THESIS TITLE: Specification and Verification of Actor Protocols with Finite-State Machines

AUTHOR: Jonathan Schuster

Ph.D. Thesis Approved to complete all degree requirements for the Ph.D. Degree in Computer Science.

Olie Shvans
Thesis Advisor

Date 2019/5/6

Date 6 May 2019

Date 6 May 2019

Date 6 May 2019

Date

GRADUATE SCHOOL APPROVAL:

Date 5/10/2019

Director, Graduate School

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

Date 5/10/19

Recipient's Signature

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI website.
Specification and Verification of Actor Protocols with Finite-State Machines

Jonathan Schuster

May 6, 2019

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

College of Computer and Information Science
Northeastern University
Boston, Massachusetts
Abstract

Many programmers use the actor model to build distributed systems. The communication aspects of such systems are notoriously hard to implement correctly, however, leading programmers to spend more time debugging protocol implementations and less time focusing on application logic. Furthermore, the common approach of specifying a protocol as a finite-state machine and verifying that the program implements this protocol is insufficient, because standard FSMs do not account for the dynamic, evolving communication topologies in actor programs.

To address this problem, this dissertation defines a specification language that augments finite-state machines with the ability to describe address-passing aspects of actor protocols. Additionally, the dissertation develops a series of proof techniques for such specifications, as well as a model-checking algorithm that verifies whether a program conforms to its specification. When applied to realistic actor programs and specifications, the model checker can both detect protocol-violating bugs and prove conformance in a reasonable amount of time.
Acknowledgments

A dissertation may be written by just one person, but no one earns a Ph.D. without lots of help from their colleagues and loved ones. As a result, I have many people to thank for helping me get this far, starting with my thesis committee:

- My advisor Olin Shivers has overseen my growth as a researcher ever since my early days at Northeastern. Over the course of many whiteboard chats that included complicated diagrams, hand-waving explanations, and Olin telling me “I didn’t understand that: explain that to me again”, I learned how to pare my ideas down to their essence and clearly explain the main points. He also constantly encouraged me along the way, especially on the days where I doubted I was good enough to be a “real” researcher. Finally, Olin has always been quick to remind me to spend quality time with family and friends and not let work take over my life.

- Matthias Felleisen introduced me to the formal study of programming languages when he taught the Ph.D.-level course during my first semester at Northeastern. Two years later when Olin went on sabbatical, Matthias acted as a substitute advisor for me and was instrumental in helping me find a research project when I had been spinning my wheels for the previous year. Along with Stephen Chang, he helped me turn that project into my first paper, which became the basis for this dissertation. I’ve also had several chats with Matthias over the years about grad school and the sort of career I’d like to have, and I am thankful for all of his advice.

- Amal Ahmed and I didn’t have many chances to work together directly, but she was nevertheless a critical part of my grad-school experience. Whenever I needed someone in the department to discuss the difficulties I was facing, Amal was there to lend an empathetic ear, and I always came away from our meetings feeling better about my situation. Although I may not have worked with her on technical matters as much as I did with my other committee members, Amal’s support and advice were equally important to my success in grad school.

- Jon Rossie was initially my manager at Cisco during a summer internship after my first year of grad school, where I was exposed to the kinds of fascinating ideas that can come from applying academic techniques to
real-world problems. Jon helped guide my research in my first few years after that internship, helping me staying grounded in problems relevant to industry programmers. We also had a number of fun calls and emails trading interesting research papers—a practice I hope to carry on with him and others as I transition into an industry career.

Mitch Wand, while not on my thesis committee, was extraordinarily generous with his time to help with my dissertation. Reading through my work line-by-line with him helped me become a much better mathematician and taught me how to present math in a rigorous way. Furthermore, while I may have thought I was a decent writer before, Mitch's feedback improved my skills immensely.

Cisco as a whole and Jon's team in particular deserve my thanks for two reasons. First, they funded me for several years of my Ph.D. studies. Second, the research in this dissertation was inspired by some of the work the team is doing and problems they have run into, so none of this would exist without that project. Thank you to all of the project team members for creating a fun space to work in and for giving me a chance to contribute in my own small way.

Past and present members of Northeastern's Programming Research Lab have had a major impact on my time in grad school. Much of what I learned about research and programming languages came not from reading assigned papers or working on my research, but from conversations with my fellow grad students. Additionally, although getting a Ph.D. can be a stressful time for just about anyone who attempts it, I was privileged to work with a group of colleagues who support one another through all of the highs and lows. I should make special mention of Stephen Chang, who as previously mentioned helped me work out the initial ideas of this dissertation. I would also be remiss if I didn't single out Ben Lerner. Ben has been there for the best and worst moments of my grad-school experience, but through it all he has constantly encouraged me to press on, and he has become a good friend in the process.

Outside of the research world, my parents Greg and Diane worked hard throughout their lives to ensure that my sisters and I had access to a top-quality education. They also taught me to have the kind of do-it-yourself attitude that is so essential for independent research (although I'm sure they would be happy to tell you stories of when I was young and a little too eager to be helpful). Their encouraging words and advice have been a constant comfort over the last eight years. My sisters Kate and Lauren and my in-laws in the Wagner family have cheered me along in the process, as well.

Finally, there is my wife Claire, the love of my life. Meeting her early in my grad-school experience has made the journey a thousand times better, and I am incredibly lucky that she agreed to spend her life with me. She has been an unwavering source of support, a wise counselor, a role model for work ethic, a smiling face at the end of a long day, and my number-one fan. Her love and support mean the world to me, and I can only hope to be as wonderful a partner to her over the many years we will share together as she has been to me.

Thank you to one and all. I wouldn't be here without you.

— Jonathan Schuster
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 The Problem and My Thesis</td>
<td>1</td>
</tr>
<tr>
<td>1.3 The Current Landscape</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Structure of the Dissertation</td>
<td>5</td>
</tr>
<tr>
<td><strong>2 CSA: Actors as Finite-State Machines</strong></td>
<td>7</td>
</tr>
<tr>
<td>2.1 Notation</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Syntax and Informal Semantics</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Example: Stream Processing</td>
<td>12</td>
</tr>
<tr>
<td>2.4 Type System</td>
<td>16</td>
</tr>
<tr>
<td>2.5 Formal Semantics</td>
<td>18</td>
</tr>
<tr>
<td>2.6 Related Work</td>
<td>30</td>
</tr>
<tr>
<td><strong>3 APS: Actor Protocols as Finite-State Machines</strong></td>
<td>33</td>
</tr>
<tr>
<td>3.1 Syntax and Intuitive Semantics</td>
<td>34</td>
</tr>
<tr>
<td>3.2 Extended Example: Stream-Processing</td>
<td>46</td>
</tr>
<tr>
<td>3.3 APS Strengths and Weaknesses</td>
<td>49</td>
</tr>
<tr>
<td>3.4 History Markers</td>
<td>50</td>
</tr>
<tr>
<td>3.5 Marked Transition Semantics for CSA</td>
<td>57</td>
</tr>
<tr>
<td>3.6 Marked Transition Semantics for APS</td>
<td>64</td>
</tr>
<tr>
<td>3.7 Conformance</td>
<td>74</td>
</tr>
<tr>
<td>3.8 Refinements to Conformance</td>
<td>77</td>
</tr>
<tr>
<td>3.9 Related Work</td>
<td>78</td>
</tr>
<tr>
<td><strong>4 A Manual Conformance Proof</strong></td>
<td>83</td>
</tr>
<tr>
<td><strong>5 Abstracting CSA</strong></td>
<td>93</td>
</tr>
<tr>
<td>5.1 Overview of the Abstract Interpretation</td>
<td>94</td>
</tr>
<tr>
<td>5.2 Abstract Interpretation for Programs</td>
<td>95</td>
</tr>
<tr>
<td>5.3 Abstract Transitions for PSMs</td>
<td>118</td>
</tr>
</tbody>
</table>
CONTENTS

B Externals-Only Conformance 231
B.1 Definitions for Proofs . . . . . . . . . . . . . . . . . . . . . . . . . . 232
B.2 EO Input-Pattern-Matching Lemma . . . . . . . . . . . . . . . . . 235
B.3 EO Output-Pattern-Matching Lemma . . . . . . . . . . . . . . . . . 236
B.4 Externals-Only PSM Input Lemma . . . . . . . . . . . . . . . . . . 236
B.5 Externals-Only Specification Input Lemma . . . . . . . . . . . . . 238
B.6 Externals-Only PSM Output Lemma . . . . . . . . . . . . . . . . . 239
B.7 Externals-Only Specification Output Lemma . . . . . . . . . . . . 240
B.8 Externals-Only Silent Step Lemma . . . . . . . . . . . . . . . . . . 241
B.9 EO Specification Simulation Lemma . . . . . . . . . . . . . . . . . 241
B.10 Externals-Only Simulation Lemma . . . . . . . . . . . . . . . . . . 242
B.11 Externals-Only Conformance Theorem . . . . . . . . . . . . . . . . 249

C External-Representative Conformance 253
C.1 Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 255
C.2 External-Representative Message Lemma . . . . . . . . . . . . . . 256
C.3 ER Receptionist Lemma . . . . . . . . . . . . . . . . . . . . . . . . . 256
C.4 External-Representative Simulation Lemma . . . . . . . . . . . . . 257
C.5 ER Specification Simulation Lemma . . . . . . . . . . . . . . . . . . 258
C.6 External-Representative Conformance Theorem . . . . . . . . . . . 259

D Single-Handler Conformance 261
D.1 Definitions for the Proof . . . . . . . . . . . . . . . . . . . . . . . . . 263
D.2 Handler Step Commutativity Lemma . . . . . . . . . . . . . . . . . 263
D.3 Program Fair Suffix Lemma . . . . . . . . . . . . . . . . . . . . . . 264
D.4 Handler Termination Lemma . . . . . . . . . . . . . . . . . . . . . . 265
D.5 PSM Output Commutativity Lemma . . . . . . . . . . . . . . . . . . 266
D.6 Specification Output Commutativity Lemma . . . . . . . . . . . . . 269
D.7 Specification Step Commutativity Lemma . . . . . . . . . . . . . . . 269
D.8 Specification Fair Suffix Lemma . . . . . . . . . . . . . . . . . . . . 270
D.9 Single-Handler Isomorphism Lemma . . . . . . . . . . . . . . . . . 271
D.10 Single-Handler Prefix Lemma . . . . . . . . . . . . . . . . . . . . . 272
D.11 Handler Continuation Lemma . . . . . . . . . . . . . . . . . . . . . 272
D.12 Rearrange Lemma . . . . . . . . . . . . . . . . . . . . . . . . . . . . 276
D.13 Unrearrange Lemma . . . . . . . . . . . . . . . . . . . . . . . . . . 278
D.14 Single-Handler Conformance Theorem . . . . . . . . . . . . . . . . 281

E Deterministic-Handler Conformance 283

F Event-Step Conformance 289

G PSM Conformance 293
G.1 Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 294
G.2 Concrete Specification Well-Formed Preservation . . . . . . . . . . 294
G.3 Concrete Distinct-Marker Preservation . . . . . . . . . . . . . . . . 294
G.4 PSM Conformance Theorem . . . . . . . . . . . . . . . . . . . . . . . 295
## H Type System for Abstract CSA

- **H.1 Abstract Program Configurations** ........................................ 299
- **H.2 Abstract Behaviors** ....................................................... 300
- **H.3 Abstract State Definitions** .............................................. 300
- **H.4 Abstract Timeout Clauses** ............................................... 300
- **H.5 Abstract Expressions** ...................................................... 300
- **H.6 Labels** ................................................................................ 302
- **H.7 Type Depth** ........................................................................ 302

## I Proofs for Abstract Conformance

- **I.1 Definitions** ............................................................................ 305
- **I.2 Value Approximation Lemma** .............................................. 309
- **I.3 Abstract Substitution Lemma** .............................................. 309
- **I.4 Abstract Context Lemma** ..................................................... 310
- **I.5 Well-Formed Preservation Lemma** ....................................... 310
- **I.6 Abstract Well-Formed Preservation Lemma** ............................ 311
- **I.7 Extra Markers Lemma** .......................................................... 311
- **I.8 Deterministic Marking Lemma** ............................................. 312
- **I.9 Marker Soundness Lemma** .................................................... 312
- **I.10 New Behavior Lemma** ....................................................... 314
- **I.11 Replaced Behavior Lemma** ............................................... 316
- **I.12 Maximal Value Lemma** ...................................................... 318
- **I.13 Well-Typed Maximal Value Lemma** .................................... 318
- **I.14 Internal Address Types Lemma** .......................................... 319
- **I.15 Functional-Step Soundness Lemma** .................................... 319
- **I.16 Merge Unreachability Lemma** ............................................. 319
- **I.17 Mergeability Preservation Lemma** ....................................... 319
- **I.18 Quasi-Commutativity Theorem** .......................................... 320
- **I.19 Approximation Mergeability Lemma** ................................... 321
- **I.20 Merge Argument Lemma** ................................................... 322
- **I.21 Merge Result Lemma** ........................................................ 322
- **I.22 Multiple Merge Result Lemma** .......................................... 323
- **I.23 Fully Merged Preservation Lemma** .................................... 323
- **I.24 Message-Addition Soundness Lemma** ................................ 324
- **I.25 Soundness of Abstract CSA Lemma** .................................. 328
- **I.26 Soundness of Event Steps Lemma** ...................................... 347
- **I.27 Event-Step Execution Soundness Lemma** .............................. 349
- **I.28 Soundness of Fair Executions Lemma** .................................. 351
- **I.29 Monitored Correspondents Lemma** .................................... 352
- **I.30 Input Pattern Lemma** ........................................................ 353
- **I.31 Output Pattern Lemma** ....................................................... 353
- **I.32 Label Erasure Lemma** ........................................................ 355
- **I.33 PSM Completeness Lemma** .............................................. 356
- **I.34 Configuration Completeness Lemma** .................................. 358
- **I.35 Expression Reflexivity Lemma** .......................................... 362
- **I.36 Message-Map Reflexivity Lemma** ....................................... 363
<p>| CONTENTS |
|------------------|------------------|
| I.37 Approximation Reflexivity Lemma | 363 |
| I.38 Expression Transitivity Lemma | 364 |
| I.39 Message-Map Transitivity Lemma | 364 |
| I.40 Approximation Transitivity Lemma | 365 |
| I.41 Externals-Only Preservation Lemma | 365 |
| I.42 Abstract Externals-Only Preservation Lemma | 367 |
| I.43 Single-Handler Preservation Lemma | 367 |
| I.44 Abstract Conformance Theorem | 369 |
| I.45 Summary Conformance Theorem | 372 |
| J Type Preservation Proofs | 375 |
| J.1 Substitution Lemma | 375 |
| J.2 Instantiation Type Preservation Lemma | 376 |
| J.3 Markings Type Preservation Lemma | 377 |
| J.4 Type Inversion Lemma | 377 |
| J.5 Context Lemma | 378 |
| J.6 IntAddrTypes Correctness Lemma | 378 |
| J.7 Functional Step Type Preservation Lemma | 378 |
| J.8 Type Preservation Lemma | 379 |
| J.9 Abstraction Type Preservation | 383 |
| J.10 Subtype Inversion Lemma 1 | 383 |
| J.11 Subtype Inversion Lemma 2 | 383 |
| J.12 Abstract Canonical Forms Lemma | 384 |
| J.13 Abstract Type Inversion Lemma | 385 |
| J.14 Typed Value Depth Lemma | 386 |
| J.15 Merge Type Preservation Lemma | 387 |
| J.16 Message Addition Type Preservation Lemma | 387 |
| J.17 Abstract Typed Substitution Lemma | 388 |
| J.18 IntAddrTypes Depth Lemma | 388 |
| J.19 Abstract Functional Step Type Preservation Lemma | 389 |
| J.20 Abstract Type Preservation Lemma | 390 |
| K Proofs for Transformation Conformance | 395 |
| K.1 Miscellaneous Definitions | 395 |
| K.2 Conformance Reflection | 397 |
| K.3 Definition of SimExecs | 401 |
| K.4 Fair Suffix Lemma | 403 |
| K.5 Specification Well-Formed Preservation | 404 |
| K.6 Empty Matchable Output Lemma | 406 |
| K.7 Externals-Only Label Lemma | 407 |
| K.8 Monitored Matchable Markers Lemma | 410 |
| K.9 Used Marker Lemma | 411 |
| K.10 Used/Monitored Marker Lemma | 412 |
| K.11 Distinct-Marker Silent Preservation Lemma | 413 |
| K.12 Distinct-Marker Preservation Lemma | 414 |
| K.13 Silent-Step Partition Lemma | 418 |</p>
<table>
<thead>
<tr>
<th></th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>K.14</td>
<td>Communication-Step Partition Lemma ........................................ 419</td>
</tr>
<tr>
<td>K.15</td>
<td>Weak-Silent-Step Partition Lemma ........................................... 420</td>
</tr>
<tr>
<td>K.16</td>
<td>Specification Weak-Step Partition Lemma ................................... 421</td>
</tr>
<tr>
<td>K.17</td>
<td>Matchable Labels Lemma ................................................................ 423</td>
</tr>
<tr>
<td>K.18</td>
<td>Fresh Matchables Lemma ................................................................ 424</td>
</tr>
<tr>
<td>K.19</td>
<td>Distinct Matchables Lemma ...... .................. .......................... 424</td>
</tr>
<tr>
<td>K.20</td>
<td>Specification Summary Partition Lemma ...................................... 426</td>
</tr>
<tr>
<td>K.21</td>
<td>Monitored Marker Permanence Lemma .......................................... 431</td>
</tr>
<tr>
<td>K.22</td>
<td>Post-Fulfillment Monitoring Lemma ........................................... 432</td>
</tr>
<tr>
<td>K.23</td>
<td>Fair Specification Suffix Lemma ................................................ 433</td>
</tr>
<tr>
<td>K.24</td>
<td>SimExecs Simulation Lemma ...... .............................................. 435</td>
</tr>
<tr>
<td>K.25</td>
<td>SimExecs Non-Emptiness Lemma .................................................. 440</td>
</tr>
<tr>
<td>K.26</td>
<td>SimExecs Determinism Lemma ..................................................... 446</td>
</tr>
<tr>
<td>K.27</td>
<td>SimExecs Common Prefix Lemma .................................................. 447</td>
</tr>
<tr>
<td>K.28</td>
<td>SimExecs Fulfillment Lemma ....................................................... 450</td>
</tr>
<tr>
<td>K.29</td>
<td>Transformation Conformance Theorem .......................................... 455</td>
</tr>
<tr>
<td>L.1</td>
<td>Conformance-Reflection Proofs .................................................... 459</td>
</tr>
<tr>
<td>L.2</td>
<td>Definitions ........................................................................... 459</td>
</tr>
<tr>
<td>L.3</td>
<td>No Monitored Markers Lemma ...................................................... 461</td>
</tr>
<tr>
<td>L.4</td>
<td>No Silent Transitions Lemma ..................................................... 462</td>
</tr>
<tr>
<td>L.5</td>
<td>Distinct Monitored Markers Lemma ............................................. 464</td>
</tr>
<tr>
<td>L.6</td>
<td>Summary Synchronization Lemma .................................................. 466</td>
</tr>
<tr>
<td>L.7</td>
<td>Split Silent Transitions Lemma .................................................. 468</td>
</tr>
<tr>
<td>L.8</td>
<td>Split Transitions Lemma ................................................................ 472</td>
</tr>
<tr>
<td>L.9</td>
<td>Split Weak-Step Transitions Lemma ............................................. 479</td>
</tr>
<tr>
<td>L.10</td>
<td>Split Event-Step Transitions Lemma ........................................... 480</td>
</tr>
<tr>
<td>L.11</td>
<td>Split Summary Transitions Lemma ............................................... 483</td>
</tr>
<tr>
<td>L.12</td>
<td>Split Conformance Reflection Theorem ......................................... 485</td>
</tr>
<tr>
<td>L.13</td>
<td>Remap Well-Formed Preservation Lemma ........................................ 491</td>
</tr>
<tr>
<td>L.14</td>
<td>Remap Externals-Only Preservation Lemma .................................... 492</td>
</tr>
<tr>
<td>L.15</td>
<td>Abstract Single-Handler Preservation Lemma .................................. 492</td>
</tr>
<tr>
<td>L.16</td>
<td>Abstract Extra Markers Lemma .................................................... 493</td>
</tr>
<tr>
<td>L.17</td>
<td>Abstract Marker Soundness Lemma .............................................. 494</td>
</tr>
<tr>
<td>L.18</td>
<td>Approximation Substitution Lemma .............................................. 494</td>
</tr>
<tr>
<td>L.19</td>
<td>Abstract Internal Address Types Lemma ....................................... 495</td>
</tr>
<tr>
<td>L.20</td>
<td>Abstract Functional-Step Soundness Lemma .................................... 496</td>
</tr>
<tr>
<td>L.21</td>
<td>Approximation Soundness Lemma .................................................. 496</td>
</tr>
<tr>
<td>L.22</td>
<td>Event-Step Approximation Soundness Lemma ................................... 515</td>
</tr>
<tr>
<td>L.23</td>
<td>Fair Execution Approximation Soundness Lemma ................................ 517</td>
</tr>
<tr>
<td>L.24</td>
<td>Label Sequence Construction Lemma ........................................... 519</td>
</tr>
<tr>
<td>L.25</td>
<td>Abstract Configuration Completeness Lemma .................................. 523</td>
</tr>
<tr>
<td>L.26</td>
<td>Approximating Transformation Lemma ........................................... 527</td>
</tr>
<tr>
<td>L.27</td>
<td>Fully Merged Expansion Lemma ................................................... 534</td>
</tr>
<tr>
<td>L.28</td>
<td>Remap Approximation Lemma ....................................................... 535</td>
</tr>
</tbody>
</table>
## CONTENTS

- N.12 Label Names Lemma ........................................... 604
- N.13 Bounded Merge Lemma ........................................ 606
- N.14 Program Origin Preservation Lemma .......................... 606
- N.15 Specification Origin Preservation Lemma .................... 607
- N.16 Bounded Types Lemma ......................................... 610
- N.17 Bounded Expressions Lemma .................................. 610
- N.18 Bounded Timeout Clauses Lemma ............................... 612
- N.19 Bounded State Definitions Lemma ............................. 613
- N.20 Bounded Behaviors Lemma .................................... 614
- N.21 Bounded Program Configurations Lemma ..................... 615
- N.22 Bounded PSMs Lemma .......................................... 617
- N.23 Bounded Pairs Lemma .......................................... 617
- N.24 Unique Context Lemma ........................................ 619
- N.25 Finite Maximal Values Lemma ................................ 619
- N.26 Finite Triggers Lemma ......................................... 620
- N.27 Finite Loop Allocations Lemma ................................ 621
- N.28 Finite Allocations Lemma ...................................... 623
- N.29 ProgSteps Termination Lemma ................................ 627
- N.30 Finite Output Matches Lemma .................................. 629
- N.31 PsmSimulateOutput Termination Lemma ....................... 629
- N.32 SimulateOutput Termination Lemma ........................... 630
- N.33 SimulateOutputs Termination Lemma ........................... 631
- N.34 MatchingSpecSteps Termination Lemma ....................... 631
- N.35 Explore Termination Lemma .................................... 631
- N.36 Finite Transformation Steps Lemma ............................ 636
- N.37 Prune Termination Lemma ...................................... 636
- N.38 FindFulfillingPairs Termination Lemma ....................... 637
- N.39 ModelCheck Termination Theorem .............................. 639
Chapter 1

Introduction

1.1 Background

Distributed systems are one of the most widely used means to structure software systems. Such a system harnesses the collective computing power of all of its constituent machines, thus allowing the system to scale even as the exponential improvements in hardware performance from Moore’s law begin to taper off.

Although distributed systems can be built directly in terms of networking primitives such as sockets, programmers sometimes turn to languages or frameworks based on the actor model that provide more suitable abstractions. A program in the actor model consists of many different processes, called actors, that communicate by asynchronous message-passing. Upon receiving a message, an actor can spawn new actors, send messages to other actors, or change how it will handle future received messages. First developed by Hewitt et al. [59] and further investigated by Agha et al. [3, 4], the model is now the basis for Erlang [10] and Scala’s Akka framework [5].

A distinguishing characteristic of actors is their address-passing capability: that is, their ability to include actor addresses in messages. This allows actors to learn about other actors as the program evolves, resulting in a dynamic communication topology. This capability increases the expressive power of the language, but it also makes reasoning about programs difficult.

1.2 The Problem and My Thesis

Because an actor program’s behavior is defined in terms of its communication with the outside world, its correctness is defined in terms of a protocol that describes its expected communication behavior. For example, the protocol for a program that computes a running average of an incoming stream of numbers might require that

- upon receiving an element of the stream, the program sends back an acknowledgment,
• every request for the current running average receives a response, and
• after the program receives an end-of-stream message, it sends no further acknowledgments.

When reasoning about such protocols, it can be useful to focus solely on the expected patterns of communication while ignoring the computational aspects. For example, the properties described above concern the messages the program is expected to send in response to inputs from the outside world, but do not specify how to compute the running average itself. However, even such lightweight specifications describe behaviors that are easy to implement incorrectly.

To specify these kinds of communication patterns, programmers often use finite-state machines (FSMs). Each transition of the FSM describes the program’s expected reaction to a given input, and the different states allow for different behavior depending on the sequence of messages received so far. This model is used for network protocols such as TCP [90] and the Alternating Bit Protocol [15].

Traditional FSMs are insufficient for describing protocols for actor programs (hereafter called actor protocols), however. The dynamic communication topology of an actor program means that a protocol must describe not only what messages a program can send and receive, but also how the addresses carried in messages should be used. For example, a request for the running average in the above example might contain an address, and the protocol would specify that the response must be sent to that address. Therefore, to describe actor protocols, FSMs would have to be augmented with the ability to describe address-passing patterns.

Even if FSMs were extended to specify actor protocols in this way, implementing such protocols would still be difficult. It can be tedious to ensure that every message is handled correctly in every state, and therefore it is easy to make mistakes. Checking that every rule is followed correctly in every possible case is a job better left to a computer than to a human, so it would be useful to have a tool that could automatically verify whether a program implements a given protocol correctly. Such a tool would help programmers avoid the “obvious” sorts of errors and allow them to focus on the more complicated aspects of their program that are harder to reason about automatically.

Verification is a hard problem in general, but there are three advantages in this situation. First, because these FSM-like specifications would describe only high-level patterns of communication rather than low-level computational details, it should be easier to prove that a program satisfies such a specification. Second, actor protocols are often implemented directly as FSMs, to the extent that Erlang and Akka provide built-in support for this purpose in the form of the gen_fsm behavior and the FSM trait, respectively. Therefore, when thinking about verifying an actor program against an FSM-based specification, it makes
sense to consider a setting in which the program is written in a language geared
towards building actors with FSMs. Third, this dissertation will show that many
actor protocols can be written in a language that syntactically forces all event-
handler expressions to be terminating (i.e., they may not contain unbounded
loops or recursion), which lends further reasoning power.

This leads me to my thesis.

**Finite-state, address-passing specifications can be used to au-
tomatically verify non-trivial protocols in actor programs.**

As evidence for this thesis, this dissertation presents the following contribu-
tions:

1. a specification language for describing actor protocols
   • The language uses address-passing, finite-state machines and it
   comes with a notion of conformance that formally defines what it
   means for a program in an FSM-based actor language to implement
   (conform to) a specification. The language can express both safety and
   liveness properties of actor protocols.

2. a series of refinements to conformance to make proving conformance easier

3. an abstract interpretation for the FSM-based programming language

4. a state-space-reduction technique
   • The technique further abstracts the program at each step during
   model checking based on what is relevant to its specification.

5. a model-checking algorithm for verifying conformance to a specification
   • The algorithm is sound, but not complete, meaning that a successful
   verification result implies that the program does indeed conform to
   its specification, but there are some conforming programs which the
   algorithm is unable to verify.

6. a set of optimizations for the model-checking algorithm

7. an empirical validation of the model-checking algorithm’s precision

### 1.3 The Current Landscape

Later chapters discuss the related work in more detail, but to provide the reader
with a sense of perspective, this section summarizes the state of the art in ensuring
that actor programs are correct.
FSMs for Protocols  The FSMs used to describe protocols are often informal diagrams, but there are also formal languages such as statecharts [55] and SDL [102] that endow these diagrams with a formal semantics. None of these languages have a simple means for expressing address-passing, however. I am unaware of any work that attempts to statically verify an actor program directly against an FSM-like specification. Chapter 3 discusses these works in more detail.

Temporal Logics  As an alternative to FSMs, some researchers have proposed the use of temporal logics to specify the expected behavior of programs. Dam et al. [36] use the first-order $\mu$-calculus to specify properties of Erlang programs similar to the kinds of behavior described in my specification language, but their language is more expressive and therefore its associated theorem prover [48] requires more human interaction than my model-checking algorithm. Lamport's Temporal Logic of Actions (TLA) [70], and especially its extension TLA+ [72] are a popular means for specifying the behavior of distributed systems in industry, but it does not yet appear to be used with actor-based technologies. Chapter 3 further compares my work to temporal logics.

Testing  When it comes to verifying the behavior of an actor program, testing is still the most common approach in industry. Unit testing is of course common, but more rigorous approaches are also used. Both Quviq QuickCheck [12] and PropEr [87], are property-based testers for Erlang that generate random test cases for checking user-defined properties. The P [40] and P# [37] projects (which also structure actor-like processes as FSMs) both have schedulers that allow the same test case to be run under many different schedulers, thereby detecting assertion violations that can occur as a result of different execution orders. I view all of these dynamic techniques as complementary to my static approach. Chapter 7 discusses these dynamic approaches in more detail.

Model-Checking  A variety of model checkers [11, 43, 49, 64] have been built for checking properties of Erlang programs. Most of these are focused on verifying individual properties of a protocol specified with temporal logic, such as checking that a certain bad state is never reached, rather than checking that an entire FSM-based protocol description is implemented correctly. Several of the existing model checkers explore only a bounded subset of the program’s state space, or require programmers to devise their own abstractions to explore the entire state space. Chapter 7 describes these in more detail.

Type Systems  Finally, various type systems have been designed for message-passing programs. A type system can be seen as both a specification and a verification tool, in that a type for a communication channel specifies the protocol that should be followed when sending or receiving on that channel, while type-checking verifies that the program uses the channel as specified. There are many such type systems, but the most well-known are session types [62]. A session type
1.4 STRUCTURE OF THE DISSERTATION

describes how a given communication channel (“session”) should be used by a
fixed set of interacting processes. Type-checking ensures that the processes coor-
dinate with each other correctly, usually to prevent conditions such as deadlock.
Session types are difficult to apply to dynamic settings such as actor programs,
in which different processes can join and leave conversations at any time, and
where a single “conversation” can occur over multiple actor addresses. Chapter 3
discusses session types in more detail, as well as other relevant type systems.

1.4 Structure of the Dissertation

The first part of this dissertation (chapters 2–4) focuses on defining both the pro-
gramming language and specification language, and on showing how to prove
conformance by hand. The latter part of the dissertation (chapters 5–9) then
develops techniques to build an automatic model checker for programs and spec-
ifications written in these two languages, and it evaluates the resulting tool on
realistic examples. The chapter-by-chapter breakdown is as follows:

- Chapter 2 defines CSA, a programming language for implementing actors
  as communicating finite-state machines.
- Chapter 3 defines APS, a specification language for describing actor pro-
tocols as restricted, address-passing finite-state machines. The formal se-
manitics of APS is defined in terms of a conformance relation that defines
what it means for a CSA program to implement the protocol described by
an APS specification.
- Chapter 4 presents an example conformance proof for a running example
  built up in chapters 2 and 3, to provide a sense of the “shape” of such proofs.
The job of the model checker is to automate the construction of such proofs.
- Chapter 5 develops an abstract interpretation for CSA programs that helps
  reduce their possible state-space.
- Chapter 6 builds on top of chapter 5 to define a new notion of conformance
  that further abstracts the program as it evolves, depending on what aspects
  of the program are relevant to the specification.
- Chapter 7 presents the model-checking algorithm itself.
- Chapter 8 defines a set of optimizations for the model-checking algorithm.
- Chapter 9 evaluates the model-checking algorithm on a set of realistic actor
  programs and specifications.
- Chapter 10 describes ideas for future work and concludes.

Chapters 2, 3, 5, and 7 also discuss related work for the topics introduced in
those chapters.
Chapter 2

CSA: Actors as Finite-State Machines

This chapter introduces CSA (Communicating State Actors), a communication-focused programming language that incorporates the actor-as-FSM pattern as its core computational model.¹ The language is used as a basis for the rest of the dissertation.

CSA models the core constructs of an internal research language explored by developers at Cisco. That language and the problems faced by programmers working in it directly inspired this dissertation. Section 2.2.4 further describes the Cisco language’s influence on CSA.

The next few sections describe the language via its syntax, intuitive semantics, and a small example. The subsequent sections then define CSA’s formal semantics and type rules, and the chapter concludes with related work.

2.1 Notation

This section introduces notation used in the rest of this dissertation. \(\mathcal{P}(A)\) stands for the powerset of \(A\), \(\mathcal{F}(A)\) stands for the set of all finite subsets of \(A\), and \(\mathcal{M}(A)\) stands for the set of all finite multisets of \(A\). Both the disjoint union of two sets \(A\) and \(B\) and the multiset sum of two multisets \(A\) and \(B\) is written \(A \uplus B\); the context will always make it obvious which meaning is intended. Tuples are denoted with angle brackets, as in the 3-tuple \(\langle a_1, a_2, a_3 \rangle\). A term \(\bar{a}\) stands for a finite sequence of \(a\)'s, \(\epsilon\) stands for an empty sequence, and \(A^*\) stands for the set of all finite sequences of elements of \(A\).

The set of partial functions from \(A\) to \(B\) is written \(A \rightarrow B\). The empty partial function (i.e., the function whose domain is the empty set) is written \(\varnothing\). The notation \(f[x \mapsto y]\) stands for the partial function that maps \(x\) to \(y\) and maps all other inputs \(x'\) to \(f(x')\). When \(f = \varnothing\), this can be shortened simply to \([x \mapsto y]\).

¹This is an updated version of the language presented in a previous paper [97].
disjoint union of two partial functions $f \cup g$ is defined such that $(f \cup g)(x) = x'$ if and only if $f(x) = x'$ or $g(x) = x'$; the disjoint union is defined only if $\text{dom}(f) \cap \text{dom}(g) = \emptyset$. The composition $g \circ f$ of two partial functions is defined such that $(g \circ f)(x) = g(f(x))$ for all $x$; it is undefined if either $f(x)$ or $g(f(x))$ is undefined. The restriction of a partial function $f$ to a set $A$, written $f\big|_A$, is defined such that $f\big|_A(x) = f(x)$ for all $x \in \text{dom}(f) \cap A$, and undefined otherwise.

2.2 Syntax and Informal Semantics

CSA programs consist of independent processes called actors that can communicate with other actors via message-passing. An actor is an event-handling process: whenever it receives a message, it can send messages, spawn new actors, and finally update its state before suspending execution until the next event occurs. Message-passing in CSA is asynchronous; each actor has a mailbox in which messages sent to it reside until the actor handles them. Messages are unordered, so the mailbox functions as a bag (multiset) of messages rather than a queue.

A CSA program runs in the context of some environment made up of other unknown actors that communicate with the program. Those unknown actors are called external actors, while actors in the program are called internal actors. Thus, a program can serve as a single component of some larger system. Modeling programs in this way allows us to separate a program's public interface (i.e., its communication with the environment) from its private implementation details.

Figure 2.1 lists the S-expression-based syntax for CSA programs. A program $P$ defines its interface with the environment and the initial set of actors. The receptionists clause lists the internal actors initially advertised to the environment, which allows actors in the environment to send them messages. Conversely, the externals clause names the external actors initially known to the program, enabling actors inside the program to communicate with the environment. The type associated with each receptionist (external) indicates the type of messages the environment (program, respectively) may send to that actor. Then the sub-clauses of the let-actors clause each spawn a new actor and bind it to the declared variable $x$. The final sequence of variables $\overline{x}$ provides the list of actor addresses to be bound as receptionists (some actors are exposed at more than one name, others are not exposed as receptionists at all). A CSA actor is structured as a finite-state machine, with each state $q$ parameterized over some arguments $\overline{x}_q$. In a program's initial spawn clauses, each actor's goto expression provides its initial state, and its state definitions $Q_q$ define its transitions. Each initial state argument is an open value $ov$. An open value is a value-like term that may have free variables. Throughout this disser-

---

2 The language and semantic presentation is inspired by Agha et al. [3].
3 The notion of receptionist here differs slightly from that given by Agha et al. to account for the addition of types. Whereas a receptionist in their work is a particular actor (or that actor's address), the term "receptionist" here refers to an address paired with a type, because the program may provide an actor's address to the environment at multiple different types.
\[ P ::= (\text{program} (\text{receptionists} \ [x \ \tau]) \ (\text{externals} \ [x \ \tau]) \quad \text{(Programs)} \\
\quad (\text{let-actors} \ (\{x \ (\text{spawn} \ \tau \ (\text{goto} \ q \ \overline{Q})) \}) \ \overline{Q})) \]

\[ Q ::= (\text{define-state} \ (q \ [x_{s} \ \tau]) \ x_{m} \ e \ t_{c}) \quad \text{(State Definitions)} \]

\[ t_{c} ::= \text{no-timeout} \ | \ [ (\text{timeout} \ ov) \ e \] \quad \text{(Timeout Clause)} \]

\[ e ::= (\text{spawn} \ \tau \ e \ \overline{Q}) \ | \ (\text{goto} \ q \ \overline{e}) \ | \ (\text{send} \ e \ e) \quad \text{(Expressions)} \]

\[ \overline{e} ::= \text{(begin} \ \overline{e} \ \overline{e} \ 	ext{)} \ | \ (\text{case} \ e \ [(t \ \overline{e})] \ (\text{fold} \ \tau \ e) \ | \ (\text{unfold} \ \tau \ e) \ | \ \text{(list} \ \overline{\overline{e}}) \]

\[ \overline{\overline{e}} ::= \text{(dict} \ [\overline{ov} \ \overline{ov}] \ | \ (\text{for/fold} \ [x \ e] \ [x \ e] \ e \ | \ (o \ \overline{e}) \ | \ n \ | \ \text{str} \ | \ x \]

\[ \overline{ov} ::= (\text{record} \ [r \ \overline{ov}] \ | \ (\text{variant} \ t \ \overline{ov}) \ | \ (\text{fold} \ \tau \ \overline{ov}) \quad \text{(Open Values)} \]

\[ \overline{ov} ::= \text{(list} \ \overline{ov}) \ | \ (\text{dict} \ [\overline{ov} \ \overline{ov}] \ | \ n \ | \ \text{str} \ | \ x \]

\[ \tau ::= \text{Nat} \ | \ \text{String} \ | \ (\text{Variant} \ [t \ \tau]) \ | \ (\text{Record} \ [r \ \tau]) \ | \ (\text{Addr} \ \tau) \quad \text{(Types)} \]

\[ \tau ::= \text{rec} \ X \ (\text{Addr} \ \tau) \ | \ X \ | \ (\text{List} \ \tau) \ | \ (\text{Dict} \ \tau \ \tau) \]

\[ q \in \text{StateName} \quad \text{(State Names)} \]

\[ x \in \text{Var} \quad \text{(Variables)} \]

\[ X \in \text{TypeVar} \quad \text{(Type Variables)} \]

\[ n \in \mathbb{N} \quad \text{(Natural Numbers)} \]

\[ \text{str} \in \text{String} \quad \text{(Strings)} \]

\[ o \in \text{PrimOp} \quad \text{(Primitive Operations)} \]

\[ \ell \in \text{ProgLoc} \cup \text{union} \ \text{EnvLoc} \quad \text{(Syntactic Locations)} \]

\[ r \in \text{RecField} \quad \text{(Field Names)} \]

\[ t \in \text{Tag} \quad \text{(Variant Tags)} \]

Figure 2.1: CSA syntax (keywords in \textbf{bold}, all parentheses and brackets are literals)
CHAPTER 2. CSA: ACTORS AS FINITE-STATE MACHINES

tation, the term “value” on its own is assumed to mean a closed value $v$ (defined in section 2.5.1).

2.2.1 Handling Events

Two kinds of events trigger state transitions in CSA: received messages and time-outs. To handle a received message, an actor in state $q$ retrieves the message from its mailbox and evaluates the handler expression $e$ from the corresponding state definition (define-state $$(q \left[ x_s \tau \right] ) x_m e tc$). In that handler expression, each state argument is bound to the corresponding variable $x_s$, and the message is bound to the formal parameter $x_m$.

A state may also declare a timeout clause $$(timeout ov) e$$. If $ov$ evaluates to $n$ and the actor does not receive a message before $n$ milliseconds elapse, then the actor executes the timeout clause’s handler expression $e$ instead of the message handler.\footnote{In an actual implementation of CSA, a message that arrives shortly after $n$ milliseconds have elapsed may be handled instead of the timeout, as a result of imprecision in the timing. This subtlety is irrelevant for this dissertation, however.}

While handling an event, an actor can spawn other actors and send messages to actors (including itself). The expression $$(spawn \ell \tau e Q)$$ spawns a new actor that accepts messages of type $\tau$ and whose states are defined by $Q$. The expression $e$ is an initial handler expression for the actor to execute when it starts.\footnote{In a practical CSA-like language, actors would be defined in terms of class-like definitions, and $$spawn$$ would instead construct new instances of those classes. The class-less version here simplifies the presentation of the language.}

The result of a $$spawn$$ expression is the new actor’s address: a unique identifier used to send messages to that actor. The address is also bound to $$self$$ in the context of the new actor. Addresses have type $$(Addr \tau)$$, where $\tau$ is the type of message the actor can receive. Address types are contravariant in the message type, allowing an address of type $$(Addr \tau)$$ to be used in a context that sends it only messages of some subtype $\tau'$ of $\tau$.

The label $\ell$ on a $$spawn$$ expression indicates the syntactic location of that expression and is used when creating addresses. In chapter 5, we will see how an abstract interpretation for addresses takes advantage of these labels.

The expression $$(send e_a e_v)$$ sends to the address produced by $e_a$ the value produced by $e_v$. The message sits in the mailbox for $e_a$ until the actor is ready to handle it.

A $$goto$$ expression ends evaluation of a handler expression by transitioning to the named state and passing the given arguments to that state. The actor then waits in that state for the next event. As with $$return$$ statements in C-like languages, any remaining continuation is dropped when the $$goto$$ is evaluated.

2.2.2 Other Expressions and Types

The remaining expressions are standard values and control forms. Record fields are accessed using the $$:$$ form. A variant with tag $t$ is constructed with
2.2. SYNTAX AND INFORMAL SEMANTICS

(\texttt{variant} \(t \tau\)), and the \texttt{case} expression deconstructs variants, binding the variant's fields in the matching clause. These are similar to ML-style sum-of-products datatypes and have types marked with the keyword \texttt{Variant}. The primitive operations \(o\) include standard operations on natural numbers and strings as well as operations to add and retrieve values from lists and dictionaries. For polymorphic types such as \((\texttt{List}\ \tau)\), every possible type has its own set of operators for that type (e.g., \texttt{cons}_{\texttt{Nat}}, \texttt{cons}_{\texttt{String}}, etc.).

To enable an abstract interpretation of CSA in chapter 5, list and dictionary expressions may contain only open values. The more general form can easily be desugared into this one by using a \texttt{case} statement to bind variables, e.g., the expression \((\texttt{list}\ e)\) can be desugared into the following:

\[
\begin{align*}
&\texttt{(case (variant Binding e)} \\
&\hspace{1em}[(\texttt{Binding } x) \ (\texttt{list} \ x)]\]
\end{align*}
\]

The expression \((\texttt{for/fold} [x_{\text{acc}}\ e_{\text{acc}}] \ [x_i\ e_i] \ e_{\text{body}})\) is like a for-loop, but it also accumulates a result as it iterates, much like a fold in functional languages.\footnote{CSA borrows \texttt{for/fold} from Racket [46].}

The initial value of the accumulated result is \(e_{\text{acc}}\). The expression then evaluates \(e_{\text{body}}\) once per item in the list \(e_i\), with \(x_i\) bound to the current list item, and \(x_{\text{acc}}\) bound to the current accumulated result. The result of \(e_{\text{body}}\) is used as the next value of \(x_{\text{acc}}\), and the final such result is the result of the entire expression. CSA eschews unbounded while-loops in favor of \texttt{for/fold} to ensure all loops terminate.

CSA's recursive types differ slightly from the standard presentation. A type \((\texttt{rec} \ X (\texttt{Addr} \ \tau))\) indicates a greatest-fixed-point type with name \(X\), but the recursive type is limited to address types. This design choice allows messages to contain addresses of the same type as the receiving actor, which is necessary for many actor protocols, but prevents programs from creating values of unbounded (syntactic) depth. We will see later that this restriction assists in verifying that a program conforms to its specification.

Although least-fixed-point types are more common in standard functional languages, using a greatest fixed-point allows for infinite cyclic communication patterns, such as an address that can be sent to itself infinitely many times.

### 2.2.3 Messaging Guarantees

Message delivery in CSA is unduplicated, unordered, and reliable (i.e., messages are guaranteed to be delivered). The unduplicated guarantee matches that of Erlang and Akka, while the unordered guarantee is slightly weaker: those systems guarantee that when two messages have the same sender and receiver, their order is preserved. Because the rest of this dissertation assumes a weaker guarantee, the reasoning developed here also applies to systems with stronger ordering guarantees.

Neither Akka nor Erlang guarantee reliable delivery. The Erlang documentation, however, says that most programmers find it simplest to assume reliable
delivery while using Erlang’s monitoring facilities to detect failures. Thus, the CSA semantics corresponds to the mental model programmers typically use, and so the reasoning developed in this dissertation is sound up to the reliable delivery of messages.

2.2.4 Inspiration for CSA

CSA is based on a Cisco internal research programming language that adapts the actor model for implementing network protocols. I was introduced to the language during an internship at Cisco in the summer of 2012 while a team of developers were exploring the language. Further work with that Cisco team helped me understand the problems faced by both actor programmers and implementers of network protocols, and it inspired the work developed in this dissertation.

Aside from the actor-model foundation for the language, CSA borrows two of the Cisco language’s defining features. First, both languages use a finite-state machine to structure actors, while actors in other languages are arbitrary processes that either return a closure as their continuation when they finish handling an event or recursively execute a receive statement to process the next message. This state-machine orientation lends a notion of organization and predictability to actors. In this way, both CSA and the Cisco language are closer to functional programming languages in that the next state is computed and returned, rather than being a part of the current continuation as in a language such as Erlang.

Second, both CSA and the Cisco language syntactically enforce all handler expressions to be terminating: there are no unbounded loops and no recursion. The forced termination allows for more precise analyses, because it makes reasoning about unbounded continuations unnecessary.

Experience so far suggests that these differences from typical actor languages do not overly restrict actor programmers (see the discussions of example programs in chapter 9), but further evaluation is needed.

2.3 Example: Stream Processing

Bob runs a weather-tracking station just outside of Boston, and he wants to get a better understanding of the region’s infamously volatile weather. As a first step, he decides to track the average temperature at certain times of day/year, so he can answer questions such as “What’s the average temperature on a July afternoon?” or “How cold is a typical Christmas morning?”.

Bob’s station supplies him with a stream of temperature data at a rate of one reading per second, but he needs a system to filter that data and compute mean temperatures from it. Rather than reprocessing years’ worth of raw data every time he wants an updated result, he decides to design a system of actors to process this incoming data stream and compute running averages for particular time periods. In particular, Bob can turn each actor’s recording capabilities on or

\[\text{http://erlang.org/faq/academic.html#idp32851984}\]
2.3. EXAMPLE: STREAM PROCESSING

A separate manager actor will create new processor actors on demand and shut down the entire system when requested. To limit resource usage, the manager will create no more than 100 processors.

The FSM in figure 2.2 summarizes each stream processor’s behavior. It starts in the OFF state, and the system can toggle between OFF and ON with Enable and Disable messages. A Shutdown message from either state shuts down the process entirely. This FSM leaves out many details of the described protocol, though; we will return to this issue in the next chapter.

2.3.1 Implementation

Listings 2.1 and 2.2 show the program Bob came up with. Single-line comments start with ;, and block comments are nested between #| and |#. The listings also assume some bits of syntactic sugar: begin expressions and no-timeout clauses are left implicit, types are given the aliases listed in the comments, and a let expression (let ([x e]) e’) is short for (case (variant let e) [(let x e’) e’]).

The program starts with a single manager actor, which has just one state. When the manager receives a MakeProc request, if the processor limit has not been reached, it spawns a new processor and sends its address to the provided address resp (the contents of the processor-spawn expression are found in listing 2.2). The mdest address included in the request indicates where the processor should send the current mean when requested. The manager then adds the processor to its list of known processors and transitions back to the Managing state. If the limit has been reached, the manager silently ignores the request. Otherwise, upon receiving a ShutdownAll request, it sends each processor a Shutdown message before clearing its list of processors.

In listing 2.2, the processor actor’s type says it can accept five kinds of messages: AddRdg, a command to add a new temperature reading to the total; GetMean, a request for the current running mean value; Enable and Disable, commands to turn the actor’s recording capabilities on or off respectively; and Shutdown; a command to kill the entire process.
;;; ManagerMessage =
;;; (Variant [MakeProc (Addr (Addr ProcUserAPI)) (Addr Nat)]
;;; [ShutdownAll])

;;; ManagerUserAPI =
;;; (Variant [MakeProc (Addr (Addr ProcUserAPI)) (Addr Nat)])

;;; ManagerSysAPI = (Variant [ShutdownAll])

;;; ProcUserAPI =
;;; (Variant [AddRdg Nat (Addr (Variant [Ok] [NotOk]))]
;;; [GetMean]
;;; [Disable]
;;; [Enable])

;;; ProcAddr = (Addr (Variant [Shutdown]))

(program (receptionists [user-api ManagerUserAPI]
                         [sys-api ManagerSysAPI])
       (externals)
       (let-actors
        ([manager (spawn ManagerMessage
                     (goto Managing (list))
                     (define-state (Managing [processors (List ProcAddr)]) m
                       (case m
                         [(MakeProc resp mdest)
                          (case (< (length processors) 100)
                            [(True)
                             [(let (%p (spawn #|see listing 2.2|#))
                              (send resp p)
                              (goto Managing (cons p processors)))]
                             [(False) (goto Managing processors)])]
                         [(ShutdownAll)
                          (for/fold ([dummy-result (variant Shutdown)])
                           ([p processors])
                           (send p (variant Shutdown))
                           (goto Managing (list)))]
                         (goto Managing (list))))]
        (manager manager)))

Listing 2.1: The manager actor's implementation
2.3. **EXAMPLE: STREAM PROCESSING**

;;; Processor's declared type

```
(Variant [AddRdg Nat (Addr (Variant [Ok] [NotOk]))])
([GetMean]
 [Disable]
 [Enable]
 [Shutdown])
```

;;; Initial state

```
(goto Off 0 0)
```

;;; State definitions

```
(define-state (Off [sum Nat] [num-rdgs Nat]) m
  (case m
    [(AddRdg temp resp)
     (send resp (variant NotOk))
     (goto Off sum num-rdgs)]
    [(GetMean)
     (send mdest (/ sum num-rdgs))
     (goto Off sum num-rdgs)]
    [(Disable) (goto Off sum num-rdgs)]
    [(Enable) (goto On sum num-rdgs)]
    [(Shutdown) (goto Done)])
)
```

```
(define-state (On [sum Nat] [num-rdgs Nat]) m
  (case m
    [(AddRdg temp resp)
     (send resp (variant Ok))
     (goto On (+ sum temp) (+ num-rdgs 1))]  
    [(GetMean)
     (send mdest (/ sum num-rdgs))
     (goto On sum num-rdgs)]
    [(Disable) (goto Off sum num-rdgs)]
    [(Enable) (goto On sum num-rdgs)]
    [(Shutdown) (goto Done)])
)
```

```
(define-state (Done) m (goto Done))
```

Listing 2.2: The processor actor's implementation
The actor starts in the Off state, with an initial sum of 0 for all temperature data and 0 readings recorded so far. In that state, the actor rejects all new temperature readings by sending a NotOk message back to the sender. To reply for a request for the mean, the actor simply calculates the average so far and sends the result to mdest. The Disable message does not affect the actor in this state, but the Enable message causes it to transition to the On state. Finally, a shut-down request causes the actor to transition to the Done state, dropping its current totals.

The On state is similar to Off, except that the actor accepts new temperature readings with an Ok response and updates its counts. Finally, once it shuts down and enters the Done state, the processor ignores all messages it receives.

We will return to this example in the next chapter.

2.4 Type System

CSA’s type system is relatively standard. This section formally defines the type system and explains any non-standard features.

2.4.1 Type-Checking Programs

The top-level judgment is a type-check of complete programs, written \( \vdash \text{prog} P \) and defined in figure 2.3. It requires the externals and initial actors in a program to have distinct names, as well as the externals and receptionists. Every spawn expression in the let-actors clause must type-check as some address type when given types for the externals and previous internal actors. Finally, every address used as a receptionist must also type-check as an address of the type expected for that receptionist.

To type-check expressions, in addition to a standard term environment \( \Gamma \), the type system uses a state environment \( \Theta \) that maps state names to the sequence of types for their arguments (see figure 2.4).

Figure 2.5 defines the type rules for effectful expressions. The rule to type-check a goto expression requires that the argument types match the arguments.
2.4. TYPE SYSTEM

\[ \begin{align*}
\Gamma, \epsilon & \in \text{Var} \rightarrow \text{Type} & & \text{(Term Environments)} \\
\Theta, \epsilon & \in \text{StateName} \rightarrow \text{Type}^* & & \text{(State Environments)} \\
\tau & \in \text{Type} ::= \ldots | \bot & & \text{(Types)}
\end{align*} \]

Figure 2.4: Extra environments and types for type system

\[ \begin{align*}
\Theta(q) = \tau_1, \ldots, \tau_n & \quad \Gamma, \Theta \vdash e_i : \tau_i \text{ for all } i \in 1 \ldots n \\
\Gamma, \Theta \vdash (\text{goto } q \ e_1 \ldots e_n) : \bot
\end{align*} \]

\[ \begin{align*}
Q_i = (\text{define-state } (q_i \ [x_{i,1} \ \tau'_{i,1}] \ldots [x_{i,m} \ \tau'_{i,m}]) \ x'_{i} \ e'_i \ t_c_i) \text{ for all } i \in 1 \ldots n \\
q_1, \ldots, q_n \text{ are distinct} \\
\Gamma' = \Gamma[\text{self} \rightarrow (\text{Addr } \tau)] \\
\Theta' = [q_1 \rightarrow (\tau_{1,1}, \ldots, \tau_{1,m}), \ldots, q_n \rightarrow (\tau_{n,1}, \ldots, \tau_{n,m})] \\
\Gamma', \Theta' \vdash e : \bot \\
\Gamma', \Theta' \vdash \text{state } Q_i \text{ for all } i \in 1 \ldots n \\
\Gamma, \Theta \vdash (\text{spawn } \ell \ e \ Q_1 \ldots Q_n) : (\text{Addr } \tau)
\end{align*} \]

Figure 2.5: Expression type rules for effectful expressions

\[ \begin{align*}
\Gamma, \Theta \vdash e_1 : (\text{Addr } \tau) \\
\Gamma, \Theta \vdash e_2 : \tau \\
\Gamma, \Theta \vdash (\text{send } e_1 \ e_2) : (\text{Variant } [\text{Unit}])
\end{align*} \]

declared for that state in \( \Theta \). Because a \text{goto} expression is purely a control expression that does not evaluate to a value, it gets the special type \( \bot \).

The rule for \text{spawn} requires that all states have distinct names and creates a new state environment \( \Theta' \) in which to type-check the state definitions and initial expression of the spawned actor. The type of the initial expression \( e \) must be \( \bot \), indicating that the expression eventually transitions to one of the actor’s states. Type-checking rules for state definitions and their timeout clauses are found in figure 2.6; similar to the \text{spawn} expression, they require that all handler expressions have type \( \bot \).

Finally, the rule for \text{send} merely checks that the message being sent matches the address’s type.

Figure 2.7 lists the remaining (largely standard) type rules for expressions, including a subsumption rule (see section 2.4.2 for a description of subtyping in CSA). The type for an address is determined by the function \text{ActorType}. For type-checking primitive operations \( (o \ e_1 \ldots e_n) \), the function \text{PrimOpTypes} gives the expected argument types and return type of each operation.

The requirement that the clauses in a \text{case} expressions must be in the same order as the tags in a \text{Variant} type is only for simplification, as the subtype rules allow the tag-clauses of a \text{Variant} type to be rearranged.
2.4.2 Subtyping

Variant types are a convenient means for defining the different messages an actor can accept. To allow an address to be presented to different parts of a program with a restricted set of possible messages, CSA has a width-subtyping rule on Variant types that allows subtypes to drop tag clauses. Address types are contravariant, so an address that accepts more possible messages is a subtype of one that accepts less, allowing the “capabilities” of an address to be narrowed via the type system.

Aside from that rule and a permutation rule for Variant types, the other sub-type rules (defined in figure 2.8) are the standard reflexivity, transitivity, bottom, and depth rules. The rule for recursive types uses the standard Amber rule [23], where \( \Upsilon \) is a set of sub-type assumptions of the form \( X <: X' \). Judgments of the form \( \tau <: \tau' \) are short for \( \emptyset \vdash \tau <: \tau' \).

2.5 Formal Semantics

This section first introduces some extra syntax needed to define the evaluation semantics for CSA, then defines the transition relation on program configurations and related concepts.

2.5.1 Program Configurations

In CSA, the state of a running program is called a configuration. A configuration must keep track of three things:

- the set of running actors and the state each one is in,
- the set of in-transit messages (i.e., those that have been sent but not yet received), and
- the current set of receptionists (i.e., the internal addresses known to the environment and the type at which the environment has access to them).
2.5. **FORMAL SEMANTICS**

![Formal Semantics Diagram](image-url)

Figure 2.7: Other expression type rules
\begin{align*}
Y \vdash \tau_1 <: \tau &\quad Y \vdash \tau <: \tau' \quad Y \vdash \bot <: \tau \\
Y \vdash \text{(Variant } [t_1 \tau_1] \ldots [t_n \tau_n]\text{)} <: \text{(Variant } [t_1 \tau'_1] \ldots [t_n \tau'_{n+k}]\text{)}
\end{align*}

\begin{align*}
\tau_i = \tau_{i,1} \ldots \tau_{i,m} &\quad \text{for all } i \in 1 \ldots n \\
\tau'_i = \tau'_{i,1} \ldots \tau'_{i,m} &\quad \text{for all } i \in 1 \ldots n \\
Y \vdash \tau_{i,j} <: \tau'_{i,j} &\quad \text{for all } i \in 1 \ldots n \text{ and all } j \in 1 \ldots m
\end{align*}

\begin{align*}
(Y \vdash \text{(Variant } [t_1 \tau_1] \ldots [t_n \tau_n]\text{)} \text{ is a permutation of } ([t'_1 \tau'_1] \ldots [t'_n \tau'_{n}])) \quad Y \vdash \text{(Variant } [t_1 \tau_1] \ldots [t_n \tau_n]\text{)} <: \text{(Variant } [t'_1 \tau'_1] \ldots [t'_n \tau'_n]\text{)}
\end{align*}

\begin{align*}
X'' \notin \text{freeTypeVars}(\tau) \cup \text{freeTypeVars}(\tau') \\
Y \cup \{X <: X''\} \vdash \tau <: \tau'[X' \leftarrow X'']
\end{align*}

\begin{align*}
Y \vdash \text{(rec } X \tau) <: \text{(rec } X' \tau')
\end{align*}

\begin{align*}
(X <: X') \in Y &\quad Y \vdash \tau' <: \tau \\
Y \vdash X <: X' &\quad Y \vdash \text{(Addr } \tau) <: \text{(Addr } \tau') \\
Y \vdash \tau <: \tau' &\quad Y \vdash \text{(List } \tau) <: \text{(List } \tau')
\end{align*}

\begin{align*}
Y \vdash \tau <: \tau'' &\quad Y \vdash \tau' <: \tau'' \\
Y \vdash \text{(Dict } \tau \tau') <: \text{(Dict } \tau'' \tau''')
\end{align*}

\begin{align*}
Y \vdash \tau_i <: \tau'_i &\quad \text{for all } i \in 1 \ldots n \\
Y \vdash \text{(Record } [r_1 \tau_1] \ldots [r_n \tau_n]\text{)} <: \text{(Record } [r_1 \tau'_1] \ldots [r_n \tau'_n]\text{)}
\end{align*}

Figure 2.8: Subtyping rules
2.5. FORMAL SEMANTICS

Environment
Knows about:
• $a_m$ with type ManagerUserAPI
• $a_m$ with type ManagerSysAPI
• $a_{p1}$ with type ProcUserAPI
• $a_{p2}$ with type ProcUserAPI

Figure 2.9: Illustration of a configuration of the stream-processing example
CHAPTER 2. CSA: ACTORS AS FINITE-STATE MACHINES

\[ K ::= \langle \beta | \mu \rangle^\rho \quad \text{(Program Configurations)} \]

\[ \beta \in \text{Addr} \rightarrow \text{Beh} \quad \text{(Actor-Behavior Maps)} \]

\[ \mu \in \mathcal{M}(\text{Addr} \times \text{Val}) \quad \text{(Message Multisets)} \]

\[ \rho \in \mathcal{F}(\text{Addr} \times \text{Type}) \quad \text{(Receptionist Sets)} \]

\[ b \in \text{Beh} ::= \langle \overline{Q}, e \rangle \mid \langle \overline{Q}, (\text{receive } x e \text{ tc}) \rangle \quad \text{(Behaviors)} \]

\[ a \in \text{Addr} ::= (\text{addr } \ell n) \quad \text{(Addresses)} \]

\[ v \in \text{Val} ::= a \mid (\text{record } [r v]) \mid (\text{variant } t \overline{v}) \mid (\text{fold } \tau v) \]

\[ \mid (\text{list } \overline{v}) \mid (\text{dict } [v v]) \mid n \mid \text{str} \]

\[ e ::= \ldots \mid a \quad \text{(Expressions)} \]

\[ ov ::= \ldots \mid a \quad \text{(Open Values)} \]

Figure 2.10: Program-configuration syntax

Figure 2.9 illustrates a configuration of the stream-processing example with one manager actor and two processor actors, where there is a Shutdown message on its way to each of those two processor actors. The figure also illustrates the state each actor is in, along with its current state parameters.

The environment in which this program executes knows about the addresses for each of those actors at various types. There are two receptionists for the actor at \( a_m \): one representing the capability to request new processors (ManagerUserAPI), and another representing the capability to shut down the system (ManagerSysAPI). The processors are each known to the environment at the ProcUserAPI type.

Figure 2.10 defines the formal syntax for such a configuration configuration. A program configuration \( K \) (represented by the metavariable \( K \)) consists of an actor-behavior map \( \beta \), a message multiset \( \mu \), and a receptionist set \( \rho \).

The set of actors in a configuration (corresponding to the actors on the left-hand side of figure 2.9) is represented as a map \( \beta \) from an actor's address to its behavior. An actor's behavior \( b \) contains its state definitions \( \overline{Q} \) (not shown in figure 2.9) and either its currently executing handler expression \( e \) or a special receive term that indicates the actor is suspended and waiting for the next event. The receive term specifies the message handler \( e \) and timeout clause \( tc \) to be used for the next event, with \( x \) binding the received message in \( e \). The states of the actors in figure 2.9 correspond to this latter form of a behavior.

A message is a pair of a destination address \( a \) and a communicated value \( v \). The message multiset \( \mu \), corresponding to the messages on the right-hand side of figure 2.9, contains all sent messages not yet handled by the receiving actor.

Finally, the receptionist set \( \rho \) records the addresses the program has provided.

\[ ^8 \text{Similar permutation and width rules for Record types are possible, but unnecessary for our purposes.} \]
to the environment and at what types the environment has access to them. In figure 2.9, this corresponds to the set of addresses that the environment knows about and the types at which it knows them. The type of a receptionist may be different from the base type of the identified actor (as is the case for all receptionists in figure 2.9) so that an actor’s public interface can differ from its private implementation. This set of receptionists initially matches the declarations from the program, but adds more elements as the program sends more addresses to the environment.

The environment is assumed to be made up of actors from other CSA programs, and therefore those actors are assumed to send only messages with appropriate types. A more robust implementation of CSA may wish to first type-check messages sent to receptionists and reject any with the wrong type.

An address \( a \) is defined by the syntactic location \( \ell \) of the spawn expression that created it, along with a unique identifier \( n \) to distinguish it from other actors created at that location. The rest of this dissertation will sometimes refer to an actor by its address, as in, “the actor at \( a \) has behavior \( b \)”.

The locations are partitioned into two sets: \( \text{ProgLoc} \) is the set of all syntactic locations from the original program, and \( \text{EnvLoc} \) is the set of all locations for actors from the environment. An address \( (\text{addr } \ell n) \) is internal if \( \ell \in \text{ProgLoc} \) and external if \( \ell \in \text{EnvLoc} \).

It is assumed there is a function \( \text{ActorType} \) from locations to types such that \( \text{ActorType}(\ell) \) is the type of message that an actor spawned at \( \ell \) can receive. Furthermore, it is assumed that for every possible type \( \tau \) there are infinitely many external locations \( \ell \) such that \( \text{ActorType}(\ell) = \tau \). The function extends to addresses based on the spawn location for that address; i.e., if \( a = (\text{addr } \ell n) \), then \( \text{ActorType}(a) = \text{ActorType}(\ell) \).

### 2.5.2 Type-Checking Configurations and Addresses

The soundness of some of the proof techniques introduced in this dissertation rely on a type preservation lemma similar to the following (the actual lemma is phrased in terms of marked configurations, defined in chapter 3):

**Lemma.** For all \( K, K', I, \) and \( l \), if \( \vdash_{\text{cfg}} K \) and \( K \xrightarrow{I} K' \), then \( \vdash_{\text{cfg}} K' \).

As a result, a type-checking judgment \( \vdash_{\text{cfg}} \) for configurations is required. The rule in figure 2.11 defines the judgment. The rule checks that

- every receptionist corresponds to an actor in the program,
- every receptionist’s type is a subtype of the messages that address can receive,
- the address for every actor in the program is internal,
- the actor-behavior map has a mapping for every internal address in the configuration,
∀ (a, τ) ∈ ρ. a ∈ dom(β) and φ, φ ⊢ a : (Addr τ) ∀ a ∈ dom(β). a is internal
∀ a appearing in β, μ, or ρ. a ∈ dom(β) if a is internal
∀ a ∈ dom(β). Ξ. ActorType(a) = τ and τ, φ ⊢ beh b where b = β(a)
∀ ⟨a', v⟩ ∈ μ. Ξr. φ, φ ⊢ a' : (Addr τ) and φ, φ ⊢ v : τ

\[ \text{\textbf{Figure 2.11: Program configuration and behavior type rules}} \]

\[ \text{\textbf{Figure 2.12: Address typing rule}} \]

- every actor’s behavior type-checks, and
- the type of every in-transit message matches its destination address’s type.

The type-checking rules for behaviors are similar to the rule for spawn expressions, ensuring that both the state definitions and any handler expressions and timeout clauses type-check. The main difference is that instances of the self keyword in a behavior have already been replaced with the actor’s address, so the rule does not add self to the type environment. This judgment also takes as input an extra type τ, indicating the type of messages the actor can receive.

Because configurations may contain addresses, addresses must be able to be type-checked, as well. The rule, for an address a, given in figure 2.12, simply gives the type for that address reported by the ActorType function.

### 2.5.3 Instantiation

The function Inst in figure 2.13 converts a program into its initial configuration. It takes a sequence of addresses to assign as the declared externals and creates
2.5. FORMAL SEMANTICS

\[ \text{Inst} : \text{Prog} \times \text{Addr}^* \rightarrow \text{ProgConfig} \]
\[ \text{Inst}(P, a_1 \ldots a_m) = \left\langle \left\langle a_i' \rightarrow b_1, \ldots, a_i'' \rightarrow b_o \middle\vert \emptyset \right\rangle \right\rangle^{(a_1, \tau_1) \ldots (a_n, \tau_n)} \]

where \( P = \) (program (receptionists \( [x_1 \tau_1]^{i \in 1 \ldots n} \)) (externals \( [x'_j \tau'_j]^{j \in 1 \ldots o} \))
\[ \text{let-actors} \begin{cases} \ell_k: \text{InstAct}(\text{AddrSubst}) \rightarrow \text{Beh} \\ \text{InstAct}(\text{AddrSubst}) \end{cases} \]

The initial behavior for each actor, substituting in the actor’s own address, the external addresses, and all previously declared internal actors.

The \( \text{InstAct} \) function in figure 2.14 instantiates an actor by applying the given substitutions to its state definitions and \text{goto} arguments, then transitioning it into its initial state where it waits for the next message or timeout.

2.5.4 Transition Semantics

The transition relation for CSA is a labeled transition relation of the form \( K \xrightarrow{l} K' \), defined in figure 2.16. The transition-step label \( l \) (defined in figure 2.15)

\[ E ::= [[ \mid \text{goto} q \overline{v} E \overline{v} ] \mid \text{send} v E ] \mid \text{send} v E \]  
\[ \mid \text{begin} E \overline{v} ] \mid \text{record} [r v] [r E] [r e] \]  
\[ \mid \text{case} E \begin{cases} (t \overline{v}) \mid \text{fold} t E \mid \text{unfold} t E \end{cases} \]
\[ \mid \text{for/fold} [x E] [x e] \mid \text{for/fold} [x v] [x E] e \mid (o \overline{v} E \overline{v}) \]
\[ l ::= a: \text{rcv-ext}(v, r) \mid a: \text{rcv-int}(v) \]
\[ \mid a: \text{send-ext}(a, v) \mid a: \text{send-int}(a, v) \mid a: \text{timeout} \mid a: \text{func} \mid a: \text{goto} \]
\[ a: \text{spawn}(a) \]

Figure 2.15: Miscellaneous evaluation syntax
distinguishes steps for the sake of a fairness condition for CSA (see section 2.5.6). The label also indicates any communication with the environment (send or rcv), and whether that communication was internal to the program (int) or external (ext). The type in the rcv-ext label indicates the type of the receptionist that allows the message to be received. The address in a spawn label identifies the spawned actor. The address used as the prefix a of a transition step's label indicates the active actor for that step. A func label indicates a purely functional step of a message handler, such as extracting a field from a record or evaluating a primitive operation.

The handler-start labels are those labels of the form a : timeout, a : rcv-int(v), or a : rcv-ext(v, τ), because they each indicate the start of a new event handler. All others are handler-continuation labels, which represent a step taken while executing an event handler. Because handler expressions are deterministic (modulo choice of address for newly spawned actors), at most one step with a handler-continuation label is enabled in a given configuration for each actor.

Figure 2.16 lists the transition rules of CSA's operational semantics. The rule E-GOTO transitions an actor into its next state (represented by a receive term), in which it waits to receive a message. The message variable, handler expression, and timeout clause of the receive term come from the named state. This step ends evaluation of the handler expression by dropping the remaining context E.

E-RECEIVE-INTERNAL picks an arbitrary message sent to the actor at a and substitutes it into the handler expression.

E-RECEIVE-EXTERNAL is similar to E-RECEIVE-INTERNAL, with the message coming from the environment. If ρ contains a receptionist ⟨a, τ⟩, then the environment can send any message of type τ to a.

The rule requires that any received internal addresses are used in positions that respect their type as given in ρ. The function IntAddrTypes, defined in figure 2.17, returns a set \{⟨a_1', τ_1'⟩, ..., ⟨a_n', τ_n'⟩\} indicating the types of all internal addresses in the message when v is viewed as the receptionist's type τ. Subtyping is defined in section 2.4.2.

E-TIMEOUT allows an actor to run its state's timeout handler. The semantics does not model a clock for timeouts.

E-SEND-INTERNAL pairs the message v with its destination a' and adds it to the set of messages. The expression itself evaluates to a Unit value.

E-SEND-EXTERNAL outputs to the environment a message directed to one of the external addresses. The sent message may contain addresses the environment does not know about yet, or present previously known addresses with different types (because of the subtyping rules, see section 2.4.2), necessitating an update to the receptionists set ρ to reflect the newly exposed receptionists. For instance, ρ may have an entry ⟨a, (Variant [GetItem Nat])⟩ (meaning the actor at a can accept messages of the form (variant GetItem n)), but the program may send a message that uses a in a context where it has type (Addr (Variant [SetName String])) (indicating the actor at a can also accept messages of the form (variant SetName str)). Therefore the environment learns that a can receive an additional kind of message, so the program adds an entry ⟨a, (Variant SetName String)⟩ to ρ.
2.5. FORMAL SEMANTICS

**E-GOTO**
\[
\langle \beta [a \rightarrow (Q, E[\text{goto } q \rightarrow v])] \rangle \mu \xrightarrow{a: \text{goto}} \langle \beta [a \rightarrow (Q, (\text{receive } x e tc) \mid x \rightarrow v)] \rangle \mu
\]
if (\text{define-state} (q \mid x : r) \ast e tc) is in \(Q\)

**E-RECEIVE**
\[
\langle \beta [a \rightarrow (Q, (\text{receive } x e tc))] \rangle \mu \xrightarrow{a: \text{rcv-int(v)}} \langle \beta [a \rightarrow (Q, e[x \rightarrow v])] \rangle \mu
\]

**E-RECEIVEINT**
\[
\langle \beta [a \rightarrow (Q, (\text{receive } x e tc))] \rangle \mu \xrightarrow{a: \text{rcv-ext(r)}} \langle \beta [a \rightarrow (Q, e[x \rightarrow v])] \rangle \mu
\]
if \(\langle a, r \rangle \in \rho\) and \(\emptyset, \emptyset : r \vdash v : r\)
and \(\forall \langle a', r' \rangle \in \text{IntAddrTypes}(v, r). \exists r''. \langle a', r'' \rangle \in \rho\) and \(r'' < r'\)

**E-TIMEOUT**
\[
\langle \beta [a \rightarrow (Q, (\text{receive } x e' [\text{timeout } n e])] \rangle \mu \xrightarrow{a: \text{timeout}} \langle \beta [a \rightarrow (Q, e)] \rangle \mu
\]

**E-SENDINTERNAL**
\[
\langle \beta [a \rightarrow (Q, E[\text{send } a' v])] \rangle \mu \xrightarrow{a: \text{send-int}(a', v)} \langle \beta [a \rightarrow (Q, E[\text{variant } \text{Unit}])] \rangle \mu \cup \langle a', v \rangle
\]
if \(a'\) is internal

**E-SENDEXTERNAL**
\[
\langle \beta [a \rightarrow (Q, E[\text{send } a' v])] \rangle \mu \xrightarrow{a: \text{send-ext}(a', v)} \langle \beta [a \rightarrow (Q, E[\text{variant } \text{Unit}])] \rangle \mu \cap \langle v \cup \langle a', v \rangle \rangle
\]
if \(a'\) is external, \(\text{ActorType}(a') = \tau\), and \(\text{IntAddrTypes}(v, \tau) = \rho'\)

**E-SPAWN**
\[
\langle \beta [a \rightarrow (Q, E[\text{spawn} \tau e Q])] \rangle \mu \xrightarrow{a: \text{spawn}(a')} \langle \beta [a \rightarrow (Q, E[a'], a' \rightarrow b')] \rangle \mu
\]
such that \(a' = (\text{addr } \ell n)\) and \(a' \notin \{a\} \cup \text{dom}(\beta)\)
and \(b' = (Q[\text{self} \rightarrow a'], e[\text{self} \rightarrow a'])\)

**E-FUNC**
\[
\langle \beta [a \rightarrow (Q, E[e])] \rangle \mu \xrightarrow{a: \text{func}} \langle \beta [a \rightarrow (Q, E[e'])] \rangle \mu
\]
if \(e \rightarrow e'\)

Figure 2.16: Program configuration transition rules
\[ \text{IntAddrTypes}(v, \tau) = \begin{cases} \emptyset & \text{if } a \text{ is external, else } \{(a, \tau')\} \\
\emptyset & \text{Case } v = n, \tau = \text{Nat} \\
\emptyset & \text{Case } v = str, \tau = \text{String} \\
\bigcup_{i \in 1...n} \text{IntAddrTypes}(v_i, \tau_i) & \text{Case } v = (\text{variant } t v_1 ... v_n), \tau = (\text{Variant } [t' \tau'_1 ... \tau'_m] [t \tau_1 ... \tau_n] [t'' \tau''_1 ... \tau''_l]) \\
\bigcup_{i \in 1...n} \text{IntAddrTypes}(v_i, \tau_i) & \text{Case } v = (\text{record } [r_1 v_1] ... [r_n v_n]), \tau = (\text{Record } [r_1 \tau_1] ... [r_n \tau_n]) \\
\bigcup_{i \in 1...n} \text{IntAddrTypes}(v_i, \tau_i) & \text{Case } v = (\text{fold } \tau v'), \tau = (\text{rec } X \tau') \\
\bigcup_{i \in 1...n} \text{IntAddrTypes}(v_i, \tau_i) & \text{Case } v = (\text{list } v_1 ... v_n), \tau = (\text{List } \tau') \\
\bigcup_{i \in 1...n} \text{IntAddrTypes}(v_i, \tau') & \text{Case } v = (\text{dict } [v_1 v'_1] ... [v_n v'_n]), \tau = (\text{Dict } \tau' \tau'') \\
\bigcup_{i \in 1...n} (\text{IntAddrTypes}(v_i, \tau') \cup \text{IntAddrTypes}(v_i', \tau'')) & \text{Otherwise, undefined} \\
\end{cases} \]

Figure 2.17: Internal address type extraction
2.5. FORMAL SEMANTICS

\[
\begin{align*}
\text{(begin } v \text{)} & \mapsto v \\
\text{(begin } v \text{ e } e' \text{)} & \mapsto (\text{begin } e \text{ e'} ) \\
(\text{\texttt{begin}}) & \mapsto \text{v} \\
(\text{\texttt{begin}} & \text{ e } e' \text{)} \mapsto (\text{\texttt{begin}} e \text{ e'}) \\
(\text{\texttt{case}} & \text{ (variant } t \text{ v}) \text{ _ (} (t \text{ x}) \text{ e) } \_ \text{)} \mapsto e[x \leftarrow v] \\
(\text{\texttt{unfold}} & \text{ (fold } \tau' \text{ v}) \text{) } \mapsto v \\
(\text{\texttt{o v}) & \mapsto \text{EvalPrimop}(o,v) \\
(\text{\texttt{for/fold}} & \text{ [x v] [x' (list } \nu' \text{)] e) } \mapsto (\text{\texttt{for/fold}} [x e'] [x' (list } \nu' \text{)] e) \\
\text{where } e' & = e[ x \leftarrow v] [ x' \leftarrow \nu']
\end{align*}
\]

Figure 2.18: Functional reduction steps

E-Spawn creates an actor with a globally unique address.\textsuperscript{9} The initialization expression becomes the actor’s current handler expression. The result of the \texttt{spawn} is the new address \texttt{a’}, which is substituted for \texttt{self} in the new actor.

E-F Unc allows the actor to step if its current handler expression can take a functional reduction step, defined in figure 2.18. The functional rules are standard. Exceptional behavior such as dividing by zero causes the evaluating actor to get stuck.

CSA does not model crashed actors or dropped messages, which sometimes occurs in distributed systems. Reasoning about systems without these properties is still useful, though, because an error in such a failure-proof system would also be an error in a failure-susceptible one.

2.5.5 Related Semantic Notions

The transition semantics gives rise to a variety of related notions defined here and used later in this dissertation. A transition step labeled with \texttt{l} is \textit{enabled} from a configuration \texttt{K}, written \texttt{K \texttt{\rightarrow l}}, if there exists some \texttt{K’} such that \texttt{K \rightarrow l K’}.

Intuitively, actors alternate between waiting for messages and handling some event. To formalize this, we say that a behavior \texttt{b} is \textit{handling an event} if \texttt{b} = \texttt{⟨Q,e⟩} for some \texttt{Q} and \texttt{e}; otherwise, we say the behavior is \textit{awaiting an event}. We also use the same terms for actors to say that their associated behaviors have these properties.

A program configuration \texttt{K} = \texttt{⟨⟨ś⟩,μ⟩} is \textit{stuck} if it contains some actor with address \texttt{a} that is handling an event but there is no step labeled with \texttt{l} in which \texttt{a} is the active actor’s address (meaning that particular actor is stuck).

\textsuperscript{9}It is assumed that the syntactic \texttt{spawn} locations in a program are distinct from the \texttt{spawn} locations of actors in the program’s environment, so that the address of any newly spawned actor is distinct from all external addresses that might be learned over time. An actual implementation of CSA can use various schemes to ensure global uniqueness, such as using the MAC address of the current machine and the PID of the CSA process as part of the actor’s address.
An execution of a program configuration $K$ is a (possibly infinite) sequence of transition steps $K_1 \xrightarrow{l_1} K_2 \xrightarrow{l_2} \ldots$ with $K_1 = K$. An execution is stuck if it contains a stuck configuration. The length of an execution $\text{len}(K_1 \xrightarrow{l_1} K_2 \xrightarrow{l_2} \ldots)$ is the number of transition steps, where $\omega$ is the length of an infinite execution.\(^{10}\)

### 2.5.6 Fairness

As defined so far, CSA’s formal semantics allows many executions that would not occur in a real implementation of the language. For instance, consider a configuration with two actors, $X$ and $Y$, such that whenever either one receives a message (variant Ping $a$), it replies to $a$ with (variant Pong). A natural correctness property would be that whenever $X$ receives a Ping, it sends back a Pong to the included address. However, the execution in which $X$ receives a message but then $Y$ takes every step from there on would violate this property, because $X$ never gets a chance to send its response.

Such an execution would likely never occur in a real actor system, because such systems are designed to allow every actor to eventually run if it has some work to do. We also expect a similar guarantee that a message sent to an internal actor will eventually be received (i.e., no message sits in the mailbox forever).

CSA comes with a fairness constraint on executions that encodes these conditions. An execution $K_1 \xrightarrow{l_1} K_2 \xrightarrow{l_2} \ldots$ is fair when the following conditions hold:

1. For all configurations $K_i$ in the execution, if $l$ is a non-external-receive step (i.e., $l \neq a : \text{rcv-ext}(v, r)$) enabled on $K_i$ with active actor $a$, there exists some $j \geq i$ such that $K_j \xrightarrow{l_j} K_{j+1}$ is a step in the execution such that $a$ is the active actor for $l_j$.

2. For all configurations $K_i$ in the execution, and for all enabled transition steps $a : \text{rcv-int}(v)$ from $K_i$, there exists some $j \geq i$ such that $K_j \xrightarrow{a : \text{rcv-int}(v)} K_{j+1}$ is a step in the execution.

We might expect the first condition to require that the enabled step labeled with $l$ is eventually taken. However, if the step is a timeout, it would be fair for the actor to receive a message from the environment and transition to some other state rather than execute its timeout. Therefore fairness requires only that the actor take some step in the remainder of the execution.

### 2.6 Related Work

The Actor Model  
Carl Hewitt et al. [59] originally developed the actor model as a method for structuring artificial intelligence programs. Many researchers later studied the model, refined it, and proposed various semantics for it. The

\(^{10}\) The length of an execution is assumed to be defined similarly for all notions of “executions” defined in this dissertation.
2.6. RELATED WORK

presentation here is based on the semantics given by Agha et al. [3]. CSA differs from that semantics mainly by

- structuring actor behaviors as finite-state machines,
- syntactically restricting all event handlers to be terminating, and
- adding a simple type system.

There are many languages and frameworks based on the actor model [17, 54, 89, 107], with Erlang [10] and Scala's Akka framework [5] being the most well-known. Programming an actor as a finite-state machine is a common pattern in these systems; CSA promotes this idea to a full-fledged language feature. Other aspects of these actor systems, such as the ability to defer handling some messages until a later time, are omitted from CSA. Incorporating those features into the current framework is an area for future work.

Akka also has a typed variant, Akka Typed [7], with actor types similar to the Addr type in CSA. I am unaware of any formal semantic models of a typed actor system, however.

Rebeca [100] is another actor-based language that, like CSA, syntactically forbids loops and recursion in event handlers so that all handlers must terminate.

**FSMs for Communicating Systems** Programmers have used finite-state machines to describe protocols since the early days of computer networks. In 1969, the year the ARPANET transmitted its first message, Bartlett et al. [15] described their Alternating Bit Protocol using FSMs. Brand and Zafiropulo [19] later refined and formalized this idea into communicating finite-state machines that describe and analyze interactions between components. CSA follows in this tradition but extends the finite-state machines by parameterizing each state over arbitrary value arguments. CSA also inherits a dynamic communication topology from the actor model, whereas these early works assumed a static topology.

Aside from Cisco’s research language already mentioned, many programming languages make state machines a central feature of the language (e.g., Esterel [18], Executable UML [78]). In the context of the actor model, Actor-eUML [77] adapts the hierarchical state machines of Executable UML to implement actors, thereby enabling a model-driven architecture approach to programming with actors. The P [40] and P# [37] languages from Microsoft Research also employ message-passing state machines to implement concurrent programs such as device drivers and distributed protocols. CSA is a pure programming model for programming actors in this vein.

**Process Calculi** Process calculi such as CCS [80], CSP [60], and the $\pi$-calculus [81] are often used as an alternative to actors for modeling communicating systems. CCS and CSP are limited to static communication topologies, because they do not allow messages to contain communication channels. The
π-calculus, on the other hand, was developed specifically to model dynamic communication topologies. The main difference from the actor model is that the π-calculus distinguishes communication channels from processes, so that multiple processes can both send and receive messages on the same channel. As a result, the kinds of protocols programmers implement in the π-calculus tend to differ from those implemented in the actor model.

**Session Types** Session types [62, 63] are a typing discipline used to ensure that the communication of some process (such as one written in the π-calculus) follows some expected pattern. A CSA actor takes the form of a finite-state machine to give structure to an actor’s behavior, but does not otherwise restrict its communication. Compared to session types, the Addr types in CSA are basic types that describe only the kinds of messages that may be sent to an address (e.g., an actor may not send a string to another actor that accepts only integers). The related work section in chapter 3 compares session types with the specification framework developed in that chapter.
Chapter 3

APS: Actor Protocols as Finite-State Machines

To describe how a component can be used as part of a larger system, programmers use specifications (both formal and informal) that describe its expected behavior. In a functional setting, a specification can be understood as simply associating each input with its expected output. Specifications for communicating systems, however, are more complex because they must describe the valid sequences of messages sent and received over time. For instance, a specification for the stream-processing system of the previous chapter might include requirements such as

- every GetMean request results in a response to the mdest address from the processor's creation, and furthermore, those are the only messages sent to that address,
- the processor actor responds to temperature readings with Ok only when it is enabled, and
- no responses are sent after a processor receives a Shutdown message.

The common theme with these requirements is to capture cause-and-effect relationships between the messages that the environment sends to the program and those that the program sends back to the environment.

To accommodate this computational model, programmers often specify a program’s communication patterns (i.e., its protocols) with finite-state machines. Such specifications guide the programmer during development and constitute the basis for formal methods such as test-case generation and automatic verification. Finite-state machines are a time-tested method for distinguishing the expected behavior in each state without cluttering the specification with extraneous details. For example, almost everyone who has studied network protocols is familiar with the TCP state machine [90], and SDL [102] and UML [95] are two popular modeling frameworks that specify protocols with FSMs.
The standard FSM approach assumes a static topology, however, so the dynamic topology inherent to the actor model poses significant problems. First, a specification may need to describe how the program uses addresses provided by the environment, as with the stream-processor actor that sends Ok/NotOk responses to the address included in an AddRdg message. Second, when the program sends messages that themselves carry addresses in the message, a specification may need to describe how the program responds to messages received at those addresses. For instance, in the stream-processing example, the address that the manager sends in response to a MakeProc message refers to an actor that can process temperature readings and respond to requests for the mean. Intuitively, this situation has similarities to contextual equivalence in higher-order languages, where equivalence depends not only on the shape of a result, but also on the behavior of embedded functions.

As one of the major contributions of this dissertation, this chapter defines APS (Actor Protocol Specifications), a specification language for CSA that adapts the familiar finite-state-machine method to the dynamic topology of the actor model. Each specification approximates a program’s expected behavior in terms of the messages it sends to and receives from the environment. The language is designed to express non-trivial communication patterns of CSA programs, yet be amenable to automated verification (as we will see in chapter 7).

The following section introduces the language and explains how it deals with dynamic topologies. Some toy examples in that section help make the new concepts concrete. Section 3.2 gives a larger example specification for the stream-processing example, and section 3.3 summarizes the expressiveness of APS. Section 3.4 develops an annotated version of the CSA semantics needed to define conformance to an APS specification, and sections 3.6 and 3.7 defines the formal semantics for APS itself. The next chapter illustrates the typical conformance proof structure with an example proof.

3.1 Syntax and Intuitive Semantics

3.1.1 Overview: Protocol Specification Machines

A specification in APS consists of a dynamically growing collection of FSM-like structures called protocol specification machines, or PSMs. A PSM is a communicating process that articulates a program’s expected communication with its environment. A PSM over-approximates the specified program’s behavior: that is, a PSM might have more possible transition steps than a given program, but every step of a conforming program must be a valid step of its specifying PSM. Because a PSM is itself a process, this dissertation sometimes describes a PSM in terms of sending or receiving messages, although one should interpret such phrases to mean that the PSM expects the program it specifies to send or receive such a message.

---

1This is based on the specification language presented in a previous paper [97].
Approximating all of a program's behavior with a single FSM is infeasible, however, because a program may take part in many conversations over unboundedly many addresses. Instead, a PSM describes a program's expected behavior only in terms of messages the program receives from the environment at a single receptionist address, called the PSM's input-monitored address, and messages the program sends back to a subset of the external addresses, called the PSM's output-monitored addresses. Thus, any given PSM focuses on specifying a single protocol carried out over only a handful of the addresses with which the specified program and its environment interact.

Different PSMs can describe different aspects of a program's expected behavior, depending on the addresses each PSM monitors. For example, consider an order-processing program that has access to two external addresses: an address $W$ to send orders to a warehouse, and an address $L$ to send log messages. To specify the communication with the warehouse while leaving the logging behavior unspecified, one could write a PSM with $W$ as its only output-monitored address. Similarly, one could specify just the logging aspects with a PSM with $L$ as its lone output-monitored address, or specify both aspects at once with a PSM that has both addresses in its output-monitored set.

It is important to note that a PSM is not an abstract actor. Although a PSM specifies the expected responses from sending messages to a particular actor, a program might implement that behavior with a collection of actors working together in concert, with the actor at the PSM's input-monitored address merely acting as an external facade for that collection. Thus, rather than modeling an individual actor, a PSM models an entire program in terms of that program's communication with its environment.

The main benefit of limiting each PSM to a single input-monitored address is that it enables specifications to associate expected behavior with addresses sent back to the environment. For example, a response to a `MakeProc` request from the stream-processing example is expected to be not just any address, but an address referring to an actor that follows the processor-actor protocol. To express such a specification, a PSM can associate the address in an expected `MakeProc` response with a new PSM, in a technique called forking. The sent address would act as the forked PSM's input-monitored address, and the PSM would also have its own separate set of output-monitored addresses. Thus, a forked PSM enforces that a conforming program satisfy some additional protocol with respect to the messages received at and sent to those addresses.

A specification configuration, then, is a collection of PSMs that merges together the restrictions that each PSM places on the specified program, forming a single unified model of the program's expected behavior. PSMs forked during the transition are added to the configuration for subsequent steps. Intuitively, a program conforms to its specification if

1. for every possible step the program can take, the specification configuration can take a step that relates to it, and
2. the program eventually performs all actions the specification requires it to perform (see the description of obligations in section 3.1.2).
Just as PSMs do not model individual actors, a specification configuration is not a collection of entities interacting with each other. Indeed, PSMs never interact with each other once they are created. Rather, it is best to view the specification configuration logically as a conjunction of specifications: it says that the specified program must behave as specified by the first PSM, and as by the second PSM, and as by the third PSM, and so on.

A Brief Note on the Specification Paradigm

APS places no requirements on the behavior of the environment other than to respect the type system. In a distributed system, one often does not control all of the interacting parties. Therefore, it is important to ensure that a program running in such a system is robust enough to handle both expected and unexpected communication.

By contrast, a framework that specifies the environment’s behavior in addition to that of the program could use APS as a foundation. For example, by associating a PSM to an address received from the environment, one might be able to describe the environment’s reaction to messages the program sends to that address. This is left to future work, however.
3.1. SYNTAX AND INTUITIVE SEMANTICS

3.1.2 Specification Syntax

Figure 3.1 lists the syntax for APS specifications. Instantiating a specification by providing it with the initial receptionists and externals of a program that should be checked for conformance (see section 3.6.2) yields a configuration with just a single PSM, although the configuration may create more PSMs as it evolves.

The **mon-receptionist** clause in the specification declares some named receptionist on a specified program as the PSM's initial input-monitored address. Of all the messages a specified program may receive from its environment, the PSM specifies the program's reaction only to those messages sent to that input-monitored address. The expected reactions to messages sent to other addresses are left unspecified.

Similarly, the **mon-externals** clause defines addresses that are output-monitored in the PSM. The only outgoing messages the PSM specifies are those the program sends to output-monitored addresses.

A PSM may start without an input-monitored address (indicated by the keyword **no-mon-receptionist**). This represents a kind of anonymous PSM that specifies the messages sent to the output-monitored addresses, regardless of the messages received from the environment. The discussion of effects below explains how such a PSM might gain an input-monitored address over time through use of the **self-addr** output pattern.

The remainder of the specification defines a kind of finite-state machine that governs the PSM's behavior. The state machine's structure is inspired by the state diagrams often used to describe network protocols: a state transition is triggered by receiving a matching message at the input-monitored address, and the effects of a transition include messages to send to output-monitored addresses.

The specification's **goto** statement names the PSM's initial state. Like CSA, states in APS are parameterized; unlike CSA, however, those parameters may only be external addresses. Thus, APS can name different addresses to which a specified program may send messages (thereby accounting for the dynamic topology of actor programs), while it abstracts over computational details that are irrelevant to the high-level protocol. A PSM's state parameters are always output-monitored addresses, so initially they must come from the **mon-externals** list.

The final components of the specification, $\Phi$, define the PSM’s states. Every state definition $\Phi$ provides the state’s name $\varphi$, formal parameters $\vec{x}$, and a set of transitions $\delta$, described below.

**State Transitions**

The transitions of an APS state machine are more restricted than those of CSA actors, consisting of three components: a **firing condition** $c$, some effects $\vec{f}$, and a **goto** expression naming the next state. The firing condition indicates when a transition can happen: the condition $pi$ requires that the PSM receive a message matching the input pattern $pi$ on its input-monitored address, while a **free** transition can occur at any time. The **free** transitions account for events other than
messages received at the input-monitored address that may affect the expected behavior, such as a timeout, the reception of an internal message, or the reception of a message at some other address. When a transition fires, it performs each of its effects (see below), and the PSM transitions to the declared state. A PSM can have multiple transitions in the same state with the same firing condition, making the semantics non-deterministic.

In the input patterns, * is a wildcard that matches any value. Any received addresses matched under * are left unmonitored, because the wildcard signals that the uses of any such received address are irrelevant to this specification.

If a received address is relevant, a specification matches it with an input pattern variable \( x \). That pattern binds the address in the scope of the transition's effects and goto statement in order to specify messages sent to it, and the address becomes output-monitored so that the specification describes its legal uses. Such name-binding addresses the first problem mentioned with dynamic communication topologies: how to specify the uses of addresses received from the environment.

### Effects

Transitions have two kinds of effects. The first is an **obligation**, which is a requirement that the specified program must send to the named address \( x \) a message matching the output pattern \( po \) (although not in any particular order or amount of time, to account for the asynchrony inherent in the actor model). Every message sent to an output-monitored address must fulfill some existing obligation. Thus, APS specifications constrain outputs both from above (no message may be sent without an obligation) and below (every obligation must eventually be fulfilled).

One of the major ideas of APS is that output patterns describe not just structural properties of communicated values, but also behavioral properties of communicated addresses. For instance, a **fork-addr** or **delayed-fork-addr** pattern matches an address \( a \) and creates a new PSM specifying how future messages sent to \( a \) should be handled. (Section 3.1.4 describes the difference between fork-addr and delayed-fork-addr.) A conforming program's behavior must match the protocols of both the new PSM (with input-monitored address \( a \)) and the original PSM. A **self-addr** pattern also matches an address \( a \), but then instead sets \( a \) as the current PSM's input-monitored address if the PSM does not yet have one: messages received at that address will trigger state transitions for this PSM.

The second kind of effect is a **fork**. Like the fork-addr and delayed-fork-addr patterns, fork creates a new PSM specifying a new protocol that a conforming program must implement. Unlike those patterns, however, a PSM created by a fork effect has no input-monitored address. As with an initial PSM with no-mon-receptionist, this represents a kind of anonymous PSM.

Whether created from a fork effect or a fork-addr or delayed-fork-addr pattern, a forked PSM's goto statements must only refer to the states defined in its own definitions \( \Phi \); they may not refer to the states of the parent PSM. Also, the addresses the forked PSM uses (e.g., as state parameters) must be disjoint.
from those used by the current PSM or other forked PSMs created in this transition. Because a PSM monitors all of the addresses it uses, and because a PSM must account for all messages sent to addresses it monitors, this latter restriction helps to eliminate conflicting uses of an address by multiple PSMs. Some logic introduced in section 3.6.4 formalizes this condition.

### 3.1.3 Simple Examples

#### Simple Ping/Pong

A few small examples may help make these ideas concrete. We start with a simple ping/pong server. Whenever the server receives a Ping, it sends a Pong to some well-known address. The interface that program exposes to the environment might look something like the following:

```plaintext
(program (receptionists [s (Variant [Ping]]))
  (externals [dest (Variant [Pong])])
  ;; program body elided
)
```

The program has a receptionist s to which the environment can send Ping messages, and the environment is expected to initially provide some destination dest for the Pong responses.

A specification for such a program might look like the following:

```plaintext
(specification
  (mon-receptionist s)
  (mon-externals dest)
  (goto Running dest)
  (define-state (Running d)
    [(variant Ping) ->
      (obligation d (variant Pong))
      (goto Running d)])
)
```

In this case, all receptionists and externals are monitored (according to the mon-receptionist and mon-externals clauses), so this specification describes all messages between the program and its environment.

The initial goto says the PSM starts in its only state, Running, with the external address dest as its parameter d. Within that state, the lone transition says that the PSM can react to receiving a Ping by requiring the server to (eventually) send a Pong to d and returning to its current state. APS requires that all messages sent to output-monitored addresses be accounted for in the specification, so this specification does not allow a Pong to be sent without a preceding Ping.

It may seem odd to have a state parameter d for the Pong destination even though that destination never changes. The only variables in the scope of a state transition, however, are the state parameters and any pattern-bound variables, so the dest address is out of scope inside the Running state. This scoping rule
simplifies a portion of the formal semantics that determines which addresses each PSM uses (see section 3.6).

### Ping/Pong with Multiple Externals

The previous example monitored all of the program’s communication with its environment, but sometimes a specification writer will want to describe only part of the program’s communication, because it may be involved in many disparate conversations. For example, in addition to sending a Pong, the ping/pong server might also send the number of Pings received so far to a metrics-recording system elsewhere in the environment. The program would then have an additional initial external [metrics Nat]. One option would be to specify the outputs to both external addresses, as in the following specification:

```
(specification
 (mon-receptionist s)
 (mon-externals dest metrics)
 (goto Running dest metrics)
 (define-state (Running d m)
   [(variant Ping) ->
    (obligation d (variant Pong))
    (obligation m *)
    (goto Running d m)]))
```

As in the previous example, this specification monitors all of the available receptionists, so this PSM specifies all outputs to the environment. In addition to the obligation from the previous specification, receiving a Ping results in an obligation to send a message to the address m of the metrics system.

Perhaps the metrics are unrelated to the server’s main purpose, though, and the specification writer wishes to focus only on the Pong responses and leave the metrics unspecified. In that case, the specification from the previous example could be used instead. That specification does not include metrics as an initial output-monitored address, so the specification says that the program can send any messages to that address at any time, regardless of the messages the environment sends to the server.
Ping/Pong with Multiple Receptionists

Although leaving an external address unmonitored is usually just a matter of omitting it from the `mon-externals` list, leaving a receptionist unmonitored is not always so simple. For instance, suppose that the ping/pong server can accept Pings on two different addresses, s1 and s2, but still sends all Pongs to the same external address dest. The program would look like the following:

```plaintext
(program (receptionists [s1 (Variant [Ping])] [s2 (Variant [Ping])])
  (externals [dest (Variant [Pong])])
;; program body elided)
```

A PSM has at most one input-monitored address, so a single PSM can describe the server's behavior in terms of messages received at s1 or s2, but not both. Nevertheless, a PSM that specifies just the inputs to s1 and the outputs to dest must account for the Pongs caused by Pings to s2, as well, because those Pongs are also sent to dest. The example below represents one possible way to handle this situation.

```plaintext
(specification
 (mon-receptionist s1)
 (mon-externals dest)
 (goto Running dest)
 (define-state (Running d)
   [(variant Ping) ->
     (obligation d (variant Pong))
     (goto Running d)]
   [free ->
     (obligation d (variant Pong))
     (goto Running d)])
)
```

This PSM's input-monitored address is just s1; it does not describe messages sent to s2. The additional `free` transition in the `Running` state says that at any time, the PSM might incur an additional Pong obligation without receiving a message via s1, thus accounting for the extra messages.

This specification is not as precise as one might like, because it allows for more possible Pong responses than are intended. This is because of the restriction that a PSM can have at most one input-monitored address. That restriction lends conceptual simplicity to APS specifications (i.e., that a PSM models how the environment can expect to use a given address), but the tradeoff is a loss of precision in cases like this.
CHAPTER 3. APS: ACTOR PROTOCOLS AS FINITE-STATE MACHINES

Ping/Pong with Dynamic Response Address

Another variation is a ping/pong server that sends responses to an address included with the Ping message, rather than to a static well-known address:

(specification
  (mon-receptionist s)
  (mon-externals)
  (goto Running)
  (define-state (Running)
    [(variant Ping d) ->
      (obligation d (variant Pong))
      (goto Running)])
)

Here, the PSM initially has no output-monitored addresses. Instead, the transition names the destination address $d$ included in the message (thereby adding $d$ as an output-monitored address external) and uses that address as the target of the obligation. Over the course of modeling a program’s execution, the PSM may incur Pong obligations on many different addresses.

Multi-State Ping/Pong

Of course, a finite-state machine is not terribly interesting if it has only one state. The following specification describes a server that starts in an Off state and does not respond to Pings until it receives an Init message with the destination for responses:

(specification
  (mon-receptionist s)
  (mon-externals)
  (goto Off)
  (define-state (Off)
    [(variant Ping) -> (goto Off)]
    [(variant Init d) -> (goto On d)])
  (define-state (On d)
    [(variant Ping) ->
      (obligation d (variant Pong))
      (goto On d)]
    [(variant Init new-d) -> (goto On d)])
)

In the transition from Off to On, $d$ is used as the state parameter to On so that it can be used in future transitions. Within the On state, further Init messages cause no further changes.

Ping/Pong Factory

Finally, as an example of forking PSMs, one might have a factory process for creating ping/pong-servers. The Make message sent to the factory includes an
3.1. SYNTAX AND INTUITIVE SEMANTICS

Listing 3.1: A ping/pong factory specification using `fork-addr`

A specification for such a program might look like the one in listing 3.1. Upon receiving a `Make` message, the PSM (which in this case describes the factory portion of the specified program) creates a new obligation for the observed program: it must send an address `a` to `d`. Furthermore, any messages sent to `a` must be handled as specified by the inner PSM, which corresponds to the PSM from the initial example.

3.1.4 Fork-Addr/Delayed-Fork-Addr

Some subtle details in the semantics necessitate splitting the patterns for fork addresses into two classes, `fork-addr` and `delayed-fork-addr`. In the example specification from listing 3.1, the obligation is incurred on the received address `d`, and the forked PSM also uses that address as its initial state argument so that it can incur `Pong` obligations on it. With the `fork-addr` pattern, the new PSM is created as soon as the parent incurs the obligation.
(specification
  (mon-receptionist f)
  (mon-externals mon)
  (goto Making m)
  (define-state (Making m)
    [(variant Make *) ->
     (obligation m
      (delayed-fork-addr
       ;; Behavior for the created ping/pong server:
       (goto Running)
       (define-state (Running)
        [(variant KeepAlive r) ->
         (obligation r (variant Ok))
          (goto Running)]))))
    (goto Making m))
  )

Listing 3.2: A ping/pong factory specification using delayed-fork-addr

Now compare that example with the situation where in addition to sending the new server’s address to the client, the factory also sends that address to a special monitor actor. The monitor maintains a list of all of the generated servers and periodically sends each one a keep-alive message so that it can detect if any server crashes. The program might then expose the following interface:

(program
  ;; Response = (Variant [Server (Addr (_variant [Ping]))]
  ;; [Pong])
  ;; MonitorAPI = (Variant [KeepAlive (Addr (variant [Ok]))])

  (receptionists [f (variant [Make (Addr Response)])])
  (externals [mon (variant (Addr MonitorAPI))])
  ;; program body elided
)

One might want to specify each server’s behavior in terms of its communication with the monitor rather than with the client, as in the specification in listing 3.2.

This specification has a largely similar shape to the previous one, but uses delayed-fork-addr instead of fork-addr. The difference relates to whether the forked PSM holds onto the obligation’s destination address (e.g., as a state argument), or the forking PSM holds onto it instead.\(^2\) When the child PSM holds onto the obligation’s address, as with \(d\) in the first example, that PSM must keep that

\(^2\)Recall from section 3.1.2 that to avoid conflicts, at most one PSM may use a given external address.
address as an output-monitored address and therefore maintains the *fork-addr* obligation for that address. Then when some message fulfills that obligation by providing the new server’s address, the child PSM knows to set that address as its own input-monitored address.

In the second case, though, the parent PSM specifies the obligation address \( m \) instead, because it keeps that address as an argument to its \( \text{Making} \) state. This means the parent PSM is also the one to track the *delayed-fork-addr* obligation until it is fulfilled. But if the child PSM were to be forked off immediately, the parent would be the one to learn the child’s input-monitored address when the obligation is fulfilled, with no way to communicate that information back to the forked child.

Instead, a *delayed-fork-addr* pattern waits to create the PSM (hence “delayed”) until the corresponding obligation is fulfilled by some message. That way, the parent continues to maintain the obligation address as an output-monitored address, and it can provide the forked PSM its input-monitored address at creation time.

A side effect of this delay is that from the time the obligation is incurred to the time it is fulfilled, any messages sent to the child PSM’s output-monitored addresses would not be checked against their obligations, because the only PSM assigned to do so would not yet be running. To express this invariant, a *delayed-fork-addr* is not allowed to take any state arguments, implying that it initially has no output-monitored addresses.

These restrictions on forks are one of the downsides of specifying all messages sent to output-monitored addresses (*i.e.*, no message may be sent without an obligation and every obligation must be fulfilled). As we will see in chapter 9, however, these limited capabilities are sufficient for describing several different communication patterns in real-world applications.

To summarize, with a *fork-addr* pattern

- the child PSM is created when the obligation is incurred as part of a state transition, and
- the child PSM keeps the target address for the obligation as an output-monitored address (and therefore maintains the *fork-addr* obligation, as well);

whereas with a *delayed-fork-addr* pattern

- the child PSM is created when the obligation is fulfilled by a sent message,
- the parent PSM keeps the target address for the obligation as an output-monitored address (and maintains the obligation itself), and
- the child PSM’s initial state takes no arguments.
CHAPTER 3. APS: ACTOR PROTOCOLS AS FINITE-STATE MACHINES

3.2 Extended Example: Stream-Processing

While Bob is designing his stream-processing system for computing temperature averages, he realizes the interaction between the manager, processors, and their clients (i.e., external actors that request new processors, supply readings, ask for the current average, etc.) involves a non-trivial protocol. As a result, he thinks it would be useful to specify the protocol from the point of view of a client, apart from the computational aspects of his system. That way he can ensure the protocol makes sense and he can check his implementation against the specification as he goes.

In particular, Bob wants the specification to answer questions such as

- What transitions are possible among the processor’s states? Of those, what transitions might be caused by a message other than those the client might send?
- How does the processor react in each state to each kind of received message?
- What is the behavior of the actor referred to by the response to a MakeProc request?

3.2.1 The Stream-Processing Specification

Listing 3.3 shows Bob’s APS specification for the manager actor. The receptionists and externals match the declarations from the program in the previous chapter. Line 2 tells us that user-api (which can receive MakeProc messages) is the PSM’s monitored receptionist. That means that whenever the manager receives a MakeProc message, its subsequent behavior must correspond to one of the two state transitions listed in the PSM (either it sends nothing to the environment, or it eventually sends one address to resp). By contrast, a ShutdownAll message can be received only by sys-api, which is an unmonitored receptionist. In that case, no particular behavior is specified in response to receiving a ShutdownAll message. This specification describes the server’s

**Listing 3.3: APS specification for the manager actor**

```plaintext
(specification
  (mon-receptionist user-api)
  (mon-externals)
  (goto Managing)
  (define-state (Managing)
    [((variant MakeProc resp mdest) -> (goto Managing))]
    [((variant MakeProc resp mdest) ->
      [obligation resp (fork-addr #|see listing 3.4|#)]
       (goto Managing))])
```

3.2 Extended Example: Stream-Processing

While Bob is designing his stream-processing system for computing temperature averages, he realizes the interaction between the manager, processors, and their clients (i.e., external actors that request new processors, supply readings, ask for the current average, etc.) involves a non-trivial protocol. As a result, he thinks it would be useful to specify the protocol from the point of view of a client, apart from the computational aspects of his system. That way he can ensure the protocol makes sense and he can check his implementation against the specification as he goes.

In particular, Bob wants the specification to answer questions such as

- What transitions are possible among the processor’s states? Of those, what transitions might be caused by a message other than those the client might send?
- How does the processor react in each state to each kind of received message?
- What is the behavior of the actor referred to by the response to a MakeProc request?

3.2.1 The Stream-Processing Specification

Listing 3.3 shows Bob’s APS specification for the manager actor. The receptionists and externals match the declarations from the program in the previous chapter. Line 2 tells us that user-api (which can receive MakeProc messages) is the PSM’s monitored receptionist. That means that whenever the manager receives a MakeProc message, its subsequent behavior must correspond to one of the two state transitions listed in the PSM (either it sends nothing to the environment, or it eventually sends one address to resp). By contrast, a ShutdownAll message can be received only by sys-api, which is an unmonitored receptionist. In that case, no particular behavior is specified in response to receiving a ShutdownAll message. This specification describes the server's
communication from the point-of-view of a client with only the capability to send MakeProc requests.

The specification starts in its only state, Managing. On lines 6–7, that state lists two transitions triggered by a MakeProc message, indicating there are two possible ways to react to that request. Both transitions’ patterns bind the included addresses resp and mdest, thereby marking them as output-monitored addresses in each case. The first says that the manager can silently ignore the request by sending no response at all. The other says the manager can eventually send back an address to resp, and the actor associated with the new address must behave as specified in the fork-addr term. For space reasons, the contents of the fork-addr term are displayed separately.

The forked PSM, shown in listing 3.4, specifies the processor’s behavior from the perspective of the client that requested it. In its initial Off state, the first two clauses are simple response-only transitions that do not change the state: AddRdg messages should be rejected with NotOk responses while in the Off state, and requests for the current average should get some response on mdest (recall that the name Off refers only to the processor’s temperature-recording capabilities, not its ability to respond to requests). The exact value of the response, however, is unspecified, because APS cannot express computational details such as numeric calculations.

A Disable message has no effect in the Off state, but an Enable message causes the actor to transition to On.

The last transition in the Off state accounts for the case where the processor actor receives a Shutdown message. Because this PSM does not describe the reception of those messages, the transition merely expresses that the processor may transition from Off to the Done state without any triggering message to the input-monitored address.

The On state is similar to Off, except that all temperature readings should receive an acknowledgment with an Ok response. Finally, the Done state declares that all messages should be ignored and the actor should send no further responses on received addresses.

The next chapter provides a proof that the program from chapter 2 conforms to this specification.

3.2.2 Specifications as FSM Diagrams

Although textual representations of specifications are better for formal presentations and automated tools, APS specifications are easily representable in the style of FSM diagrams. Figure 3.2 illustrates the stream-processing specification in the style of network protocol state machines, such as in the TCP specification [90]. The diagram includes the structure of the basic FSM from figure 2.2 of the previous chapter, but augments that model with the behavior expressed by the formal protocol specification.

Even with the extra information from the specification, though, the diagram is still easy to refer to at a glance. This suggests that programmers can rely on
(goto Off mdest)

(define-state (Off mdest)
  [(variant AddRdg * resp) ->
   [obligation resp (variant NotOk)]
   (goto Off mdest)]
  [(variant GetMean) -> [obligation mdest *] (goto Off mdest)]
  [(variant Disable) -> (goto Off mdest)]
  [(variant Enable) -> (goto On mdest)]
  [free -> (goto Done)])

(define-state (On mdest)
  [(variant AddRdg * resp) ->
   [obligation resp (variant Ok)]
   (goto On mdest)]
  [(variant GetMean) -> [obligation mdest *] (goto On mdest)]
  [(variant Disable) -> (goto Off mdest)]
  [(variant Enable) -> (goto On mdest)]
  [free -> (goto Done)])

(define-state (Done)
  [(variant AddRdg * resp) -> (goto Done)]
  [(variant GetMean) -> (goto Done)]
  [(variant Disable) -> (goto Done)]
  [(variant Enable) -> (goto Done)])

Listing 3.4: APS specification fragment for the processor actor
3.3. APS STRENGTHS AND WEAKNESSES

The previous example represents one possible use of APS. Generally, APS is best suited for specifying high-level patterns of communication that use a bounded number of addresses. It is also best used for protocols in which the state transitions and effects depend more on the kind of message received (e.g., something distinguished by a variant tag) than on the exact contents of the message in terms of primitive values such as numbers and strings.

APS is not intended for complete protocol specifications such as those for network protocols. For instance, although APS can approximate some of the state

Figure 3.2: An FSM representation of the stream-processing specification

t heir existing intuitions about state machine diagrams to read and write APS specifications, and that such diagrams might be useful during development.

3.3 APS Strengths and Weaknesses

The previous example represents one possible use of APS. Generally, APS is best suited for specifying high-level patterns of communication that use a bounded number of addresses. It is also best used for protocols in which the state transitions and effects depend more on the kind of message received (e.g., something distinguished by a variant tag) than on the exact contents of the message in terms of primitive values such as numbers and strings.

APS is not intended for complete protocol specifications such as those for network protocols. For instance, although APS can approximate some of the state
transitions of TCP (as we will see in chapter 9), the full TCP specification [90] requires reasoning about lower-level details such as sequence numbers and sliding-window sizes that APS is unable to express. An APS specification should act as a high-level guide for programmers (as well as for tools such as verifiers and test generators) rather than a complete protocol specification.

APS is also not well-suited for specifying communication involving an unbounded number of participants, such as broadcast and multicast. Because an APS state is parameterized over a bounded number of messages, it is unable to express the dynamically sized lists of addresses required for that sort of communication.

Finally, APS is unable to specify cause-and-effect relationships across actors. For instance, in the stream-processing example, APS is unable to express that a $\texttt{ShutdownAll}$ message sent to the manager actor will stop all of the processor actors. Once the processor specification has been forked, that forked PSM does not specify the reactions to events on the manager.

### 3.4 History Markers

The formal semantics for APS requires an extension to the CSA semantics. Up to this point, PSMs have been described in terms of input-monitored and output-monitored addresses, with the assumption that all occurrences of the same address are equivalent. However, specifying cause-and-effect relationships requires distinguishing even two occurrences of the same external address if each occurrence comes from a different external message. A similar distinction is needed for addresses the program provides to the environment. The following scenario shows how identifying the target of send and receive actions by destination address is insufficient. Section 3.4.2 introduces a new mechanism, called history markers, to provide the required distinctions. A second scenario then explains why some internal actions must be considered in addition to all external communication actions when checking whether a program conforms to its specification. Section 3.5 formalizes these ideas as an annotated semantics for CSA.

#### 3.4.1 Scenario 1: Replicated External Addresses

Figure 3.3 illustrates a partial execution sequence of Bob’s stream-processing program on the left-hand side, along with the processor PSM described in listing 3.4 attempting to simulate that execution on the right. The figure omits non-communication steps for the sake of the example, and the $\texttt{rcv-ext}$ labels omit their types for space reasons.

The initial program configuration $K$ has two processor actors, with addresses $a_{p1}$ and $a_{p2}$, respectively. The $\texttt{mdest}$ address for the first processor is $a_d$. $K$ exposes each processor’s address to the environment as a receptionist.

The box at the top of the right-hand sequence represents a PSM specifying the behavior a client expects to see from interactions with the first processor. The
Figure 3.3: A partial program execution (left-hand side), and a specification configuration’s failed attempt to simulate that execution (right-hand side). The specification is unable to complete the simulation because it cannot distinguish two uses of the same address for different callbacks.
PSM starts in the state On, with an address \(a_d\) as its \(mdest\) parameter. Its input-monitored address is \(a_{p1}\), meaning the PSM must take one of the transitions in the On state to simulate receiving a message on that address. The address \(a_{p2}\) is unmonitored, however, so the PSM can simulate an input on that address without an accompanying state transition. The PSM's only output-monitored address is \(a_d\), and there are no outstanding obligations.

In the program execution on the left, the environment first sends an AddRdg message to \(a_{p1}\) with \(a_r\) (an external address) as the callback. Then before the processor has a chance to finish handling that message, the environment sends to the address \(a_{p2}\) of the second processor a similar message, with the same callback. Each processor then responds with an Ok reply to \(a_r\), as expected.

The right-hand side shows how the PSM from listing 3.4 might transition to model the program's external communication. To express a PSM's transitions in terms of only what the environment observes, rather than what the program does internally, PSM transition steps use labels similar to those of the \(\pi\)-calculus [81]: \(a ? v\) represents the receptionist actor at address \(a\) receiving a message \(v\) from the environment, and \(a! v\) represents the program sending a message \(v\) to some actor in the environment with address \(a\).

The initial receive action happens on the PSM's input-monitored address \(a_{p1}\), so to simulate it, the PSM must take one of the state transitions from the On state. The only transition in listing 3.4 with a matching pattern in that state is the following:

\[
((\text{variant AddRdg } * \text{ resp}) \rightarrow \neg
[\text{obligation \text{ resp (variant Ok)}]} \neg
(\text{goto On mdest})]
\]

The transition binds \(a_r\) to \text{ resp} (making the address output-monitored), and incurs an obligation to send \text{ Ok} on that address. Simulating the program's second step, however, neither requires a state transition nor incurs an obligation on \(a_r\), because the receiving address \(a_{p2}\) is not an input-monitored address for the PSM. This leads to a problem when the PSM attempts to simulate the two output steps: because \(a_r\) is output-monitored, the PSM must discharge a matching obligation to send each output, but only one such obligation is available. Thus the interaction on the left-hand side of figure 3.3 apparently violates the specification of listings 3.3–3.4.

### 3.4.2 Solution: Marking Addresses in Transitions

Intuitively, this execution sequence matches the expected behavior: every AddRdg message to a processor results in exactly one Ok message sent to the included callback address. So why does this sequence appear to violate the specification modeled by the PSM? The root of the problem is that intuitively, the PSM specifies how the particular copy of \(a_r\) received in step 1 should be used, but the transition step labels as written do not distinguish between different copies of the same address. The solution is to annotate each occurrence of an address in
the execution in figure 3.3 with history markers (or simply markers) that identify the transmission of each occurrence. The PSM can then use history markers to identify each distinct address-copy.

A history marker identifies a particular communication step in the execution of a program along with a position in the communicated message. By analogy to plane travel, one can think of a history marker as denoting a particular flight (a send or receive step) and a particular seat on that flight (a position in the sent/received message). Then a marker annotating an occurrence of an address denotes a use of that address in some previous message from which this occurrence flowed. That is, the marker denotes part of the occurrence’s history as a transmitted value. A PSM can use a history marker to refer to just the copies of an address stemming from a particular message, rather than all copies of the address across the program configuration that it models.

Conformance to a PSM with history markers means that there is some marking of the initial program configuration that allows it to match that PSM (although it turns out that all markings that give distinct markers to distinct address occurrences are equivalent). Therefore, history markers are merely a formalism necessary at conformance-checking time to make the necessary distinctions; they do not change the behavior of a PSM. Neither the author of a program nor the author of a specification need be aware of the idea of history markers.

Figure 3.4 adapts the example execution for the use of markers (here given as natural-number superscripts in bold). In $K$, the receptionists $a_p^1$ and $a_p^2$ exposed to the environment are marked with 1 and 2, respectively. The copy of $a_d$ initially provided to the program (not shown) is marked with 3. The marker on the receiving actor’s address for a rcv-ext step identifies the particular copy of that address to which the received message was sent.

The PSM in the figure replaces its state arguments, input-monitored address, and output-monitored addresses with markers that identify the relevant copy of each address. In the PSM’s initial state, it has 1 as its lone input-monitored marker, meaning it must use a state transition to simulate receiving a message sent to any occurrence of an address marked with 1. Similarly, the use of 3 as the only output-monitored marker means that simulating a send to an address occurrence marked with 3 (e.g., $a_3^3$) requires a corresponding obligation.

During execution, every communication between the program and the environment adds a fresh marker to every address contained in the transmitted message. So the first step applies a fresh marker 4 to the received copy of $a_r$, indicating the address’s origin, and the second step similarly applies the marker 5 to its received copy of $a_r$. The markers then distinguish which copy was used in each of the subsequent outputs: the destination in step 3 is marked with 4 (meaning that occurrence flowed from the first AddRdg message), while the destination in step 4 is the same address, but marked with 5.

This distinction between the two output actions now allows the program execution to satisfy the specification. In the first step, the PSM records marker 4 as an output-monitored marker and incurs an obligation for that marker. Then it discharges its obligation in the third step to match the output to an address marked with 4. No such obligation is needed for the final output step, though,
Figure 3.4: By referring to address occurrences by history marker rather than by the address itself, the specification can simulate the entire program execution.
because the marker 5 on the destination is not output-monitored. Thus, history markers distinguish uses of the same address received in different messages, matching the intuitive goal of APS.

### 3.4.3 Scenario 2: Redirecting to Internal Addresses

The previous scenario dealt with the case in which the environment provides the same external address in two different messages. This next scenario demonstrates what might go wrong in a naive semantics for APS when the environment provides an internal address as a callback rather than an external address.

For the sake of the example, suppose Bob decided to add a testing facility into his stream-processing system. In particular, he modified the manager actor so that it additionally accepts messages of the form `(variant SendGetMean a)`, which causes the manager to send a GetMean message to a.

Figure 3.5 illustrates an example usage of this message. The left-hand sequence is again a partial execution of the program configuration K, and the right-hand sequence represents a PSM’s attempt to simulate that execution.

The initial program configuration K, along with the processors from the previous example, has a manager actor with address am, with a copy marked with 6 exposed to the environment. In the first step, the environment sends the message `(SendGetMean a17)` to the manager. The included address is marked with 1, indicating the environment sent in the particular instance of a1p1 to which it had access (and for which the PSM specifies all input actions); 7 is the fresh marker applied in this step. Internally, the program sends a GetMean message to the processor, and the processor picks it up for handling. Finally, the processor sends the current mean value, 45, to the chosen mean-destination address ad in the environment.

The obvious, naive semantics for APS would be to label all internal transitions with the silent label • (similar to a τ action in the π-calculus), because internal actions are invisible to the environment. Under that rule, the PSM takes the first step without changing its internal state (because the marker 6 on the destination address is not input-monitored). The next two steps similarly do not change its state because there is no observable action to specify. However, the last step requires the PSM to simulate an output to an address marked with 3, which it is unable to do because the marker 3 is output-monitored but has no obligations.

### 3.4.4 Solution: Marking in Internal Transitions

In this case, the problem is that the specification is intended to describe the results of all messages sent to the address-occurrence a17p1, but this naive semantics observes only the subset of messages that are sent by the environment. Therefore, the PSM does not see that the first processor receives a GetMean message in the third step, and so it is unable to take a state transition to incur the corresponding response obligation on ad.

The solution involves modifying which steps we consider relevant to the specification. Specifically, instead of marking transmitted addresses and exposing the
Figure 3.5: Because the first step provides internal address $a^{1,7}_{p_1}$ as a callback, the PSM cannot see the reception of the message sent to that address, and therefore does not incur the obligation necessary to simulate the response to $a^{3}_{d}$.
communication to the specification whenever the communication crosses the pro-
gram/environment boundary, the marking and exposure both happen whenever
the destination of the send/receive action is annotated with at least one history
marker. All receptionists provided to the environment and externals provided
to the program are marked, so this rule applies to strictly more transition steps
than the previous rule.

Think of a marker as a kind of remote observation device the environment at-
taches to every address received from or sent to the program. The device reports
back to the environment whenever that copy of the address is used to send or
receive a message, and it additionally attaches a new observation device to any
addresses transmitted in messages sent to this copy of the address. The upshot is
that the environment (and therefore the PSM) observes all communication that
can be transitively traced back to some message the environment sent to the
program, but all other communication remains hidden away.

Figure 3.6 illustrates this change for the second scenario. Because every com-
munication step happens on a marked address, the destination and message of
every communication are exposed to the PSM. The crucial difference occurs in
the PSM's third step: because the label now includes the reception of a message
sent to an address instance marked with 1, the PSM must take a state transition
corresponding to that message. The transition incurs an obligation * for output-
monitored marker 3, which the following step uses to simulate the program's
output, finishing the simulation of the execution.

Exposing these internal actions to the specification is necessary to define how
a PSM can simulate all possible executions of a (conforming) program. A theo-
rem defined in appendix B, however, states that to prove that a program conforms
to its specification, it is sufficient to consider only those messages from the en-
vironment that do not contain internal addresses. In that case, the transition
steps using marked addresses are exactly those involving communication to or
from the environment, so the theorem both recovers the intuition of “black-box”
specifications and allows conformance proofs to avoid the complexities introduced
here.

3.5  Marked Transition Semantics for CSA

The addition of history markers to CSA requires modifications to the struc-
ture of program configurations and their components, the instantiation func-
tion \( \text{Inst} \), and the transition rules for sending and receiving, as well as a new
well-formedness condition. The additions are concerned only with adding and
maintaining history markers throughout executions, so it is easy to see that the
marked semantics is otherwise identical to the unmarked semantics. History
markers do not affect the typing rules, however, so the typing rules for marked
program configurations are analogous to the rules in chapter 2 and therefore
omitted.
Figure 3.6: When the specification monitors the communication on all marked addresses, it can observe the reception of the message sent to $a^{1,7}_{p_1}$ and generate an obligation for the expected response to $a^3_{a_d}$. 
3.5. MARKED TRANSITION SEMANTICS FOR CSA

\[ \eta \in \text{HistMark} \]  \hspace{1cm} \text{(History Markers)}
\[ H \in \mathcal{P}(\text{HistMark}) \]  \hspace{1cm} \text{(History Marker Sets)}
\[ \hat{K} := \left\langle \hat{\beta} \mid \hat{\mu} \mid H \right\rangle \hat{\rho} \]  \hspace{1cm} \text{(Marked Program Configurations)}
\[ \hat{\beta} \in \text{Addr} \rightarrow \text{MarkedBeh} \]  \hspace{1cm} \text{(Marked Actor-Behavior Maps)}
\[ \hat{\mu} \in \mathcal{M}(\text{MarkedAddr} \times \text{MarkedVal}) \]  \hspace{1cm} \text{(Marked Message Multisets)}
\[ \hat{\rho} \in \mathcal{P}(\text{MarkedAddr} \times \text{Type}) \]  \hspace{1cm} \text{(Marked Receptionist Sets)}
\[ \hat{\nu} ::= a@H \mid (\text{record} \bar{t} \bar{v}) \mid (\text{variant} \tau \bar{v}) \mid (\text{fold} \tau \bar{v}) \]  \hspace{1cm} \text{(Marked Values)}
\[ \mid (\text{list} \bar{v}) \mid (\text{dict} \bar{v} \bar{v}) \mid n \mid \text{str} \]
\[ \hat{I} ::= a: \text{rcv-ext}(H,\bar{v},\tau) \mid a: \text{rcv-int}(H,\bar{v}) \]  \hspace{1cm} \text{(Marked Transition-Step Labels)}
\[ \mid a: \text{send-ext}(a@H,\bar{v}) \mid a: \text{send-int}(a@H,\bar{v}) \mid a: \text{timeout} \mid a: \text{func} \]
\[ \mid a: \text{goto} \mid a: \text{spawn}(a) \]

Figure 3.7: Marked program configurations

3.5.1 Program Configurations

Figure 3.7 lists the significant changes for marked program configurations and their components. Omitted non-terminals are the obvious replacement of their original components with corresponding marked components.

A history marker \( \eta \) is an element of a set \( \text{HistMark} \), left abstract for generality. The previous sections used the set of natural numbers as markers, but the semantics requires only that \( \text{HistMark} \) is an infinite set with a well-order\(^3\) and without a greatest element. Every set of markers \( H \) appearing in a program configuration or specification configuration is finite by construction of that configuration. These conditions guarantee that for any given set of markers, there exists a marker greater than all markers in the set, so that a new marker can always be applied to a transmitted address.

Occurrences of addresses in expressions, receptionist sets, and message maps are all \textit{marked} addresses \( a@H \), where \( H \) is a set of markers. The markers in the set represent the previous send or receive steps that this copy of the address flowed from. For an in-flight message \( (a@H,\bar{v}) \in \hat{\mu}, a@H \) is the marked address to which the message was sent. For a receptionist \( (a@H,\tau) \), \( a@H \) is the marked instance that the environment received in some message and can use for subsequent transmissions back to the program.

The program configuration also stores the set \( H \) of markers used so far. Even after all other occurrences of a marker \( \eta \) disappear from a program configuration, a specification configuration may still refer to that marker, so the program configuration uses this set to determine the next fresh marker to apply to a sent or received address.

\(^3\) A well order is a total order such that every non-empty subset has a least element.
inst(P,a_1'...a_m') = \langle \tilde{K}, [x_1 \rightarrow \eta_1,...,x_n \rightarrow \eta_n], [x'_1 \rightarrow \eta'_1,...,x'_m \rightarrow \eta'_m] \rangle

where $P =$

\begin{align*}
\text{(program (receptionists \{x_l \rightarrow \tau_l\}_{l \in 1..n}) (externals \{x'_j \rightarrow \tau'_j\}_{j \in 1..m}))}
\end{align*}

and $\tilde{K} = \langle \{a_1'' \rightarrow \tilde{b}_1,...,a'_m'' \rightarrow \tilde{b}_m\} | \emptyset | \{\eta_1,...,\eta_n,\eta'_1,...,\eta'_m\} \rangle \langle \{a_1\@\{\eta_1,\tau_1\},...,a_n\@\{\eta_n,\tau_n\}\} \rangle$

and $a_k'' = (\text{addr} \ell_k 0)$
and $\eta_1,...,\eta_n,\eta'_1,...,\eta'_m$ are distinct
and $\tilde{b}_k = \text{InstAct}(\tilde{Q}_k,e_k,[\text{self} \rightarrow a''_k\@\emptyset][x''_1 \rightarrow a''_1\@\emptyset]...[x''_m \rightarrow a''_m\@\emptyset][x'_1 \rightarrow a'_1\@\{\eta'_1\}]...[x'_m \rightarrow a'_m\@\{\eta'_m\}])$
and $a_i = x''_1'...a''_1'...x'_m'...a'_m'$

Figure 3.8: Marked instantiation for program configurations

\begin{align*}
\text{InstAct}(\tilde{Q}, (\text{goto} q \text{ ov}_1 \ldots \text{ ov}_n), [x_1 \rightarrow \tilde{v}_1]...[x_m \rightarrow \tilde{v}_m]) =

\langle \tilde{Q}', (\text{receive} x'' \text{ ev} tc) [x'_1 \rightarrow \tilde{v}'_1]...[x'_m \rightarrow \tilde{v}'_m] \rangle
\end{align*}

where $\tilde{Q}' = \tilde{Q}[x_1 \rightarrow \tilde{v}_1]...[x_m \rightarrow \tilde{v}_m]$
and (\text{define-state} \langle q [x'_1 \rightarrow \tau_1] ... [x'_m \rightarrow \tau_m] \rangle \text{ x'' ev tc} ) is in $\tilde{Q}'$
and $\tilde{v}'_i = \text{ov}_i[x_1 \rightarrow \tilde{v}_1]...[x_m \rightarrow \tilde{v}_m]$

Figure 3.9: Marked instantiation for actors

Finally, each transition step label $\tilde{l}$ also records the markers on the destination address of the relevant communication (if any).

### 3.5.2 Instantiation

Marked instantiation for programs (figures 3.8 and 3.9) is similar to the original version, except that every declared receptionist and external gets its own unique marker. The function also returns the name-to-marker bindings for the receptionists and externals in addition to the instantiated configuration, for use in instantiating specifications (section 3.6.2).

### 3.5.3 Transition Relation

Figure 3.10 lists the transition rules that require non-trivial changes for this semantics; the rules not shown here are analogous to the corresponding rules from the previous chapter.

Each of the listed rules marks the addresses in the sent/received value $\tilde{v}$ with the new history markers if the destination $a\@H$ has a non-empty marker set (see below for the description of the Markings function). The new markers identify those copies of those addresses as having participated in this particular step.
3.5. MARKED TRANSITION SEMANTICS FOR CSA

\[ \langle \hat{\beta} \mid a \rightarrow \langle \hat{Q}, (\text{receive } x \; \bar{e} \; \bar{f} \; c) \rangle \mid \hat{\mu} \mid \mathcal{H}' \rangle \hat{\rho} \quad \frac{a: \text{rcv-int}(\mathcal{H}, \hat{\rho})}{\langle \hat{\beta} \mid a \rightarrow \langle \hat{Q}, e \mid x \; \bar{e} \; \bar{f} \; c \rangle \mid \hat{\mu} \mid \mathcal{H}' \rangle \hat{\rho}} \]

where \( \langle \bar{v}', \mathcal{H}' \rangle \in \text{Markings}(\bar{v}, \mathcal{H}) \) if \( \mathcal{H} \neq \varnothing \), else \( \langle \bar{v}', \mathcal{H}' \rangle = \langle \bar{v}, \mathcal{H} \rangle \)

\[ \langle \hat{\beta} \mid a \rightarrow \langle \hat{Q}, (\text{receive } x \; \bar{e} \; \bar{f} \; c) \rangle \mid \hat{\mu} \mid \mathcal{H}' \rangle \hat{\rho} \quad \frac{a: \text{rcv-ext}(\mathcal{H}, \hat{\rho}, \cdot)}{\langle \hat{\beta} \mid a \rightarrow \langle \hat{Q}, e \mid x \; \bar{e} \; \bar{f} \; c \rangle \mid \hat{\mu} \mid \mathcal{H}' \rangle \hat{\rho}} \]

if \( \exists \bar{v}. \; (a@\mathcal{H}, \tau) \in \hat{\rho} \) and \( \varnothing \vdash \bar{v} : \tau \)

where \( \langle \bar{v}', \mathcal{H}' \rangle \in \text{Markings}(\bar{v}, \mathcal{H}) \)

and \( \forall \langle a'@\mathcal{H}'', \tau'' \rangle \in \text{IntAddrTypes}(\bar{v}, \tau). \exists \tau'''. \langle a'@\mathcal{H}'', \tau''' \rangle \in \hat{\rho} \) and \( \tau''' < : \tau' \)

and \( \forall a'@\mathcal{H}'' \) in \( \bar{v} \), if \( a' \) is external, then \( \mathcal{H}'' = \varnothing \)

\[ \langle \hat{\beta} \mid a \rightarrow \langle \hat{Q}, E1(\text{send } a@\mathcal{H} \; \bar{v}) \rangle \mid \hat{\mu} \mid \mathcal{H}' \rangle \hat{\rho} \quad \frac{a: \text{send-int}(a@\mathcal{H}, \hat{\rho})}{\langle \hat{\beta} \mid a \rightarrow \langle \hat{Q}, E1(\text{send } a@\mathcal{H} \; \bar{v}) \rangle \mid \hat{\mu} \mid \mathcal{H}' \rangle \hat{\rho}} \]

if \( a' \) is internal

where \( \langle \bar{v}', \mathcal{H}' \rangle \in \text{Markings}(\bar{v}, \mathcal{H}) \) if \( \mathcal{H} \neq \varnothing \), else \( \langle \bar{v}', \mathcal{H}' \rangle = \langle \bar{v}, \mathcal{H} \rangle \)

and \( \bar{v}' = (\text{variant } \text{Unit}) \)

\[ \langle \hat{\beta} \mid a \rightarrow \langle \hat{Q}, E1(\text{send } a@\mathcal{H} \; \bar{v}) \rangle \mid \hat{\mu} \mid \mathcal{H}' \rangle \hat{\rho} \quad \frac{a: \text{send-ext}(a@\mathcal{H}, \hat{\rho})}{\langle \hat{\beta} \mid a \rightarrow \langle \hat{Q}, E1(\text{send } a@\mathcal{H} \; \bar{v}) \rangle \mid \hat{\mu} \mid \mathcal{H}' \rangle \hat{\rho}} \]

if \( a' \) is external, \( \text{ActorType}(a') = \tau \), and \( \text{IntAddrTypes}(\bar{v}', \tau) = \hat{\rho}' \)

where \( \langle \bar{v}', \mathcal{H}' \rangle \in \text{Markings}(\bar{v}, \mathcal{H}) \)

Figure 3.10: Marked-program-semantics transition rules
In the M-RECEIVEEXTERNAL rule, the original message \( \bar{v} \) received from the environment (before being marked) does not appear in either the program configurations or the transition label; the rule requires merely that such a \( \bar{v} \) exists. This rule requires that all external addresses in the received message \( \bar{v} \) are unmarked. Because the CSA semantics does not model communications between two external actors, if any address with an output-monitored marker were to escape, messages the environment sent to it would not be accounted for. Therefore APS does not allow such addresses to escape to the environment (see section 3.6.5). For similar reasons, APS also enforces that all addresses with input-monitored markers must be internal (see section 3.6.6). This implies that any markers on external addresses that escape to the environment are irrelevant to the PSM, so for simplicity we can assume that the environment sends only unmarked versions of its addresses to the program. Future work could potentially address these limitations, although they have not been a hindrance for the examples built so far.

Also in that rule, the definition of \textit{IntAddrTypes} is assumed to be updated to extract \textit{marked} addresses from values, rather than unmarked ones as in the corresponding function in the previous chapter. The rules are otherwise similar to their counterparts in the unmarked semantics.

The function \textit{Markings} defines the set of all possible ways to mark a given value \( \bar{v} \) with fresh markers on every address, where a given set of used markers \( H \) determines freshness. Every member of the set is a tuple \( \langle \bar{v}', H' \rangle \), where \( \bar{v}' \) is the marked value and \( H' \) is the extended set of markers used so far. Figure 3.11 defines \textit{Markings}.

The new marker for an address must be greater than all of its existing markers, so that the newest marker is uniquely identified. The marking is non-deterministic to simplify a proof in appendix D.

### 3.5.4 Well-Formed Configurations

Some of the theorems later in this dissertation rely on a program configuration being \textit{well-formed}. Effectively, this property states that every internal address corresponds to some actor in the configuration, and that history markers have been applied appropriately to all components of the configuration. The formal definition follows:

\textbf{Definition.} A marked configuration \( \langle \langle \hat{\beta}, \hat{\mu}, H \rangle \rangle^\hat{\beta} \) is well-formed if and only if

1. for all \( a \) appearing in \( \hat{\beta}, \hat{\mu}, \text{ and } \hat{\rho} \), if \( a \) is internal then \( a \in \text{dom}(\hat{\beta}) \),

2. for all \( a@H' \) appearing in \( \hat{\beta}, \hat{\mu}, \text{ and } \hat{\rho} \), \( H' \subseteq H \),

3. for all \( a@H' \) appearing in \( \hat{\beta} \) and \( \hat{\mu} \) such that \( a \) is external, \( H' \neq \emptyset \), and

4. for all \( \langle a@H', \tau \rangle \in \hat{\rho} \), \( H' \neq \emptyset \) and there does not exist \( \tau' \neq \tau \) such that \( \langle a@H', \tau' \rangle \in \hat{\rho} \).
Markings($\bar{v}, H$) =

- **Case $\bar{v} = a@H'$:**
  \[
  \{a@H' \cup \{\eta\} | \eta \notin H \text{ and } \eta > \eta' \text{ for all } \eta' \in H'\}
  \]

- **Case $\bar{v} = n:**
  \[
  \{n\}
  \]

- **Case $\bar{v} = \text{str}:**
  \[
  \{\text{str}\}
  \]

- **Case $\bar{v} = (\text{variant } t \bar{v}_1 \ldots \bar{v}_m):**
  \[
  \{(\text{(variant } t \bar{v}_1' \ldots \bar{v}_m'), H_{n+1}) | (\bar{v}_i', H_{i+1}) \in \text{Markings}(\bar{v}_i, H_i) \text{ where } H_1 = H\}
  \]

- **Case $\bar{v} = (\text{record } [r_i \bar{v}_i])_{i=1..n}:**
  \[
  \{(\text{(record } [r_i \bar{v}_i']_{i=1..n}, H_{n+1}) | (\bar{v}_i', H_{i+1}) \in \text{Markings}(\bar{v}_i, H_i) \text{ where } H_1 = H\}
  \]

- **Case $\bar{v} = (\text{fold } t \bar{v}'):**
  \[
  \{(\text{(fold } t \bar{v}'), H') | (\bar{v}', H') \in \text{Markings}(\bar{v}, H)\}
  \]

- **Case $\bar{v} = (\text{list } \bar{v}_1 \ldots \bar{v}_n):**
  \[
  \{(\text{(list } \bar{v}_1' \ldots \bar{v}_n'), H_{n+1}) | (\bar{v}_i', H_{i+1}) \in \text{Markings}(\bar{v}_i, H_i) \text{ where } H_1 = H\}
  \]

- **Case $\bar{v} = (\text{dict } [\bar{v}_1' \bar{v}_1''] \ldots [\bar{v}_n' \bar{v}_n'']) :**
  \[
  \{\{(\text{dict } [\bar{v}_1' \bar{v}_1''] \ldots [\bar{v}_n' \bar{v}_n'']), H_{n+1}) | (\bar{v}_i', H_{i+1}) \in \text{Markings}(\bar{v}_i, H_i) \text{ and } (\bar{v}_i'', H_{i+1}) \in \text{Markings}(\bar{v}_i', H_i') \text{ where } H_1 = H\}
  \]

Figure 3.11: The sets of valid markings
The second condition requires that the set \( H \) of used markers contain every marker appearing throughout the configuration. The third and fourth conditions require that all external addresses and all receptionists are marked with at least one marker: those addresses represent the communication link with the environment, so all communication over them should be observable. The fourth condition also requires that no marked address is exposed as a receptionist at two different types; this is necessary for a proof in appendix B. Receptionists created by the \texttt{M-SENDEXTERNAL} rule are guaranteed to preserve this property, because each one is marked with a fresh marker in that step, making it unique among the other receptionists.

**Lemma** (Well-Formed Preservation). For all \( \bar{K}, \bar{l}, \) and \( \bar{K}' = \langle \bar{H}', \bar{H}, \bar{\varphi}, \bar{\eta}, \bar{\Phi}, O \rangle \), if \( \bar{K} \) is well-formed and \( \bar{K} \xrightarrow{\bar{l}} \bar{K}' \), then \( \bar{K}' \) is well-formed and every marker appearing in \( \bar{l} \) is a member of \( H' \).

Appendix I proves this lemma.

## 3.6 Marked Transition Semantics for APS

Section 3.7 defines conformance between a CSA program and an APS specification in terms of a simulation relation between a program configuration and a specification configuration. In preparation, this section formally defines specification configurations, PSMs, their transition relations, and related concepts.

### 3.6.1 Specification Configurations

The communicating process defined by an APS specification is represented by a configuration \( S \), defined in figure 3.12. Formally, a specification configuration \( S \) is a set of PSMs, with each PSM \( s \) specifying some protocol. A history marker \( \eta \) is monitored in a specification configuration if it is input-monitored or output-monitored in any of the configuration’s constituent PSMs.

A PSM is a tuple \( \langle H,H',\varphi:\eta,\Phi,O \rangle \) consisting of

- the set of input-monitored history markers \( H \) (containing at most one marker),
- the set of output-monitored history markers \( H' \)

\[ S \in \mathcal{F}(PSM) \quad \text{(Specification Configurations)} \]
\[ s \in PSM ::= \langle H,H',\varphi:\eta,\Phi,O \rangle \quad \text{(Protocol Specification Machines)} \]
\[ O \in \mathcal{M}(HistMark \times OutPat) \quad \text{(Obligation Multisets)} \]
3.6. MARKED TRANSITION SEMANTICS FOR APS

State: (On 3)
Input-Monitored: 1
Output-Monitored: \{3\}
Obligations: \{3 : *\}

 corresponds to \( \langle \eta_1, \eta_3, 0n : \eta_3, \overline{\Phi}, \{\langle \eta_3, *\rangle\} \rangle \)

Figure 3.13: Informal and formal representations of a PSM

\( \text{SpecInst}(\Sigma, [x_1 \rightarrow \eta_1, \ldots, x_n \rightarrow \eta'_n], [x'_1 \rightarrow \eta'_1, \ldots, x'_m \rightarrow \eta'_m]) = \{s\} \)

where \( \Sigma = \text{specification \( mr \) (mon-externals \( x''_1 \ldots x''_p \)) (goto \( \varphi \) \( x''_1 \ldots x''_q \) \( \overline{\Phi} \))} \)
and \( x''_j \in \{x''_1, \ldots, x''_p\} \) for all \( j \in 1 \ldots q \)
and \( H = \{x''_1[x_1 \rightarrow \eta_1] \ldots [x_n \rightarrow \eta_n]\} \) if \( mr = x'' \), else \( \varphi \)
and \( H' = \bigcup_{j \in 1 \ldots q} x''_j[x'_1 \rightarrow \eta'_1] \ldots [x'_m \rightarrow \eta'_m] \)
and \( \eta''_j = x''_j[x'_1 \rightarrow \eta'_1] \ldots [x'_m \rightarrow \eta'_m] \) for \( j \in 1 \ldots q \)
and \( s = \langle H, H', \varphi : \eta'_1 \ldots \eta'_q, \overline{\Phi}, \varphi \rangle \)

Figure 3.14: Specification instantiation

• the current state \( \varphi \) of the specification along with the state's marker arguments \( \eta_i \),
• the set of state definitions \( \overline{\Phi} \), and
• the multiset of outstanding obligations \( O \).

An obligation \( \langle \eta, po \rangle \) represents a commitment to eventually send a message matching the pattern \( po \) to an address marked with \( \eta \).

Figure 3.13 shows the correspondence between the fourth informal PSM from figure 3.6 in Scenario 2 and its formal representation. The state definitions \( \overline{\Phi} \) define the stream processor specification's states, and the markers \( \eta_1 \) and \( \eta_3 \) are distinct history markers corresponding to the informal markers 1 and 3, respectively.

3.6.2 Instantiation

The function \( \text{SpecInst} \) in figure 3.14 instantiates specifications into specification configurations, similar to the \( \text{Inst} \) function for CSA programs. It takes as input the specification \( \Sigma \), the bindings \([x_1 \rightarrow \eta_1, \ldots, x_n \rightarrow \eta'_n]\) for the initial receptionist markers, and the bindings \([x'_1 \rightarrow \eta'_1, \ldots, x'_m \rightarrow \eta'_m]\) for the initial external markers.

The function returns the initial configuration consisting of just a single PSM \( s \).

The instantiation is possible only if every initial state argument \( x''_j \) is one of the specification's monitored externals \( \{x''_1, \ldots, x''_p\} \). The returned PSM's initial input-monitored markers \( H \) (if any) are defined by the clause \( mr \), with the receptionist markers \( \eta_1, \ldots, \eta_n \) substituted into the variable if present. Similarly, the initial output-monitored markers \( H' \) are defined by substituting the external
$f ::= \ldots \mid (\text{fork } (\text{goto } \varphi \eta) \overline{\Phi}) \mid (\text{obligation } \eta po)$ \hspace{1cm} (Effects)

$po ::= \ldots \mid (\text{fork-addr } (\text{goto } \varphi \eta) \overline{\Phi})$ \hspace{1cm} (Output Patterns)

Figure 3.15: Effects and patterns extended to allow markers as arguments

$$
\text{Perform}(f) = \\
\left\{ \\
\varphi, \{ \varphi, \{ \eta_1, \ldots, \eta_n \}, \varphi : \eta_1 \ldots \eta_n, \overline{\Phi}, \varphi \} \right\}, \varphi \\
\text{if } f = (\text{fork } (\text{goto } \varphi \eta_1 \ldots \eta_n) \overline{\Phi}) \\
\left\{ \{ \langle \eta, po' \rangle \}, S, H \right\} \\
\text{if } f = (\text{obligation } \eta po) \text{ and Extract}(po, \eta) = \langle po', S, H \rangle
$$

Figure 3.16: Semantics of performing effects

$\text{PerformAll}(f_1 \ldots f_n) = \langle \bigcup_{i \in 1 \ldots n} O_i, \bigcup_{i \in 1 \ldots n} S_i, \bigcup_{i \in 1 \ldots n} H_i \rangle$

where $\text{Perform}(f_i) = \langle O_i, S_i, H_i \rangle$ for all $i \in 1 \ldots n$

markers $\eta'_1, \ldots, \eta'_m$ into the names given in the mon-externals clause. Finally, the external markers are also substituted for the state arguments $x'_1, \ldots, x'_q$. The PSM starts with no obligations.

For convenience, the configurations $\overline{K}$ and $S$ are said to be instan-
tiable from $P$ and $\Sigma$ if the respective instantiations use corresponding mark-
ers. That is, there exist $[x_1 \leftarrow \eta_1, \ldots, x_n \leftarrow \eta_n]$ and $[x'_1 \leftarrow \eta'_1, \ldots, x'_m \leftarrow \eta'_m]$ such that $\text{Inst}(P, \overline{\alpha}) = \langle \overline{K}, [x_1 \leftarrow \eta_1, \ldots, x_n \leftarrow \eta_n], [x'_1 \leftarrow \eta'_1, \ldots, x'_m \leftarrow \eta'_m] \rangle$ and $\text{SpecInst}(\Sigma, [x_1 \leftarrow \eta_1, \ldots, x_n \leftarrow \eta_n], [x'_1 \leftarrow \eta'_1, \ldots, x'_m \leftarrow \eta'_m]) = S$.

3.6.3 Performing Effects

The transition relation for PSMs defined in section 3.6.5 requires two helper functions: one to perform the effects of a state transition, and another to distribute a transition’s generated obligations among the current and forked PSMs. This section defines the former; the next section introduces the latter.

The function $\text{Perform}$ (figure 3.16) determines the obligations $O$ and forked PSMs $S$ from performing an effect of a state transition. It also reports which markers $H$ acquire obligations with a self-addr pattern, which the PSM transition rules need in order to distribute the obligations among the PSMs (see section 3.6.5). It assumes an extended syntax for effects $f$ and output patterns $po$, shown in figure 3.15, that allows markers $\overline{\eta}$ to substitute for variables $\overline{x}$.

A fork effect creates a new PSM with no input-monitored marker, the state arguments $\eta_1, \ldots, \eta_n$ as output-monitored markers, and no obligations.

An obligation effect creates a new obligation, but also must extract any forked
PSMs from the specified pattern. The Extract function in figure 3.17 takes a marker and pattern for an obligation, and it returns an updated pattern, the extracted PSMs, and a set of markers (explained below). In addition to its state arguments, each PSM extracted from a fork-addr also output-monitors the marker used for the obligation itself, as discussed in section 3.1.4. The fork-addr pattern itself turns into a self-addr pattern, because the obligation will be transferred to the forked PSM.

An instance of self-addr in the original pattern should refer to the current PSM rather than any forked PSMs. Therefore when such a pattern is found, Extract additionally returns a set containing the marker for this obligation, indicating the marker has an obligation related to the current PSM. The PSM transition rules use the set of such markers found to ensure obligations with self-addr patterns are not transferred to forked PSMs.

A helper function PerformAll, also defined in figure 3.16, collects the results from performing several individual effects.

### 3.6.4 Distributing Pattern-Bound Markers and Obligations

Once PerformAll generates new obligations, they must each be distributed to either the current PSM or one of the child PSMs created as part of the state transition. The Dist function, defined in figure 3.18, describes that distribution.

The function takes the current PSM \( s_1 \), the forked PSMs \( s_2, \ldots, s_n \), and the obligations \( O \) to distribute. It first checks that the output-monitored-marker sets \( H'_i \) of all involved PSMs are disjoint in order to prevent conflicts; otherwise the function is undefined. Such a conflict usually indicates a poorly written specification, because it describes multiple conflicting uses of the same address. A static well-formedness check on specifications could likely rule out such cases, but is omitted for simplicity.

If that condition holds, then the given obligations are split into disjoint multisets \( O'_i \), one per PSM, such that every obligation \( \langle \eta', po \rangle \) is assigned to the PSM that output-monitors the obligation’s destination marker (i.e., \( \eta' \in H'_i \)). The result is the collection of PSMs \( s'_1, \ldots, s'_n \) updated with the new obligations, with the current PSM distinguished from the forked children.

### 3.6.5 PSM Transition Relation

The transition relation for PSMs formally defines how a PSM simulates the communication of a program conforming to the specification. The relation is independent of the transition relation for CSA; the notion of conformance in section 3.7 relates the two.

The transition step labels for PSMs and specification configurations, introduced informally in section 3.4, are defined below.

\[
\lambda ::= a@H?\delta \mid a@H!\delta \mid \bullet \quad \text{(Specification Transition-Step Labels)}
\]

The labels define the “observed” information from each step (recall that one may think of history markers as remotely observing addresses and reporting us-
\[(\ast, \varnothing, \varnothing)\]

if \(po = \ast\)

\[\langle \text{or } po' \ldots po'' \rangle, \bigcup_{i \in 1 \ldots n} S_i, \bigcup_{i \in 1 \ldots n} H_i \]

if \(po = \langle \text{or } po' \ldots po'' \rangle\)
where \(\text{Extract}(po', \eta) = \langle po'_i, S_i, H_i \rangle\) for all \(i \in 1 \ldots n\)

\[\langle \text{variant } t \text{ po'} \ldots \text{ po''} \rangle, \bigcup_{i \in 1 \ldots n} S_i, \bigcup_{i \in 1 \ldots n} H_i \]

if \(po = \langle \text{variant } t \text{ po'} \ldots \text{ po''} \rangle\)
where \(\text{Extract}(po'_i, \eta) = \langle po''_i, S_i, H_i \rangle\) for all \(i \in 1 \ldots n\)

\[\langle \text{record } \left[ r_1 \text{ po'} \right] \ldots \left[ r_n \text{ po'} \right] \rangle, \bigcup_{i \in 1 \ldots n} S_i, \bigcup_{i \in 1 \ldots n} H_i \]

if \(po = \langle \text{record } \left[ r_1 \text{ po'} \right] \ldots \left[ r_n \text{ po'} \right] \rangle\)
where \(\text{Extract}(po'_i, \eta) = \langle po''_i, S_i, H_i \rangle\) for all \(i \in 1 \ldots n\)

\[\langle \text{self-addr}, \{\varnothing, \eta_1', \ldots, \eta_n'\}, \varnothing : \eta_1' \ldots \eta_n', \varnothing, \varnothing \rangle, \varnothing\]

if \(po = \langle\text{fork-addr} \ (\text{goto} \ \varnothing \ \eta_1' \ldots \eta_n') \ \varnothing \rangle\)

\[\langle \text{delayed-fork-addr} \ (\text{goto} \ \varnothing) \ \varnothing, \varnothing, \varnothing \rangle\]

if \(po = \langle\text{delayed-fork-addr} \ (\text{goto} \ \varnothing) \ \varnothing \rangle\)

\[\langle \text{self-addr}, \varnothing, \{\eta\} \rangle\]

if \(po = \text{self-addr}\)

---

**Figure 3.17**: Fork extraction

\[\text{Dist}(s_1, s_2, \ldots, s_n, O) = \langle s'_1, s'_2, \ldots, s'_n \rangle\]

if \(s_i = \langle H_i, H'_i, \psi_i : \eta_i, \varnothing, O'_i \rangle\) for all \(i \in 1 \ldots n\)
and \(H'_i \cap H'_j = \varnothing\) for all \(i, j \in 1 \ldots n\) such that \(i \neq j\)
and \(O = \bigcup_{i \in 1 \ldots n} O'_i\)
and \(\eta' \in H'_i\) for all \(i \in 1 \ldots n\) and all \(\langle \eta', po \rangle \in O''_i\)
where \(s'_i = \langle H_i, H'_i, \psi_i : \eta_i, \varnothing, O'_i \cup O''_i \rangle\) for all \(i \in 1 \ldots n\)

**Figure 3.18**: Distribution of obligations among PSMs
age back to the environment). The label $a@H?\overrightarrow{v}$ represents an actor at $a$ receiving message $\overrightarrow{v}$ sent to marked address $a@H$, the label $a@H!\overrightarrow{v}$ represents the sending of a message $\overrightarrow{v}$ to marked address $a@H$, and $\bullet$ represents a step with no observed effects.

Figure 3.19 defines the rules of the transition relation itself. A transition step has the form $s, \lambda, O \rightarrow s'$, which means the PSM $s$ can take a step with label $\lambda$ to become $s'$, fulfilling the obligations $O$ and forking a set of PSMs $S$ in the process.

The rule P-UnmonitoredReceive allows the PSM to receive any message sent to any marked address $a@H'''$, as long as the PSM does not input-monitor any of those markers (i.e., $H \cap H''' = \emptyset$). Because the received message is irrelevant to the PSM’s protocol, the transition step does not modify the PSM nor fork any children. This is how the PSM simulates receiving a message that is not explicitly specified in its FSM-like description.

If a received message was sent to an address marked with some input-monitored marker, though, then P-MonitoredReceive applies instead. To simulate that step, the PSM’s current state $\varphi$ must have a state transition with a pattern $pi$ as its firing condition. The line $\overrightarrow{v} \sim pi \triangleright [x_{n+1} \rightarrow \eta_{n+1}, \ldots, x_{n+p} \rightarrow \eta_{n+p}]$ is a judgment of the input-pattern-matching relation (defined in section 3.6.6), saying that $\overrightarrow{v}$ matches $pi$ and generates the bindings $[x_{n+1} \rightarrow \eta_{n+1}, \ldots, x_{n+p} \rightarrow \eta_{n+p}]$. The rule performs the state transition’s effects (after substituting in the state arguments $\eta_1, \ldots, \eta_n$ and the pattern-bound markers $\eta_{n+1}, \ldots, \eta_{n+p}$), yielding new obligations $O'$ and new PSMs $S$. The rule then updates the current PSM with new output-monitored markers $H'''$, state $\varphi'$, and state arguments $\eta_1', \ldots, \eta_m'$, and distributes the new obligations $O'$ over it and the new children $S$. Any pattern-bound marker not output-monitored in one of the forked PSMs is added as an output-monitored marker in the current PSM so that some PSM specifies its use ($OutMon(S)$ stands for the set of output-monitored markers in the PSMs $S$).

P-FreeTransition is similar to P-MonitoredReceive, except that it uses a free state transition rather than one with an input pattern. Thus, no pattern variables are bound, and there are no new output-monitored markers. Such a step involves no communication, so it uses the silent label $\bullet$.

Finally, P-Send lets the PSM simulate sending a message. To do so, the PSM must discharge one obligation for each of the output-monitored markers on the message’s destination $a@H''$, with the sent message $\overrightarrow{v}$ matching each discharged obligation’s pattern $po_i$. The output-pattern-match $\overrightarrow{v} \sim po_i \triangleright H''', S_i$ (defined in section 3.6.6) produces new input-monitored markers $H''$ (from self-addr patterns) and forked PSMs $S_i$ (from delayed-fork-addr patterns). The rule checks that adding the new input-monitored markers $H''' \cup \ldots \cup H''''$ does not violate the invariant that a PSM can have at most one input-monitored marker. As mentioned in section 3.5.3, the rule also enforces that no marked address $a'@H'''$ in the message with a output-monitored marker can escape to the environment. Finally, the transition step’s label reports the fulfilled obligations $O'$. 
CHAPTER 3. APS: ACTOR PROTOCOLS AS FINITE-STATE MACHINES

### P-UNMONITOREDRECEIVE

\[
H \cap H'' = \emptyset
\]

\[
\left< H, H', \varphi : \eta, \Phi, O \right> \xrightarrow{a @ H'' ? \ell, \sigma, \phi} \left< H, H', \varphi : \eta, \Phi, O \right>
\]

### P-MONITOREDRECEIVE

\[
H \cap H'' \neq \emptyset
\]

**Define-state** \( (\varphi \ x_1 \ldots \ x_n) \overset{\delta}{\rightarrow} \left[ \text{goto } \varphi' \ x'_1 \ldots x'_m \right] \sigma' \in \Phi \)

\( \sigma \sim po \triangleright \[ x_{n+1} \rightarrow \eta_{n+1}, \ldots, x_{n+p} \rightarrow \eta_{n+p} \] \)

**PerformAll** \( \left[ \left[ x_1 \rightarrow \eta_1 \right] \ldots \left[ x_{n+p} \rightarrow \eta_{n+p} \right] \right] = \langle O', S, H'' \rangle \)

\( \eta'_j = x'_j[x_1 \rightarrow \eta_1] \ldots [x_{n+p} \rightarrow \eta_{n+p}] \) for all \( j \in 1 \ldots m \)

\( H''' = \{ \eta'_1, \ldots, \eta'_m \} \cup H'' \cup \{ \eta_{n+1}, \ldots, \eta_{n+p} \} - \text{OutMon}(S) \)

**Dist** \( \langle H, H', \varphi : \eta' \ldots \eta_m, \Phi, O \rangle, S, O' \rangle = \langle s', S' \rangle \)

\[
\left< H, H', \varphi : \eta_1 \ldots \eta_n, \Phi, O \right> \xrightarrow{a @ H'' ? \ell, \sigma, S'} s'
\]

### P-FREETRANSITION

**Define-state** \( (\varphi \ x_1 \ldots x_n) \overset{\delta}{\rightarrow} \left[ \text{free } \rightarrow \right. \left. \left[ \text{goto } \varphi' \ x'_1 \ldots x'_m \right] \sigma' \right] \}

**PerformAll** \( \left[ \left[ x_1 \rightarrow \eta_1 \right] \ldots \left[ x_n \rightarrow \eta_n \right] \right] = \langle O', S, H'' \rangle \)

\( \eta'_j = x'_j[x_1 \rightarrow \eta_1] \ldots [x_n \rightarrow \eta_n] \) for all \( j \in 1 \ldots m \)

**Dist** \( \langle H, H', \varphi : \eta'_1 \ldots \eta'_m, \Phi, O \rangle, S, O' \rangle = \langle s', S' \rangle \)

\[
\left< H, H', \varphi : \eta_1 \ldots \eta_n, \Phi, O \right> \xrightarrow{a @ H'' ? \ell, \sigma, S'} s'
\]

### P-SEND

\( H' \cap H'' = \{ \eta'_1, \ldots, \eta'_n \} \)

\( O' = \{ \langle \eta'_1, po_1 \rangle, \ldots, \langle \eta'_n, po_n \rangle \} \)

\( \sigma \sim po \triangleright H''', S_i \) for all \( i \in 1 \ldots n \)

\( |H \cup H'_1 \cup \ldots \cup H'_n| \leq 1 \)

\( H''' \cap H' = \emptyset \) for all \( a' @ H'''' \) in \( \sigma \)

\[
\left< H, H', \varphi : \eta, \Phi, O \cup O' \right> \xrightarrow{a @ H'''' ? \ell, S_1 \cup \ldots \cup S_n} \left< H \cup H'''' \cup \ldots \cup H'''_n, H', \varphi : \eta, \Phi, O \right>
\]

Figure 3.19: PSM transition rules
\[ \eta = \max(H) \]

\[ \bar{v} \sim \emptyset \triangleright \emptyset \]

\[ a@H \sim x \triangleright [x \mapsto \eta] \]

\[ \bar{v}_i \sim p_i \triangleright [x_{i,1} \mapsto \eta_{i,1}, \ldots, x_{i,m} \mapsto \eta_{i,m}] \text{ for all } i \in 1 \ldots n \]

\[(\text{variant } t \bar{v}_i \in \{1 \ldots n\}) \sim (\text{variant } t p_i \in \{1 \ldots n\}) \triangleright \bigcup_{i \in 1 \ldots n} [x_{i,1} \mapsto \eta_{i,1}, \ldots, x_{i,m} \mapsto \eta_{i,m}] \]

\[ \bar{v}_i \sim p_i \triangleright [x_{i,1} \mapsto \eta_{i,1}, \ldots, x_{i,m} \mapsto \eta_{i,m}] \text{ for all } i \in 1 \ldots n \]

\[(\text{record } [r_i \bar{v}_i \in \{1 \ldots n\}] \sim (\text{record } [r_i p_i \in \{1 \ldots n\}] \triangleright \bigcup_{i \in 1 \ldots n} [x_{i,1} \mapsto \eta_{i,1}, \ldots, x_{i,m} \mapsto \eta_{i,m}] \]

\[ \bar{u} \sim p \triangleright [x_1 \mapsto \eta_1, \ldots, x_n \mapsto \eta_n] \]

\[(\text{fold } \tau \bar{v}) \sim p \triangleright [x_1 \mapsto \eta_1, \ldots, x_n \mapsto \eta_n] \]

Figure 3.20: Pattern-matching for input patterns

### 3.6.6 Pattern Matching

Figure 3.20 defines the relation \( \bar{v} \sim p \triangleright [x_1 \mapsto \eta_1, \ldots, x_n \mapsto \eta_n] \) for matching input patterns. The judgment produces the bindings \([x_1 \mapsto \eta_1, \ldots, x_n \mapsto \eta_n]\) created by matching the pattern \(p\) against a value \(\bar{v}\). A pattern variable matches a marked address and binds the maximal marker on that address (i.e., the most recently applied marker). Other patterns use the obvious component-wise matching. Patterns are assumed to be linear, i.e., every pattern uses a given variable at most once.

Figure 3.21 defines pattern-matching for output patterns, written \( \bar{v} \sim po \triangleright H, S \). Matching a value \(\bar{v}\) against an output pattern \(po\) produces a set of markers \(H\) (the new input-monitored markers discovered via self-addr patterns), and a set of new PSMs \(S\) from delayed-fork-addr patterns.

A delayed-fork-addr pattern matches a marked address and creates the new PSM, with the maximal (i.e., newest) marker used as the PSM's input-monitored marker. As mentioned in section 3.5.3 that address must be internal because the semantics does not model messages the environment sends to external addresses. The rule for self-addr is similar, except it returns the marker in a set on its own rather than creating a PSM, because that marker refers to the current PSM rather than a new forked child. The or pattern matches if one of its constituent patterns matches the value. The other pattern-matching rules merely combine the results from their sub-matches.

### 3.6.7 Configuration Transition Relation

As mentioned in section 3.1, a specification configuration is effectively a conjunction of PSMs, enforcing that a conforming program acts as specified by each
\[ \begin{align*}
\delta \sim \ast & \triangleright \varnothing, \varnothing \\
\delta \sim (\text{or} \ p_{o1} \ p_{o2}) & \triangleright H, S \\
\delta_i \sim p_{oi} & \triangleright H_i, S_i \\
(\text{variant} \ t \ \delta_i^{i \in 1 \ldots n}) \sim (\text{variant} \ t \ p_{oi}^{i \in 1 \ldots n}) & \triangleright \bigcup_{i \in 1 \ldots n} H_i, \bigcup_{i \in 1 \ldots n} S_i \\
\delta_i \sim p_{oi} & \triangleright H_i, S_i \\
(\text{record} \ [r_i \ \delta_i]^{i \in 1 \ldots n}) \sim (\text{record} \ [r_i \ p_{oi}]^{i \in 1 \ldots n}) & \triangleright \bigcup_{i \in 1 \ldots n} H_i, \bigcup_{i \in 1 \ldots n} S_i \\
\delta \sim p_{o} & \triangleright H, S \\
(\text{fold} \ \tau \ \delta) & \sim p_{o} \triangleright H, S
\end{align*} \]

\[ a \text{ is internal} \quad \eta = \max(H) \]

\[ a@H \sim (\text{delayed-fork-addr} \ (\text{goto} \ \varnothing \ \Phi)) \triangleright \varnothing, \{ \{ \eta \}, \varnothing, \varnothing : \epsilon, \Phi, \varnothing \} \]

\[ a \text{ is internal} \quad \eta = \max(H) \]

\[ a@H \sim \text{self-addr} \triangleright \{ \eta \}, \varnothing \]

Figure 3.21: Pattern-matching for output patterns

\[ \text{S-SENDORRECEIVE} \]

\[ \lambda \neq \bullet \quad s_i \xrightarrow{\lambda, O_i, S_i} s'_i \text{ for all } i \in 1 \ldots n \]

\[ \{s_1, \ldots, s_n\} \xrightarrow{\lambda, O_1 \ldots \ldots O_n} \{s'_1, \ldots, s'_n\} \cup S_1 \cup \ldots \cup S_n \]

\[ \text{S-FREETRANSITION} \]

\[ s \xrightarrow{\ast, \varnothing, S'} s' \]

\[ \{s\} \cup S \xrightarrow{\ast, \varnothing} \{s'\} \cup S \cup S' \]

Figure 3.22: Specification configuration transition rules
constituent PSM. Therefore a specification configuration has its own transition relation, so that it models the communication expected by all of its PSMs. Figure 3.22 defines this relation.

The main rule is S-SENDORRECEIVE. The rule says that the configuration can take a step labeled with \( a@H?v \) or \( a@H!v \) whenever all of its PSMs can take a step with that label. Every PSM in the configuration specifies the program’s behavior from some point of view defined by its monitored markers, so each one must account for any communication it monitors. The result is a configuration containing the stepped original PSMs \( (s'_1, ..., s'_n) \) and the sets of forked PSMs created during this step \( (S_1 \cup ... \cup S_n) \). The step additionally exposes the multiset \( O_1 \cup ... \cup O_n \) of obligations fulfilled by each PSM.

The other rule, S-FREETRANSITION, allows any single PSM to take one of its free transitions at any time. The step does not involve any communication that might be monitored, so the rule does not require the other PSMs in the configuration to take similar transitions. Any forked PSMs are again added to the configuration. Such a step does not fulfill any obligations.

Conformance is partially based on the idea of a weak simulation [79] that ignores silent steps. Therefore, specification configurations have a weak-step transition relation \( \lambda, O \rightarrow \lambda, O \), indicating that the configuration takes a step labeled with \( \lambda \) along with any number of • transition steps before or after that step. This is defined below.

**Definition.** The weak-step transition relation \( \rightarrow \) for specification configurations is the smallest relation such that \( S \lambda, O \rightarrow S' \) if and only if one of the following holds:

- \( \lambda = •, O = \emptyset \), and there exist \( S''_1, ..., S''_n \) (for \( n > 0 \)) such that \( S''_1 = S, S''_n = S' \), and \( S_1 \rightarrow^* ... \rightarrow^* S_n \).

- There exist \( S'' \) and \( S''' \) such that \( S \rightarrow^* S'' \rightarrow^* \lambda, O \rightarrow \lambda, O \rightarrow^* S''' \rightarrow^* S' \).

The original transition step \( \lambda \rightarrow \) may be referred to as the strong-step transition relation. For both the strong-step and weak-step relations, the fulfilled obligations \( O \) may be omitted when irrelevant. Thus, \( S \lambda \rightarrow S' \) if and only if there exists \( O \) such that \( S \lambda, O \rightarrow S' \), and \( S \rightarrow \lambda \rightarrow S' \) if and only if there exists \( O \) such that \( S \lambda, O \rightarrow S' \).

### 3.6.8 Executions and Fairness

As with program configurations, an execution of a specification configuration \( S \) is a transition sequence of the form \( S_1 \lambda_1, O_1 \rightarrow S_2 \rightarrow S_3 \rightarrow \cdots \) with \( S_1 = S \). Then a fair execution is an execution in which every PSM eventually discharges all of its obligations, formalized below.
Definition. Let $Obls\left(\{(H, H', \phi, \eta, \Phi, O)\}\right) = O$, and let $Obls((s_1, \ldots, s_n)) = \bigcup_{i=1}^{n} Obls(s_i)$. Then an execution $S_1 \xrightarrow{\lambda_1, O_1} S_2 \xrightarrow{\lambda_2, O_2} \ldots$ is fair if and only if for all $S_1$, $po$, and $\eta$ such that $(\eta, po) \in Obls(S_1)$, there exists a step $S_{i+j} \xrightarrow{\lambda_{i+j}, O_{i+j}} S_{i+j+1}$ later in the execution such that $(\eta, po) \in O_{i+j}$.

Similarly, a weak-step execution is a sequence $S_1 \xrightarrow{\lambda_1, O_1} S_2 \xrightarrow{\lambda_2, O_2} \ldots$ of weak-step transitions. Such an execution is fair if it can be written equivalently in terms of a fair strong-step execution, i.e., with each step $S_i \xrightarrow{\lambda_i, O_i} S_{i+1}$ rewritten as $S_i \xrightarrow{\lambda_i, O_i} S'_{i,1} \xrightarrow{\lambda_i, O_i} \ldots \xrightarrow{\lambda_i, O_i} S'_{i,n} \xrightarrow{\lambda_i, O_i} \ldots \xrightarrow{\lambda_i, O_i} S''_{i,m} \xrightarrow{\lambda_i, O_i} S_{i+1}$. Because weak-step executions are used more commonly than strong-step executions, this dissertation sometimes refers to them simply as “executions”; the symbol for the transition relation used resolves any ambiguity.

### 3.7 Conformance

Simulation relations (such as the conformance relation for APS) are often used to relate a pair of communicating processes in which one acts as a specification of the other. Loosely, a process $P$ simulates a process $Q$ if whenever $Q$ takes a step to $Q'$, $P$ can take a similar step to some $P'$, and $P'$ simulates $Q'$. Such a relation provides a safety property, ensuring that $Q$ (the program) never does something that $P$ (the specification) does not allow. In the case of APS and CSA, safety means that the program does not send undesired messages.

In addition to safety, however, APS aims to enforce a liveness property: every fair execution must fulfill all the obligations it incurs over the course of its run. Therefore, conformance for APS is defined in terms of the simulation relation of Grumberg and Long [53], which adapts simulation relations to account for fairness conditions.

To relate the executions of a program and a specification, an operation $\lfloor \_ \rfloor$ translates program transition-step labels to specification transition-step labels. It follows the rule stated in section 3.4.3, that a communication step is observed whenever the marker set $H$ of its destination address is non-empty.

\[
\lfloor a : \text{rcv-ext}(H, \hat{v}, \tau) \rfloor = a@H?\hat{v} \text{ if } H \neq \emptyset, \text{ else } \ast
\]
\[
\lfloor a : \text{rcv-int}(H, \hat{v}) \rfloor = a@H?\hat{v} \text{ if } H \neq \emptyset, \text{ else } \ast
\]
\[
\lfloor a : \text{send-ext}(a'@H, \hat{v}) \rfloor = a'@H!\hat{v} \text{ if } H \neq \emptyset, \text{ else } \ast
\]
\[
\lfloor a : \text{send-int}(a'@H, \hat{v}) \rfloor = a'@H!\hat{v} \text{ if } H \neq \emptyset, \text{ else } \ast
\]
\[
\lfloor a : \text{timeout} \rfloor = \ast
\]
\[
\lfloor a : \text{func} \rfloor = \ast
\]
\[
\lfloor a : \text{goto} \rfloor = \ast
\]
\[
\lfloor a : \text{spawn}(a) \rfloor = \ast
\]

The definition of conformance first requires a notion of a conformance-dense relation, which defines what it means for a specification configuration to be able to match every step of a program configuration.
3.7. CONFORMANCE

**Definition.** A relation $R$ on pairs of program/specification configurations is conformance-dense if and only if for all $\langle \tilde{K}_1, S_1 \rangle \in R$ and all fair, non-stuck program executions $\tilde{K}_1 \overset{i_1}{\to} \tilde{K}_2 \overset{i_2}{\to} \ldots$, there exists a fair specification execution $S_1 \overset{i_1}{\to} S_2 \overset{i_2}{\to} \ldots$ with the same length such that $\langle \tilde{K}_i, S_i \rangle \in R$ for all $\tilde{K}_i$ and $S_i$ in the respective executions.

This is similar to a simulation relation, in that for every step a program configuration $\tilde{K}_i$ can take, the corresponding specification configuration $S_i$ must be able to match that step. Additionally, however, any such sequence of matching steps representing a fair program execution must fulfill all obligations in the specification configurations of that sequence: this enforces the liveness condition for APS.

For example, say $R_{SP}$ is a conformance-dense relation for a program containing just one stream-processor actor from the previous chapter and its corresponding PSM from listing 3.4. Then $R_{SP}$ would have to include one pair with the actor and PSM both in their respective Off states, another in their respective On states, another in their respective Done states, and many others for all of the intermediate configurations while the actor is handling a message. It would not include, however, a pair with the actor in its On state and the PSM in the Off state, because the program’s response to AddRdg messages in that state does not match the specified response in the PSM’s Off state: this violates the safety aspect of conformance. It would also not include a pair where the program and PSM are both in their Done states, but with an additional obligation to send an Ok message in the PSM, because the program would never send such a message from that state, meaning it would violate the liveness aspect of conformance.

Conformance density ignores stuck executions because detecting run-time errors is outside the scope of APS specifications. Rather, conformance to an APS specification expresses that if a program has no run-time errors, then its communication behavior will match the specification.

Conformance density then entails a definition of conformance between a program configuration and a specification configuration.

**Definition.** A program configuration $\tilde{K}$ conforms to a specification configuration $S$, written $\tilde{K} \models S$, if and only if there exists a conformance-dense relation $R$ such that $\langle \tilde{K}, S \rangle \in R$.

It may be helpful to think of a conformance-dense relation $R$ as the evidence that some program conforms to its specification. A program/specification-configuration pair conforms only if the specification can match every step of the program. $R$ is self-consistent in that sense: it provides those matching steps from every member of $R$. So a program configuration conforms to a specification configuration if and only if we can provide suitable evidence for its conformance. This is similar to the difference between a simulation relation and similarity in process algebras: a process $P$ is similar to a process $Q$ if and only if $\langle P, Q \rangle$ is a member of some simulation relation.

---

4Appendix F describes a technique for reasoning only in terms of the non-intermediate states.
Of course, what we actually want is a notion of conformance between the program and specification that a programmer writes down, not the configurations that they induce. We can easily adapt the configuration-based notion of conformance to static programs and specifications by instantiating each one, as follows:

**Definition.** A CSA program $P$ conforms to an APS specification $\Sigma$, written $P \models \Sigma$, if and only if there exists at least one instantiation $\bar{K}$ and $S$ of $P$ and $\Sigma$, and for all such instantiations, $\bar{K} \models S$.

Checking for the existence of at least one instantiation eliminates vacuous conformance results in which the “interface” (defined by the receptionists and externals) of $P$ and $\Sigma$ differ.

### 3.7.1 Maximal Instantiations

Although the conformance is defined in terms of all possible instantiations of a program $P$ and specification $\Sigma$, it is sufficient to check just one representative instantiation for conformance. This allows the model checker of chapter 7 to check just one initial configuration pair rather than infinitely many, and also simplifies the relationship to stronger notions of conformance introduced later.

Only a subset of the possible instantiations are usable for this purpose, however, because only some of them demonstrate all possible behaviors of the program. For example, say $P$ expects to be instantiated with an external address that can accept messages of type $(\text{Addr} \ (\text{Variant} \ [A] \ [B]))$. That is, the environmental actor identified by that address should be able to accept another address as a message, where that address in turn can accept $A$ or $B$ messages. The following sequence might happen:

1. The program $P$ is instantiated with an address $a$ such that $\text{ActorType}(a) = (\text{Addr} \ (\text{Variant} \ [A]))$.
2. After instantiation, the program sends to $a$ an address $a'$ such that $\text{ActorType}(a') = (\text{Variant} \ [A] \ [B])$.
3. Because the environment receives $a'$ in a context where it expects only something of type $(\text{Addr} \ (\text{Variant} \ [A]))$, the environment can send an $A$ to $a'$, but is unable to send a $B$.

To be useful as a representative instantiation for checking conformance, that instantiation must be “maximal” in terms of the types of addresses provided as externals. Formally, an instantiation $\bar{K}$ and $S$ of $P$ and $\Sigma$ is maximal if and only if for every address $a_i$ assigned to each external $[x_i, \tau_i]$ in the instantiation of $P$, $\text{ActorType}(a_i) = \tau_i$. This definition leads to the following theorem:

**Theorem** (Maximal Instantiation). For all CSA programs $P$ and APS specifications $\Sigma$, $P \models \Sigma$ if and only if there exists some maximal instantiation $\bar{K}$ and $S$ of $P$ and $\Sigma$ such that $\bar{K} \models S$. 
In light of this result, all other notions of conformance defined later in this dissertation are defined in terms of a maximal instantiation, rather than all instantiations. Appendix A gives a proof of this theorem. The main idea is to show that a maximal instantiation of a program can simulate the transitions of any other instantiation of that program.

3.8 Refinements to Conformance

To prove that a program conforms to its specification, one must show that every reachable configuration of the program conforms to some appropriate configuration of that specification. Ideally, a conformance proof would categorize these program/specification-configuration pairs into a small number of sets and prove conformance for each set as a whole. The wide variety of reachable configurations, however, defies easy categorization.

To alleviate this problem, there is a series of refinements to conformance that reduce the number of conforming configuration pairs needed for a conformance proof. Specifically, these refinements enable a conformance proof to

- ignore messages from the environment that include one of the program’s own actor addresses (appendix B),
- let the program’s reaction to a given external message to stand for its reaction to similar messages containing different addresses (appendix C),
- assume that an actor cannot start handling an event unless all other actors are idle (appendix D),
- assume that all fresh names (addresses, markers) are generated via some deterministic scheme (appendix E),
- reason in terms of “event steps” that represent the entire sequence of actions an actor takes to handle an event, rather than individual transition steps (appendix F), and
- prove conformance to a single PSM at a time rather than to a configuration consisting of many PSMs (appendix G).

The final notion of conformance that encompasses all of these refinements is called PSM conformance. It is denoted with the symbol $\models_{\text{PSM}}$ rather than $\models$. The following chapter gives an example proof using PSM conformance, and chapter 5 develops a notion of conformance for abstract program configurations that is based on PSM conformance.

After applying these refinements, the remaining pairs are easier to partition by similarity into a small number of sets, thereby resulting in manageable conformance proofs. In fact, each of these refinements is equivalent to the conformance definition given above,\(^5\) leading to the following theorem:

\(^5\)At least for well-typed programs, which are the only programs for which conformance matters.
Theorem (Conformance Equivalence). For all $P$ and $\Sigma$ such that $\vdash_{\text{prog}} P$, $P \models_{\text{PSM}} \Sigma$ if and only if $P \models \Sigma$.

Proof. Let there be some $P$ and $\Sigma$ such that $\vdash_{\text{prog}} P$.

- By the PSM Conformance theorem (appendix G), $P \models_{\text{PSM}} \Sigma$ if and only if $P \models_{\text{EV}} \Sigma$.
- By the Event-Step Conformance theorem (appendix F), $P \models_{\text{EV}} \Sigma$ if and only if $P \models_{\text{D}} \Sigma$.
- By the Deterministic-Handler Conformance theorem (appendix E), $P \models_{\text{D}} \Sigma$ if and only if $P \models_{\text{SH}} \Sigma$.
- By the Single-Handler Conformance theorem (appendix D), $P \models_{\text{SH}} \Sigma$ if and only if $P \models_{\text{ER}} \Sigma$.
- By the External-Representative Conformance theorem (appendix C), $P \models_{\text{ER}} \Sigma$ if and only if $P \models_{\text{EO}} \Sigma$.
- By the Externals-Only Conformance theorem (appendix B), $P \models_{\text{EO}} \Sigma$ if and only if $P \models \Sigma$.

Therefore, $P \models_{\text{PSM}} \Sigma$ if and only if $P \models \Sigma$. \qed

The description of each of these refinements to conformance and its equivalence proof can be found in the appendix listed above, but the body of this dissertation does not describe them further so as to focus on the development of a model checker for APS.

Also, the transition relation $\rightarrow_R$ allows only the transitions allowed in the first four refinements above (the remaining two refinements do not restrict the possible transitions). Appendices B–E define transition relations that add those refinements one at a time, so appendix E defines $\rightarrow_R$ itself. The abstract interpretation of CSA introduced in chapter 5 is sound with respect to this relation.

### 3.9 Related Work

**Session Types** Session types [62, 63] are another well-studied means for specifying communication between processes, including dynamic communication topology. A session type describes the possible sequences of messages to be exchanged over a given channel during a given communication session. Type-checking is then a matter of first projecting this global view of the session into a set of local views that describe how each session participant should use the channel, then checking that each participant satisfies that local type. Type safety in such a program usually guarantees properties like deadlock-freedom and session progress.

Session types can express properties similar to APS; indeed, there is work in this area to use communicating FSMs as specifications [39]. The fixed nature of
an actor's address, however, makes session types a bad fit for the actor model. In a session-type setting, messages are sent and received over a unique channel per session. In an actor-based setting, however, an actor must use the same address to receive messages from many different sessions. Thus, an actor address cannot have a type for just a single session, but must account for a potentially unbounded number of sessions.

Three projects have added session types to actor models. Crafa [35] presents a type system that checks the communication actions of actors that halt within a fixed number of steps. Mostrous and Vasconcelos [82] add a session-type system to Erlang that prevents incomplete sessions and unreceived messages, but the session-type encoding restricts the possible shapes of messages. Neykova and Yoshida [84] propose a hybrid actor/session-type system that requires a central message broker to connect actors. The broker does not allow evolving communication topologies. Generally, sessions encourage a different method of communication from idiomatic actor programming.

Other Type Systems  There are many existing type systems for specifying communication behavior, usually falling under the heading of behavioral type systems. The survey by Hüttel et al. [65] gives an overview of this space.

Contracts for mobile processes [24] are another behavioral type system for specifying dynamic communication behavior. A contract is a type structured much like a term in a process algebra, with send actions, receive actions, and channel creation. Contracts have not been applied to actor programs, and it is not clear whether contracts can express APS features such as unobservability and output commitments.

Conversation types [22] provide a behavioral type discipline for the conversation calculus [109]. The conversation calculus is an extension of the π-calculus in which all communication takes place over the medium of nested conversations, which processes may join or leave independently. Messages in individual conversations can have different labels (known to the type system) to distinguish between different messages. The type system guarantees session fidelity and linear usage of labels that should be used linearly. Conversation types do not have the ability to express unobserved messages or the concept of output commitments that are eventually satisfied. They also do not have the ability to specify that a message in one conversation causes a message to be sent in some other conversation—conversations are kept separate.

Regular types [85] are a type system for objects with non-uniform service availability to approximate the times when an object is able to handle a message of a given type. Others have extended this idea to message-passing systems [30, 83]. Puntigam [91] presents another type system in this vein, but in which messages additionally carry “type marks” that give the receiving process the capability to send more messages. The focus in those systems is to prevent cases in which a sent message is never handled, whereas I focus on checking the expected usage of an address in a context with uniform service availability.

Kobayashi [68] defines a type system that describes the usage of a channel
with a process-like type. Each send or receive action additionally has obligation and capability levels to indicate when an action must happen, or when it may happen. The mailbox types of de’Liguoro and Padovani [38] also describe the valid usages of a channel, but using commutative regular expressions over types. Both of these type systems can express many properties expressible by APS, but they cannot express cause-and-effect relationships across multiple messages. For example, neither one can express that the response to the GetMean message in the running example should be sent to the mdest address obtained when the processor actor was created.

A running theme in these works is that most type systems for communication give types for channels or processes, but not both. There are limitations to either choice. If the type system describes only expected channel usage, then it is unable to talk about cause-and-effect relationships across multiple messages, as in the GetMean example mentioned above. On the other hand, if the type system describes only the actions taken by a given process, then it can be difficult to show how multiple processes working together in concert can satisfy a single specification.

**Temporal Logics** Researchers have used a variety of temporal logics to specify properties of message-passing programs. The RMC model checker for Rebeca [100] specifies properties in both LTL (linear temporal logic) [88] and CTL (computation tree logic) [28]. These logics are unable to quantify over the addresses that may be received, however, making it impossible to express properties about dynamic communication topologies. Also, the LTL and CTL formulas are defined in terms of propositions over the states a program can be in, rather than over the (communication) actions that cause transitions between those states. As a result, such specifications are forced to mandate certain implementation details of the program.

Lamport’s TLA [70] (especially its extended TLA+ form [72]) is commonly used to specify reactive systems. It is based on set theory, first-order logic, temporal logic, and a notion of actions, which are formulas that express the expected value of state variables after a step. Because the formulas are so state-focused, expressing specifications about the communications that occur in transitions between states is more difficult than in APS.

The METT system of Bauer et al. [16] extends LTL with quantification over process identifiers. They use this to describe the communication behavior of groups of processes in a dynamic communication topology, but in which processes cannot create other processes and there is only one type of process. METT’s lack of recursion would make encoding FSM-like specifications difficult, however, if not impossible. It also does not appear to be possible to specify linear usage of process identifiers in this system.

Dam et al. [36] use a first-order extension of the modal µ-calculus to specify Erlang programs. The µ-calculus is a logic over a program’s labeled transition system with constructs to indicate whether all continuations of a transition have a given property, or whether there exists a transition whose continuation has
3.9. RELATED WORK

some property. The calculus was designed for a general proof system rather than for model-checking specifically, so it is generally more expressive than APS, but its associated theorem prover [48] requires more human interaction than the model checker I present in chapter 7.

Summers and Müller [105] develop a logic-like reasoning system for actors based on actor services. An actor service is somewhat like a state transition in APS, in that it says a message received from the environment can trigger some output message to be sent eventually. Actor services can be chained together so that each actor satisfies some local actor service, while the composition of all of those services satisfies some global requirement. Actor services can specify only liveness requirements, however; they cannot specify when a given message should not be sent. The system also supports arbitrary predicates on communicated messages, so it is oriented more towards a manual proof system rather than automated model checking.

**Automata for Specifications** Harel’s statecharts [55] are a well-known extension of FSMs for creating modular specifications of reactive systems. Statecharts allow for grouping states into hierarchies of “superstates” and for separating orthogonal aspects of state machines, among other features. Statecharts cannot easily express the idea that a given event (such as sending a message) eventually happens, though, nor can they dynamically create new state machines.

I/O automata [75] are a more specialized kind of state machine designed for specifying distributed algorithms, with the transition labels partitioned into input, output, and internal actions. Attie and Lynch [13] extended this model to dynamic I/O automata that can synchronize with each other and spawn new automata as part of their state transitions. Kurnia [69] uses these dynamic I/O automata to specify actor systems. Specifications in Kurnia’s system are made up of two kinds of automata: actor automata that specify the behavior of an individual actor, and configuration automata that specify the behavior of groups of actors. Both the number of states and number of transitions for actor automata are completely arbitrary (they can even be infinite), so it is up to a human prover to show how an actor satisfies its specification.

SDL [102] is a protocol-specification language based on extended finite-state machines that accommodates both flow-chart and textual syntax. The language is more prescriptive than APS; specifying the behavior of a component also dictates its implementation. As a result, SDL is better suited for full-fledged protocol descriptions rather than lightweight specifications.

Specification diagrams [101] formalize and extend SDL-like diagrams with arbitrary mathematical constraints. Smith and Talcott present a refinement relation between specifications similar to my treatment of conformance, although the arbitrary constraints can lead to complex proofs. In contrast, the restricted structure of specifications in APS makes the correspondence between a program and its specification almost obvious.
History Markers  History markers were somewhat inspired by the markers used by Venet [108], Feret [45], and Garoche et al. [51] to develop abstract interpretations for communicating systems. My final design ended up quite different from theirs, however; e.g., their markers are attached to processes, whereas mine are attached to addresses. Chapter 5 further discusses these works. There is also a relationship to contours in Shivers’ k-CFA [99], in that both contours and history markers are extra annotations used to distinguish values that would be otherwise indistinguishable (during execution in my work, and during abstract interpretation in his).

State-Space Reduction  Some of the refinements to conformance defined in appendices B–G are instances of standard reduction techniques used in model checking. Single-handler conformance (appendix D) examines only those executions in which an actor does not start handling an event until all other running event handlers have finished. This is a form of partial order reduction [52], which reduces the number of states that need to be explored when multiple transitions can commute with each other without affecting the final result.

External-representative conformance (appendix C) and deterministic-handler conformance (appendix E) both assume there is some deterministic scheme for choosing the new addresses and history markers introduced during program transitions. In either case, it is sufficient to examine only the executions that use this scheme, thus treating each such execution as a representative of some equivalence class of executions. This is an instance of symmetry reduction [110], which exploits symmetry in the graph of a program’s possible transitions to avoid exploring redundant states.
Chapter 4

A Manual Conformance Proof

This chapter proves that the stream-processing program from chapter 2 conforms to its specification given in chapter 3.

**Theorem.** Let $P_{\text{stream}}$ be the stream-processing CSA program from chapter 2, and let $\Sigma_{\text{stream}}$ be the associated specification from chapter 3. Then $P_{\text{stream}} \models \Sigma_{\text{stream}}$.

Listings 4.1 and 4.2 reproduce the CSA implementations of those actors for reference, and listings 4.3 and 4.4 reproduce their APS specifications.

**Proof.** As mentioned in the previous chapter, PSM conformance is equivalent to standard conformance (as long as the given program type-checks). $P_{\text{stream}}$ type-checks, so therefore it is sufficient to show that $P_{\text{stream}} \models \text{PSM} \Sigma_{\text{stream}}$. $P_{\text{stream}}$ declares no externals, so let

\[
\text{Inst}(P_{\text{stream}}, e) = (\bar{K}_{\text{init}}, \langle a_m @ \{\eta_u \}, a_m @ \{\eta_s \}, e \rangle),
\]

with $a_m @ \{\eta_u \}$ and $a_m @ \{\eta_s \}$ as the ManagerUserAPI and ManagerSysAPI receptionists for the manager in $\bar{K}_{\text{init}}$, respectively. Let

\[
\text{SpecInst}(\Sigma_{\text{stream}}, (\eta_u, \eta_s), e) = S_{\text{init}} = \{s_{\text{init}}\}.
\]

We must show that the initial program configuration $\bar{K}_{\text{init}}$ PSM-conforms to the initial PSM $s_{\text{init}}$, i.e., $\bar{K}_{\text{init}} \models \text{PSM} s_{\text{init}}$.

The details of PSM conformance are given in appendices B–G, but for the sake of this proof, it is important only to know that PSM conformance is defined in terms of event steps rather than individual transitions. Roughly, an event step $\bar{K} \xrightarrow{\bar{i}_1, \ldots, \bar{i}_n} \bar{K}'$ represents the sequence of transitions $\bar{K} \xrightarrow{\bar{i}_1} \ldots \xrightarrow{\bar{i}_n} \bar{K}'$ the configuration takes to handle a single message or timeout; see appendix F for details. Specification configurations have a similar kind of step $S \xrightarrow{\lambda_1, \ldots, \lambda_n} S'$ that holds if and only if $S \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_n} S'$, also defined in appendix F.

83
;;; ManagerMessage =
;;; (Variant [MakeProc (Addr (Addr ProcUserAPI)) (Addr Nat)]
;;; [ShutdownAll])

;;; ManagerUserAPI =
;;; (Variant [MakeProc (Addr (Addr ProcUserAPI)) (Addr Nat)])

;;; ManagerSysAPI = (Variant [ShutdownAll])

;;; ProcUserAPI =
;;; (Variant [AddRdg Nat (Addr (Variant [Ok] [NotOk]]))]
;;; [GetMean]
;;; [Disable]
;;; [Enable])

;;; ProcAddr = (Addr (Variant [Shutdown]))

(program (receptionists [user-api ManagerUserAPI]
              [sys-api ManagerSysAPI])
      (externals)
      (let-actors
       ([manager (spawn ManagerMessage
                  (goto Managing (list))
                  (define-state (Managing [processors (List ProcAddr)]) m
                  (case m
                    [[(MakeProc resp mdest)
                      (case (< (length processors) 100)
                      [(True)
                        ([let ([(p (spawn #|see listing 2.2|#))]
                            (send resp p)
                            (goto Managing (cons p processors)))]))
                      [(False) (goto Managing processors)])]
                    [(ShutdownAll)
                      (for/fold ([dummy-result (variant Shutdown)])
                        ([p processors])
                        (send p (variant Shutdown))))
                      (goto Managing (list)))]))))
     (manager manager)))

Listing 4.1: CSA implementation of the manager actor
;; Processor's declared type
(\texttt{Variant} \texttt{[AddRdg Nat (Addr (Variant [Ok] [NotOk])]}]
 [GetMean]
 [Disable]
 [Enable]
 [Shutdown])

;; Initial state
(goto Off 0 0)

;; State definitions
(define-state (\texttt{Off} \texttt{[sum Nat] [num-\texttt{rdgs Nat}]}) m
  (case m
    [(AddRdg temp resp)
     (send resp (\texttt{variant NotOk}))
     (goto Off sum num-\texttt{rdgs})]
    [(GetMean)
     (send mdest (/ sum num-\texttt{rdgs}))
     (goto Off sum num-\texttt{rdgs})]
    [(Disable) (goto Off sum num-\texttt{rdgs})]
    [(Enable) (goto On sum num-\texttt{rdgs})]
    [(Shutdown) (goto Done)])

(define-state (\texttt{On} \texttt{[sum Nat] [num-\texttt{rdgs Nat}]}) m
  (case m
    [(AddRdg temp resp)
     (send resp (\texttt{variant Ok}))
     (goto On (+ sum temp) (+ num-\texttt{rdgs 1}))]
    [(GetMean)
     (send mdest (/ sum num-\texttt{rdgs}))
     (goto On sum num-\texttt{rdgs})]
    [(Disable) (goto Off sum num-\texttt{rdgs})]
    [(Enable) (goto On sum num-\texttt{rdgs})]
    [(Shutdown) (goto Done)])

(define-state (\texttt{Done}) m (goto Done))

Listing 4.2: CSA implementation of the processor actor
(specification
  (mon-receptionist user-api)
  (mon-externals)
  (goto Managing)
  (define-state (Managing)
    [(variant MakeProc resp mdest) -> (goto Managing)]
    [(variant MakeProc resp mdest) ->
      [obligation resp (fork-addr #|see listing 3.4|#)]
      (goto Managing)])

Listing 4.3: APS specification for the manager actor

(goto Off mdest)

(define-state (Off mdest)
  [(variant AddRdg * resp) ->
    [obligation resp (variant NotOk)]
    (goto Off mdest)]
  [(variant GetMean) -> [obligation mdest *] (goto Off mdest)]
  [(variant Disable) -> (goto Off mdest)]
  [(variant Enable) -> (goto On mdest)]
  [free -> (goto Done)])

(define-state (On mdest)
  [(variant AddRdg * resp) ->
    [obligation resp (variant Ok)]
    (goto On mdest)]
  [(variant GetMean) -> [obligation mdest *] (goto On mdest)]
  [(variant Disable) -> (goto Off mdest)]
  [(variant Enable) -> (goto On mdest)]
  [free -> (goto Done)])

(define-state (Done)
  [(variant AddRdg * resp) -> (goto Done)]
  [(variant GetMean) -> (goto Done)]
  [(variant Disable) -> (goto Done)]
  [(variant Enable) -> (goto Done)])

Listing 4.4: APS specification for the processor actor
Proving that $\langle K_{\text{init}}, s_{\text{init}} \rangle$ involves showing that $\langle K_{\text{init}}, s_{\text{init}} \rangle$ is a member of some PSM-conformance-dense relation $R$. Thus, the proof requires two main steps. First, we must define a suitable relation $R$ for this program and specification that contains $\langle K_{\text{init}}, s_{\text{init}} \rangle$. Second, we must show that $R$ is PSM-conformance-dense by showing for each pair $\langle K, s \rangle \in R$ how a configuration containing just $s$ can simulate all fair, non-stuck event-step executions from $K$.

These steps require some intuition as to why the program should conform to its specification. Whenever a PSM has multiple choices of transition steps it can take to match a step of the program, it is up to the human prover to choose which step describes the intended behavior. Similarly, it is up to the human prover to find some way to partition the infinite set of reachable program-configuration/PSM pairs into a small number of provable cases.

For this example, let the relation $R$ be defined as the set of pairs $\langle K, s \rangle$ such that

- the actors in $K$ are a single manager actor and any number of processor actors,
- the only in-flight messages in $K$ are Shutdown messages to the processor actors,
- the receptionists on $K$ are those for the manager described above as well as one receptionist $\langle a_p@\{\eta_p\}, \text{ProcUserAPI} \rangle$ per processor (where ProcUserAPI stands for the type given in listing 4.1),
- the manager actor in $K$ is in the Managing state, with its state argument processors containing the addresses of all processors that have not been sent a Shutdown message,
- $s$ is either
  - a PSM $\langle \{\eta_u\}, \{\eta'_1, \ldots, \eta'_n\}, \text{Managing} : e, \Phi_M, \emptyset \rangle$ with $\Phi_M$ corresponding to the states in listing 4.3 and where no address in $K$ is marked with any marker in $\{\eta'_1, \ldots, \eta'_n\}$, or
  - a PSM $\langle \{\eta_p\}, \{\eta_m, \eta'_1, \ldots, \eta'_n\}, \phi : \eta''_1, \ldots, \eta''_p, \Phi_P, \emptyset \rangle$ with $\Phi_P$ corresponding to the states in listing 4.4 and where some marked address $a_m@\{\eta_m\}$ is the mdest address for the processor and no address in $K$ is marked with a marker in $\{\eta''_1, \ldots, \eta''_n\}$, and
- $\langle K, s \rangle$ additionally fulfills the conditions of one of the below cases.

The first four bullets in the definition of $R$ just list some invariants of the reachable configurations for this program. The conditions on a PSM in the Managing state effectively say that the manager actor throws away response addresses for MakeProc messages once it sends the response, and that every incurred obligation is satisfied in the same event step where it is incurred; we can see that this is the case by tracing through the code. The conditions for a processor PSM similarly say that no obligations for this PSM remain after any given
event step, that the processor actor throws away AddRdg response addresses by
the end of each event step, and that the only remaining output-monitored ad-
dress is the destination for the running mean. The last bullet refers to the pos-
sible states the program can be in, and the state of the specification each one
should correspond to. These pairs come from an intuitive understanding of how
the program should correspond to the specification.

We have that \( \langle \hat{K}_{\text{init}}, s_{\text{init}} \rangle \in R \) (it corresponds to case 1 below), so it remains to
show that \( R \) is PSM-conformance-dense.

The remainder of the proof proceeds by cases on the configuration pairs \( \langle \hat{K}, s \rangle \)
in \( R \). Each case shows how, for each event step \( \hat{K} \xrightarrow{l_1, \ldots, l_n} \hat{K}' \), the PSM can
use some transition sequence \( \{s\} \xrightarrow{l_1, \ldots, l_n} S \) to simulate that event step and
reach another configuration/PSM pair in \( R \). Each case also describes how any
obligations incurred are later fulfilled. Then for any fair, non-stuck event-step
execution \( \hat{K}_1 \xrightarrow{l_1, \ldots, l_{1,m}} \ldots \), the simulating specification execution is defined by
chaining together the simulating steps for each individual event step.

There are four cases, corresponding to the four states in the specification: one
state for the manager PSM, and three for the processor PSM. The possible con-
figurations of this example program nicely categorize into these four cases, but
that is not necessarily always the case: conformance proofs for other programs
or specifications may have no correspondence between the number of proof cases
and the number of PSM states.

**Case 1: Manager in the Managing State**

**Case Conditions:** The PSM is a manager PSM (i.e., its state definitions \( \Phi_M \) cor-
respond to those in listing 4.3).

We first consider each of the event steps triggered by the manager actor re-
ceiving some message, followed by all other event steps.

**NewProcessor** The manager actor may receive a message (variant NewProcessor \( a_r@\{\eta_r\} a_d@\{\eta_d\} \)) via the receptionist \( a_m@\{\eta_u\} \).
The marker \( \eta_u \) is the PSM’s input-monitored marker, so the PSM must take one
of its matching state transitions to simulate that step.

There are two possible outcomes. If the manager’s list of processor addresses
already contains 100 items, then the processor limit has been reached, and the
manager simply drops the request and waits for the next message.

In that case, the PSM \( s \) can take its first transition within its Managing state,
resulting in a sequence of steps \( \{s\} \xrightarrow{l_1, \ldots, l_n} \{s'\} \) that generates no new obliga-
tions or forks, nor fulfills any obligations. Then \( \langle \hat{K}', s' \rangle \in R \) because it is an in-
stance of this case.

If the manager has not reached the processor limit yet, then it instead spawns
a new processor with \( a_d@\{\eta_d\} \) as its \( \text{mdest} \) address. It then sends the processor’s
address $a_P$ back to $a_r@\{\eta_r\}$, with the address getting marked with some fresh marker $\{\eta_p\}$.

To simulate this event step, $s$ can instead take the second transition within its Managing state, which incurs an obligation to send back an address corresponding to some new PSM (as described in the fork-addr pattern). That pattern immediately creates a new PSM with input-monitored marker $\eta_p$ and output-monitored markers $\{\eta_r, \eta_d\}$ (because $\eta_r$ is the target of the obligation, and $\eta_d$ is the mdest argument to the new PSM's state). The new PSM also maintains the new obligation (now transformed to a self-addr obligation by Extract), because $\eta_r$ is one of its output-monitored markers. Meanwhile, the original PSM does not change, because the state transition goes back to the same state.

The weak-step transition relation for APS (section 3.6.7) allows each PSM to simulate the purely internal steps of the event handler without otherwise changing its state. The only way to simulate sending the processor's address back to the environment, however, is by taking a transition step using the P-SEND rule (section 3.6.5). As a reminder, that rule says that a PSM can send a message only by discharging one obligation for each of its output-monitored markers on the destination address. The original PSM does not monitor the marker $\eta_r$ of the response address $a_r$, so it can take such a step without any extra conditions: outputs to an address marked with $\eta_r$ are irrelevant to this PSM. The forked PSM, however, does monitor $\eta_r$, so it discharges the obligation incurred at the beginning of this event step. Matching the marker $\eta_p$ against the pattern self-addr in that obligation sets $\eta_p$ as the input-monitored marker for that forked PSM. This matches the intent of the specification: the forked PSM should model the reactions to messages the environment sends to the spawned processor. Thus, the only obligation incurred in this event step is fulfilled.

In total, this results in a specification step sequence $\{s\}$ $\downarrow_{l_1,\ldots,l_n}$ $\{s,s'\}$, where $s'$ is the forked PSM after fulfilling the obligation. The pair $\langle \bar{K}', s \rangle$ is in $R$ because it is an instance of the current case. Because there remains no address in $\bar{K}'$ marked with $\eta_r$, $\langle \bar{K}', s' \rangle$ is an instance of the following case and therefore also a member of $R$.

**ShutdownAll** The manager actor may also receive a message ShutdownAll via the receptionist $a_m@\{\eta_s\}$). The PSM $s$ does not monitor $\eta_s$, and none of the remaining communication in this event step is on a marked address. Therefore, $s$ can step with the sequence $\{s\}$ $\downarrow_{l_1,\ldots,l_n}$ $\{s\}$, with no obligations fulfilled, and $\langle \bar{K}', s \rangle \in R$ because it is an instance of the current case.

**Other Steps** None of the remaining possible event-steps $\bar{K} \downarrow_{l_1,\ldots,l_n} \bar{K}'$ involve communication using the monitored markers of $s$. Therefore in each case we have $\{s\}$ $\downarrow_{l_1,\ldots,l_n}$ $\{s\}$, with no obligations fulfilled, and $\langle \bar{K}', s \rangle \in R$ because it is an instance of the current case.
CHAPTER 4. A MANUAL CONFORMANCE PROOF

Case 2: Processor in the Off State

Case Conditions:

- The PSM s is a processor PSM (i.e., its state definitions $\Phi_P$ correspond to those in listing 4.4),
- s is in its Off state,
- there is a receptionist $a_p @ \{ \eta_p \}$ in $\overline{K}$ such that $\eta_p$ is the input-monitored marker on s and $a_p$ refers to a processor actor in its Off state,
- the $\text{mdest}$ address for that processor actor is some marked address $a_d @ \{ \eta_d \}$, and
- and the $\text{mdest}$ argument on the PSM is $\eta_d$.

There are four message variants the environment may send to the receptionist $a_p @ \{ \eta_p \}$ associated with the PSM’s input-monitored marker $\eta_p$: AddRdg, GetMean, Enable, and Disable. Let us first consider each of these steps in turn, then consider all other event steps.

AddRdg The processor actor reacts to an AddRdg message by sending a NotOk response to the included response address $a_g @ \{ \eta_g \}$ and transitioning back to the Off state without changing its running totals. Because s monitors $\eta_p$, it has to take its only matching transition for AddRdg messages. This incurs an obligation on $\eta_p$, which the PSM can later discharge to simulate the NotOk response. There is no other communication in this step, so we have $\{s\}_{[l_1 \ldots l_n]} \{s'\}$, where $s'$ is identical to s except it has $\eta_d$ as an additional output-monitored marker. The sequence fulfills the obligation $\langle \eta_g, (\text{variant NotOk}) \rangle$. Because $\overline{K}'$ has no remaining address marked with $\eta_g$, $\langle \overline{K}', s' \rangle$ is an instance of this case and therefore a member of $R$.

GetMean If num-rdgs is 0 when the processor receives a GetMean message, the program will get stuck because of a divide-by-zero error. PSM conformance considers only steps that lead to non-stuck configurations, however, so we can ignore such a step. Otherwise, the processor actor sends the resulting average to $a_d @ \{ \eta_d \}$ and transitions back to Off. Similar to the AddRdg case, the PSM can simulate this by taking its only matching transition for GetMean and then discharging the incurred obligation on $\eta_d$ to simulate the response. The resulting pair $\langle \overline{K}', s' \rangle$ is an instance of this case and therefore a member of $R$.

Enable Upon receiving an Enable message, the processor merely transitions to the On state. The PSM can do the same with its matching transition, and the resulting pair $\langle \overline{K}', s' \rangle$ is an instance of the next case and therefore a member of $R$. 
Disable  Similar to Enable, the program transitions to the Off state (its current state), and the PSM does the same. The resulting pair \( \langle \bar{K}', s' \rangle \) is an instance of the current case and therefore a member of \( R \).

Other Steps  The only other event step on the processor at \( a_p \) is to receive a Shutdown message. The PSM matches that step with the free transition to the Done state. The reached configuration pair \( \langle \bar{K}', s' \rangle \) is an instance of case 4 and therefore a member of \( R \).

The inputs and outputs of all other event steps do not involve markers this PSM monitors. Therefore, the PSM can take no transition, as with the steps with no monitored communication in case 1. The resulting configuration pair \( \langle \bar{K}', s \rangle \) is an instance of this case and therefore a member of \( R \).

Case 3: Processor in the On State

Case Conditions:

- The PSM \( s \) is a processor PSM (i.e., its state definitions \( \bar{\Phi}_P \) correspond to those in listing 4.4),
- \( s \) is in its On state,
- there is a receptionist \( a_p@\{\eta_p\} \) in \( \bar{K} \) such that \( \eta_p \) is the input-monitored marker on \( s \) and \( a_p \) refers to a processor actor in its On state,
- the \texttt{mdest} address for that processor actor is some marked address \( a_d@\{\eta_d\} \), and
- and the \texttt{mdest} argument on the PSM is \( \eta_d \).

Similar to Case 2, except that the response to an AddRdg message is Ok rather than NotOk.

Case 4: Processor in the Done State

Case Conditions:

- The PSM \( s \) is a processor PSM (i.e., its state definitions \( \bar{\Phi}_P \) correspond to those in listing 4.4),
- \( s \) is in its Done state, and
- there is a receptionist \( a_p@\{\eta_p\} \) in \( \bar{K} \) such that \( \eta_p \) is the input-monitored marker on \( s \) and \( a_p \) refers to a processor actor in its Done state.
Because the processor actor at $a_P$ is in the Done state, whenever it receives a message it immediately transitions to Done again without causing any other effects. The PSM can match each such event-step with its corresponding transitions, or in the case of an internal Shutdown message, the PSM takes no transition. In each case, the reached PSM $s'$ differs from $s$ only by having possibly more output-monitored markers that do not appear on addresses in $\tilde{K}'$. Then $\langle \tilde{K}', s' \rangle \in R$ because it is an instance of the current case.

The inputs and outputs of all other event steps do not involve markers this PSM monitors, so the PSM can again simulate them by taking no transition. Again, $\langle \tilde{K}', s \rangle \in R$ because it is an instance of the current case. \qed
Chapter 5

Abstracting CSA

Even for a small program such as the stream-processing system, manual conformance proofs quickly become tedious, not to mention error-prone. To solve this issue, this dissertation develops a model checker that automatically verifies whether a CSA program conforms to an APS specification. Specifically, chapter 7 defines an algorithm $\text{ModelCheck}$ such that for a given program $P$ and specification $\Sigma$, whenever $\text{ModelCheck}(P, \Sigma)$ returns true, $P \models \Sigma$. The model checker is sound, but not complete: there exist some pairs $(P, \Sigma)$ such that $P \models \Sigma$, but $\text{ModelCheck}(P, \Sigma)$ returns false, meaning it was unable to prove this fact.

Model checking is a technique for verifying whether a program satisfies a specification by constructing a model of the program and exhaustively checking that all states of the model satisfy that specification [28, 92]. In the context of CSA and APS, this means checking that for every state of the model and every transition from that state, the specification has some way to match that transition.

Hence, we must find some way to represent the program’s possibly infinite state space as a finite model. To that end, this chapter develops an abstract interpretation for CSA$^1$ that removes most sources of the infinite state space, along with a corresponding notion of conformance called abstract conformance. Section 5.5 then introduces summary conformance, which makes abstract conformance more practical by providing finite representations for the infinitely many ways an abstract handler expression can reduce.

Looking ahead, chapter 6 builds on top of this step to develop one last notion of conformance, called transformation conformance, that further limits the state

---

$^1$Bear in mind that an APS specification is not an abstraction of a CSA program. While both a specification configuration and an abstract program configuration denote a set of concrete program configurations (either the conforming program configurations for the former or the concrete program configurations abstracted by the latter), they have very different purposes. An APS specification configuration describes what a program should do, and it omits internal implementation details. An abstract program configuration, by contrast, describes what a program actually does, and it includes most implementation details from the original program. The situation is similar to symbolic model checking, in which the model checker determines whether some abstraction of the given program satisfies the given specification.
space that must be explored. Chapter 7 then defines ModelCheck as an algorithm that checks whether a program $P$ transformation-conforms to a specification $\Sigma$. Theorems stated in this chapter and chapter 6 show that transformation conformance implies concrete conformance, so this approach is a sound method for checking whether $P \models \Sigma$.

5.1 Overview of the Abstract Interpretation

Abstract interpretation is a method for analyzing programs by executing them with some analysis-specific abstraction of their semantics. The result is a sound (but not necessarily precise) analysis of the program’s behavior. First developed by Cousot and Cousot [34], researchers have used abstract interpretation for a variety of applications, such as higher-order flow analysis [99], taint analysis [106], and type inference [33].

The abstract interpretation developed here approximates the communication behavior of a CSA program. The abstraction is sound: an abstract program configuration models every execution path of the concrete program configuration from which it was abstracted. Therefore, if the abstract configuration conforms to some APS specification, then so does its corresponding concrete configuration. A theorem at the end of this chapter formalizes this statement.

The purpose of the abstract interpretation is to abstract away the aspects of CSA program configurations that can lead to an infinite state space. Those are:

1. lists and dictionaries of unbounded size,
2. infinitely many primitive values,
3. an unbounded number of actor addresses,
4. an unbounded number of history markers, and
5. unbounded quantities of in-flight messages.

The first two cases can lead to an infinite state space if, for example, an actor adds a new item to a list or increments a counter every time it receives a message. The third case is an issue for any program that can repeatedly spawn new actors without limit, such as a hypothetical variation on the stream-processing example that spawns a new processor in response to every MakeProc response, regardless of the number that already exist. History markers can cause state-space issues if, say, an actor repeatedly sends its address to the environment every time it receives a message, marking that address with a distinct marker each time. Because a program configuration maintains the set of used history markers, every distinct marker leads to a new state. Finally, the quantity of in-flight messages can be a problem when, for example, an actor sends itself two messages for every one that it receives: the number of messages in-transit to that actor would grow continuously over time.

Because APS aims to describe the high-level structure of a protocol rather than its low-level computational details, the precision of abstractions for lists,
dictionaries, and primitive values are unlikely to affect conformance. Simple, coarse abstractions for these values should be acceptable. Strings and numbers are abstracted to wildcard values that represent the concept of “any string” or “any number”. Collections (i.e., lists and dictionaries) are abstracted to sets of abstract values, representing a superset of the items contained in the concrete collection.

The remaining three issues, however, require a careful design of their abstract representations to maintain the essence of the protocol’s implementation across the abstraction. For example, imagine if all of the stream-processor actors from our running example were represented as a single mega-processor actor. That abstract model would not be precise enough to distinguish between a processor’s behavior in the On or Off states, because the abstract actor would have to simultaneously model every processor’s state.

Instead, the abstract interpretation partitions actors into atomic actors and collective actors. An atomic actor is just like a concrete actor: it represents just one actor from the concrete program, and it is in exactly one state. A collective actor, on the other hand, represents zero or more actors all spawned from the same location $\ell$. Rather than just a single behavior $b$, a collective actor has a set of behaviors representing the possible states of all of the actors it abstracts.

The abstraction for history markers exploits the fact that a given specification might monitor only a subset of the markers used in a given program. So when checking whether some program configuration $K$ conforms to a PSM $s$, it is sufficient to abstract a set of markers $H$ marking an address in $K$ by just the subset monitored by $s$. Thus, an abstract configuration abstracts away some of the (irrelevant) information about where each address came from.

To bound the number of in-flight messages, the abstract interpretation of the multiset $\mu$ of messages abstracts the quantity of any given abstract message to either single (one) or many (any number, including zero). This level of abstraction allows the checker to reason precisely about the common case in which we are concerned with just one particular request moving through the system.

APS specifications add one more source of infinity with which the model checker must contend: an unbounded number of obligations for any given marker. The abstract interpretation ignores this issue; we will see how the algorithm handles it in chapter 6.

The majority of this chapter details the design of the abstractions for CSA and defines a notion of conformance for abstract program configurations. The last section defines two additional notions of conformance that further reduce the number of executions the model checker must explore to prove conformance.

### 5.2 Abstract Interpretation for Programs

Figure 5.1 defines abstract program configurations and their components. Much of the components are similar to their concrete counterparts; the discussion here focuses on the significant differences.

The first major change is in the abstract actor-behavior map $\hat{\beta}$: rather than
\[ \hat{K} := \langle \hat{\beta} \mid H \rangle \hat{\mu} \]  
\text{(Abstract Program Configurations)}

\[ \hat{\beta} \in \text{AbsAddr} \to \mathcal{F}(\text{AbsBeh}) \]  
\text{(Abstract Actor-Behavior Maps)}

\[ \hat{\mu} \in (\text{AbsMarkedAddr} \times \text{AbsVal}) \to \text{Quant} \]  
\text{(Abstract Message Maps)}

\[ \hat{\rho} \in \mathcal{F}(\text{AbsMarkedAddr} \times \text{Type}) \]  
\text{(Abstract Receptionist Sets)}

\[ \hat{b} \in \text{AbsBeh} := \langle \hat{Q}, C[\hat{e}] \rangle \mid \langle \hat{Q}, (\text{receive } x \hat{e} \hat{t}c) \rangle \]  
\text{(Abstract Behaviors)}

\[ \hat{B} \in \mathcal{F}(\text{AbsBeh}) \]  
\text{(Sets of Abstract Behaviors)}

\[ C ::= [] \mid (\text{in-loop } []) \]  
\text{(Parent Contexts)}

\[ \hat{a} \in \text{AbsAddr} ::= (\text{addr } \ell n) \mid (\text{collective-addr } \ell) \]  
\text{(Abstract Addresses)}

\[ m \in \text{Quant} ::= \text{single} \mid \text{many} \]  
\text{(Quantities)}

\[ \hat{e} ::= (\text{spawn } \ell \tau \hat{Q}) \mid (\text{goto } q \hat{e}) \]  
\text{(Abstract Expressions)}

\[ \hat{\alpha v} ::= \hat{a}@H \mid (\text{record } \hat{r} \hat{\alpha v}) \]  
\text{(Abstract Open Values)}

\[ \hat{\nu} \in \text{AbsVal} ::= \hat{a}@H \mid (\text{record } \hat{r} \hat{\nu}) \]  
\text{(Abstract Closed Values)}

\[ \hat{Q} ::= (\text{define-state } (q \hat{x} \tau) x \hat{e} \hat{t}c) \]  
\text{(Abstract State Definitions)}

\[ \hat{t}c ::= \text{no-timeout} \mid [(\text{timeout } \hat{\alpha v}) \hat{e}] \]  
\text{(Abstract Timeout Clause)}

Figure 5.1: CSA abstract domains
mapping each address to a behavior, it maps each one to a set of behaviors. A
collective actor abstracts a set of concrete actors, so its set of behaviors abstracts
all the behaviors that those concrete actors may have. An atomic actor always has
exactly one element in its behavior-set, though, representing a single concrete
actor’s behavior. A message sent to a collective actor may be handled by any of
that actor’s current behaviors.

Next, rather than storing messages in a multiset as in concrete configurations,
the abstract message map \( \hat{\mu} \) maps from an abstract marked-address/value
pair to its abstract quantity. An abstract quantity of single corresponds to exactly
one concrete message, while many indicates any number of in-flight messages.

An abstract behavior \( \hat{b} \) of the form \( \langle \hat{Q}, C[\hat{e}] \rangle \) represents an actor handling an
event in some parent context \( C \). A parent context (in-loop []) indicates that the
actor is initializing itself after its parent spawned it while executing a for/fold
loop. The empty parent context [] is used for all other situations. Section 5.2.3 de-
scribes how the abstract transition rules use this information to avoid redundant
transitions.

An abstract address \( \hat{a} \) can take two different forms. Atomic ad-
dresses (addr \( \ell \ n \)) are identical to concrete addresses. Collective addresses
(collective-addr \( \ell \)) drop the extra distinguishing identifier \( n \), because a collec-
tive address for a spawn location \( \ell \) collects the behavior of all non-atomic actors
spawned at that location.

An abstract list expression (list \( \{ \hat{ov} \} \)) abstracts a list expression whose el-
ements come from the set \( \{ \hat{ov} \} \). Similarly, an abstract dictionary expression
(dict \( \{ \hat{ov}_k \} \{ \hat{ov}_v \} \)) represents a dictionary expression with keys from \( \{ \hat{ov}_k \} \) and
values from \( \{ \hat{ov}_v \} \). The abstraction is over-approximating; each set represents
a superset of the items found in a concrete collection. Substitution is defined
slightly differently for lists and dictionaries to accommodate this set-based rep-
resentation, as follows:

\[
(\text{list } \{ \hat{ov}_1, \ldots, \hat{ov}_n \})[x \leftarrow \hat{v}] = (\text{list } \bigcup_{i \in 1 \ldots n} \{ \hat{ov}_i[x \leftarrow \hat{v}] \})
\]

\[
(\text{dict } \{ \hat{ov}_1, \ldots, \hat{ov}_n \} \{ \hat{ov}_1', \ldots, \hat{ov}_m' \})[x \leftarrow \hat{v}] =
(\text{dict } \bigcup_{i \in 1 \ldots n} \{ \hat{ov}_i[x \leftarrow \hat{v}] \} \bigcup_{j \in 1 \ldots m} \{ \hat{ov}_j[x \leftarrow \hat{v}] \})
\]

The last new forms are the ones for natural numbers and strings. The expres-
sion abs-nat abstracts any natural number, and abs-string abstracts any string.

Appendix H lists the rules of the type system for abstract CSA. The rules are
largely similar to their concrete analogs.

### 5.2.1 The Abstraction Function

This section defines the abstraction function \( |\cdot| \) from concrete CSA configurations
into abstract ones. Abstraction for a configuration is defined as the component-
Abstracting CSA

The abstraction function leaves every internal actor in the configuration atomic, so there is no need for the abstraction function to modify the set of receptionists $\hat{\beta}$. The next chapter will describe the conditions under which atomic actors are converted into collective ones. Abstraction for the actor-behavior map $\beta$ is defined inductively:

$$|\hat{\beta}| = \begin{cases} \emptyset & \text{if } \hat{\beta} = \emptyset \\ |\hat{\beta}'[a \rightarrow \{\hat{b}\}]| & \text{if } \hat{\beta} = \hat{\beta}'[a \rightarrow \hat{b}] \end{cases}$$

For an abstract message map $\hat{\mu}$, soundness requires that $\text{dom}(\hat{\mu})$ contain no more than one abstract representation $\langle \hat{a}@H, \hat{v} \rangle$ of a given concrete message $\langle a@H', v \rangle$. As a result, the abstraction of a concrete message multiset $\hat{\mu}$ merges all "similar" messages into an abstract value representing their upper bound.

Figure 5.2 defines the operation $\text{Merge}$ to merge similar abstract values together. In the figure, the symbol $\hat{V}$ stands for a set of abstract values. Effectively, the operation takes the union of elements contained in lists and dictionaries, but otherwise requires all components of merged values to be identical. The operation is a partial function, so the merge is undefined whenever none of the listed cases apply.
the values in the sequence and associative. As a result, we write $\text{Merge}$ the quantity single inductively:

$$\hat{\mu} \oplus \langle \hat{\alpha} @ H, \hat{\nu} \rangle = \hat{\mu}'' \left\{ \langle \hat{\alpha} @ H, \text{Merge}(\hat{\nu}, \hat{\nu}_1', ..., \hat{\nu}_n') \rangle \mapsto m \right\}$$

where $\hat{\mu} = \hat{\mu}' \oplus \hat{\mu}''$ and $\forall \langle \hat{\alpha} @ H', \hat{\nu}' \rangle \in \text{dom}(\hat{\mu}')$, $\hat{\alpha} @ H = \hat{\alpha}' @ H'$ and $\text{Merge}(\hat{\nu}, \hat{\nu}')$ is defined and $\forall \langle \hat{\alpha} @ H', \hat{\nu}' \rangle \in \text{dom}(\hat{\mu}'')$, $\hat{\alpha} @ H \neq \hat{\alpha}' @ H'$ or $\text{Merge}(\hat{\nu}, \hat{\nu}')$ is undefined and $\{\hat{\nu}_1', ..., \hat{\nu}_n'\} = \{\hat{\nu}' \mid \langle \hat{\alpha} @ H, \hat{\nu}' \rangle \in \text{dom}(\hat{\mu}')\}$ and $m = \text{single}$ if $n = 0$, $\text{many}$ otherwise

Figure 5.3: Abstract message-addition operation

From the definition in figure 5.2, it is easy to see that $\text{Merge}$ is commutative and associative. As a result, we write $\text{Merge}(\hat{\nu}_1, ..., \hat{\nu}_n)$ for the merging of all of the values in the sequence $\hat{\nu}_1, ..., \hat{\nu}_n$.

Figure 5.3 then defines an operation $\oplus$ for adding an abstract message $\langle \hat{\alpha} @ H, \hat{\nu} \rangle$ to an abstract message map $\hat{\mu}$ by merging that message with all similar messages in $\hat{\mu}$. The given message can be merged with another existing message $\langle \hat{\alpha}' @ H', \hat{\nu}' \rangle \in \text{dom}(\hat{\mu})$ if

- their marked destination addresses are the same (i.e., $\hat{\alpha} @ H = \hat{\alpha}' @ H'$), and
- the merge of the two values (i.e., $\text{Merge}(\hat{\nu}, \hat{\nu}')$) is defined.

If $\hat{\mu}$ contains any such similar messages, the abstract quantity for the new merged value is set to $\text{many}$; otherwise $\hat{\nu}$ is added to the map on its own with the quantity $\text{single}$. Any messages in $\hat{\mu}$ not able to merge with the given message are left in the map unchanged.

Given that operation, abstraction for concrete message multisets is defined inductively:

$$|\hat{\mu}| = \begin{cases} \emptyset & \text{if } \hat{\mu} = \emptyset \\ |\hat{\mu}'| \oplus |\langle \alpha @ H, \hat{\nu} \rangle| & \text{if } \hat{\mu} = \hat{\mu}' \cup \{\langle \alpha @ H, \hat{\nu} \rangle\} \end{cases}$$

The Quasi-Commutativity theorem in appendix I proves that the order in which the messages are added doesn’t matter (i.e., $\hat{\mu} \oplus \langle \alpha @ H, \hat{\nu} \rangle \oplus \langle \alpha' @ H', \hat{\nu}' \rangle = \hat{\mu} \oplus \langle \alpha' @ H', \hat{\nu}' \rangle \oplus \langle \alpha @ H, \hat{\nu} \rangle$), so $|\hat{\mu}|$ is well-defined.

Abstraction for behaviors $\hat{b}$, state definitions $\hat{Q}$, timeout clauses $\hat{t}c$, evaluation contexts $\hat{E}$, and transition labels $\hat{l}$ are the obvious component-wise abstractions, so those definitions are omitted for brevity.

Figure 5.4 provides the definitions for the notable expression cases; other cases are the obvious component-wise abstractions. All internal addresses are left atomic, which matches the abstraction of the actor map $\hat{\beta}$. $\text{External}$ addresses, on the other hand, are converted into collective addresses, because the markers on external addresses provide the only distinction between them needed for conformance proofs. The abstractions for lists and dictionaries take the union of the abstractions of their elements.

5.2.2 Loss of Precision

When abstracting a program, some precise information about that program’s behavior is lost. This means that even if a concrete program conforms to its spec-
\[ |e| = \]

Case \( e = a@H \) where \( a = (\text{addr } \ell \ n) \):
- \( a@H \) if \( a \) is internal, else \((\text{collective-addr } \ell)@H\)

Case \( e = (\text{list } \tilde{o}v_1 \ldots \tilde{o}v_n) \):
- \((\text{list } \bigcup_{i \in 1 \ldots n} \{|\tilde{o}v_i|\})\)

Case \( e = (\text{dict } [\tilde{o}v_1 \tilde{o}v'_1] \ldots [\tilde{o}v_n \tilde{o}v'_n]) \):
- \((\text{dict } \bigcup_{i \in 1 \ldots n} \{|\tilde{o}v_i|\} \bigcup_{i \in 1 \ldots n} \{|\tilde{o}v'_i|\})\)

Case \( e = n \):
- abs-nat

Case \( e = \text{str} \):
- abs-string

(other cases use obvious component-wise abstraction; omitted for brevity)

Figure 5.4: Abstraction of expressions

Abstraction, the abstraction of that program might not. For example, consider this segment of an example specification from chapter 3, which specifies that every received Ping message should result in a Pong sent back to the address contained in the Ping message.

\[
(\text{define-state} \ (\text{Running}))
\]

\[
[(\text{variant} \ Ping \ d) ->
(\text{obligation} \ d \ (\text{variant} \ Pong))
(\text{goto} \ \text{Running})]]
\]

One could implement such a specification with an actor that starts in the following state:

\[
(\text{define-state} \ (\text{Running}) \ m)
\]

\[
(\text{case} \ m
[(Ping \ d)
(\text{case} \ (= 1 \ 2)
[(True) (send \ d \ (\text{variant} \ BadResponse))
[(False) (send \ d \ (\text{variant} \ Pong))]])])
\]

Such an actor conforms to the given specification, because the False branch always executes, so the actor always sends the correct response. However, abstracting that actor would turn the expression \( (= 1 \ 2) \) into the abstract expression \( (= \ \text{abs-nat} \ \text{abs-nat}) \), which non-deterministically evaluates to True or False. Upon receiving a Ping, the abstract actor would non-deterministically choose one of its branches to execute, possibly sending a BadResponse instead of a Pong. So although the concrete program conforms to its specification, its abstraction does not, because it over-approximates the program’s concrete behavior.
The abstract interpretation is designed with this loss of precision in mind. For instance, the abstract interpretation does not abstract away the distinction between different variant values, such as \(\text{variant Enable}\) and \(\text{variant Disable}\), nor between records with different fields, such as \(\text{record [sum 5]}\) and \(\text{record [id 5]}\). The specific variants and records found in messages tend to drive the “shape” of a program’s communication with its environment, so it is critical that the abstract interpretation maintain precise representations of their tags and field names. (This is also why APS has patterns to distinguish those structures.)

### 5.2.3 Abstract Transition Relation

This section defines a transition relation for abstract program configurations. The relation is sound with respect to the concrete transition relation: whenever a concrete program configuration can take a step, any abstraction of that configuration can take a similar step. Section 5.2.7 formalizes the related soundness lemma.

Figure 5.5 defines the abstract evaluation contexts and transition labels needed for the transition relation. Each of these are the obvious abstract versions of their concrete counterparts.

Figures 5.6, 5.7, and 5.8 lists the abstract transition rules. The rules are similar to the concrete marked transition rules from chapter 3, with differences described below.

The transition rules use a special update operation \(\oplus\) on abstract actor-behavior maps \(\hat{\beta}\) to account for the differences between atomic and collective actors, defined as follows:

\[
\hat{\beta} \oplus [\hat{a} \mapsto \hat{b}] = \begin{cases} 
\hat{\beta} [\hat{a} \mapsto \{\hat{b}\}] & \text{if } \hat{a} \text{ is atomic} \\
\hat{\beta} [\hat{a} \mapsto \hat{\beta}(\hat{a}) \cup \{\hat{b}\}] & \text{otherwise}
\end{cases}
\]

Updating the actor at address \(\hat{a}\) with a behavior \(\hat{b}\) replaces the existing behavior if \(\hat{a}\) identifies an atomic actor, but merely adds the new behavior to the existing
Figure 5.6: CSA abstract transition rules for handling received messages and timeouts

\[
\begin{align*}
\dot{\psi} & = \langle [\varnothing] (\text{receive } x, (\text{timeout } a, H)) \rangle \\
& \xrightarrow{\text{timeout}} \langle [\varnothing] (\text{receive } x, (\text{receive } x, H)) \rangle
\end{align*}
\]
\[ \text{A-SendInternal} \]
\[
\begin{array}{c}
\langle \hat{\beta}[\hat{a} \rightarrow \hat{B} \downarrow \{ \langle \hat{Q}, C \mid (\text{send } \hat{a}' @ \hat{v}) \rangle \} \mid \mu] \mid H]\rangle^\hat{\beta} \\
\xrightarrow{\hat{a} : \text{send-internal}(\hat{a}' @ H, \hat{v})} \\
\langle \hat{\beta}[\hat{a} \rightarrow \hat{B} \cup \{ \hat{b} \} \mid \mu \oplus (\hat{a}' @ H', \hat{v})] \mid H'' \rangle^\hat{\beta}
\end{array}
\]
if \( \hat{a}' \) is internal
where \( \langle \hat{v}', H'' \rangle \in \text{Markings}(\hat{v}, H') \) if \( H \neq \emptyset \) and \( \hat{E} \neq \hat{E}' \) \([\text{for/fold } [x \leftarrow \hat{E}'''] [x' \leftarrow \hat{v}'''] \hat{v}'] \) and \( C = [], \) else \( \langle \hat{v}', H'' \rangle = \langle \hat{v}, H' \rangle \) and \( \hat{b} = \langle \hat{Q}, C \mid \hat{E} \mid \text{(variant Unit)} \rangle \)

\[ \text{A-SendExternal} \]
\[
\begin{array}{c}
\langle \hat{\beta}[\hat{a} \rightarrow \hat{B} \downarrow \{ \langle \hat{Q}, C \mid (\text{send } \hat{a}' @ \hat{v}) \rangle \} \mid \mu] \mid H]\rangle^\hat{\beta} \\
\xrightarrow{\hat{a} : \text{send-external}(\hat{a}' @ H, \hat{v})} \\
\langle \hat{\beta}[\hat{a} \rightarrow \hat{B} \cup \{ \hat{b} \} \mid \mu] \mid H'' \rangle^\hat{\beta} \oplus \hat{\rho}'
\end{array}
\]
if \( \hat{a}' \) is external, \( \text{ActorType}(\hat{a}') = \tau \) and \( \text{IntAddrTypes}(\hat{v}', \tau) = \hat{\rho}' \)
where \( \langle \hat{v}', H'' \rangle \in \text{Markings}(\hat{v}, H') \) if \( \hat{E} \neq \hat{E}' \) \([\text{for/fold } [x \leftarrow \hat{E}'''] [x' \leftarrow \hat{v}'''] \hat{v}'] \) and \( C = [], \) else \( \langle \hat{v}', H'' \rangle = \langle \hat{v}, H' \rangle \) and \( \hat{b} = \langle \hat{Q}, C \mid \hat{E} \mid \text{(variant Unit)} \rangle \)

Figure 5.7: CSA abstract transition rules for sending messages
Figure 5.8: CSA abstract transition rules for spawns, state transitions, and functional reduction:

\[ \llangle H \mid \{ [\beta \land \gamma] : \theta \} \cap g \rightarrow g \rrangle \]

\[ \llangle H \mid \{ [\beta \land \gamma] : \theta \} \cap g \rightarrow g \rrangle \]
set if \( \hat{a} \) identifies a collective actor instead.

Up to this point, the intuition has been that the behaviors in a collective actor abstract the behaviors of some set of concrete actors. That implies each abstract behavior \( \hat{b} \) may correspond to a set of concrete behaviors \( \{b_1, \ldots, b_n\} \). If the transition rules were to follow this intuition naively, however, every time a collective actor took a step, the new behavior would not replace the old one, but rather be added to the actor’s set of behaviors. This is because that step might abstract a step in which many concrete actors in the initial configuration have the same behavior, but only one of them takes a step. With such an approach, every collective actor would amass a set of behaviors representing every intermediate step of every event handler it executed.

Instead, the abstract semantics says that if a behavior \( \hat{b} \) is handling an event, then it represents exactly one concrete actor. Appendix I proves that this approach is sound with respect to the single-handler transition relation, which executes only a single event handler at a time.

The definition of the transition relation in figures 5.6, 5.7, and 5.8 implements this approach. The rules that start a new event handler (A-RECEIVE INTERNAL, A-RECEIVE EXTERNAL, and A-TIMEOUT) all add the new behavior to the full existing set of behaviors, so that the previous behavior is still present if the actor is a collective actor. The remaining rules, however, use disjoint union (\( \sqcup \)) to distinguish the behavior performing the transition from all other behaviors \( \hat{B} \), and they replace that behavior by adding it just to \( \hat{B} \).

The abstract semantics also requires a method to remove a message from the abstract message map \( \hat{\mu} \) when the message is received in A-RECEIVE INTERNAL. This is accomplished via an abstract removal operation (\( \ominus \)), defined as follows:

\[
\hat{\mu} \ominus (\hat{a}@H, \hat{v}) = \begin{cases} 
\hat{\mu}' & \text{if } \hat{\mu} = \hat{\mu}'[(\hat{a}@H, \hat{v}) \rightarrow \text{single}] \\
\hat{\mu} & \text{otherwise}
\end{cases}
\]

Abstractly removing a single message actually removes it from the abstract message map. Abstractly removing a many message has no effect, however, because many indicates that the map may have any number of copies of that message, and removing a single message would not change that abstract quantity.

Figure 5.9 defines the abstract version of the Markings function. When marking values, the abstract interpretation does not mark addresses inside lists and dictionaries. The purpose of marking addresses is to distinguish different instances of addresses, but because each abstract value in a list or dict expression stands for any number of concrete values, marking the addresses in such an abstract value would not accomplish that purpose. This means that PSMs cannot monitor addresses found in abstract lists and dictionaries, but APS patterns are unable to bind addresses inside lists and dictionaries anyway, so the problem is moot. The function is otherwise analogous to the concrete version from chapter 3.

As with the concrete semantics in chapter 2, the abstract semantics does not model a clock, so the A-TIMEOUT rule allows an actor to evaluate its timeout handler at any time. If the concrete semantics did model a clock, then reasoning precisely about timeouts would necessitate a different timeout abstraction.
Markings(\hat{v}, H) = 
\begin{align*}
\text{Case } \hat{v} &= a \oplus H': \\
\{a \oplus H' \cup \eta \mid \eta \notin H \text{ and } \eta > \eta' \text{ for all } \eta' \in H'\} \\
\text{Case } \hat{v} &= n: \\
\{n\} \\
\text{Case } \hat{v} &= str: \\
\{str\} \\
\text{Case } \hat{v} &= (\text{variant } t \hat{v}_1 \ldots \hat{v}_m): \\
\{\langle \text{variant } t \hat{v}_1' \ldots \hat{v}_m', H_{n+1} \rangle \mid \langle \hat{v}_i', H_{i+1} \rangle \in \text{Markings}(\hat{v}_i, H_i) \} \\
\text{Case } \hat{v} &= (\text{record } [r_1 \hat{v}_1]_{i=1}^{\ldots n}): \\
\{\langle \text{record } [r_1 \hat{v}_i']_{i=1}^{\ldots n}, H_{n+1} \rangle \mid \langle \hat{v}_i', H_{i+1} \rangle \in \text{Markings}(\hat{v}_i, H_i) \} \\
\text{Case } \hat{v} &= (\text{fold } \tau \hat{v}): \\
\{\langle \text{fold } \tau \hat{v}', H' \rangle \mid \langle \hat{v}', H' \rangle \in \text{Markings}(\hat{v}, H)\} \\
\text{Case } \hat{v} &= (\text{list } \hat{v}_1 \ldots \hat{v}_n): \\
\{\langle \text{list } \{\hat{v}_1, \ldots, \hat{v}_n\}, H \rangle\} \\
\text{Case } \hat{v} &= (\text{dict } \{\hat{v}_1, \ldots, \hat{v}_n\} \{\hat{v}_1', \ldots, \hat{v}_m'\}): \\
\{\langle \text{dict } \{\hat{v}_1, \ldots, \hat{v}_n\} \{\hat{v}_1', \ldots, \hat{v}_m'\}, H \rangle\}
\end{align*}

Figure 5.9: The abstract Markings function

The A-SENDINTERNAL and A-SENDEXTERNAL rules mark sent messages only when they are sent outside the context of a for/fold loop or an in-loop context. The abstract interpretation must be able to evaluate all abstract executions of the loop in finitely many steps (including the initialization of any actors spawned during the loop), so this constraint prevents fresh marker allocations from leading to infinitely many possible abstract executions. The trade-off is that without markers on the sent addresses, our model checker will not be able to verify that addresses sent to the environment during a loop match self-addr or delayed-fork-addr output patterns.

In A-SPAWN, the new actor is spawned as a collective actor in two cases: if the spawning actor is itself collective, or if the spawn expression is in the context of a for/fold loop. Spawning a collective actor loses precision compared to an atomic actor, but precision is typically not important in these cases, and spawning a collective actor here helps reduce the state space.

The new actor’s initialization expression \( \hat{e} \) is evaluated inside an in-loop context if the spawning actor is currently evaluating a for/fold loop or is itself executing inside an in-loop context. This prevents the initialization expression from allocating fresh markers, as discussed above.

As with concrete configurations, an abstract program configuration \( \hat{K} \) is stuck
5.2. ABSTRACT INTERPRETATION FOR PROGRAMS

\[(\text{begin } \hat{v}) \mapsto \hat{v}\]
\[(\text{begin } \hat{v} \in \hat{e}) \mapsto (\text{begin } \hat{e})\]
\[(: (\text{record } [r' \hat{v}] [r \hat{v}] [r'' \hat{v}'']) r) \mapsto (\text{begin } \hat{e})\]
\[(\text{case } (\text{variant } t \hat{v})_\ldots[(t \hat{v})_\ldots]_\ldots) \mapsto \hat{e}[x \leftarrow \hat{v}]\]
\[(\text{unfold } \tau (\text{fold } t' \hat{v})) \mapsto \hat{v}\]
\[(\text{for/fold }[x \hat{v}][x' \{x \in 1 \ldots n\}]_\ldots) \mapsto \hat{v}\]
\[(\text{for/fold }[x \hat{v}][x' \{x \in 1 \ldots n\}]_\ldots) \mapsto (\text{for/fold }[x \hat{v}'][x' \{x \in 1 \ldots n\}]_\ldots)\]
\[\text{where } \hat{v}' = \hat{e}[x \leftarrow \hat{v}][x' \leftarrow \hat{v}'] \text{ for some } j \in 1 \ldots n\]

Figure 5.10: Abstract functional reduction rules

if it contains some actor with address \(\hat{a}\) that is handling an event but there is no transition \(\hat{l}\) in which \(\hat{a}\) is the active actor’s address (that is, in which the actor at \(\hat{a}\) is the one that would perform that step).

Functional Reduction Rules

Figure 5.10 updates the functional reduction rules for the abstract semantics, used in the A-FUNC rule in figure 5.8. Evaluation for primitives calls out to a new function, \(\text{EvalAbsPrimop}\), which computes the possible abstract results for a given primitive operation. Its full definition is omitted, but it interprets collection-related operations such as \texttt{cons} and \texttt{list-ref} to match the coarse abstraction on collections, so operations that add items just add the abstract value to the set, selection operations select an arbitrary element from the abstract collection’s set, and removal operations are no-ops. Arithmetic and string operations simply return \texttt{abs-nat} or \texttt{abs-string} as appropriate. Some of these operations have multiple possible results, so the reduction rule chooses one nondeterministically.

The other difference in the functional rules is the evaluation of \texttt{for/fold} loops. Because a \texttt{list} expression represents a list that may or may not contain each of the listed values, in any order or quantity, the loop can nondeterministically either end iteration or iterate with an arbitrary abstract value from the list. The other reduction rules are standard.

5.2.4 Fairness

Abstract executions come with a notion of fairness similar to the one for concrete executions, with two exceptions. First, a collective actor awaiting an event may represent a set of zero concrete actors, so fairness does not require it to eventually take a step. Second, only single abstract messages must eventually be received, because an abstract message with many copies may represent zero concrete instances of that message. The quantity of an abstract message may change from
single to many as the configuration evolves, in which case the message does not have to be received.

The below definition formalizes what it means for an abstract actor to have work to do.

**Definition.** An actor at address \( \hat{a} \) is necessarily enabled in an abstract program configuration \( \hat{K} = \langle \langle \hat{\beta} \mid \hat{\mu} \mid H \rangle \rangle^\hat{\rho} \) if and only if there exists \( \hat{l} \) such that

- an abstract step labeled with \( \hat{l} \) is enabled in \( \hat{K} \), where \( \hat{a} \) identifies the active actor for \( \hat{l} \),
- \( \hat{l} \) is not a rcv-ext label,
- if \( \hat{a} \) is collective, then \( \hat{l} \) is not a rcv-int or timeout label, and
- if \( \hat{l} = \hat{a} : \text{rcv-int}(H', \hat{v}) \) for some \( \hat{v} \) and \( H' \), then \( \hat{\mu}(\hat{a}@H', \hat{v}) = \text{single} \).

That definition then enables a notion of fairness for abstract executions.

**Definition.** An abstract execution \( \hat{K}_1 \xrightarrow{\hat{l}_1} \hat{K}_2 \xrightarrow{\hat{l}_2} \ldots \) is fair if and only if the following conditions hold:

1. For all configurations \( \hat{K}_i \) in the execution, if an actor at \( \hat{a} \) is necessarily enabled in \( \hat{K}_i \), then there exists some \( j \geq i \) such that either the actor at \( \hat{a} \) in \( \hat{K}_j \) is not necessarily enabled, or \( \hat{K}_j \xrightarrow{\hat{l}_j} \hat{K}_{j+1} \) is a step in the execution and \( \hat{a} \) is the active actor for \( \hat{l}_j \).

2. For all configurations \( \hat{K}_i = \langle \langle \hat{\beta} \mid \hat{\mu} \mid H \rangle \rangle^\hat{\rho} \) in the execution, and for all \( \hat{a}, H', \) and \( \hat{v} \) such that \( \hat{\mu}(\hat{a}@H', \hat{v}) = \text{single} \), there exists some \( j \geq i \) such that either

   - \( \hat{K}_j = \langle \langle \hat{\beta}' \mid \hat{\mu}' \mid H'' \rangle \rangle^\hat{\rho} \) is a configuration in the execution such that either \( \langle \hat{a}@H', \hat{v} \rangle \notin \text{dom}(\hat{\mu}') \) or \( \hat{\mu}(\hat{a}@H', \hat{v}) = \text{many} \), or
   - \( \hat{\tilde{K}}_j \xrightarrow{\hat{a} : \text{rcv-int}(H', \hat{v})} \hat{K}_{j+1} \) is a step in the execution.

This is a conservative approximation of fairness: every fair concrete execution can be abstracted by a fair abstract execution, but a fair abstract execution might represent some unfair concrete executions, as well. Section 5.2.7 states this formally as a lemma after defining an approximation relation for abstract configurations.

### 5.2.5 Comparing Abstract Configurations

An abstract program configuration \( \hat{K} \) represents a set of concrete program configurations. That set is defined in terms of a partial order \( \sqsubseteq \) that defines what it means for one configuration \( \hat{K} \) to approximate another configuration \( \hat{K}' \). Then
an abstract configuration \( \hat{K} \) represents a concrete configuration \( \hat{\bar{K}} \) if and only if \( \hat{K} \) approximates \( \hat{\bar{K}} \)'s abstraction, i.e., \( |\hat{K}| \preceq \hat{\bar{K}} \).

The definition of \( \preceq \) is largely structural, but comparisons for some subcomponents of the configurations require information about each configuration as a whole. For example, suppose \( \hat{K} \) has actors at the addresses \( (\text{addr } \ell 1) \) and \( (\text{addr } \ell 2) \), while \( \hat{K}' \) has actors at the addresses \( (\text{addr } \ell 1) \) and \( (\text{collective-addr } \ell) \). In other words, \( \hat{K}' \) has assimilated the latter atomic actor into a collective actor. Then when each configuration tries to spawn a new actor for location \( \ell \), \( \hat{K} \) will assign its next fresh address \( (\text{addr } \ell 3) \) to that actor, while \( \hat{K}' \) will assign its next fresh address \( (\text{addr } \ell 2) \) to that actor, instead.

Although these two addresses are distinct, all resulting occurrences of \( (\text{addr } \ell 3) \) in the first configuration correspond to the occurrences of \( (\text{addr } \ell 2) \) in the second configuration. Thus to determine whether an address \( \hat{a} \) in one configuration corresponds to an address \( \hat{a}' \) in another, the relation \( \preceq \) is indexed over a partial function \( A : \text{AbsAddr} \rightarrow \text{AbsAddr} \) that defines the correspondence between the addresses in each configuration. The relation is also indexed over a second partial function \( M : \text{HistMark} \rightarrow \text{HistMark} \) that provides a similar correspondence for history markers.

In certain contexts, the first index is irrelevant, and in other contexts neither index is relevant. Thus, \( \hat{K} \preceq_M \hat{K}' \) if and only if there exists \( A \) such that \( \hat{K} \preceq_{A,M} \hat{K}' \), and \( \hat{K} \preceq \hat{K}' \) if and only if there exists \( M \) such that \( \hat{K} \preceq_M \hat{K}' \).

**Definition of the Approximation Relation**

The rule in figure 5.11 defines \( \preceq_{A,M} \) for abstract program configurations. The “more abstract” configuration is on the right-hand side, so one can read \( \hat{K} \preceq_{A,M} \hat{K}' \) as “\( \hat{K} \) is approximated by \( \hat{K}' \)”, or equivalently, “\( \hat{K}' \) approximates \( \hat{K} \)”.

The rule requires that \( M \) be a one-to-one function so that each marker in the first configuration have at most one correspondent in the second. The rest of the comparison is defined component-wise.

As defined in the rule in figure 5.12, an actor-behavior map \( \hat{\beta}' \) approximates another map \( \hat{\beta} \) if

- every actor in \( \hat{\beta} \) has some approximation in \( \hat{\beta}' \),
- every atomic actor in \( \hat{\beta}' \) has a unique correspondent in \( \hat{\beta} \), and
- every actor handling an event in \( \hat{\beta}' \) has a corresponding actor similarly handling an event in \( \hat{\beta} \).
∀ \alpha \in \text{dom}(\hat{\beta}). \exists \alpha'. \hat{\alpha} \subseteq_A \hat{\alpha}' \text{ and } \hat{\beta}(\hat{\alpha}) \subseteq_{A,M} \hat{\beta}'(\hat{\alpha}')

∀ \alpha' \in \text{dom}(\hat{\beta}). \text{ if }\hat{\alpha}' \text{ is atomic, then } \exists \hat{\alpha}' \in \text{dom}(\hat{\beta}). \hat{\alpha} \subseteq_A \hat{\alpha}'

∀ \alpha', \hat{b}' \in \hat{\beta}(\hat{\alpha}'). \text{if } \hat{b}' \text{ is handling an event, then } \exists \hat{\alpha}, \hat{b} \in \hat{\beta}(\hat{\alpha}). \hat{\alpha} \subseteq_A \hat{\alpha}' \text{ and } \hat{b} \subseteq_{A,M} \hat{b}'

\hat{\beta} \subseteq_{A,M} \hat{\beta}'

Figure 5.12: Approximation rule for actor-behavior maps

∀ (\hat{a}@H, \hat{v}) \in \text{dom}(\hat{\mu}). \exists (\hat{a}'@H', \hat{v}') \in \text{dom}(\hat{\mu}'). \langle \hat{\mu}, \hat{a}@H, \hat{v} \rangle \subseteq_{A,M} \langle \hat{\mu}', \hat{a}'@H', \hat{v}' \rangle

∀ (\hat{a}'@H', \hat{v}') \text{ such that } \hat{\mu}'(\hat{a}'@H', \hat{v}') = \text{single}. \exists (\hat{a}@H, \hat{v}) \in \text{dom}(\hat{\mu}). \langle \hat{\mu}, \hat{a}@H, \hat{v} \rangle \subseteq_{A,M} \langle \hat{\mu}', \hat{a}@H', \hat{v}' \rangle

∀ \hat{a}, H, \hat{v}, \hat{v}'. \text{if } (\hat{a}@H, \hat{v}) \in \text{dom}(\hat{\mu}) \text{ and } (\hat{a}@H, \hat{v}') \in \text{dom}(\hat{\mu}'), \text{then } \hat{\mu} = \hat{\mu}' \text{ or } \text{Merge}(\hat{\mu}, \hat{\mu}') \text{ is undefined}

\hat{\mu} \subseteq_{A,M} \hat{\mu}'

Figure 5.13: Approximation rule for message maps

The third condition accounts for the fact that, like an atomic actor, a collective actor handling a message or timeout represents exactly one actor (see section 5.2.3).

For an abstract message map \hat{\mu}' to approximate another map \hat{\mu} (as defined in figure 5.13), every message in \hat{\mu} must have exactly one approximating message in \hat{\mu}', with an approximation of its quantity. The comparison \langle \hat{\mu}, \hat{a}@H, \hat{v} \rangle \subseteq_{A,M} \langle \hat{\mu}', \hat{a}@H', \hat{v}' \rangle \text{ is a shorthand for comparing the addresses, values, and quantities of those two messages in those two message maps, as defined in figure 5.14. Furthermore, every single message in \hat{\mu}' must correspond to exactly one message in \hat{\mu}, because that quantity represents exactly one concrete message. Finally, \hat{\mu}' may not contain two distinct messages that can be merged.}

Figure 5.15 defines the rules for abstract quantities. The quantity many approximates single, because many can stand for any quantity (including one). Any abstract quantity approximates itself.

An approximation \hat{\rho}' of a set of receptionists \hat{\rho} must have the same type for every corresponding marked address in \hat{\rho}, as defined in figure 5.16.

Approximation for sets of behaviors, defined in figure 5.17, requires that the approximating set have an approximating behavior \hat{b}' for every behavior \hat{b} in the approximated set, and that every actor in \hat{\rho}' has at most one behavior handling an event. As with the comparison for actor-behavior maps above, the latter con-

∀ \alpha \in \text{dom}(\hat{\rho}). \exists \alpha'. \hat{\alpha} \subseteq_A \hat{\alpha}' \text{ and } \hat{\rho}(\hat{\alpha}) \subseteq_{A,M} \hat{\rho}'(\hat{\alpha}')

∀ \alpha' \in \text{dom}(\hat{\rho}). \text{ if }\hat{\alpha}' \text{ is atomic, then } \exists \hat{\alpha}' \in \text{dom}(\hat{\rho}). \hat{\alpha} \subseteq_A \hat{\alpha}'

∀ \alpha', \hat{b}' \in \hat{\rho}(\hat{\alpha}'). \text{if } \hat{b}' \text{ is handling an event, then } \exists \hat{\alpha}, \hat{b} \in \hat{\rho}(\hat{\alpha}). \hat{\alpha} \subseteq_A \hat{\alpha}' \text{ and } \hat{b} \subseteq_{A,M} \hat{b}'

\hat{\rho} \subseteq_{A,M} \hat{\rho}'

Figure 5.14: Approximation rule for messages
5.2. ABSTRACT INTERPRETATION FOR PROGRAMS

| single ⊑ single | single ⊑ many | many ⊑ many |

Figure 5.15: Approximation rules for abstract quantities

\[
\forall \langle \hat{a}@H, \tau \rangle \in \hat{\rho}. \exists \langle \hat{a'}@H', \tau \rangle \in \hat{\rho}'. \hat{a}@H \sqsubseteq_{A,M} \hat{a'}@H' \\
\hat{\rho} \sqsubseteq_{A,M} \hat{\rho}'
\]

Figure 5.16: Approximation rule for receptionists

dition accounts for the fact that a collective actor handling a message or timeout represents exactly one actor.

The partial orders for behaviors, state definitions, and timeout clauses are the obvious component-wise comparisons, defined in figure 5.18.

Because concrete program configurations do not have the concept of parent contexts, the abstraction of a concrete behavior never has an \textbf{in-loop} context. Such contexts may arise during evaluation though, so soundness requires that an \textbf{in-loop} context approximate an empty one. Figure 5.19 formalizes these rules.

The comparisons for evaluation contexts, transition labels, and all non-value expressions (other than lists and dictionaries, shown above) are the obvious component-wise definitions; those rules are omitted for brevity. The address-correspondence function \(A\) used for labels is often irrelevant and therefore omitted, so \(\hat{l} \sqsubseteq_{M} \hat{l}'\) if and only if there exists \(A\) such that \(\hat{l} \sqsubseteq_{A,M} \hat{l}'\).

Figure 5.20 gives the approximation rules for values. The significant rules are those for lists, dictionaries, and marked addresses. A list approximates another list if the former contains an approximating expression for every item in the latter, and the free variables occurring in the approximating list are a subset of those in the approximated one. The latter condition ensures that the first list is a value if and only if the second list is, as well (see the Value Preservation lemma in appendix I). The rules for dictionaries are similar to those for lists, comparing both the sets of keys and the sets of values.

Because a dictionary acts like a function, one might expect the ordering on

\[
\forall i \in 1\ldots n. \exists j \in 1\ldots m. \hat{b}_i \sqsubseteq_{A,M} \hat{b}'_j \\
\{\hat{b}_1,\ldots,\hat{b}_m\} \sqsubseteq_{A,M} \{\hat{b'}_1,\ldots,\hat{b'}_m\}
\]

Figure 5.17: Approximation rule for sets of behaviors
\[ Q_{1,i} \subseteq_{A,M} Q_{2,i} \text{ for } i \in 1,\ldots,n \quad C_1 \subseteq C_2 \quad \hat{e}_1 \subseteq_{A,M} \hat{e}_2 \]
\[
\langle Q_{1,1} \ldots Q_{1,n}, C_1[\hat{e}_1] \rangle \subseteq_{A,M} \langle Q_{2,1} \ldots Q_{2,n}, C_2[\hat{e}_2] \rangle
\]
\[ Q_{1,i} \subseteq_{A,M} \hat{Q}_{2,i} \text{ for } i \in 1,\ldots,n \quad \hat{e}_1 \subseteq_{A,M} \hat{e}_2 \quad \hat{f}_c_1 \subseteq_{A,M} \hat{f}_c_2 \]
\[
\langle \hat{Q}_{1,1} \ldots \hat{Q}_{1,n}, (\text{receive } x \hat{e}_1 \hat{f}_c_1) \rangle \subseteq_{A,M} \langle \hat{Q}_{2,1} \ldots \hat{Q}_{2,n}, (\text{receive } x \hat{e}_2 \hat{f}_c_2) \rangle
\]
\[ (\text{define-state } (q \ [x \ \tau]) (x_m) \hat{e}_1 \hat{f}_c_1) \subseteq_{A,M} (\text{define-state } (q \ [x \ \tau]) (x_m) \hat{e}_2 \hat{f}_c_2) \]
\[ \hat{e}_1 \subseteq_{A,M} \hat{e}_2 \quad \hat{f}_c_1 \subseteq_{A,M} \hat{f}_c_2 \]
\[ \text{no-timeout} \subseteq_{A,M} \text{no-timeout} \]
\[ \langle \text{timeout } \hat{o}_v_1 \rangle \subseteq_{A,M} \langle \text{timeout } \hat{o}_v_2 \rangle \]
\[ \langle \text{timeout } \hat{o}_v_1 \rangle \subseteq_{A,M} \langle \text{timeout } \hat{o}_v_2 \rangle \]

Figure 5.18: Approximation rules for behaviors, state definitions, and timeout clauses

\[ \boxed{[]} \subseteq [] \quad \boxed{[]} \subseteq (\text{in-loop } \boxed{[]}) \quad (\text{in-loop } \boxed{[]}) \subseteq (\text{in-loop } \boxed{[]}) \]

Figure 5.19: Approximation rules for parent contexts
∀ i ∈ 1...n. ∃ j ∈ 1...m. \( \overline{v}_i \sqsubseteq_{A,M} \overline{v}'_j \)

every free variable occurring in \( \{ \overline{v}'_1, ..., \overline{v}'_m \} \) occurs in \( \{ \overline{v}_1, ..., \overline{v}_n \} \)

(list \( \{ \overline{v}_1, ..., \overline{v}_n \} \) \( \sqsubseteq_{A,M} \) (list \( \{ \overline{v}'_1, ..., \overline{v}'_m \} \))

∀ i ∈ 1...n. ∃ j ∈ 1...o. \( \overline{v}_i \sqsubseteq_{A,M} \overline{v}''_j \)

∀ i ∈ 1...m. ∃ j ∈ 1...p. \( \overline{v}'_j \sqsubseteq_{A,M} \overline{v}'''_j \)

every free variable occurring in \( \{ \overline{v}''_1, ..., \overline{v}'''_m \} \) occurs in \( \{ \overline{v}_1, ..., \overline{v}_n, \overline{v}'_1, ..., \overline{v}'_m \} \)

(list \( \{ \overline{v}_1, ..., \overline{v}_n \} \) \( \sqsubseteq_{A,M} \) (list \( \{ \overline{v}'_1, ..., \overline{v}'_m \} \))

\( \hat{a} \sqsubseteq_{A} \hat{a}' \quad H \sqsubseteq_{M} H' \)
\( \hat{a}@H \sqsubseteq_{A,M} \hat{a}'@H' \)

∀ i ∈ 1...n. \( \hat{v}_i \sqsubseteq_{A,M} \hat{v}'_i \)

(record \( [r_1 \hat{v}_1] ... [r_n \hat{v}_n] \) \( \sqsubseteq_{A,M} \) (record \( [r_1 \hat{v}'_1] ... [r_n \hat{v}'_n] \))

∀ i ∈ 1...n. \( \hat{v}_i \sqsubseteq_{A,M} \hat{v}'_i \)

\( \hat{v} \sqsubseteq_{A,M} \hat{v}' \)

(fold \( \tau \hat{v} \) \( \sqsubseteq_{A,M} \) (fold \( \tau \hat{v}' \))

abs-nat \( \sqsubseteq_{A,M} \) abs-nat

abs-string \( \sqsubseteq_{A,M} \) abs-string

Figure 5.20: Approximation rules for value expressions
Abstract conformance is based on PSM conformance, which considers only a subset of all the possible transitions a program configuration can take. As a result, the abstract interpretation of CSA is designed to be sound only with respect to those restricted transitions. This section provides some definitions needed to incorporate those restrictions at the abstract level.

First, abstract program configurations require a transition relation that incorporates the restrictions from the \( \rightarrow_R \) relation at the concrete level. Specifically, those restrictions are the following:

- The environment never sends the program a message containing an internal address.
Each address in a message from the environment is the “representative” message from its equivalence class, as described in appendix C.

Only one actor (plus the children it spawns while handling an event) run at a time.

New addresses and history markers are allocated using a deterministic scheme.

Thus, a new relation $\rightarrow_{RA}$, called the restricted abstract-transition relation, is like the abstract transition relation $\rightarrow$ defined in section 5.2.3, except that it incorporates the above restrictions. This is analogous to the relationship between $\rightarrow$ and $\rightarrow_{R}$ for concrete configurations. The formal definition of $\rightarrow_{RA}$ is found in appendix I.

The soundness lemma is also defined with respect to concrete program configurations reached only by taking those restricted steps. The below definitions formalize the necessary properties for those configurations, which come from externals-only conformance (appendix B) and single-handler conformance (appendix D).

**Definition.** A marked program configuration $\langle \hat{\beta} | \hat{\mu} | H \rangle^{\hat{\rho}}$ is an externals-only configuration if and only if

- for all $a@H'$ appearing in either $\hat{\beta}$ or $\hat{\mu}$, $H' = \emptyset$ if $a$ is internal, and $|H'| = 1$ otherwise, and
- for all $\langle a@H', \tau \rangle \in \hat{\rho}$, $|H'| = 1$.

**Definition.** A marked program configuration $\langle \hat{\beta} | \hat{\mu} | H \rangle^{\hat{\rho}}$ is a single-handler configuration if and only if for all $a = (\text{addr } \ell n)$, if $\hat{\beta}(a)$ is handling an event, then there is no $a' = (\text{addr } \ell n')$ such that $a \neq a'$ and $\hat{\beta}(a')$ is handling an event.

**Maximal Received Values**

On top of the four restrictions listed above, the relation $\rightarrow_{RA}$ includes one additional technique to reduce the number of reachable program configurations. Consider an abstract program configuration $\hat{K}$ that can receive from the environment a message of type $\text{List (Variant } [A] [B] \text{ )}$, i.e., a list containing A’s and B’s. There are four abstract messages that configuration may receive:

1. $(\text{list})$, which represents only the empty list;
2. $(\text{list } \{\text{variant A}\})$, which represents any list containing just A’s (including the empty list);
3. $(\text{list } \{\text{variant B}\})$, which represents any list containing just B’s; and
4. $(\text{list } \{(\text{variant A}), (\text{variant B})\})$, which represents all lists of A’s and B’s.
The last abstract message in that list approximates each of the other three; there is nothing that \( \hat{K} \) can do in response to one of the first three messages that it cannot do in response to the fourth. To rule out such redundant transitions, the relation \( \rightarrow_{RA} \) allows a configuration to receive only “maximal” values for a given type, such as the fourth value in the above list. Appendix I formalizes this idea in the definition of \( \rightarrow_{RA} \).

5.2.7 Soundness

The purpose of the abstract interpretation is to define an abstract notion of conformance such that if an abstract program configuration \( \hat{K} \) conforms to a PSM \( s \), then so does any concrete configuration \( \hat{K} \) that it abstracts. The proof of this property requires showing that the abstract interpretation of CSA is a sound abstraction of concrete CSA, including its fairness properties. This section formalizes that idea.

Transition Soundness

At the level of individual transitions, soundness is defined in terms of a simulation. Roughly, for every step \( \hat{K} \overset{\hat{l}}{\rightarrow} \hat{K}' \) that a concrete, externals-only, single-handler configuration \( \hat{K} \) can take, an abstracting configuration \( \hat{K} \) can take a similar step \( \hat{K} \overset{\hat{l}}{\rightarrow}_{RA} \hat{K}' \), and \( \hat{l} \) and \( \hat{K}' \) abstract \( \hat{l} \) and \( \hat{K}' \), respectively. The below lemma formalizes this idea.

**Lemma** (Soundness of Transitions for Abstract CSA). For all \( \hat{K} = \langle \{ \hat{\beta} \mid \hat{\mu} \mid H \} \rangle^{\hat{\beta}}, \hat{K}' = \langle \{ \hat{\beta}' \mid \hat{\mu}' \mid H' \} \rangle^{\hat{\beta}'}, \hat{l}, \hat{K}, A, \) and \( M \), if

- \( \hat{K} \) is a well-formed, externals-only, single-handler configuration,
- \( |\hat{K}| \sqsubseteq_{A,M} \hat{K} \),
- \( \overset{\hat{l}}{\rightarrow} \hat{K}' \),
- \( \text{dom}(A) = \text{dom}(\hat{\beta}) \cup \text{ExtAddr} \),
- \( \text{dom}(M) \subseteq H \), and
- for all \( \hat{\alpha} \in \text{ExtAddr} \), \( A(\hat{\alpha}) = \hat{\alpha} \),

then there exist \( \hat{l}, \hat{K}', A', \) and \( M' \) such that

- \( \overset{\hat{l}}{\rightarrow}_{RA} \hat{K}' \),
- \( |\hat{l}| \sqsubseteq_{A \cup A', M \cup M'} \hat{l} \),
- \( |\hat{K}'| \sqsubseteq_{A \cup A', M \cup M'} \hat{K}' \),
- \( \text{dom}(A') = \text{dom}(\hat{\beta}') - \text{dom}(\hat{\beta}) \), and
• \( \text{dom}(M') \subseteq H' - H \).

The new substitution \( M' \) provides the correspondence between the new markers from \( \hat{K} \xrightarrow{R} \hat{K}' \) and \( \hat{K} \xrightarrow{RA} \hat{K}' \). The non-stuckness conditions are necessary for showing the soundness of abstract conformance (defined in section 5.4). Appendix I contains a proof of this lemma.

**Fair-Execution Soundness**

Because conformance to an APS specification is defined in terms of fair executions of CSA program, rather than individual transitions, it is also necessary that every fair concrete execution corresponds to some fair abstract execution. For PSM conformance, those executions are sequences of “event steps”. An event step \( \hat{K} \xrightarrow{l_1,...,l_n} K' \) represents the full sequence of (restricted) transitions an actor takes to handle an event (i.e., message or timeout)—appendix F describes them in more detail. Thus, abstract conformance needs an abstract notion of event step. This is defined below.

**Definition.** The abstract event-step relation \( \xrightarrow{\text{RA}} \) is defined such that

\[
\hat{K}_1 \xrightarrow{l_1,...,l_n} K' \xrightarrow{l_{n+1}} K_{n+1}
\]

if and only if there exist \( \hat{K}_2, \ldots, \hat{K}_n \) such that

• \( \hat{K}_1 \xrightarrow{\text{RA}} \hat{K}_2 \xrightarrow{l_2,...,l_n} K_{n+1} \),

• \( n > 0 \),

• some actor in \( \hat{K}_i \) is handling an event for all \( i \in 2 \ldots n \), and

• no actor in \( \hat{K}_{n+1} \) is handling an event.

A abstract-event-step execution is a (possibly infinite) sequence of transitions

\[
\hat{K} \xrightarrow{l_{1,1} \ldots l_{1,n_1}} \hat{K}' \xrightarrow{l_{2,1} \ldots l_{2,n_2}} \ldots
\]

Such an execution is stuck if any of its configurations are stuck (since the reached configuration must have no actor handling an event, this effectively means that only the zero-step execution starting with a stuck configuration is stuck). Fairness for such executions is similar to the definition in section 5.2.4, except that each condition checks for the existence of a step in which the first label of a step (i.e., some \( l_{1,1} \)) meets the given criteria.

Whenever an abstract event-step execution simulates a fair concrete event-step execution, the abstract execution is also fair. The following lemma formalizes this statement:

**Lemma (Soundness of Fair Executions).** For all concrete event-step executions

\[
\hat{K}_1 \xrightarrow{l_{1,1} \ldots l_{1,n_1}} \hat{K}_2 \xrightarrow{l_{2,1} \ldots l_{2,n_2}} \ldots
\]

all abstract event-step executions

\[
\hat{K}_1 \xrightarrow{l_{1,1} \ldots l_{1,n_1}} \hat{K}_2 \xrightarrow{l_{2,1} \ldots l_{2,n_2}} \ldots
\]

of the same length, and all A and M, if

• the concrete execution is fair,

• all of the \( \hat{K}_1 \) in the concrete execution are externals-only configurations,
• $|\overline{K}_i| \subseteq_{A,M} \hat{K}_i$ for all corresponding configurations in the two executions, and

• $|\overline{l}_{i,j}| \subseteq_{A,M} \hat{l}_{i,j}$ for all corresponding labels in the two executions,

then the abstract execution is also fair.

Appendix I contains a proof of this lemma.

### 5.3 Abstract Transitions for PSMs

The abstract interpretation of CSA comes with a notion of abstract conformance so that the model checker developed in the next two chapters can take the following approach:

1. Abstract the initial configuration of the given program into an abstract configuration $\hat{K}$.

2. Check whether $\hat{K}$ conforms to the initial PSM $s$ of the given specification; return true if so and false otherwise.

As we will see in section 5.4, abstract conformance is sound with respect to concrete conformance. So if the model checker returns true, then the given program definitely conforms to the given specification.

Abstract conformance does not require a change to the structure of specification configurations, but it does require new transition relations that allow a PSM or specification configuration to simulate abstract program transitions rather than concrete ones. The relation is almost identical to the concrete relation defined in chapter 3, except for a few minor changes that enable a PSM/specification configuration to communicate abstract values to abstract addresses.

First, the new transition relations use slightly different labels to represent the communication of abstract values. These are defined below.

$$\hat{l} ::= \hat{a}@H?\hat{v} \mid \hat{a}@H!\hat{v} \mid \cdot$$  

(Abstract Specification Transition-Step Labels)

Next, the label-conversion function $\lfloor \_ \rfloor$ is adapted for abstract transition labels. To match the abstract transition semantics for program configurations from section 5.2.3, it converts a CSA transition label to a send or receive label if the communication is to or from the environment, rather than if the destination address is marked as in the concrete version of this function.

$$[\hat{l}] = \begin{cases} 
\hat{a}@H?\hat{v} & \text{if } \hat{l} = \hat{a} : \text{rcv-ext}(H, \hat{v}, \tau) \\
\hat{a}@H!\hat{v} & \text{if } \hat{l} = \hat{a} : \text{send-ext}(\hat{a}@H, \hat{v}) \\
\cdot & \text{otherwise}
\end{cases}$$

Finally, the pattern-matching rules are adapted to match abstract values instead of concrete ones. The rules are defined in figures 5.23 and 5.24; they are analogous to the concrete rules from chapter 3.

The new abstract transition relations for PSMs and specification configurations use the same arrow as their corresponding concrete relations; the label on
5.3. ABSTRACT TRANSITIONS FOR PSMS

\[ \hat{v} \sim \ast \triangleright \emptyset \]
\[ \eta = \max(H) \]
\[ \hat{a}@H \sim x \triangleright [x \mapsto \eta] \]
\[ \hat{v}_i \sim p\hat{i}_i \triangleright [x_i \mapsto \eta_i, \ldots, x_{i,m} \mapsto \eta_{i,m}] \text{ for all } i \in 1 \ldots n \]
\[ (\text{variant } t \hat{v}_i^{\in [1..n]}) \sim (\text{variant } t p\hat{i}_i^{\in [1..n]}) \triangleright \bigcup_{i \in 1 \ldots n} [x_i \mapsto \eta_i, \ldots, x_{i,m} \mapsto \eta_{i,m}] \]
\[ \hat{v}_i \sim p\hat{i}_i \triangleright [x_i \mapsto \eta_i, \ldots, x_{i,m} \mapsto \eta_{i,m}] \text{ for all } i \in 1 \ldots n \]
\[ (\text{record } [r_i \hat{v}_i^{\in [1..n]}) \sim (\text{record } [r_i p\hat{i}_i^{\in [1..n]}) \triangleright \bigcup_{i \in 1 \ldots n} [x_i \mapsto \eta_i, \ldots, x_{i,m} \mapsto \eta_{i,m}] \]
\[ \hat{v} \sim po \triangleright [x_1 \mapsto \eta_1, \ldots, x_n \mapsto \eta_n] \]
\[ (\text{fold } \tau \hat{v}) \sim po \triangleright [x_1 \mapsto \eta_1, \ldots, x_n \mapsto \eta_n] \]

Figure 5.23: Abstract pattern-matching for input patterns

\[ \hat{v} \sim po \triangleright H, S \]
\[ \hat{v} \sim (\text{or } po' po po'') \triangleright H, S \]
\[ \hat{v}_i \sim po_1 \triangleright H_i, S_i \]
\[ (\text{variant } t \hat{v}_i^{\in [1..n]}) \sim (\text{variant } t po_i^{\in [1..n]}) \triangleright \bigcup_{i \in 1 \ldots n} H_i, \bigcup_{i \in 1 \ldots n} S_i \]
\[ \hat{v}_i \sim po_1 \triangleright H_i, S_i \]
\[ (\text{record } [r_i \hat{v}_i^{\in [1..n]}) \sim (\text{record } [r_i po_i^{\in [1..n]}) \triangleright \bigcup_{i \in 1 \ldots n} H_i, \bigcup_{i \in 1 \ldots n} S_i \]
\[ \hat{v} \sim po \triangleright H, S \]
\[ (\text{fold } \tau \hat{v}) \sim po \triangleright H, S \]

\[ \hat{a} \text{ is internal } \eta = \max(H) \]
\[ \hat{a}@H \sim (\text{delayed-fork-addr } (\text{goto } \varphi) \overrightarrow{\Phi}) \triangleright \emptyset, \{ \{ \eta \}, \varphi, \varphi : \epsilon, \vec{\Phi}, \varphi \} \]
\[ \hat{a} \text{ is internal } \eta = \max(H) \]
\[ \hat{a}@H \sim \text{self-addr} \triangleright \{ \eta \}, \emptyset \]

Figure 5.24: Abstract pattern-matching for output patterns
the arrow will always disambiguate between the abstract and concrete relations where necessary. The abstract specification-configuration transition relation → has a weak-step transition relation → defined analogously to the one in chapter 3. Fairness is also defined analogously to the definition in chapter 3.

Specification configurations also have an event-step-like transition relation → which merely combines multiple → transitions into one.

**Definition.** The relation → is defined such that $S_1 \overset{\vec{\lambda}_1, O_1}{\cdots} \overset{\vec{\lambda}_n, O_n}{\cdots} S_{n+1}$ if and only if there exist $S_2, \ldots, S_n$ such that $S_1 \overset{\vec{\lambda}_1, O_1}{\cdots} \overset{\vec{\lambda}_n, O_n}{\cdots} S_{n+1}$.

As with the → relation, the obligations $O_i$ can be dropped where irrelevant. An execution $S_1 \overset{\vec{\lambda}_1, O_1}{\cdots} \overset{\vec{\lambda}_n, O_n}{\cdots} \cdots$ of such transitions is fair if and only if there exists some fair → execution $S_1 \overset{\vec{\lambda}_1, O_1}{\cdots} \overset{\vec{\lambda}_n, O_n}{\cdots} \cdots$ that it denotes.

### 5.4 Abstract Conformance

Abstract conformance is defined analogously to PSM conformance (found in appendix G).

**Definition.** A relation $R$ is abstract-conformance-dense if and only if for all $\langle \vec{K}_1, s \rangle \in R$ and all fair, non-stuck executions $\vec{K}_1 \overset{\vec{\lambda}_{1,1}, \ldots, \vec{\lambda}_{1,n}}{\cdots} \cdots$, there exists a fair specification execution $S_1 \overset{\vec{\lambda}_{1,1}, O_{1,1}}{\cdots} \overset{\vec{\lambda}_{1,n}, O_{1,n}}{\cdots} \cdots$ with the same length such that $S_1 = \{s\}$ and for all $\vec{K}_i$ and $S_i$ in the respective executions and all $s' \in S_i$, $\langle \vec{K}_i, s' \rangle \in R$.

For the reader who has not read appendices F and G, the “shape” of this definition is similar to the one from chapter 3, in that for every fair execution of the program there is a fair execution of the specification. There are just two main differences relative to (concrete-)conformance-density:

1. Abstract-conformance-density requires only that the program-configuration/PSM pairs between event steps be in the relation, rather than the pairs between every individual transition step.

2. Abstract-conformance-density is defined in relation to an individual PSM $s$ rather than an entire specification configuration $\Sigma$.

For a further discussion of those two techniques, see appendices F and G, respectively.

Conformance at the level of a program and specification (rather than configurations of those) is defined similarly to concrete conformance, except for the abstraction of the initial program configuration $\vec{K}$.

**Definition.** An abstract program configuration $\vec{K}$ abstract-conforms to a PSM $s$, written $\vec{K} \models_A s$, if and only if there exists an abstract-conformance-dense relation $R$ such that $\langle \vec{K}, s \rangle \in R$. Furthermore, a program $P$ abstract-conforms to a specification $\Sigma$, written $P \models_A \Sigma$, if and only if there exists some maximal instantiation $\langle \vec{K}, \{s\} \rangle$ of $P$ and $\Sigma$ such that $|\vec{K}| \models_A s$. 
Our end goal is to be able to prove concrete conformance, as defined in chapter 3, so abstract conformance is useful only if it can be used to prove concrete conformance. The Conformance Equivalence theorem from the end of chapter 3 proves that PSM conformance is equivalent to concrete conformance, so the following theorem closes the gap by stating that abstract conformance implies PSM conformance:

**Theorem (Abstract Conformance).** For all \( P \) and \( \Sigma \), if \( P \models_A \Sigma \), then \( P \models_{PSM} \Sigma \).

Appendix I proves the theorem. The main obligation of the proof is to prove the Soundness of Abstract CSA lemma mentioned above: that is, that the abstract interpretation for CSA is sound with respect to the concrete transitions. Section 5.2.2 gave an example showing that the converse of this theorem does not hold.

### 5.5 Summary Conformance

The abstract interpretation of CSA greatly reduces the number of program-configuration/PSM pairs needed to prove conformance for a given program. Nevertheless, an abstract expression may have infinitely many different ways to execute, because the non-deterministic semantics for for/fold loops removes the bound on loop iterations. This section introduces a notion of conformance that solves this issue.

#### 5.5.1 Summary Transitions

As an example, consider the following expression that sends a Purchase request to some store actor for each book in a list of books (represented by their ISBN):

```plaintext
(for/fold ([result (variant Unit)])
  ([book books])
  (send store (variant Purchase book)))
```

When evaluating this expression during the abstract interpretation, we might end up with an expression like the following:

```plaintext
(for/fold ([result (variant Unit)])
  ([book (list (abs-nat))])
  (send store (variant Purchase book)))
```

Because the abstract value `(list (abs-nat))` can represent a list containing any number of ISBNs, there are infinitely many possible ways to evaluate that expression. This poses a problem for the model-checking algorithm, which needs to exhaustively check all possible transition steps in a finite amount of time.

The solution is to define a new kind of transition step, called a **summary transition**. A summary transition summarizes the observable effects from some
event step: namely, the event that kicked off that step, and the messages sent back to the environment. This section also defines a new notion of conformance based on these transitions, so that the model checker can use them to represent infinitely many event steps with finitely many summary transitions.

A summary-transition label $L$ encodes the summary of an event step. It is a pair with the following form:

$$L := (\hat{l}, \hat{\mu}) \quad \text{(Program Summary-Transition Labels)}$$

The first part of the pair is the label $\hat{l}$ of the event that started the event step.

The second part is an abstract message map $\hat{\mu}$ that records the abstract quantity (i.e., either single or many) of every message sent to the environment during that event step.

For example, say there is an abstract event step in which the actor with address $\hat{a}$ times out and runs the above loop multiple times to send Purchase requests to a store actor at address $\hat{a}'$. The initial label of that sequence would be $\hat{l}_1 = \hat{a} : \text{timeout}$, and we could summarize the sent messages with an abstract message map $\hat{\mu}_{\text{reqs}} = \{\langle \hat{a}' @ \overline{\varnothing}, \text{(variant Purchase abs-nat)} \rangle \mapsto \text{many} \}$. Thus, the summary label $L = (\hat{l}_1, \hat{\mu}_{\text{reqs}})$ describes that event step. This idea is formalized as follows:

**Definition.** A summary label $L = (\hat{l}, \hat{\mu})$ summarizes an abstract label sequence $\hat{l}_1, \ldots, \hat{l}_n$ if and only if there exist $\hat{a}_1, \ldots, \hat{a}_m, \hat{a}_1', \ldots, \hat{a}_m', H_1, \ldots, H_m$, and $\hat{v}_1, \ldots, \hat{v}_m$ such that

- $\hat{l} = \hat{l}_1$,
- $\hat{l}_1$ is not a send-ext label,
- there is no rcv-ext label in $\hat{l}_2, \ldots, \hat{l}_n$,
- $\hat{a}_1 : \text{send-ext}(\hat{a}_1' @ H_1, \hat{v}_1), \ldots, \hat{a}_m : \text{send-ext}(\hat{a}_m' @ H_m, \hat{v}_m)$ are the send-ext labels in $\hat{l}_1, \ldots, \hat{l}_n$, and
- $\varnothing \oplus \langle \hat{a}_1' @ H_1, \hat{v}_1 \rangle \ldots \oplus \langle \hat{a}_m' @ H_m, \hat{v}_m \rangle \subseteq_{\text{id, id}} \hat{\mu}$.

Messages summarized with a quantity of many often come from for/fold loops, as in the above example, but this is not always the case. For example, consider the following variation on the above example that sends exactly two Purchase requests:

```plaintext
(begin
  (send store (variant Purchase abs-nat))
  (send store (variant Purchase abs-nat)))
```

These two sends would also be summarized with an abstract quantity many, because the actor executing this code would send the same abstract message to the same destination twice in the same event handler.

The transition relation is formalized as follows:
5.5. SUMMARY CONFORMANCE

**Definition.** The summary transition relation is defined such that $\delta \xrightarrow{L} \delta'$ for $L = \langle \hat{I}_1, \hat{\mu} \rangle$ if and only if there exist $\hat{I}_2, \ldots, \hat{I}_n$ such that

- $\delta \xrightarrow{\hat{I}_1, \ldots, \hat{I}_n} \delta'$,
- $L$ summarizes $\hat{I}_1, \ldots, \hat{I}_n$, and
- for all $\hat{a}$, $H$, and $\hat{v}$ such that $\hat{\mu}(\hat{a}@H, \hat{v}) = \text{many}$, there exist an abstract event step $\hat{\delta} \xrightarrow{\hat{I}'_1, \ldots, \hat{I}'_n} \hat{\delta}'$, two distinct labels $\hat{I}'_i$ and $\hat{I}'_j$ from that step, $\hat{v}'$, and $\hat{v}''$ such that
  - $\hat{I}'_i = \hat{a}@H!\hat{v}'$,
  - $\hat{I}'_j = \hat{a}@H!\hat{v}''$, and
  - $\text{Merge}(\hat{v}', \hat{v}'')$ and $\text{Merge}(\hat{v}, \hat{v}'')$ are defined.

The comparison $\sqsubseteq_{id,id}$ implies that the message map $\hat{\mu}$ may contain additional messages with a quantity of $\text{many}$, and that the messages may be approximations of the ones actually sent (see the definition of $\sqsubseteq$ in section 5.2.5). The last requirement effectively says that for every output $(\hat{a}@H, \hat{v})$ in the summary with a quantity of $\text{many}$, there must be an event step that sends at least two messages corresponding to that summary. This is necessary for some of the proofs in appendix K.

As a shorthand, we say that $\delta \xrightarrow{L} \delta'$ summarizes $\delta \xrightarrow{\hat{I}_1, \ldots, \hat{I}_n} \delta'$ if and only if

$$
\delta \xrightarrow{L} \delta', \delta \xrightarrow{\hat{I}_1, \ldots, \hat{I}_n} \delta', \text{ and } L \text{ summarizes } \hat{I}_1, \ldots, \hat{I}_n.
$$

A summary execution is a (possibly infinite) sequence of summary transitions $\delta_1 \xrightarrow{L_1} \delta_2 \xrightarrow{L_2} \ldots$, and it summarizes any abstract-event-step execution for which each individual summary transition summarizes each abstract event step. Such an execution is fair if and only if the following conditions hold:

1. For all configurations $\hat{\delta}_i$ in the execution, if an actor at $\hat{a}$ is necessarily enabled in $\hat{\delta}_i$, then $\exists j \geq i$ such that $\hat{\delta}_j \xrightarrow{\langle \hat{I}_i, \hat{\mu} \rangle} \hat{\delta}_{j+1}$ is a step in the execution for some $\hat{\mu}$ and $\hat{a}$ is the active actor for $\hat{I}_i$.

2. For all configurations $\hat{\delta}_i = \langle \hat{\mu} \mid \hat{H} \rangle$ in the execution, and for all $\hat{a}, \hat{H}'$, and $\hat{v}$ such that $\hat{\mu}(\hat{a}@\hat{H}', \hat{v}) = \text{single}$, there exists some $j \geq i$ such that either $\hat{\delta}_j = \langle \hat{\mu}' \mid \hat{H}' \rangle$ is a configuration in the execution and $\hat{\mu}(\hat{a}@\hat{H}', \hat{v}) = \text{many}$, or $\hat{\delta}_j \xrightarrow{\langle \hat{a}, \text{rcv-int}(\hat{H}', \hat{v}), \hat{v}'' \rangle} \hat{\delta}_{j+1}$ is a step in the execution for some $\hat{\mu}''$.

### 5.5.2 Summary Transitions for Specifications

To define a notion of conformance in terms of summary transitions, we need to define what it means for a specification configuration to simulate a summary
transition. Therefore, specification configurations have a similar notion of summary transition, with the transition label $\Lambda$ defined below.

$$\Lambda ::= \langle L, O \rangle \quad \text{(Specification Summary-Transition Labels)}.$$ 

The label couples a program summary-transition label with a multiset of fulfilled obligations. Intuitively, a specification summary transition says that for any sequence summarized by $L$, there is some specification transition sequence that transitions with those labels and fulfills at least the obligations in $O$. The definition of the transition relation is below.

**Definition.** The specification-configuration summary transition relation is defined such that $S \xrightarrow{\Lambda} S'$ if and only if $\Lambda = \langle L, O \rangle$ and for every label sequence $\hat{l}_1, \ldots, \hat{l}_n$ summarized by $L$, there exist $O'_1, \ldots, O'_n$ such that

- $S \xrightarrow{\langle \hat{l}_1, O'_1 \rangle} \ldots \xrightarrow{\langle \hat{l}_n, O'_n \rangle} S'$,
- $O \subseteq O'_1 \uplus \ldots \uplus O'_n$,
- for all $\langle \eta, po \rangle \in \text{Obls}(S), \langle \eta, po \rangle \in \text{Obls}(S') \uplus O$, and
- either
  - there exists $s$ such that $S = \{s\}$, $\text{Mon}(s) = \emptyset$, and there is no $s'$ such that $s \xrightarrow{\emptyset} s'$, or
  - for all $\langle H, H', \varphi : \eta, \Phi, O'' \rangle \in S', H \cup H' \neq \emptyset$.

A specification summary execution is a (possibly infinite) sequence of such transitions. Similar to a standard specification execution, such an execution is fair if and only if every obligation is eventually fulfilled. Formally, an execution $S_1 \xrightarrow{L_1, O_1} \ldots$ is fair if and only if for all $s_i, po$, and $\eta$ such that $\langle \eta, po \rangle \in \text{Obls}(S_i)$, there exists a step $S_{i+j} \xrightarrow{\langle \hat{l}_{i+j}, O_{i+j} \rangle} S_{i+j+1}$ later in the execution such that $\langle \eta, po \rangle \in O_{i+j}$.

### 5.5.3 Formal Definition of Summary Conformance

With summary transitions for both program and specification configurations defined, we are ready to define a notion of conformance that takes advantage of these transitions. Summary conformance is based on abstract conformance, except that it requires a simulating specification execution for a summary of each possible abstract event step, rather than for each abstract event step directly.

---

2This condition requires that all obligations appearing in the initial configuration must be accounted for in either the step's fulfilled obligations or the obligations of the reached configuration. This is required for a proof in appendix K.

3This condition simplifies a proof in appendix K. A PSM that monitors no markers does not specify anything, so a programmer would not normally write a specification that results in such a PSM.
5.5. SUMMARY CONFORMANCE

Definition. A relation $R$ is summary-conformance-dense if and only if for all $⟨ˆK_1, s⟩ ∈ R$ and all fair, non-stuck executions $ˆK_1 \xrightarrow{l_1 \ldots l_n} \ldots$, there exists a fair summary execution $ˆK_1 \xrightarrow{L_1} \ldots$ that summarizes it and a fair specification summary execution $S_1 \xrightarrow{(l_1,o_1)} S_2 \xrightarrow{(l_2,o_2)} \ldots$ with the same length as $ˆK_1 \xrightarrow{L_1} \ldots$ such that $S_1 = \{s\}$ and $⟨ˆK_i, s'⟩ ∈ R$ for all $ˆK_i$ and $S_i$ in the respective executions and all $s' ∈ S_i$.

Definition. $K$ summary-conforms to $S$, written $K ⊨_S S$, if there exists a summary-conformance-dense relation $R$ such that $⟨K, S⟩ ∈ R$. Furthermore, a program $P$ summary-conforms to a specification $Σ$, written $P ⊨_S Σ$, if and only if there exists some maximal instantiation $⟨˘K, \{s\}⟩$ of $P$ and $Σ$ such that $|˘K| ⊨_S s$.

As with abstract conformance, summary conformance is useful only to the extent it can be used to prove concrete conformance. Thus, the following theorem states that summary conformance implies abstract conformance, which we know from previous theorems implies concrete conformance:

Theorem (Summary Conformance). For all $P$ and $Σ$, if $P ⊨_S Σ$, then $P ⊨_A Σ$.

Appendix I includes a proof of this theorem. The converse does not hold: summary conformance is strictly weaker than abstract conformance. To see why, consider the following state of a specification that expects exactly two Pong responses for a given Ping:

(\texttt{define-state} (Running)
[(\texttt{variant} Ping d) ->
 (\texttt{obligation} d (\texttt{variant} Pong))
 (\texttt{obligation} d (\texttt{variant} Pong))
 (\texttt{goto} Running)])

A simple implementation of that behavior in a CSA program might look like the following:

(\texttt{define-state} (Running) m
(\texttt{case} m
  [((Ping d)
   (\texttt{begin}
    (\texttt{send} d (\texttt{variant} Pong))
    (\texttt{send} d (\texttt{variant} Pong))
    (\texttt{goto} Running))]))

Such a program would abstract-conform to a specification with the above state. However, any summary for the above event handler would summarize the two Pong outputs with the abstract quantity \texttt{many}. Because of this over-approximation, the program would summary-conform only to a specification that can accept \texttt{any} number of Pong responses, so it would not summary-conform to the above specification.
5.6 Related Work

Huch [64] defines an abstract interpretation for Core Erlang programs that abstracts values according to the term constructors that appear in match-patterns across the program; all values not corresponding to a pattern are abstracted to a bottom ($\bot$) value. The resulting abstraction is a finite system suitable for LTL model checking. However, the approach works for programs that create only a bounded number of actors, and in which all actor mailboxes have bounded size.

Venet [108] used abstract interpretation to analyze the communication topologies of processes in the $\pi$-calculus. The main idea in that work is to distinguish between processes spawned at the same lexical location by approximating the sequence of events that led to each one’s creation. Feret [45] extended Venet’s work to analyze open systems as well as add an occurrence-counting analysis. Garoche et al. [51] adapted Venet and Feret’s work to CAP [31], a calculus of distributed, actor-like objects, and then used the abstract interpretation to determine whether actor addresses are used linearly, or whether a program will spawn only a bounded number of actors over its execution. As mentioned in chapter 3, my history markers were somewhat inspired by the markers used in these works.

Soter [43] uses Petri nets to abstract Erlang programs, leaving the number of processes and the sizes of mailboxes unbounded. This abstraction can be checked only for properties that can be expressed in terms of the number of processes in each state and the number of each type of message in the mailboxes of a certain class of actors.

Several abstract interpretations of Concurrent ML [93], another language with dynamic communication topology, have been designed for static analyses. Colby [32] identifies both channels and processes by the control paths that led to their creation, then abstracts each path into some partition of the set of paths. Reppy and Xiao [94] use a similar notion of control paths to abstract processes, but they abstract each channel to just the variable it is bound to when created. Martel and Gengler [76] abstract each process as a nondeterministic finite automaton that approximates the process’s transitions, then they compute the product of all such automata in a program to find matching send and receive points. My abstract interpretation differs from all of these in that it can single out a specific actor using the atomic/collective-actor distinction. This capability is necessary for precise model checking against an APS specification.

The idea of abstracting together objects created at the same lexical location is a common way to start developing an abstract interpretation [103]. Although developed independently, the atomic/collective-actor distinction in abstract CSA is similar to the distinction made by the “special contour” in Shivers’ reflow analysis [99]. Both allow an analysis to reason precisely about one particular entity while abstracting others together.
Chapter 6

Transformation Conformance

Even a proof of summary conformance may need to show conformance for infinitely many program/specification-configuration pairs, because the abstract interpretation does not bound the number of actors or distinct history markers in each such pair. In pursuit of a model-checking algorithm that can show conformance with just finitely many such pairs, this chapter defines a general framework for resolving the remaining unboundedness. The framework is parameterized over a function that reduces the state space as the program evolves. This chapter shows four different ways to instantiate that function, and chapter 8 gives three more.

The framework is structured as a final conformance relation, called transformation conformance. Transformation conformance implies summary conformance, which through several intermediate steps implies concrete conformance. Therefore, the next chapter uses this framework to define an algorithm that checks whether a CSA program conforms to an APS specification.

6.1 Overview

The abstract interpretation for CSA eliminates some sources of the infinite state space, but each program/specification-configuration pair \( \langle \hat{K}, s \rangle \) explored in an abstract-conformance proof may still have

- unboundedly many outstanding obligations in \( s \),
- unboundedly many history markers either appearing in \( \hat{K} \) or monitored by \( s \), and
- unboundedly many atomic actors in \( \hat{K} \) (which entails unboundedly many atomic addresses in \( \hat{K} \), as well).
To resolve the first issue, the model-checking algorithm defined in the next chapter conservatively assumes that no program configuration can conform to a PSM with multiple copies of the same obligation and therefore does not attempt to prove conformance for such pairs (see section 7.3). This assumption may be overly conservative and prevent the algorithm from proving conformance in some cases, but it has not been problematic in the applications evaluated so far (see chapter 9). Because the output patterns in obligations are bound by the number of patterns written in the original specification, this coarse approach bounds the number of obligations in explored PSMs once the number of history markers is bounded.

The remaining issues are not as simple to resolve. Further abstraction can help, but the challenge is to abstract just enough of each configuration to bound the state space without abstracting away the evidence that a program conforms to its specification.

The solution presented here is to let the particular specification drive the abstraction. By abstracting away only the parts of a program configuration that are irrelevant to its specification, this technique can bound the state space while leaving a reasonable hope of proving conformance to that specification. Of course, determining which aspects of a program configuration are relevant to a specification is undecidable in general, so we must instead rely on heuristics. As a result, this technique may result in some false negatives, but it can help prove conformance for many common cases.

To that end, this chapter develops one final refinement to conformance, called transformation conformance. The main idea is that at every point where summary conformance requires some pair \( \langle \hat{K}, s \rangle \) to be in a summary-conformance-dense relation \( R \), this technique instead applies a special function, called a transformation, to \( \langle \hat{K}, s \rangle \) to get a set of pairs \( \{ \langle \hat{K}_1, s'_1 \rangle, \ldots, \langle \hat{K}_n, s'_n \rangle \} \) that must be in \( R \) instead. The intent is that it should be easier to prove conformance for those pairs, yet conformance for all of them should imply conformance for \( \langle \hat{K}, s \rangle \).

Specifically, this chapter defines a transformation to bound the set of reachable pairs by taking the following steps:

1. **Split** Dividing \( s \) into an “equivalent” set of PSMs \( \{s_1, \ldots, s_n\} \) representing the total behavior specified by \( s \), such that each new PSM monitors only boundedly many history markers, resulting in the set of pairs \( \{ \langle \hat{K}, s_1 \rangle, \ldots, \langle \hat{K}, s_n \rangle \} \) (see section 6.4).

2. **Unmark** For each pair \( \langle \hat{K}, s_i \rangle \) from the previous step, remove from \( \hat{K} \) all history markers not monitored by \( s_i \), because those markers are irrelevant for conformance (see section 6.5).

3. **Assimilate** For each pair \( \langle \hat{K}', s' \rangle \) produced by the previous step, convert all but one atomic actor per spawn location \( \ell \) in \( \hat{K}' \) to a collective actor. Use heuristics to determine which actors might be relevant to the behavior specified by \( s' \) and should therefore be left atomic to enable precise reasoning about their behavior (see section 6.6).
Notice that in the latter two steps, the PSM itself drives the abstraction of the program configuration.

Altogether, these steps bound the sets of atomic actors and history markers appearing in each individual program-configuration/PSM pair, but the set of markers and atomic actors across all reachable pairs is still unbounded. For example, even if every encountered program configuration has just one atomic actor after these transformations, it might be the actor at \((\text{addr} \ell 1)\) in \(\hat{K}_1\), the actor at \((\text{addr} \ell 2)\) in \(\hat{K}_2\), the actor at \((\text{addr} \ell 3)\) in \(\hat{K}_3\), and so on. Thus, an infinite execution may involve infinitely many actor addresses.

However, addresses and history markers are merely opaque tokens, uninterpreted in CSA. If we consider equivalence up to the renaming of addresses and markers, then the number of equivalence classes of the produced pairs is bounded. Thus, one final step is sufficient to bound the state space:

4. **Canonicalize** Rename the history markers and atomic addresses in each pair \(\langle \hat{K}'', s'' \rangle\) produced by the previous step to some canonical naming for that pair’s equivalence class (see section 6.7).

In fact, each of the above steps defines an independent transformation from a pair \(\langle \hat{K}, s \rangle\) to a set of pairs \(\{\langle \hat{K}'_1, s'_1 \rangle, \ldots, \langle \hat{K}'_n, s'_n \rangle\}\). Therefore, the transformation used by the model checker is actually the composition of these four.

The following sections formally define transformations, transformation conformance, and each of the individual transformations described above. Chapter 8 defines three additional transformations used as optimizations for the model checker.

### 6.2 Transformations

#### 6.2.1 Formal Definition

As described above, a transformation is a function that turns a program-configuration/PSM pair \(\langle \hat{K}, s \rangle\) into a set of such pairs, with the intent of using those pairs to prove conformance for \(\langle \hat{K}, s \rangle\). The formal definition follows:

**Definition.** A transformation is a (total) function \(T\) over program-configuration/PSM pairs such that for any pair \(\langle \hat{K}, s \rangle\), \(T(\hat{K}, s)\) is a set of tuples \(\{\langle \hat{K}'_1, s'_1, A_1, M_1 \rangle, \ldots, \langle \hat{K}'_n, s'_n, A_n, M_n \rangle\}\). Each \(A_i\) is a partial function from addresses to addresses, and each \(M_i\) is a partial function from history markers to history markers.

Similar to their use in the approximation relation \(\subseteq\) in the previous chapter, these two functions provide maps from the addresses and markers in \(\langle \hat{K}, s \rangle\) to those in \(\langle \hat{K}', s' \rangle\).

#### 6.2.2 Conformance Reflection

Of course, not every possible transformation \(T\) is useful for proving conformance. For example, let \(\hat{K}_{\text{empty}}\) be the empty configuration \(\langle \emptyset | \emptyset | \emptyset \rangle\), and let \(s_{\text{empty}}\)
be the PSM \( \langle \varnothing, \varnothing, \text{(goto } S) \rangle, \text{(define-state } S \rangle, \varnothing \rangle \). No transitions are possible from \( \hat{K}_{\text{empty}} \), and the PSM \( s_{\text{empty}} \) does not monitor any markers, so \( \hat{K}_{\text{empty}} \) trivially conforms to \( s_{\text{empty}} \). Now, let \( \text{Trivial} \) be the transformation such that for any input pair \( \langle \hat{K}, s \rangle \), \( \text{Trivial} \langle \hat{K}, s \rangle = \{ \langle \hat{K}_{\text{empty}}, s_{\text{empty}}, id, id \rangle \} \). It is easy to prove conformance for all pairs produced by \( \text{Trivial} \), but that does nothing to help prove conformance for \( \langle \hat{K}, s \rangle \), which defeats the point of the transformation. Clearly, there needs to be some sort of relationship between the pair given to a transformation \( T \) and the set of pairs \( T \) produces.

For transformation conformance to be a sound refinement of summary conformance, the transformation must be \text{conformance-reflecting}. Intuitively, a transformation \( T \) is \text{conformance-reflecting} if for all program-configuration/PSM pairs \( \langle \hat{K}, s \rangle \), if all the pairs in \( T \langle \hat{K}, s \rangle \) are conforming, then so is \( \langle \hat{K}, s \rangle \). For example, consider the transformation \( \text{Unmark} \) that removes from the given program configuration any markers that the given PSM does not monitor. \( \text{Unmark} \) is \text{conformance-reflecting} because if some program configuration \( \hat{K} \) conforms to \( s \), then undoing \( \text{Unmark} \) by adding more (unmonitored) markers to \( \hat{K} \) will not affect its conformance to \( s \). So as to not distract from the main contributions of this chapter, the full definition of conformance reflection is given in appendix K. The above intuition is sufficient for understanding the uses of transformations in this chapter.

### 6.2.3 Composition

Transformations can also be composed to combine the benefits of multiple transformations. For example, the model checker described in the next chapter uses the composition of \( \text{Split} \), \( \text{Unmark} \), \( \text{Assimilate} \), and \( \text{Canonicalize} \) as its transformation to fully bound the explored state space. Applying the composition of transformations \( T_1 \) and \( T_2 \) (denoted \( T_1 \odot T_2 \)) entails first applying \( T_2 \) to the given pair, then applying \( T_1 \) to the results from \( T_2 \) and composing the created correspondence functions. Formally, the composition is defined as follows:

\[
(T_1 \odot T_2) \langle \hat{K}, s \rangle = \left\{ \langle \hat{K}'', s'', A_1 \circ A_2, M_1 \circ M_2 \rangle \mid \exists \hat{K}', s'. \right. \\
\langle \hat{K}', s', A_2, M_2 \rangle \in T_2 \langle \hat{K}, s \rangle \text{ and } \langle \hat{K}'', s'', A_1, M_1 \rangle \in T_1 \langle \hat{K}', s' \rangle \right\}
\]

**Lemma** (Composition Conformance Reflection). For all conformance-reflecting transformations \( T_1 \) and \( T_2 \), \( T_1 \odot T_2 \) is a conformance-reflecting transformation.

Appendix L proves this lemma.

### 6.3 Conformance as a Verification Game

A summary-conformance proof can be thought of as a game played between two players, called the Verifier and the Falsifier. The Verifier’s goal is to prove that
6.3. CONFORMANCE AS A VERIFICATION GAME

Figure 6.1: An example of the game to show that a relation $R$ is summary-conformance-dense.

the given relation $R$ is summary-conformance-dense, while the Falsifier’s goal is to prevent the Verifier from finding such a proof.$^{1}$

Figure 6.1 shows an example game. At the start of the game, the Falsifier selects some pair $\langle K_1, s \rangle$ from $R$ and sets the initial specification configuration $S_1 = \{s\}$. Then at the beginning of round 1, the Falsifier chooses as its first “move” some event step $\hat{K}_1 \xrightarrow{\hat{l}_{1,1},...\hat{l}_{1,n}} \hat{K}_2$ from $\hat{K}_1$. In response, the Verifier must choose some transition $S_1 \xrightarrow{\langle L_1, O_1 \rangle} S_2$ such that $\hat{K}_1 \xrightarrow{L_1} \hat{K}_2$ summarizes $\hat{K}_1 \xrightarrow{\hat{l}_{1,1},...\hat{l}_{1,n}} \hat{K}_2$. Each of the following rounds continues the same way, with the Falsifier choosing another event step $\hat{K}_i \xrightarrow{\hat{l}_{i,1},...\hat{l}_{i,n}} \hat{K}_{i+1}$ from the program configuration $\hat{K}_i$ reached in the previous round, and the Verifier choosing a matching summary step $S_i \xrightarrow{\langle L_i, O_i \rangle} S_{i+1}$ from the specification configuration $S_i$ reached in the previous round.

To win the game, the Verifier must satisfy two separate conditions, corresponding to the safety and liveness properties required for conformance.

1. Safety: for every event step $\hat{K}_i \xrightarrow{\hat{l}_{i,1},...\hat{l}_{i,n}} \hat{K}_{i+1}$ the Falsifier chooses, the Verifier must always be able to come up with some transition $S_i \xrightarrow{\langle L_i, O_i \rangle} S_{i+1}$ to match it.

2. Liveness: whenever the sequence of moves taken by the Falsifier represents a fair program execution (including an infinite sequence for an infinitely long game), the matching sequence of moves taken by the Verifier must represent a fair specification execution.

In effect, the safety property says that the program never takes a step forbidden by the specification (i.e., the specification simulates the program), and the liveness property says that every fair execution of the program fulfills all obligations.

For every notion of conformance defined so far, the corresponding conformance-proof game is effectively the same. Transformation conformance,

$^{1}$This game-theoretic view of conformance is based on the work of Henzinger et al. on fair simulations [58].
CHAPTER 6. TRANSFORMATION CONFORMANCE

Round 1

Figure 6.2: A single round of the transformation-conformance game

However, changes the rules of the game, as figure 6.2 illustrates. The round begins the same way, with the Falsifier selecting an event step $\hat{K}_1 \xrightarrow{\hat{l}_1,1,\ldots,\hat{l}_1,n} \hat{K}'_2$ and the Verifier selecting a transition $S_1 \xrightarrow{(L_1,O_1)} S'_2$ to match it. Instead of starting the next round with $\hat{K}'_2$ and $S'_2$, however, the Falsifier

1. chooses some PSM $s'$ from $S'_2$,

2. transforms the pair $\langle \hat{K}'_2, s' \rangle$ with the transformation $T$ (chosen by the Verifier at the start of the game), then

3. chooses one of the resulting pairs $\langle \hat{K}_2, s'' \rangle$ as the pair for the next round of the game, setting $S_2 = \{ s'' \}$.

As mentioned in section 6.1, transforming the resulting pair after each transition allows us to further abstract the program configuration and bound the state space. However, this change also makes it impossible to define transformation conformance similarly to previous notions of conformance. In all previous games, every sequence of moves corresponds to an execution, so the liveness requirement for conformance can be defined in terms of those sequences. Each round of the transformation-conformance game, however, contains an intermediate use of the transformation $T$ to connect each pair $\langle \hat{K}'_{i+1}, S'_{i+1} \rangle$ reached by those transitions to the starting pair $\langle \hat{K}_{i+1}, S_{i+1} \rangle$ for the next round, so the sequences of moves are no longer executions. Therefore, we can no longer talk about the fairness of such sequences.

Instead, transformation conformance is defined directly in terms of this game. After all, a transformation $T$ does is transform a pair $\langle \hat{K},s \rangle$ into a set of “similar” pairs whose conformance helps prove conformance for $\langle \hat{K},s \rangle$, so the move sequences of each game should have some relationship to actual executions and therefore be useful for checking the liveness properties required for conformance.

To realize this new style of definition, section 6.3.1 formalizes the notion of the strategy the Verifier uses to select its moves, which acts as a witness in a transformation-conformance proof. Section 6.3.2 defines a transition relation
6.3. CONFORMANCE AS A VERIFICATION GAME

that encodes the moves of this game, along with associated notions of fairness. Section 6.3.3 then defines transformation conformance in terms of two conditions, corresponding to the two win conditions for the transformation-conformance game:

1. the Verifier’s strategy must include a way to match every possible event step the Falsifier might take, and
2. every game that represents a fair program execution must also represent a fair specification execution.

6.3.1 Strategies

In order to define transformation conformance in terms of the above game, we need to know what moves the Verifier takes over the course of the game. This is encoded by a strategy. A winning strategy acts as a witness for a proof of transformation conformance: it shows how the Verifier can win the game. We will see in chapter 7 that the model-checking algorithm’s main job is to construct a strategy suitable for proving that a given program conforms to a given specification.

Intuitively, a strategy is a function that, given the current state of the game, returns the Verifier’s next move. At the start of the Verifier’s turn, the relevant pieces of game state are

- the summarization $\hat{K} \xrightarrow{L} \hat{K}'$ of the Falsifier’s most recent event step $\hat{K} \xrightarrow{l_1,\ldots,l_n} \hat{K}'$ and
- the lone PSM $s$ in the current specification configuration $S$.

Technically, the summary transition $\hat{K} \xrightarrow{L} \hat{K}'$ is unknown at the beginning of the Verifier’s turn: it is up to the Verifier to choose such a summarization. In practice, however, coming up with a summarization $\hat{K} \xrightarrow{L} \hat{K}'$ of a given event step is easy; the hard part is to find some transition $s \xrightarrow{(L,O)} S$ that simulates it. Therefore, a strategy models each summary transition as a given and records only how a specification configuration can simulate that transition. This allows the model checker in the next chapter to use a finite strategy to prove transformation conformance, because a finite number of summary transitions can represent infinitely many event steps possible from a given program configuration.

Formally, a strategy is defined as follows:

**Definition.** A strategy is a total function $Z$ such that whenever $Z(\hat{K} \xrightarrow{L} \hat{K}', s) = (\langle O_1, S_1 \rangle, \ldots, \langle O_n, S_n \rangle, s) \xrightarrow{(L,O)} S_i$ for all $i \in 1 \ldots n$.

In other words, each result $\langle O_i, S_i \rangle$ given by a strategy indicates that a specification configuration containing just the given PSM $s$ can simulate a program summary step $\hat{K} \xrightarrow{L} \hat{K}'$, fulfilling at least the obligations $O_i$ in the process, and resulting in the PSMs $S_i$. 
A strategy \( Z \) is redundant in that it can contain multiple answers for any given inputs. This enables the model checker in the next chapter to initially try many possible matching transitions, then gradually prune away those that turn out to not work (section 7.1 explains this idea in more detail).

### 6.3.2 Transformation Steps

The idea of a strategy allows us to define the transformation-conformance game in terms of a transition relation, called the transformation-step relation. Specifically, the transformation-step relation is a relation between program-configuration/PSM pairs such that each step defines a single round of the game. Thus, a step of the transition relation simultaneously represents

- a step taken by a program configuration \( \hat{K} \),
- a matching step of a corresponding PSM \( s \) given by a strategy \( Z \), and
- the transformation \( T \) applied to a pair \( \langle \hat{K}', s' \rangle \) reached by those steps.

Formally, the relation is defined as follows:

**Definition.** The transformation-step relation \( \rightarrow \) is defined such that

\[
\langle \hat{K}, s \rangle \xrightarrow{L,K',Z,O,S,T,A,M} \langle \hat{K}'', s'' \rangle \quad \text{if and only if}
\]

- \( \hat{K} \xrightarrow{L} \hat{K}' \),
- \( \langle O, S \rangle \in Z(\hat{K} \xrightarrow{L} \hat{K}', s) \),
- \( \langle \hat{K}'', s'', A, M \rangle \in T(\hat{K}', s') \) for some \( s' \in S \).

Compare to the illustration of the game in figure 6.2. The transformation-step relation requires that the specific \( O \) and \( S \) come from the given strategy \( Z \) (rather than being chosen arbitrarily), but this definition otherwise encodes the steps in a round of that game. The definition here does not mention the specific event step \( \hat{K} \xrightarrow{I_1 \ldots I_n} \hat{K}' \) from the game, because \( \hat{K} \xrightarrow{L} \hat{K}' \) implies the existence of some summarized event step \( \hat{K} \xrightarrow{I_1 \ldots I_n} \hat{K}' \), and the definition of transformation conformance in section 6.3.3 does not need to name the specific event step.

Entire games are then represented as transformation-step executions, defined as follows:

**Definition.** A transformation-step execution is a sequence of transformation-step transitions, using the same strategy \( Z \) and transformation \( T \) in each step.

### Fairness for Transformation-Step Executions

As mentioned towards the beginning of section 6.3, we can treat the sequences of moves of the transformation-conformance game as being “similar to” executions for the purposes of checking whether all obligations are eventually fulfilled.
To do so, a transformation-step execution has two notions of fairness: one for its embedded program steps, and another for its embedded specification steps. Program-fairness roughly corresponds to fairness for program summary executions, and specification-fairness roughly corresponds to fairness for specification summary executions, except that they must also account for some possible complications introduced by the transformations applied at every step.

First, a transformation such as Canonicalize might rename the addresses and markers in a pair \( \langle \hat{K}, s \rangle \). For example, the actor at address \( \text{addr} \ell 3 \) in \( \hat{K} \) might correspond to the actor at address \( \text{addr} \ell 1 \) in a program configuration from Canonicalize\( (\hat{K}, s) \). Therefore, asking a question like “does the actor at \( \text{addr} \ell 3 \) ever get to run in this execution?” does not make sense, because “the same” actor might have a different address in each step of the execution. Instead, the functions \( A \) and \( M \) produced by each transformation give the necessary correspondence between addresses and markers in different steps of the execution, thereby providing a persistent notion of identity. The substitutions provide a similar service for in-flight messages, as well.

Second, an abstracting transformation such as Assimilate might turn an atomic actor into a collective one. Fairness requires that only atomic necessarily enabled actors take a step, so fairness in this context requires only that an enabled actor must eventually take a step as long as it stays an atomic, necessarily enabled actor. A similar exception exists for single messages that must eventually be received, as shown in the definition below.

Define the active actor for a summary-step label \( \ell = \langle \hat{i}, \hat{\mu} \rangle \) to be the active actor for the transition label \( \hat{i} \). Define the application of the correspondence functions \( A \) and \( M \) to an abstract program configuration \( \hat{K} \) to be the component-wise, element-wise application, except that \( \hat{M}(H) = \bigcup_{\eta \in \text{dom}(M) \cap H} \{ M(\eta) \} \) for all sets \( H \) of history markers (this allows \( M \) to drop some markers, e.g., as in the Unmark transformation in section 6.5). Then program-fairness is defined as follows:

**Definition.** A transformation-step execution \( \langle \hat{K}_1, s_1 \rangle \xrightarrow{L_1, \hat{K}_1, Z, O_1, S_1, T, A_1, M_1} \ldots \) is program-fair if and only if both of the following conditions hold:

1. For all program configurations \( \hat{K}_i \) in the execution, if an actor at address \( \hat{a} \) in \( \hat{K}_i \) is necessarily enabled, then there exist \( j \) and \( \hat{a}' = A_{i+j-1} \circ \ldots \circ A_i(\hat{a}) \) such that \( \hat{K}_{i+j} \) is a configuration in the execution and either
   - there is no actor at \( \hat{a}' \) in \( \hat{K}_{i+j} \),
   - the actor at \( \hat{a}' \) is not necessarily enabled in \( \hat{K}_{i+j} \), or
   - \( \langle \hat{K}_{i+j}, s_{i+j} \rangle \xrightarrow{L_{i+j}, \hat{K}_{i+j}, Z, O_{i+j}, S_{i+j}, T, A_{i+j}, M_{i+j}} \langle \hat{K}_{i+j+1}, s_{i+j+1} \rangle \) is a step in the execution and \( \hat{a}' \) identifies the active actor for \( L_{i+j} \).

2. For all program configurations \( \hat{K}_i = \langle \hat{\beta} | \hat{\mu} | H \rangle \hat{\rho} \) in the execution, and all \( \hat{a}, H' \), and \( \hat{\rho} \) such that \( \hat{\mu} \hat{a} = \text{single} \), there exist \( j, \hat{\beta}', \hat{\mu}', H'', \hat{\rho}', \hat{\imath}', A' = A_{i+j-1} \circ \ldots \circ A_i, M' = M_{i+j-1} \circ \ldots \circ M_i, \hat{a}' = A'(\hat{a}), H'' = M'(H') \), and
\( \varnothing^* = A'(M'(\varnothing)) \) such that \( \hat{R}_{i+j} = \langle \langle \hat{B}' \mid \hat{\mu}' \mid H'' \rangle \rangle \) is a configuration in the execution and either

- \( \langle \hat{a}'\hat{H}''', \hat{\varnothing} \rangle \notin \text{dom}(\hat{\mu}') \),
- \( \hat{\mu}'(\hat{a}'\hat{H}''', \hat{\varnothing}) = \text{many, or} \)

\[ \langle \hat{R}_{i+j}, s_{i+j} \rangle \xrightarrow{L_1, K_{i+j}, Z, O_{i+j}, S_{i+j}, T, A_{i+j}, M_{i+j}} \langle \hat{R}_{i+j+1}, s_{i+j+1} \rangle \]

is a step in the execution and \( L_{i+j} = \langle \hat{a}' : \text{rcv-int}(H''', \hat{\varnothing}), \hat{\mu}'' \rangle \) for some \( \hat{\mu}'' \).

Specification-fairness similarly uses the function \( M \) to provide mappings between history markers across the steps of each execution, thereby maintaining an obligation’s “identity” across multiple transition steps.

**Definition.** A transformation-step execution \( \langle \hat{K}_1, s_1 \rangle \xrightarrow{L_1, K_{1}, Z, O_1, S_1, T, A_1, M_1} \ldots \) is specification-fair if and only if for all \( s_i \) in the execution and all obligations \( \langle \eta, po \rangle \in \text{Obls}(s_i) \), there exist some \( j \) and \( \eta' = M_{i+j-1} \circ \ldots \circ M_1(\eta) \) such that either

- there exists some \( O_{i+j} \) in the execution such that \( \langle \eta', po \rangle \in O_{i+j} \), or
- there exists some \( s_{i+j} \) in the execution such that \( \langle \eta', po \rangle \notin \text{Obls}(s_{i+j}) \).

### 6.3.3 Formal Definition of Transformation Conformance

Transformation conformance is defined in terms of the two conditions the Verifier’s strategy must satisfy to win the game: a Simulation condition requiring that every step of the program has at least one matching step from the corresponding PSM, and a Fulfillment condition requiring that every obligation is eventually fulfilled in all games representing fair program executions. The formal definition follows:

**Definition.** A relation \( R \) is transformation-conformance-dense if there exists some strategy \( Z \) and conformance-reflecting transformation \( T \) such that for all \( \langle \hat{K}_1, s_1 \rangle \in R \), the following two conditions hold:

- **Simulation** For all \( \hat{T}_1, \ldots, \hat{T}_n \) and \( \hat{T}' \) such that \( \hat{K}_1 \xrightarrow{L} \hat{T}' \) and \( \hat{T}' \) is not stuck, there exists \( L \) such that

  - \( \hat{K}_1 \xrightarrow{L} \hat{T}' \)
  - \( Z(\hat{K}_1 \xrightarrow{L} \hat{T}', s_1) \neq \emptyset \), and
  - for all transformation steps \( \langle \hat{K}_1, s_1 \rangle \xrightarrow{L, K_{1}, Z, O_1, S_1, T, A_1, M_1} \langle \hat{K}_2, s_2 \rangle \)

- **Fulfillment** Every program-fair transformation-step-execution

\[ \langle \hat{K}_1, s_1 \rangle \xrightarrow{L, K_{1}, Z, O_1, S_1, T, A_1, M_1} \ldots \] is specification-fair.
6.3. CONFORMANCE AS A VERIFICATION GAME

When the strategy and transformation need to be named explicitly, we say that $R$ is a transformation-conformance-dense relation for $T$ with witness $Z$.

Then $\hat{K}$ transformation-conforms to $s$, written $\hat{K} \models_{TR} s$, if there exists a transformation-conformance-dense relation $R$ such that $\langle \hat{K}, s \rangle \in R$. Also, $P \models_{TR}^2 \Sigma$ if there exists some maximal instantiation $\langle \hat{K}, (s) \rangle$ of $P$ and $\Sigma$ such that $|\hat{K}| \models_{TR} s$.

**Theorem** (Transformation Conformance). For all $P$ and $\Sigma$, if $P \models_{TR} \Sigma$, then $P \models_{S} \Sigma$.

The proof of this theorem is in appendix K. It describes a process by which a program summary execution can be converted into a set of transformation-step executions, and then a matching specification summary execution can be extracted from that set.

6.3.4 Putting it all Together

The Transformation Conformance theorem provides the last implication in the chain of refinements to conformance, stretching back to the standard definition of conformance from chapter 3. If we consider each conformance relation as a set of conforming program/specification pairs, we have the following relationships:

$$\models_{TR} \subseteq \models_{S} \subset \models_{A} \subset \models_{PSM} = \models$$

It is unknown whether transformation conformance ($\models_{TR}$) is a strictly tighter relationship than summary conformance ($\models_{S}$). The use of a strategy $Z$ in transformation conformance appears to constrain some of the non-determinism available in summary conformance in terms of how a specification can simulate a program's transition steps, but further work is needed to prove that there is indeed a gap between the two. Henzinger et al. [58] show there is a gap between their notion of fair simulation and $\exists$-simulation, upon which transformation conformance and summary conformance are based, respectively, so this suggests that a similar gap exists here.

All of these different refinements to conformance developed throughout this dissertation are what make building an APS model checker feasible. The algorithm in the next chapter determines whether a given program $P$ transformation-conforms to a given specification $\Sigma$ (i.e., $P \models_{TR} \Sigma$). If so, then by the various relationships shown above, $P$ conforms to $\Sigma$ by the original notion of conformance, as well (i.e., $P \models \Sigma$).

The remaining sections in this chapter formalize the individual transformations (Split, Unmark, Assimilate, and Canonicalize) that the model-checking algorithm uses in the next chapter.

---

2In fact, the relationship between $\models_{PSM}$ and $\models$ is based on several intermediate conformance relations, defined in appendices B–G.
6.4 Split

To explain the first transformation, Split, consider the following state from a ping/pong example specification from chapter 3:

\[
\text{(define-state (Running)}
\begin{align*}
&\quad \text{(variant Ping d) ->} \\
&\quad \quad \text{(obligation d (variant Pong))} \\
&\quad \quad \text{(goto Running))}
\end{align*}
\]

To simulate receiving a Ping message, a PSM with that state can take the listed transition, which adds the marker \(\eta\) matched by the pattern variable \(d\) as an output-monitored marker and adds a Pong obligation for that marker. The transition does not save the variable \(d\) as a state argument, however; therefore, the PSM will never be able to add further obligations for \(\eta\). In effect, the PSM at that point specifies two independent protocols: the first says that a conforming program must send exactly one message (a Pong) to an address marked with \(\eta\), and the second says that a conforming program must continue to handle subsequent Pings.

This observation suggests a way to bound the number of output-monitored markers in each explored PSM: whenever an output-monitored marker and its PSM are unable to affect each other, as in the above example, that marker and its associated obligations can be split off into their own independent PSM. Repeating this process for every such independent marker leaves the PSM monitoring only its non-independent markers.

We will see in the next chapter that just as the model-checking algorithm conservatively assumes that no program configuration conforms to a PSM with multiple copies of the same obligation, it similarly assumes that no program configuration conforms to a PSM with more than \(n + 1\) output-monitored markers, where \(n\) is the number of arguments to the PSM’s current state. Thus, Split ensures that the model-checking algorithm explores only pairs \(\langle \hat{K}, s \rangle\) in which the PSM monitors a bounded number of markers.

Intuitively, if a program’s behavior matches that specified by both the split-off PSMs and the remainder of the original PSM, then it should also match the behavior specified by the entire original PSM. In other words, this transformation is conformance-reflecting.

To formalize this idea, a marker \(\eta\) is an independent marker of a PSM \(\langle H, H', \varphi : \eta_1 \ldots \eta_n, \Phi, O \rangle\) if and only if

- \(\eta \in H'\),
- \(\eta \neq \eta_i\) for all \(i \in 1 \ldots n\), and
- there is no obligation \(\langle \eta, po \rangle \in O\) such that \(po\) contains a self-addr pattern.

The last condition ensures that fulfilling an obligation for an independent marker does not change the PSM’s specified receptionist.
The transformation \( \text{Split} \) itself, defined below, splits each independent marker of a PSM into a separate PSM that models that independent protocol.

\[
\text{Split}(\tilde{\Phi},s) = \{\langle \tilde{\Phi},s',id,\text{id}\rangle,\langle \tilde{\Phi},s''_1,id,\text{id}\rangle,\ldots,\langle \tilde{\Phi},s''_m,id,\text{id}\rangle \}
\]

where \( s = \langle H,H',\varphi:\eta_1\ldots\eta_n,\bar{\Phi},O\cup O'_1\cup\ldots\cup O'_m \rangle \)

and \( \{\eta'_1,\ldots,\eta'_m\} \) are the independent markers of \( s \)
and for all \( \langle \eta'',\text{po}\rangle \in O, \eta'' \notin \{\eta'_1,\ldots,\eta'_m\} \)
and for all \( i \in 1\ldots m \) and all \( \langle \eta'',\text{po}\rangle \in O'_i, \eta'' = \eta'_i \)
and \( s' = \langle H,H' - \{\eta'_1,\ldots,\eta'_m\},\varphi:\eta_1\ldots\eta_n,\bar{\Phi},O \rangle \)
and \( s''_i = \langle \varphi,\{\eta'_i\},\text{Dummy} : \eta'_i,\{\text{define-state} \ (\text{Dummy})\},O'_i \rangle \) for all \( i \in 1\ldots m \)

Each new PSM \( s''_i \) produced by \( \text{Split} \) starts in a \text{Dummy} state that has no transitions, because the PSM’s only purpose is to specify the messages that should be sent to an address marked with \( \eta'_i \). In each returned tuple, the identity function is used for the address- and marker-correspondence functions, because this transformation does not rename the addresses or history markers in the given program configuration or PSM.

To illustrate the use of \( \text{Split} \) on the ping/pong example above, let \( \Phi_{\text{dummy}} \) be the state definition given above, and let \( s = \{\langle \eta_1,\eta_2\rangle,\text{Running,}\Phi_{\text{running}},\{\langle \eta_2,\langle \text{variant Pong} \rangle \rangle\} \} \) be the PSM just after taking the Ping transition, where \( \eta_2 \) is the marker matched by \( d \). Furthermore, let \( \tilde{\Phi} \) be an arbitrary program configuration.

The marker \( \eta_2 \) is an independent marker of \( s \), because it is not a parameter to the PSM’s current state, and there is no obligation for it containing a \text{self-addr} pattern. Therefore, \( \text{Split} \) moves \( \eta_2 \) and its corresponding obligation into a new PSM with only the \text{Dummy} state, thereby dividing the PSM \( s \) into

- \( s' = \{\langle \eta_1,\varnothing,\text{Running,}\Phi_{\text{running}},\varnothing \rangle \}, \) a PSM similar to \( s \) but without the independent marker \( \eta_2 \), and
- \( s''_1 = \{\varnothing,\{\eta_2\},\text{Dummy} : \eta_2,\Phi_{\text{dummy}},\{\langle \eta_2,\langle \text{variant Pong} \rangle \rangle\} \}, \) a new PSM that monitors \( \eta_2 \) and waits for its Pong obligation to be fulfilled.

Therefore, \( \text{Split}(\tilde{\Phi},s) \) returns the set \( \{\langle \tilde{\Phi},s',id,\text{id}\rangle,\langle \tilde{\Phi},s''_1,id,\text{id}\rangle \} \), indicating the program configuration \( \tilde{\Phi} \) must be shown to conform to both \( s' \) and \( s''_1 \).

\textbf{Theorem.} \( \text{Split} \) is a conformance-reflecting transformation.

Appendix L proves this theorem by showing that the original PSM can match the steps of a conforming program configuration by combining the actions taken in the steps from each of the split PSMs.

### 6.5 Unmark

Every PSM returned by \( \text{Split} \) monitors a bounded number of markers (at most one input-monitored marker and at most \( n + 1 \) output-monitored markers, where
n is the number of arguments to the PSM's current state). Because the monitored markers are the only ones that matter for the sake of conformance, all other markers can be removed from the current program configuration \( \hat{K} \), thereby bounding the set of history markers appearing in the resulting configuration \( \hat{K}' \).

This is the purpose of the next transformation, \textit{Unmark}.

The transformation is defined in terms of a more general-purpose function \textit{Remap} that, given some address- and marker-correspondence functions \( A \) and \( M \), performs the renaming/erasure associated with those functions throughout an entire program configuration. We will see later in this chapter that the transformations \textit{Assimilate} and \textit{Canonicalize} are also defined in terms of \textit{Remap}, as is another transformation defined in chapter 8.

The function is largely defined as the obvious component-wise, element-wise application to the components of the given program configuration, with \( \text{Remap}(\hat{a}, A, M) = A(\hat{a}) \) and \( \text{Remap}(H, A, M) = M(H \cap \text{dom}(M)) \). The two exceptions are the actor-behavior map \( \hat{\beta} \) and the message map \( \hat{\mu} \).

Because \( A \) might map two addresses \( \hat{a}_1 \) and \( \hat{a}_2 \) to the same new address \( \hat{a}' \), \textit{Remap} on an actor-behavior map \( \hat{\beta} \) must merge together the behaviors of actors that are mapped to the same address, as follows:

\[
\text{Remap}(\hat{\beta}, A, M) = \begin{cases} 
\varnothing & \text{if } \hat{\mu} = \varnothing \\
\text{Remap}(\hat{\beta'}, A, M) = \langle \hat{a}'@H', \hat{v}' \rangle & \text{if } \hat{\mu} = \hat{\mu}' \cup \{ (\hat{a}@H, \hat{v}) \rightarrow \text{single} \} \\
\text{Remap}(\hat{\mu}, A, M) = \langle \hat{a}'@H', \hat{v}' \rangle \oplus \langle \hat{a}@H', \hat{v}' \rangle & \text{if } \hat{\mu} = \hat{\mu}' \cup \{ (\hat{a}@H, \hat{v}) \rightarrow \text{many} \}
\end{cases}
\]

where \( \hat{a}' = A(\hat{a}), H' = M(H) \), and \( \hat{v}' = \text{Remap}(\hat{v}, A, M) \).

Similarly, \textit{Remap} might remap two messages \( \langle \hat{a}_1@H_1, \hat{v}_1 \rangle \) and \( \langle \hat{a}_2@H_2, \hat{v}_2 \rangle \) to the same new message \( \langle \hat{a}'@H', \hat{v}' \rangle \). To account for this, \textit{Remap} adds each remapped message back to the message map with the \( \oplus \) operation. Messages with an abstract quantity \textit{many} are added twice so that the new message map reflects that quantity.

\[
\text{Remap}(\hat{\mu}, A, M) = \begin{cases} 
\varnothing & \text{if } \hat{\mu} = \varnothing \\
\text{Remap}(\hat{\mu'}, A, M) = \langle \hat{a}@H', \hat{v} \rangle & \text{if } \hat{\mu} = \hat{\mu}' \cup \{ (\hat{a}@H, \hat{v}) \rightarrow \text{single} \} \\
\text{Remap}(\hat{\mu}, A, M) = \langle \hat{a}@H', \hat{v} \rangle \oplus \langle \hat{a}@H', \hat{v} \rangle & \text{if } \hat{\mu} = \hat{\mu}' \cup \{ (\hat{a}@H, \hat{v}) \rightarrow \text{many} \}
\end{cases}
\]

where \( \hat{a}' = A(\hat{a}), H' = M(H), \) and \( \hat{v}' = \text{Remap}(\hat{v}, A, M) \).

As with the abstraction of message maps in chapter 5, the Quasi-Commutativity theorem in appendix I proves that the order in which the messages are added doesn’t matter, so \textit{Remap}(\hat{\mu}, A, M) is well-defined.

The transformation \textit{Unmark} is then defined formally as follows:

\[
\text{Unmark}(\hat{K}, s) = \langle \text{Remap}(\hat{K}, A, M), s, A, M \rangle \\
\text{where } s = \langle H, H', \varphi : \eta, \Phi, O \rangle, \varphi = id, \text{ and } M = id |_{H \cup H'}
\]
6.6. ASSIMILATE

By restricting the domain of \( M \) to just \( H \cup H' \), this transformation effectively removes from \( \hat{K} \) all markers not in \( H \cup H' \). The functions \( A \) and \( M \) are returned to indicate this new correspondence.

**Theorem.** Unmark is a conformance-reflecting transformation.

Appendix L proves this theorem. By the definition of the abstract interpretation, a subset \( H \) of a set of markers \( H' \) is an approximation of \( H' \). Therefore this transformation returns an approximation of the given program configuration. The proof shows that all transformations that simply return an approximation of the given program configuration are conformance-reflecting.

**6.6 Assimilate**

As mentioned in section 6.1, the model checker bounds the number of atomic actors in each encountered program configuration by converting some number of them into collective actors. This conversion process is called **assimilation**. Assimilating an atomic actor involves two steps: merging the actor’s behavior \( \hat{b} \) into the set of behaviors \( \hat{B} \) for the corresponding collective actor, and renaming all uses of that actor’s address \((\text{addr} \ell n)\) to the collective actor’s address \((\text{collective-addr} \ell)\). The resulting program configuration is a sound approximation of the original configuration, because a collective actor represents any number of concrete actors with any of the collective actor’s behaviors. When applied in the context of the rest of the model-checking algorithm, the transformation **Assimilate** leaves at most one atomic actor per spawn location, so the number of atomic actors in a post-Assimilate configuration is bounded by the number of spawn locations in the original program.

As an example, say that an abstract configuration \( \hat{K} \) of the stream-processing program has an atomic processor actor with address \((\text{addr} \ell n)\) in the On state, as well as a collective processor actor with address \((\text{collective-addr} \ell)\) and with behaviors in both the Off and Done states. Assimilating that atomic processor actor would result in a configuration \( \hat{K}' \) in which

- the collective actor at \((\text{collective-addr} \ell)\) has a behavior in the On state as well as the Off and Done states, and
- every occurrence of the address \((\text{addr} \ell n)\) is replaced with \((\text{collective-addr} \ell)\).

**Heuristics**

Generally, given a pair \( \langle \hat{K}, s \rangle \) to assimilate, the only actors not converted into collective actors would ideally be those involved in the protocol specified by \( s \), because proving conformance likely means reasoning precisely about those actors. Determining which actors those are, however, is hard. Certainly the actor associated with the PSM's specified-receptionist marker is relevant for conformance, because that actor receives the input messages specified by the PSM, but that
actor may coordinate with several others to help fulfill its obligations. In general, determining which actors are involved in processing even a single message is non-trivial, and in some cases the number of actors may even be unbounded.

As a result, *Assimilate* resorts to heuristics to approximate the set of actors that are relevant to a given PSM. The key observation is that in the abstract interpretation, executing a handler expression spawns at most one atomic actor per location (spawn expressions inside for/fold loops create only collective actors). Actors spawned in the same summary step are typically involved in handling the same message, so *Assimilate* takes the strategy of either assimilating only newly spawned actors, or assimilating only the old ones, depending on which group seems more relevant to the given PSM.

To formalize the criteria used to decide which set of actors to assimilate, we require the following definitions. Let an atomic actor with address \((addr \ell n)\) be location-unique in a configuration \(\hat{K}\) if there exists no other address \((addr \ell n')\) in \(\hat{K}\) with \(n' \neq n\). Because a location-unique actor is the only actor for its spawn location, there is no need to assimilate it.

A notion of age then distinguishes the non-location-unique actors according to whether they were created as part of the most recent event handler. An age is one of two values: either old or new. For a non-location-unique atomic actor with address \((addr \ell n)\) in a configuration \(\hat{K}\), let that actor's age be old if there exists no address \((addr \ell n')\) in \(\hat{K}\) such that \(n' < n\); else, that actor's age is new. The deterministic semantics for allocating addresses (appendix E) guarantees that addresses for newly spawned actors have a greater identifier \(n\), so this formal definition corresponds to our intuition.

The transformation *Assimilate* uses the following criteria to decide which atomic actors to assimilate in a configuration/PSM pair \(⟨\hat{K}, s⟩\):

1. If \(s\) has a specified-receptionist marker \(η\) and there is a receptionist on \(\hat{K}\) with address \(\hat{a}@H\) such that \(η \in H\), then assimilate the actors with the opposite age as the actor at \(\hat{a}\). That actor receives the messages specified by \(s\), so it is almost certainly relevant for proving conformance.

2. Otherwise, if \(s\) does not have a specified-receptionist marker, let \(A\) be the set of addresses of non-location-unique atomic actors in \(\hat{K}\) whose behavior syntactically contains a specified-external marker from \(s\). This set represents actors that are likely to send messages expected by the current PSM. If \(A\) is non-empty and all actors represented by \(A\) are old, then assimilate all non-location-unique new actors (because none of those actors are likely to send an expected message). Likewise, if \(A\) is non-empty and all actors represented by \(A\) are new, then assimilate all non-location-unique old actors.

3. Otherwise, default to assimilating the non-location-unique old actors in \(\hat{K}\).

These criteria are the result of iterating over multiple possibilities and using them to prove conformance for realistic programs and specifications. There are many possible criteria for deciding which actors to assimilate; the heuristics used
by \textit{Assimilate} are one part of the model-checking algorithm that can be tuned for precision or performance.

\textbf{Formal Definition}

Formally, \textit{Assimilate} is defined as follows:

\begin{align*}
\text{Assimilate}(\hat{K}, s) &= \{(\text{Remap}(\hat{K}, A, id), A, id)\} \\
\text{where } \text{ChooseAssimilationSet}(\hat{K}, s) &= \{(\text{addr } \ell_1 n_1), \ldots, (\text{addr } \ell_m n_m)\} \\
\text{and } A &= \text{id}\left[\text{(addr } \ell_i n_i) \rightarrow (\text{collective-addr } \ell_i)\right]_{i \in 1\ldots m}
\end{align*}

Within the definition, the function \textit{ChooseAssimilationSet} implements the criteria described above and selects the addresses of actors to be assimilated. The call to \textit{Remap} then substitutes all uses of each chosen address for its corresponding collective address. \textit{Remap} also merges the behavior of each assimilated actor into the corresponding collective actor (or creates the actor if it does not exist, as defined in section 6.5).

\textit{Assimilate} also returns the mapping \(A\) from the address of each actor in \(\hat{K}\) to the address of its corresponding actor in the eventual returned configuration \(\text{Remap}(\hat{K}, A, id)\). Specifically, each assimilated actor’s original address \(\hat{a}\) maps to its new collective address \(\hat{a}'\), and all other addresses in the configuration map to themselves. \textit{Assimilate} does not change the markers in the configuration, so the identity function acts as the marker-correspondence function.

\textbf{Theorem.} \textit{Assimilate} is a conformance-reflecting transformation.

The proof of this theorem can be found in appendix L. As with \textit{Unmark}, it relies on the fact that \textit{Assimilate} yields a program configuration that approximates the given one, so conformance for the assimilated configuration implies conformance for the non-assimilated one.

\section{6.7 Canonicalize}

After dividing up each PSM with \textit{Split}, removing all unused markers with \textit{Unmark}, and assimilating atomic actors with \textit{Assimilate}, the set of program-configuration/PSM pairs explored during a run of the model-checker is almost bounded. In fact, when considering equivalence up to the renaming of addresses and history markers, the number of equivalence classes of those pairs is bounded. Addresses and markers are both uninterpreted tokens, so conformance for any one pair in such an equivalence class implies conformance for all pairs in the class. Therefore, the last step is to transform each pair into a canonical representation of its equivalence class through some renaming of its addresses and markers. Effectively, this is a kind of symmetry reduction [110], a common state-space-reduction technique in model checking.

Many possible renaming schemes are possible; the one chosen here is as follows: For addresses, each location-unique addresses \((\text{addr } \ell n)\) is renamed to
(addr ℓ 0); all other addresses are left unchanged. The model checker applies this transformation immediately after Assimilate, so there are no non-location-unique atomic actors at that point, and thus it is unnecessary to give canonical addresses to those actors.

The following function creates an address substitution encoding this scheme:

\[
\text{CanonicalAddrSubst}(\hat{K})(\hat{a}) = \begin{cases} 
\text{addr ℓ 0} & \text{if } \exists n. \hat{a} = (\text{addr ℓ } n) \text{ and } \hat{a} \text{ is location-unique in } \hat{K} \\
\hat{a} & \text{otherwise}
\end{cases}
\]

For example, let \( \hat{K} \) and \( \hat{K}' \) be two abstract configurations of the stream-processing program from chapter 2 that differ only in the address of the lone processor actor in each configuration: the actor in \( \hat{K} \) has address \( \hat{a} = (\text{addr ℓ } 3) \), whereas the corresponding atomic actor in \( \hat{K}' \) has address \( \hat{a}' = (\text{addr ℓ } 7) \). Then \( \text{CanonicalAddrSubst}(\hat{K})(\hat{a}) = \text{CanonicalAddrSubst}(\hat{K}')(\hat{a}') = (\text{addr ℓ } 0) \). Both addresses are renamed to the same new address.

To define a canonical naming scheme for history markers, we need a way to name specific markers to which Canonicalize can rename the existing markers in a pair \( \langle \hat{K}, s \rangle \). One easy way to do this is by mapping the natural numbers to markers, where 0 maps to the minimal marker out of the entire set of markers HistMark, 1 maps to the next minimal marker, and so on. The function \( \lceil_\sim \rceil \) captures this idea.

\[
\lceil_\sim n \rceil = \min(\text{HistMark} - \{\lceil_\sim 0 \rceil, \ldots, \lceil_\sim n - 1 \rceil})
\]

Each history marker \( \eta \) in a pair \( \langle \hat{K}, s \rangle \) is renamed using the following scheme:

1. If \( \eta \) is the input-monitored marker for \( s \), then rename it to \( \lceil_\sim 0 \rceil \).

2. Else, if \( \eta \) is the \( n \)th argument to the PSM’s current state (and \( \eta \) is not also the \( m \)th argument for some \( m < n \)), then rename it to \( \lceil_\sim n \rceil \).

3. Otherwise, let \( H \) be the set of markers matched by the previous two steps, plus all markers \( \eta' \) such that \( \eta' < \eta \). Then rename \( \eta \) to \( \lceil_\sim |H| + 1 \rceil \).

For a renaming-equivalence class of \( \langle \hat{K}, s \rangle \) pairs that each have at most one marker that is neither an input-monitored marker nor an argument to the PSM’s current state, this scheme renames gives canonical names to each marker. The input-monitored marker (if present) maps to \( \lceil_\sim 0 \rceil \), the \( n \) state arguments map to markers \( \lceil_\sim 1 \rceil \) through \( \lceil_\sim n \rceil \), and the remaining marker maps to \( \lceil_\sim n + 1 \rceil \). Any program-configuration/PSM pair with more markers than that is never explored by the model checker (as explained in section 6.4). For those cases, the third rule above merely ensures that no two markers are renamed to the same thing.

\text{CanonicalMarkerSubst} defines a substitution following this scheme, as fol-
lows:

\[
\text{CanonicalMarkerSubst}\left(\left\langle \left( \tilde{\beta} \left| H \right) \right]\right), \left\langle H', H'', \varphi : \eta'_1 \ldots \eta'_n, \overline{\Phi}, O \right\rangle\right)(\eta) =
\begin{cases}
\tilde{\beta} & \text{if } \eta \in H' \\
\tilde{\beta}i & \text{if } 1 \leq i \leq n, \eta = \eta'_i, \text{ and } \exists j < i. \eta = \eta'_j \\
\tilde{\beta}m + 1 & \text{otherwise, where } m = |H' \cup \{\eta'_1, \ldots, \eta'_n\} \cup \{\eta'' | \eta'' < \eta}| 
\end{cases}
\]

As an example, let \( s \) be \( \{\eta_4\}, \{\eta_5, \eta_1, \eta_3\}, \varphi : \eta_5 \eta_3, \overline{\Phi}, \{\langle \eta_1, \text{self-addr} \rangle\} \) and let \( s' \) be \( \{\eta_2\}, \{\eta_8, \eta_6, \eta_4\}, \varphi : \eta_8 \eta_4, \overline{\Phi}, \{\langle \eta_6, \text{self-addr} \rangle\} \), where \( \eta_i = \tilde{\beta}i \) for all \( i \). The two PSMs are equivalent up to the renaming of their markers. Then for any two program configurations \( \tilde{K} \) and \( \tilde{K}' \) that contain only the markers in \( s \) and \( s' \), respectively, \( \text{CanonicalMarkerSubst}(\tilde{K}, s)(s) \) and \( \text{CanonicalMarkerSubst}(\tilde{K}', s')(s') \) both yield the same PSM, \( \{\eta_0\}, \{\eta_1, \eta_2, \eta_3\}, \varphi : \eta_1 \eta_2, \overline{\Phi}, \{\langle \eta_3, \text{self-addr} \rangle\} \).

The transformation \( \text{Canonicalize} \) is then defined as merely the application of those substitutions to the components of the given pair, as follows:

\[
\text{Canonicalize}(\tilde{K}, s) = \{\langle \text{Remap}(\tilde{K}, A, M(s), A, M) \rangle\}
\]

where \( A = \text{CanonicalAddrSubst}(\tilde{K}) \) and \( M = \text{CanonicalMarkerSubst}(\tilde{K}, s) \).

**Theorem.** \( \text{Canonicalize} \) is a conformance-reflecting transformation.

Appendix L proves this theorem by showing that the “same” transition steps are possible from the given program configuration and PSM both before and after applying the substitution, modulo the renaming of addresses and markers.
Chapter 7

A Model-Checking Algorithm for APS

The benefit of transformation conformance is that when using the right transformation, a conformance proof requires exploring only a finite subset of a program’s state space. This state-space reduction in turn enables one of the contributions of this dissertation: an algorithm that automatically checks whether a CSA program conforms to an APS specification. The algorithm is sound, but not complete: there are program/specification pairs that conform, but for which this algorithm will fail to find a conformance proof. A section at the end of this chapter lists some of the algorithm’s heuristics that users can tune to trade off precision/performance.

The following sections explain the algorithm and define its individual parts. Throughout, the specific transformation \( T \) used is left unspecified. The most basic form that still guarantees termination is the composed transformation \( \text{Canonicalize} \circ \text{Assimilate} \circ \text{Unmark} \circ \text{Split} \), but the next chapter introduces additional transformations that may be added as optimizations. Section 7.7 states the two key theoretical results of this algorithm: that it always terminates, and that its results are a sound approximation of conformance, as defined in chapter 3.

7.1 Finding a Simulation Relation

Given a CSA program \( P \) and an APS specification \( \Sigma \), the goal of the ModelCheck algorithm is to determine whether \( P \vdash \Sigma \). According to the theorems defined in the previous several chapters, it is sufficient to determine merely whether \( P \) transformation-conforms to \( \Sigma \), so ModelCheck attempts to find some transformation-conformance-dense relation \( R \) and strategy \( Z \) such that \( P \equiv_{TR} \Sigma \).

The main obstacle in automating such a search is to find, for all \( \langle \hat{K}, s \rangle \in R \) and all summary transitions \( \hat{K} \xrightarrow{L} \hat{K}' \), at least one corresponding summary transition \( \langle L, O \rangle \xrightarrow{(L,O)} S \) of the specification. In other words, the main obstacle is to construct
an appropriate strategy $Z$. In the manual conformance proof from chapter 4, we relied on our intuitive understanding of the programmer’s intent for this purpose, but an algorithm has no such intuition. Instead, ModelCheck adapts Henzinger et al.’s algorithm for finding simulation relations [57], defined below.

Consider the basic definition of a simulation relation $R$ between processes $P$ and $Q$, with $l$ ranging over transition labels:

$$\text{For all } \langle P, Q \rangle \in R \text{ and all } l \text{ and } P' \text{ such that } P \xrightarrow{l} P', \text{ there exists some } Q' \text{ such that } Q \xrightarrow{l} Q' \text{ and } \langle P', Q' \rangle \in R.$$  

The key insight of the Henzinger et al. approach is to break this definition into a local part (every step $P \xrightarrow{l} P'$ has some similarly labeled step $Q \xrightarrow{l} Q'$) and a non-local part (the pair $(P', Q')$ reached by taking those steps is also in $R$). Then finding a simulation relation $R$ is a two-step process: first construct some over-approximation $R'$ of $R$ that satisfies just the local part of the definition, then remove from $R'$ all pairs that violate the non-local part. These two steps are described below.

**Step 1**

As an example, consider the two labeled transition systems in figure 7.1, each representing a process with transitions labeled with some label $l$ (the particular label is irrelevant for this example). The process $Q_1$ can simulate any transition sequence from $P_1$ by taking its right-hand path through $Q_3$ and $Q_5$. If $Q_1$ instead attempted to simulate a sequence such as $P_1 \xrightarrow{l} P_2 \xrightarrow{l} P_3 \xrightarrow{l} P_3$ with its left-hand path, however, it would fail after two transitions, because there is no transition from $Q_4$.

To show that $Q_1$ simulates $P_1$, the Henzinger et al. algorithm first builds the initial over-approximation $R'$ for these two systems by exploring the state space starting from $\langle P_1, Q_1 \rangle$. For every pair $\langle P, Q \rangle$ it explores, and for every step $P \xrightarrow{l} P'$ from $P$, the algorithm finds every step $Q \xrightarrow{l} Q'$ with a matching label and
7.1. FINDING A SIMULATION RELATION

<table>
<thead>
<tr>
<th>P Transition</th>
<th>Corresponding Q</th>
<th>Matching Q Transition(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 \xrightarrow{l} P_2 )</td>
<td>( Q_1 )</td>
<td>( Q_1 \xrightarrow{l} Q_2, Q_1 \xrightarrow{l} Q_3 )</td>
</tr>
<tr>
<td>( P_2 \xrightarrow{l} P_3 )</td>
<td>( Q_2 )</td>
<td>( Q_2 \xrightarrow{l} Q_4 )</td>
</tr>
<tr>
<td>( P_2 \xrightarrow{l} P_3 )</td>
<td>( Q_3 )</td>
<td>( Q_3 \xrightarrow{l} Q_5 )</td>
</tr>
<tr>
<td>( P_3 \xrightarrow{l} P_3 )</td>
<td>( Q_5 )</td>
<td>( Q_5 \xrightarrow{l} Q_5 )</td>
</tr>
</tbody>
</table>

Relation: \( \langle P_1, Q_1 \rangle, \langle P_2, Q_2 \rangle, \langle P_2, Q_3 \rangle, \langle P_3, Q_5 \rangle \)

Frontier: \( \{ Q_4 \} \)

Figure 7.2: Strategy, relation, and frontier after initial state-space exploration

adds \( \langle P', Q' \rangle \) to its list of pairs to explore. If every step from \( P \) has at least one step from \( Q \) that simulates it, then the algorithm adds \( \langle P, Q \rangle \) to \( R' \).

While exploring the state space, the algorithm produces two byproducts in addition to the over-approximation relation \( R' \). The first is a strategy that records all of the matching \( Q \xrightarrow{l} Q' \) steps for each \( P \xrightarrow{l} P' \). The second is the set of pairs \( \langle P', Q' \rangle \in R' \) such that some pair \( \langle P, Q \rangle \in R' \) can reach it by matching steps \( P \xrightarrow{l} P' \) and \( Q \xrightarrow{l} Q' \). This set is called the frontier: it represents the transitions from members of \( R' \) that violate the non-local part of the simulation relation.

Figure 7.2 shows the strategy, relation, and frontier resulting from that exploration. The strategy is shown as a table: for instance, the first row shows that \( Q_1 \) can match the transition \( P_1 \xrightarrow{l} P_2 \) by taking either the transition \( Q_1 \xrightarrow{l} Q_2 \) or \( Q_1 \xrightarrow{l} Q_3 \). Notice that for every pair \( \langle P, Q \rangle \) in the relation and every step \( P \xrightarrow{l} P' \), the strategy contains at least one matching step \( Q \xrightarrow{l} Q' \). Notice also that the pair \( \langle P_3, Q_4 \rangle \) is in the frontier rather than the main relation, because \( Q_4 \) has no way to match the step \( P_3 \xrightarrow{l} P_3 \).

**Step 2**

Having found a relation \( R' \) that satisfies the local part of the definition, the second step of the algorithm is to enforce the non-local part: every matching step \( Q \xrightarrow{l} Q' \) for a step \( P \xrightarrow{l} P' \) must lead to a pair in \( R' \). The frontier tells us exactly where this does not happen, so the idea is to remove from the strategy every step that leads to a pair in the frontier. In the example, this would be the transition \( Q_2 \xrightarrow{l} Q_4 \) used to match \( P_2 \xrightarrow{l} P_3 \) (as highlighted in figure 7.2), because taking those transitions leads to a pair \( \langle P_3, Q_4 \rangle \) in the frontier.

After removing that step, if the strategy had some additional way to match the step \( P_2 \xrightarrow{l} P_3 \) matched by \( Q_2 \xrightarrow{l} Q_4 \), then nothing else would have to happen:

\[^1\text{Although a strategy is a total function, the table shows only those entries with a non-empty set of matching steps.}\]
CHAPTER 7. A MODEL-CHECKING ALGORITHM FOR APS

<table>
<thead>
<tr>
<th>Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$ Transition</td>
</tr>
<tr>
<td>$P_1 \xleftarrow{l} P_2$</td>
</tr>
<tr>
<td>$P_2 \xleftarrow{l} P_3$</td>
</tr>
<tr>
<td>$P_3 \xleftarrow{l} P_3$</td>
</tr>
</tbody>
</table>

Relation: $\langle P_1, Q_1 \rangle, \langle P_2, Q_3 \rangle, \langle P_3, Q_5 \rangle$
Frontier: $\{ P_2, Q_2 \}$

Figure 7.3: Strategy, relation, and frontier after removing transitions to $\langle P_3, Q_4 \rangle$

<table>
<thead>
<tr>
<th>Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$ Transition</td>
</tr>
<tr>
<td>$P_1 \xleftarrow{l} P_2$</td>
</tr>
<tr>
<td>$P_2 \xleftarrow{l} P_3$</td>
</tr>
<tr>
<td>$P_3 \xleftarrow{l} P_3$</td>
</tr>
</tbody>
</table>

Relation: $\langle P_1, Q_1 \rangle, \langle P_2, Q_3 \rangle, \langle P_3, Q_5 \rangle$
Frontier: $\emptyset$

Figure 7.4: Final strategy, relation, and frontier

Each step from $P_2$ just needs a single matching transition from $Q_2$. In this case, however, removing $Q_2 \xleftarrow{l} Q_4$ leaves the strategy without any means to match $P_2 \xleftarrow{l} P_3$ with a step from $Q_2$. That means that $\langle P_2, Q_2 \rangle$ violates the non-local part of the definition, so the algorithm therefore removes it from $R'$ and adds it to the frontier. Generally, the matching steps from each $Q$ form a logical “or”, in that only one is necessary for each step from the corresponding $P$, whereas the steps from each $P$ form a logical “and”, in that every step must have a match from the corresponding $Q$. Figure 7.3 shows the state after this removal.

Because removing $\langle P_2, Q_2 \rangle$ results in adding it to the frontier, the algorithm must also remove steps to that pair from the strategy, too. This recursive removal process continues until the strategy contains no steps that lead to the frontier. Once all such steps are gone, whatever remains in $R'$ must satisfy the full definition for a simulation relation.

In the example, the algorithm continues by removing $Q_1 \xleftarrow{l} Q_2$ as a way to match $P_1 \xleftarrow{l} P_2$. Unlike the previous case, removing that transition does not remove the predecessor pair $\langle P_1, Q_1 \rangle$ from the relation, because $Q_1$ has another transition that can match $P_1 \xleftarrow{l} P_2$. Figure 7.4 shows the final state after that removal. The frontier is empty, so the remaining pairs $\langle P_1, Q_1 \rangle$, $\langle P_2, Q_3 \rangle$, and $\langle P_3, Q_5 \rangle$ form a simulation relation $R$, with the strategy in figure 7.4 acting as witness. Furthermore, $\langle P_1, Q_1 \rangle \in R$, so $Q_1$ simulates $P_1$. 
7.2. MODELCHECK

Adaption for APS Model-Checking

The ModelCheck algorithm in section 7.2 adapts this algorithm to find a relation that respects the Simulation condition for transformation conformance. Once such a relation is found, dealing with the Fulfillment condition for transformation conformance is relatively straightforward; section 7.2 discusses this further.

7.2 ModelCheck

7.2.1 Preliminary Definitions

Before describing the model-checking algorithm itself, we need some formal definitions to describe its intermediate results. First, the initial state-space exploration produces a relation that satisfies the local part of the Simulation rule: for each pair \( (\hat{K}, s) \in R \), the strategy provides a simulating step from the PSM \( s \) for any summary transition from \( \hat{K} \) (but the reached pairs might not be in \( R \)). This is called a local-transformation-simulation relation, defined as follows:

**Definition.** A relation \( R \) is a local-transformation-simulation relation if and only if there exists a strategy \( Z \) such that for all \( (\hat{K}, s) \in R \), all \( \hat{L}_1, \ldots, \hat{L}_n \), and all \( \hat{K}' \), if \( \hat{K} \xrightarrow{\hat{L}_1, \ldots, \hat{L}_n} \hat{K}' \) and \( \hat{K} \) is not stuck, then there exists some \( L \) such that \( \hat{K} \xrightarrow{L} \hat{K}' \) summarizes \( \hat{K} \xrightarrow{\hat{L}_1, \ldots, \hat{L}_n} \hat{K}' \) and \( Z(\hat{K} \xrightarrow{L} \hat{K}', s) \neq \emptyset \).

A strategy \( Z \) that satisfies the above conditions is a witness to such a local-transformation-simulation relation \( R \).

The notion of the frontier is then defined with respect to a local-transformation-simulation-relation.

**Definition.** A frontier with respect to a transformation \( T \) for a local-transformation-simulation relation \( R \) witnessed by \( Z \) is a set \( F \) of program-configuration/PSM pairs not in \( R \) such that for all transformation-step transitions \( (\hat{K}, s) \xrightarrow{L, \hat{K}', Z, O, S, T, A, M} (\hat{K}', s') \) from a pair \( (\hat{K}, s) \in R \), if \( (\hat{K}', s') \in R \cup F \).

The transformation \( T \) is often implied by context and therefore omitted. The frontier is defined so that it may include “extra” pairs that are not reachable from any pair in \( R \); this simplifies some of the correctness proofs in appendix M.

Finally, the result of the Henzinger et al. algorithm is a relation that satisfies the Simulation rule for a conformance-dense relation (but possibly not the Fulfillment rule). Such a relation is called a transformation-simulation relation, defined as follows:

**Definition.** A relation \( R \) is a transformation-simulation relation if and only if there exist a strategy \( Z \) and a conformance-reflecting transformation \( T \) such that for all \( (\hat{K}, s) \in R \), all \( \hat{L}_1, \ldots, \hat{L}_n \), and all \( \hat{K}' \), if \( \hat{K} \xrightarrow{\hat{L}_1, \ldots, \hat{L}_n} \hat{K}' \) and \( \hat{K} \) is not stuck, there exists some \( L \) such that

- \( \hat{K} \xrightarrow{L} \hat{K}' \) summarizes \( \hat{K} \xrightarrow{\hat{L}_1, \ldots, \hat{L}_n} \hat{K}' \),
• \( Z(\hat{K} \xrightarrow{L} \hat{K}', s) \neq \emptyset \), and
• for all transformation steps \( (\hat{K}, s) \xrightarrow{L, Z, O, S, T, A, M} (\hat{K}'', s') \in R \).

### 7.2.2 The Algorithm

Figure 7.5 defines the algorithm’s top-level function, called \( \text{ModelCheck} \), which takes as input a CSA program \( P \) and an APS specification \( \Sigma \) to check for conformance. In all algorithms in this dissertation, the syntax \( \text{let } x := y \) creates a new variable \( x \) in the current scope bound to the value of \( y \), while the \( := \) operator on its own mutates an existing variable. The \( \text{let} \) keyword may also use pattern-matching to name and select fields from tuples; the function described by the algorithm is undefined if such a pattern-match fails.

```plaintext
1 Function \( \text{ModelCheck}(P, \Sigma) = \)
2    if there exists no \( (\hat{K}, S) \) instantiable from \( P \) and \( \Sigma \) then
3        return false;
4    end
5    let \( (\hat{K}_{\text{init}}, \{s_{\text{init}}\}) := \text{MaxInstantiate}(P, \Sigma); \)
6    let \( \hat{K}_{\text{init}} := |\hat{K}_{\text{init}}|; \)
7    let \( (R_{\text{loc}}, Z_{\text{loc}}, F_{\text{loc}}) := \text{Explore}(\hat{K}_{\text{init}}, s_{\text{init}}); \)
8    let \( (R_{\text{sim}}, Z_{\text{sim}}) := \text{Prune}(R_{\text{loc}}, Z_{\text{loc}}, F_{\text{loc}}); \)
9    let \( R_{\text{fulfill}} := \text{FindFulfillingPairs}(R_{\text{sim}}, Z_{\text{sim}}); \)
10   let \( (R_{\text{conf}}, Z_{\text{conf}}) := \text{Prune}(R_{\text{fulfill}}, Z_{\text{sim}}, R_{\text{sim}} - R_{\text{fulfill}}); \)
11   return \( (\hat{K}_{\text{init}}, s_{\text{init}}) \in R_{\text{conf}}; \)
12 end
```

Figure 7.5: The top-level algorithm for the model checker

To start, the algorithm checks whether there is some way to instantiate the program and specification into corresponding initial configurations (this might fail if, for example, the specification monitors a receptionist or external that the program does not have). If not, then the algorithm returns \( false \), indicating that it is unable to prove conformance. Otherwise, it defines \( (\hat{K}_{\text{init}}, \{s_{\text{init}}\}) \) as a maximal instantiation of \( P \) and \( \Sigma \) (see section 3.6.2). Then it abstracts the initial concrete program configuration \( \hat{K}_{\text{init}} \) to \( \hat{K}_{\text{init}} \). The goal from this point is to find some transformation-conformance-dense relation \( R \) and strategy \( Z \) that prove \( \hat{K}_{\text{init}} \models_{TR} s_{\text{init}} \).

The function \( \text{Explore} \) (defined in section 7.3) performs the initial state-space-exploration portion of the Henzinger et al. algorithm, starting from the pair \( (\hat{K}_{\text{init}}, s_{\text{init}}) \). The function returns

- a local-transformation-simulation relation \( R_{\text{loc}} \),
- a strategy \( Z_{\text{loc}} \) that witnesses \( R_{\text{loc}} \), and
7.3. EXPLORE

The job of Explore is to explore the set of pairs \( (K,s) \) reachable from the given initial pair \( (K_{init},s_{init}) \) via transformation-step transitions and determine how to match each step of the program with some step of the specification. This results in a local-transformation-simulation relation \( R_{tloc} \), a witnessing redundant strategy \( Z_{tloc} \), and the frontier \( F_{tloc} \) for that relation.

Figure 7.6 lists the algorithm. It is structured as a worklist algorithm, where each item in the worklist \( W \) is a program-configuration/PSM pair \( (K,s) \) queued up for exploration. The worklist starts with just the given pair \( (K_{init},s_{init}) \). Initially, the strategy \( Z \) does not provide a matching specification step for any program step.

To process a pair \( (K,s) \) in the worklist, Explore first uses ShouldExplore (defined in figure 7.7) to determine whether the pair is worth exploring. As mentioned in sections 6.1 and 6.4, the model-checking algorithm does not attempt to explore pairs in which the PSM either contains multiple copies of an obligation, or output-monitors more than \( n + 1 \) markers (where \( n \) is the number of parameters for the PSM's current state). ShouldExplore returns false for any such PSM, in which case Explore immediately adds \( (K,s) \) to the frontier without exploring its possible transitions and moves on to the next worklist item.
Function `Explore(\(\hat{K}_{\text{init}}, s_{\text{init}}\))` =

let \(R := \emptyset\);

let \(Z := \) a function such that \(Z(\hat{K} \xrightarrow{L} \hat{K}', s) = \emptyset\) for all \(\hat{K} \xrightarrow{L} \hat{K}'\) and \(s\);

let \(F := \emptyset\);

let \(W := \{\langle \hat{K}_{\text{init}}, s_{\text{init}} \rangle\}\);

while \(W \neq \emptyset\) do

let \(\langle \hat{K}, s \rangle := \) an arbitrary item from \(W\);

\(W := W \setminus \{\langle \hat{K}, s \rangle\}\);

unless ShouldExplore(s)

\(F := F \cup \{\langle \hat{K}, s \rangle\}\);

continue;

end

let nextSteps := ProgSteps(\(\hat{K}\));

let foundUnmatchableStep := false;

for \(\langle L, \hat{K}' \rangle\) in nextSteps do

\(Z := Z[\langle \hat{K} \xrightarrow{L} \hat{K}', s \rangle \rightarrow \text{MatchingSpecSteps}(s, L)]\);

if \(Z(\hat{K} \xrightarrow{L} \hat{K}', s) = \emptyset\) then

foundUnmatchableStep := true;

break;

end

end

if foundUnmatchableStep then

\(F := F \cup \{\langle \hat{K}, s \rangle\}\);

else

\(R := R \cup \{\langle \hat{K}, s \rangle\}\);

for \(\langle L, \hat{K}' \rangle\) in nextSteps do

for \(\langle O, S \rangle \in Z(\hat{K} \xrightarrow{L} \hat{K}', s)\) do

for \(s' \in S\) do

for \(\langle \hat{K}''', s'', A, M \rangle \in T(\hat{K}', s')\) do

if \(\langle \hat{K}''', s'' \rangle \in R \cup F\) then

\(W := W \cup \{\langle \hat{K}''', s'' \rangle\}\);

end

end

end

end

end

end

return \(\langle R, Z, F \rangle\);

end

Figure 7.6: The `Explore` algorithm
7.3. EXPLORE

\[
\text{ShouldExplore}(⟨H,H',\varphi: \eta_1...\eta_n, \Phi, O⟩) =
\begin{cases}
\text{true} & \text{if every member } ⟨\eta', po⟩ \text{ of } O \text{ is unique in } O \text{ and } |H'| \leq n + 1 \\
\text{false} & \text{otherwise}
\end{cases}
\]

Figure 7.7: Conditions defining whether Explore should explore a given PSM

For all other pairs, the function ProgSteps finds a set of program summary steps that collectively summarize the possible (abstract) executions of every handler expression enabled in \(\hat{K} \). ProgSteps is a straightforward implementation of the abstract interpretation from chapter 5, not shown here. Its results are a set \(\{⟨L_1, \hat{K}'_1⟩,...,⟨L_n, \hat{K}'_n⟩\}\), such that \(\hat{K} \xrightarrow{L_i} \hat{K}'_i\) for all \(i \in 1...n\).

The algorithm's next step is to show that \(⟨\hat{K}, s⟩\) is a member of a local-transformation-simulation relation by finding the possible ways that the PSM \(s\) can simulate each of those steps from ProgSteps. This is the job of MatchingSpecSteps (described in section 7.4). Because it uses heuristics, MatchingSpecSteps may not be able to find all possible matching specification summary steps; its results instead represent a “best effort” to find matches.

Once matching steps are found, Explore records them as part of the strategy \(Z\). If there is some step \(\hat{K} \xrightarrow{L} \hat{K}'\) for which MatchingSpecSteps is unable to find any matching step from \(s\), then Explore records that fact and breaks out of the loop, ignoring the other steps to match.

Whenever the algorithm finds some program summary step from \(\hat{K}\) that MatchingSpecSteps is unable to match, it adds \(⟨\hat{K}, s⟩\) to the frontier \(F\), because it is unable to prove \(⟨\hat{K}, s⟩\) satisfies the condition for a local-transformation-simulation relation. Otherwise, \(⟨\hat{K}, s⟩\) does satisfy that condition, so the algorithm adds \(⟨\hat{K}, s⟩\) to \(R\). Then Explore must explore all program-configuration/PSM pairs reachable from this pair by a transformation-step transition, so the program:

1. finds all pairs \(⟨\hat{K}', s'⟩\) reachable by taking one of the steps from \(\hat{K}'\) along with one of the matching steps from \(s'\) recorded in \(Z\),

2. applies the transformation \(T\) to each \(⟨\hat{K}', s'⟩\) to get a set of tuples of the form \(⟨\hat{K}''', s''', A, M⟩\) (the bold typesetting for \(T\) indicates it is an implicit parameter of the algorithm), and

3. adds each \(⟨\hat{K}''', s'''⟩\) to the worklist \(W\) if that pair has not already been encountered. (The functions \(A\) and \(M\) are unimportant here; section 7.6 describes how FindFulfillingPairs uses them later on.)

Once the worklist is empty, every explored pair has been added to either \(R\) or \(F\), and the strategy \(Z\) provides at least one matching step for every program summary step from a member of \(R\). Therefore \(R\) is a local-transformation-simulation relation with witness \(Z\) and frontier \(F\), so the algorithm returns \(⟨R, Z, F⟩\).

The Explore algorithm is where all of the state-space reductions from the previous two chapters pay off. The transformations Split, Unmark, Assimilate, and
Canonicalize (applied in step 2 above as part of the transformation $T$) coupled with the conditions for ShouldExplore ensure that this algorithm explores only a bounded number of abstract-program-configuration/PSM pairs. By the definition of $\text{ProgSteps}$ and $\text{MatchingSpecSteps}$, the number of items added to the worklist is also bounded, so Explore eventually terminates. Appendix N works out this proof in detail.

### 7.4 MatchingSpecSteps

As mentioned in the previous section, the purpose of the algorithm $\text{MatchingSpecSteps}$ is to find a number of specification summary steps $S \xrightarrow{L,O} S'$ each showing how a specification configuration $S$ containing just some given PSM $s$ can simulate a given program summary step $\hat{K} \xrightarrow{L} \hat{K}'$. Explore then uses these steps as the initial strategy.

Ideally, $\text{MatchingSpecSteps}$ would return all steps from $S$ that simulate $\hat{K} \xrightarrow{L} \hat{K}'$. If the program being checked does in fact conform to the given specification, then searching through all possible steps would give the best chance for proving conformance. Unfortunately, this is not feasible in an algorithmic setting, because the number of matching summary steps from $s$ may be infinite.

For example, consider the following ping/pong specification from chapter 3. It says that a Ping sent to the monitored receptionist $s1$ causes the actor to send a Pong response to $d$, but that other unobserved events (represented by the free transition) can also cause a Pong response.

```
(specification
 (mon-receptionist s1)
 (mon-externals dest)
 (goto Running dest)
 (define-state (Running d)
   [(variant Ping) ->
    (obligation d (variant Pong))
    (goto Running d)]
   [free ->
    (obligation d (variant Pong))
    (goto Running d)])
```

How can a PSM $s$ in the Running state simulate a summary step $\hat{K} \xrightarrow{L} \hat{K}'$ in which the program receives a Ping at $s1$ and responds with a Pong to $d$? There are multiple possibilities.

1. The PSM $s$ takes the Ping transition and simulates the Pong output with the incurred obligation.

2. The PSM $s$ first takes the free transition, then takes the Ping transition and simulates the Pong output with the incurred obligation.
3. The PSM \( s \) takes two free transitions, then takes the Ping transition and simulates the Pong output with the incurred obligation.

4. (and so on)

The first possibility is the obvious choice, but in general there are infinitely many ways for \( s \) to simulate \( \tilde{K} \xrightarrow{L} \tilde{K}' \).

As a result, MatchingSpecSteps instead returns only a finite subset of the possible matching steps. The steps are selected based on heuristics that determine whether a step is likely to match the programmer’s intent. For example, in the ping/pong example above, MatchingSpecSteps might select only the step that takes just the Ping transition, because it seems unlikely that receiving a single Ping should result in more than one obligation to send a Pong message.

The criteria that make a step likely to match the programmer’s intent are open to interpretation; the implementation of MatchingSpecSteps can be changed to suit different purposes such as improved precision or performance. The specific heuristics described below were developed initially by human insight and later refined by experiments.

### 7.4.1 Reducing the Number of Summarized Sequences

To understand the approach, let’s first review what it means for a specification summary step \( S \xrightarrow{(L,O)} S' \) to simulate a program summary step \( \tilde{K} \xrightarrow{L} \tilde{K}' \). Recall from section 5.5 that a summary label \( L \) summarizes any sequence of labels \( \hat{l}_1, \ldots, \hat{l}_n \) where \( L = \langle \hat{l}_1, \hat{\mu} \rangle \) and where the messages in the abstract message map \( \hat{\mu} \) summarize the outputs of that sequence. Furthermore, recall from section 5.5 that a specification summary step \( S_1 \xrightarrow{(L,O)} S_{n+1} \) is possible if and only if \( S_1 \) can transition with every label sequence summarized by \( L \), and \( O \) is a lower bound on the fulfilled obligations in all such sequences. Thus, a matching specification summary step \( S \xrightarrow{(L,O)} S' \) for \( \tilde{K} \xrightarrow{L} \tilde{K}' \) where \( L = \langle \hat{l}_1, \hat{\mu} \rangle \) is one that describes a set of specification transition sequences that each

1. start with \( l_1 \), and
2. have outputs summarized by \( \hat{\mu} \).

Fortunately, MatchingSpecSteps need not reason about every sequence of labels summarized by \( L \) individually, for two reasons. First, the required simulation is a weak simulation, so MatchingSpecSteps can ignore all of the program’s internal steps (i.e., those labeled with some \( \hat{\ell}_i \) such that \( \lfloor \hat{\ell}_i \rfloor = \bullet \)). Second, the relative order of the sent message is irrelevant. APS was designed to allow obligations to be fulfilled in any order, so it turns out that all specification steps not involving input commute with each other. The following theorem formalizes this fact:

**Theorem** (Abstract Specification Step Commutativity). For all \( S_1, S_2, S_3, \lambda_1, \lambda_2, O_1, \) and \( O_2 \), if \( S_1 \xrightarrow{\lambda_1, O_1} S_2 \xrightarrow{\lambda_2, O_2} S_3 \), and there do not exist \( \hat{a}, H \), and \( \hat{e} \) such...
that $\lambda_1 = \hat{a}@H?\hat{v}$ for any $i \in \{1, 2\}$, then there exists $S'_2$ such that $S_1$ \xrightarrow{\lambda_2, O_2} S'_2 \xrightarrow{\lambda_1, O_1} S_3$.

**Proof.** Similar to the proof of the Specification Step Commutativity lemma in appendix D. This theorem is effectively a weaker version of that lemma, applied to abstract steps rather than concrete ones.\(^2\) The abstract/concrete distinction is irrelevant for the proof, however. \(\square\)

Thus, to find simulating steps for a step $\hat{K} \xrightarrow{(\hat{l}, \hat{\mu})} \hat{K}'$, it is sufficient to choose some arbitrary ordering of the outputs in $\hat{\mu}$, then find steps that can simulate that sequence of outputs (after simulating the initial $\hat{l}$ step). The only caveat is that for every message $\langle \hat{a}@H, \hat{v} \rangle$ such that $\hat{\mu}(\hat{a}@H, \hat{v}) = \text{many}$, each simulating sequence must be able to handle any number of corresponding outputs. Section 7.4.3 describes how to handle this issue.

### 7.4.2 The Algorithm

Figure 7.8 lists the `MatchingSpecSteps` algorithm itself. It takes as input a single PSM $s$, and the label $L$ of a program summary step that a specification configuration $S = \{s\}$ should simulate. The algorithm represents each resulting step $S \xrightarrow{(\hat{l}, O)} S''$ in its results as the pair $\langle O, S'' \rangle$ of the fulfilled obligations and the configuration reached in that step.

To start, the algorithm checks whether the given PSM is a “stuck” PSM that monitors no markers and cannot take a transition labeled with $\bullet$; this is used below to determine whether the found transitions represent a legal summary transition. constructs an initial configuration $S$ containing just the given PSM $s$. Then to determine how $S$ can simulate the initial $\hat{l}$ step from $L$, `MatchingSpecSteps` simply tries all single-step transitions from $S$ labeled with $\hat{\lambda} = [\hat{l}]$. Each of those steps corresponds to the PSM firing one of its state transitions via the P-MONITOREDRECEIVE or P-FREETRANSITION rule, or staying in the same state via P-UNMONITOREDRECEIVE if $\hat{l}$ represents receiving a message on an unmonitored receptionist (see section 3.6.5). Additionally, `MatchingSpecSteps` includes the possibility that $s$ does not transition at all if $\hat{\lambda}$ represents an internal step (i.e., $\hat{\lambda} = \bullet$), because the weak-step transition $S \xrightarrow{\bullet} S$ is also possible in that case.

After finding some weak transitions $S \xrightarrow{[\hat{l}]} S'$ to simulate the initial $\hat{l}$ step, it remains to determine how each $S'$ can simulate a sequence of outputs summarized by $\hat{\mu}$. This process is more complex than finding matching transitions for $\hat{l}$, so a separate algorithm called `SimulateOutputs` (described next) encapsulates that logic. `SimulateOutputs` returns a set of pairs of the form $\langle O, S'' \rangle$, where $O$ indicates a lower bound on the obligations fulfilled by simulating those outputs, and $S''$ is the reached configuration. `MatchingSpecSteps` adds to its results set

\(^2\)A stronger version of this theorem that corresponds exactly to the Specification Step Commutativity also exists. This weaker version is used here to simplify the presentation.
results all such pairs in which either the initial PSM is stuck as described above, or all PSMs in $S''$ monitor at least one marker (as required for specification summary transitions), then returns $results$ to its caller.

1 Function $MatchingSpecSteps(s, L) =$
2   let $\langle H, H', \varphi : \overline{\eta}, \overline{\Phi}, O' \rangle := s;$
3   let stuckPsm := $H \cup H' = \emptyset$ and $\forall s'. s \xrightarrow{\emptyset, \varphi} s'$;
4   let $S := \{s\}$;
5   let $\langle \hat{l}, \hat{\mu} \rangle := L$;
6   let $initialSteps := \{S' \mid S \xrightarrow{\hat{l}} S'\}$;
7   if $\hat{l} = \bullet$ then
8      $initialSteps := initialSteps \cup \{S\}$;
9   end
10  let $results := \emptyset$;
11 for $S' \in initialSteps$ do
12    for $(O, S'') \in SimulateOutputs(S', \hat{\mu})$ do
13      if either stuckPsm or $\forall \langle H, H', \varphi : \overline{\eta}, \overline{\Phi}, O' \rangle \in S''$. $H \cup H' \neq \emptyset$ then
14         $results := results \cup \{(O, S'')\}$;
15      end
16    end
17  end
18  return $results$;
19 end

Figure 7.8: The $MatchingSpecSteps$ algorithm

7.4.3 Simulating the Outputs

$SimulateOutputs$, shown in figure 7.9, takes as input a specification configuration $S$ and the summarized outputs $\hat{\mu}$. Its goal is to return a set of pairs each of the form $(O, S')$ such that $S$ can simulate any sequence of output steps summarized by $\hat{\mu}$, fulfilling at least the obligations in $O$ and reaching configuration $S'$. The idea of $SimulateOutputs$ is to process those outputs one at a time, so that the algorithm explores possible transition sequences of $S$. If there are no outputs to simulate (i.e., $dom(\hat{\mu}) = \emptyset$), then the algorithm returns just the trivial solution $\langle \emptyset, S \rangle$: no obligations are fulfilled, and the given configuration does not transition at all. Otherwise, it uses a separate algorithm $SimulateOutput$ (described below) to find possible ways that $S$ can simulate an arbitrary output $\langle \hat{a}@H, \hat{v} \rangle$ from $\hat{\mu}$ with quantity $m$, where each way is represented by the fulfilled obligations $O$ and the reached configuration $S'$. Then $SimulateOutputs$ calls itself recursively to determine how each $S'$ can simulate the remaining outputs in $\hat{\mu}$. $SimulateOutputs$ combines the fulfilled obligations $O'$ from each such call with those in $O$, then
**Function** SimulateOutputs(S, $\hat{\mu}$) =

1. if dom($\hat{\mu}$) = $\emptyset$ then
2. return \( \langle \emptyset, S \rangle \);
3. end
4. let \( \langle \hat{a} \hat{\in} H, \hat{v} \rangle \) := an arbitrary element of dom($\hat{\mu}$);
5. let \( m := \hat{\mu}(\hat{a} \hat{\in} H, \hat{v}) \);
6. let results := $\emptyset$;
7. for \( \langle O, S' \rangle \in \text{SimulateOutput}(S, \hat{a}, H, \hat{v}, m) \) do
8. let $\hat{\mu}'$ := $\hat{\mu} |_{\text{dom}(\hat{\mu}) - \langle \hat{a} \hat{\in} H, \hat{v} \rangle}$;
9. for \( \langle O', S'' \rangle \in \text{SimulateOutputs}(S', \hat{\mu}') \) do
10. results := results $\cup \{ \langle O \cup O', S'' \rangle \}$;
11. end
12. end
13. return results;
14. end

Figure 7.9: An algorithm to find ways for a specification S to simulate the outputs summarized by $\hat{\mu}$ returns the results.

**Simulating a Single Output**

Figure 7.10 shows the algorithm SimulateOutput. Its purpose is similar to that of SimulateOutputs, except that this algorithm determines how to simulate a single summarized output $\langle \hat{a} \hat{\in} H, \hat{v}, m \rangle$, rather than the whole collection $\hat{\mu}$ of such outputs. Simulating an output step requires that all of the configuration’s constituent PSMs simulate that step (see the S-SENDORRECEIVE rule in section 3.6.7), so the idea of SimulateOutput is to find a simulating transition sequence for each PSM in S.

In the case where S contains no PSMs, then there is nothing to transition, so the algorithm returns the trivial singleton solution \( \langle \emptyset, S \rangle \). Otherwise, the algorithm picks an arbitrary PSM s from S and uses another algorithm PsmSimulateOutput (described below) to find the ways that s can simulate the given output. For each of those ways, that algorithm returns the set O of obligations the PSM fulfilled to simulate that output, the transitioned PSM s’, and the set of PSMs S’ that s forked during the transition. SimulateOutput then recursively calls itself to get the transitions from the remaining PSMs, combines the results, and returns the full set of collected results. The forked PSMs S’ are added to the reached specification configuration S’’ because a conforming program now has to implement the protocols those PSMs represent in addition to those represented by the PSMs from S’’.

---

3 The results of the two calls are independent, so the results of the inner call are cached between iterations of the outer loop in the actual implementation.
7.4. MATCHINGSPECSTEPS

Function `SimulateOutput(S, a, H, v, m) =`

1. if `S = φ` then
2.   return `{(φ, S)}`;
3. end
4. let `s :=` an arbitrary element of `S`;
5. let `results :=` φ;
6. for `⟨O, s’, S’⟩` in `PsmSimluateOutput(s, a, H, v, m)` do
7.   let `S'' := S - {s}`;
8.   for `⟨O’, S''⟩` in `SimulateOutput(S'', a, H, v, m)` do
9.     results := results ∪ `{⟨O ⊔ O', S'' ∪ {s'} ∪ S'⟩}`;
10. end
11. end
12. return `results`;
13. end

Figure 7.10: An algorithm to find ways for a specification configuration `S` to simulate the output(s) summarized by `<a@H, v, m>`

Simulating a Single Output with a Single PSM

The algorithm `PsmSimluateOutput`, shown in figures 7.11 and 7.12 is where the real work happens, and where most of the heuristics come into play. Its goal is to find transitions that show how the given PSM `s` can simulate an output (or a sequence of outputs, depending on the value of `m`) of the given message `v` to the given marked address `a`.

The algorithm starts with three guard clauses. Recall from chapter 3 that APS does not model communication between external actors, so the PSM transition rule P-SEND forbids a conforming program from sending output-monitored markers to the environment. The first guard clause in `PsmSimluateOutput` enforces this rule by returning an empty set of solutions if `v` contains any such markers, indicating that the PSM is unable to simulate such an output. The second guard clause rejects any output whose address is marked with multiple markers. The model-checking algorithm never provides such an output, but making the check explicit here simplifies the correctness proof. The third guard clause says that if the marker `η` on the output’s destination address `a` is not monitored (i.e., this output is irrelevant to the protocol specified by `s`), then `PsmSimluateOutput` returns the trivial solution `{φ, S}`. The P-SEND rule allows a PSM to send any output to any unmonitored address without a matching obligation. Otherwise, none of those clauses matched, so `H''` must contain exactly one marker `η`.

Next, `PsmSimluateOutput` handles the case of a monitored output with abstract quantity `m = many`. A solution `{O, s', S}` for such an output is valid only if `s` can simulate sending any number of such outputs (including zero) by transitioning to `s'`, fulfilling at least the obligations `O`, and forking the PSMs `S` in the process. Because `s` has only finitely many obligations, but the message can
CHAPTER 7. A MODEL-CHECKING ALGORITHM FOR APS

Function $\text{PsmSimluateOutput}(s, \hat{a}, H'', \hat{v}, m) =$

let $\langle H, H', \varphi : \eta'_1 \ldots \eta_n, \overline{\Phi}, O \rangle := s$

if $\exists \hat{a}' @ H''$ in $\hat{v}$ such that $H'' \cap H' \neq \emptyset$ then
    return $\emptyset$;
endif

if $|H''| > 1$ then
    return $\emptyset$;
endif

if $H \cap H'' = \emptyset$ then
    return $(\langle \emptyset, s, \emptyset \rangle)$;
endif

let $\{\eta\} := H \cap H''$;

let results := $\emptyset$;

if $m = \text{many}$ then
    forall $po$ such that $s \xrightarrow{\emptyset, \emptyset} \langle H, H', \varphi : \eta'_1 \ldots \eta'_n, \overline{\Phi}, O \cup \langle \eta, po \rangle \rangle$ do
        if $\hat{v} \sim po$ then $\emptyset, \emptyset$ then
            results := results $\cup$ $(\langle \emptyset, s, \emptyset \rangle)$;
        end
    end
    return results;
endif

// continued in figure 7.12

Figure 7.11: An algorithm to find ways for a PSM $s$ to simulate the output(s) summarized by $\langle \hat{a} @ H'', \hat{v}, m \rangle$
7.4. MATCHINGSPECSTEPS

forall \langle \eta'', po \rangle \in O such that \eta'' = \eta do
  forall \ H'' and S such that \hat{v} \sim po \triangleright H'', S and \ |H \cup H''| \leq 1 do
    let \ s' := \langle H \cup H'', H', \varphi : \eta_1'' \ldots \eta_n'', \overline{O} - \langle \eta'', po \rangle \rangle;
    results := results \cup \{ \langle \langle \eta'', po \rangle, s', S \rangle \};
  end
end
if results \neq \emptyset then
  return results;
end
forall \psi' \sim, \eta_1'' \ldots \eta_m'', po such that \ s \sim, \psi, \eta_1'' \ldots \eta_m'', \overline{O} \cup \langle \eta, po \rangle \rangle do
  forall \ H'' and S such that \hat{v} \sim po \triangleright H'', S and \ |H \cup H''| \leq 1 do
    let \ s' := \langle H \cup H'', H', \varphi : \eta_1'' \ldots \eta_n'', \overline{O} - \langle \eta'', po \rangle \rangle;
    results := results \cup \{ \langle \langle \eta, po \rangle, s', S \rangle \};
  end
end
if results \neq \emptyset then
  return results;
end
forall \psi' \sim, \eta_1'' \ldots \eta_m'', po such that \ s \sim, \psi, \eta_1'' \ldots \eta_m'', \overline{O} \cup \langle \eta, po \rangle \rangle do
  forall \ H'' and S such that \hat{v} \sim po \triangleright H'', S and \ |H \cup H''| \leq 1 do
    let \ s' := \langle H \cup H'', H', \varphi : \eta_1'' \ldots \eta_n'', \overline{O} \rangle;
    results := results \cup \{ \langle \langle \eta, po \rangle, s', S \rangle \};
  end
end
return results;
end

Figure 7.12: The PsmSimulateOutput algorithm, continued
be sent an unbounded number of times, it would be futile to try to match those outputs with existing obligations on \( s \). Instead, \( \text{PsmSimluateOutput} \) determines whether \( s \) can take a transition labeled with • that

- stays in its current state,
- does not fork any PSMs, and
- incurs a single obligation \( \langle \eta, po \rangle \) that matches \( \hat{v} \).

The PSM could take such a transition any number of times (including zero) to obtain enough obligations to match the given outputs, while always ending up in the same state. Therefore \( \text{PsmSimluateOutput} \) includes each such transition as a possible simulating transition in \( \text{results} \).

The recorded solution \( \langle \emptyset, s, \emptyset \rangle \) gives \( \emptyset \) as the lower bound on the obligations fulfilled to simulate this output sequence, because no obligations will be fulfilled if the sequence contains zero outputs. Similarly, \( \text{PsmSimluateOutput} \) uses only those patterns that do not fork PSMs when matched, because such a PSM would not be created if the PSM never simulates that output.

Otherwise, \( m = \text{single} \), meaning \( s \) has to match exactly one occurrence of that output. In that case, \( \text{PsmSimluateOutput} \) tries using three different sets of obligations to match \( \hat{v} \).

1. Existing obligations in \( O \).
2. Obligations obtained by a • transition to the same state, as in the \( m = \text{many} \) case above.
3. Obligations obtained by any state transition (including those that change the current state) that incurs exactly one obligation.

For each set of obligations, if \( \text{PsmSimluateOutput} \) finds at least one match in that set, then it returns all of the results for that set rather than additionally including matches from the following sets. Experience so far has shown this approach to strike a good balance between finding enough matches to prove conformance in common cases and avoiding dead ends.

The condition \( |H \cup H''| \leq 1 \) checked when matching each obligation merely enforces the condition from the P-SEND rule that a PSM can have no more than one input-monitored marker.

### 7.5 Prune

After \( \text{ModelCheck} \) uses \( \text{Explore} \) to find a set of program-configuration/PSM pairs \( R_{loc} \) that satisfy the local part of the Simulation condition, its next step is to filter out those pairs that violate the non-local part of Simulation. This is the job of the \( \text{Prune} \) algorithm, shown in figure 7.13. Given some local-transformation-simulation relation \( R \), witness \( Z \), and frontier \( F \), its goal is to return some subset
7.5. PRUNE

Function Prune \( (R, Z, F) \) =

1. let \( R' := R \);
2. let \( Z' := Z \);
3. while \( F \neq \emptyset \) do
   4. let \( (\hat{K}, s) := \) an arbitrary item from \( F \);
   5. for all steps \( \langle \hat{K}', s' \rangle \xrightarrow{L} \langle \hat{K}, s \rangle \) do
      6. \( Z' \left[ \hat{K}' \xrightarrow{L} \hat{K}, s \right] := Z' \left( \hat{K}' \xrightarrow{L} \hat{K}, s \right) - \langle O, S \rangle \);
      7. if \( \langle \hat{K}', s' \rangle \in R' \) and \( Z' \left( \hat{K}' \xrightarrow{L} \hat{K}, s' \right) = \emptyset \) then
         8. \( R' := R' - \{ \langle \hat{K}', s' \rangle \} \);
         9. \( F := F \cup \{ \langle \hat{K}', s' \rangle \} \);
   end
   10. \( F := F - \{ \langle \hat{K}, s \rangle \} \);
11. end
12. return \( \langle R', Z' \rangle \);
end

Figure 7.13: Prune algorithm

\( R' \) of \( R \) and a strategy \( Z' \) such that \( Z' \) is a witness for \( R' \) as a transformation-simulation relation.

Prune adapts the backward-chaining algorithm from section 7.1 for transformation conformance. For each pair \( \langle \hat{K}, s \rangle \) in the frontier \( F \), Prune loops over all incoming transformation-step transitions \( \langle \hat{K}', s' \rangle \xrightarrow{L} \langle \hat{K}, s \rangle \) enabled by the strategy \( Z' \). Each such step represents a case where the program configuration can take a step \( \hat{K}' \xrightarrow{L} \hat{K} \), but where combining that step with a matching step provided by \( Z' \) leads to a pair in the frontier. That violates the non-local part of the transformation-simulation definition, so Prune removes the pair \( \langle O, S \rangle \) representing that step from \( Z' \).

Removing that step might leave \( Z' \) without any way to match the step \( \hat{K}' \xrightarrow{L} \hat{K} \) (i.e., \( Z' \left( \hat{K}' \xrightarrow{L} \hat{K} \right) = \emptyset \)). If that is the case, then \( \langle \hat{K}', s' \rangle \) no longer satisfies the rule for a transformation-simulation relation, so Prune removes that pair from \( R' \) and adds it as a new member of the frontier \( F \) instead. Prune will then eventually process \( \langle \hat{K}', s' \rangle \) and remove from \( Z' \) all steps leading to it, thus implementing the backward-chaining aspect of the algorithm.

Once the frontier \( F \) is empty, \( Z' \) provides at least one matching step for every step \( \hat{K} \xrightarrow{L} \hat{K}' \) from a pair \( \langle \hat{K}, s \rangle \) in \( R' \), and all such steps lead only to other pairs in \( R' \). Therefore, \( Z' \) is a witness to \( R' \) as a transformation-simulation relation, so

\(^4\)The actual implementation of Prune caches these steps in a separate data structure to avoid recomputing them, but figure 7.13 omits that detail to clarify the presentation.
Prune returns the tuple \( \langle R', Z' \rangle \) as its result.

### 7.6 FindFulfillingPairs

```plaintext
1 Function FindFulfillingPairs(R, Z) =
2     let R' := \emptyset;
3     for (K, s) in R do
4         let foundProgFairPath := false;
5         for (\eta, po) in Obls(s) do
6             let G := non-satisfaction graph from \( \langle K, s \rangle \) for \( \langle \eta, po \rangle \), Z, and T;
7             if G has a vertex \( \langle K', s', \eta' \rangle \) such that \( K' \) is quiescent then
8                 foundProgFairPath := true;
9                 break;
10             end
11         if there exists a program-fair SCC in G then
12             foundProgFairPath := true;
13             break;
14         end
15     unless foundProgFairPath
16         R' := R' \cup \{\langle K, s \rangle\};
17     end
18 return R';
```

Figure 7.14: The FindFulfillingPairs algorithm

Once ModelCheck finds a set of pairs \( R_{sim} \) that satisfy the Simulation condition for transformation conformance, it moves on to checking which pairs \( \langle \hat{K}_1, s_1 \rangle \) in \( R_{sim} \) satisfy the Fulfillment condition, reprinted here.

**Fulfillment** Every program-fair transformation-step-execution

\[
\langle \hat{K}_1, s_1 \rangle \xrightarrow{L_1, K_1', Z, O_1, S_1, T, A_1, M_1} \ldots \text{ is specification-fair.}
\]

Section 6.3.1 formally defines those notions of fairness, but let’s review their intuitive definitions. A transformation-step execution is program-fair when every necessarily enabled atomic actor eventually takes a step, and every single message is eventually received (while accounting for changes introduced by the transformation \( T \) at every step, as discussed in section 6.3.1). That same execution is specification-fair when every obligation appearing in some PSM in the execution is eventually fulfilled (again accounting for changes introduced by \( T \)). So checking the Fulfillment rule for a pair \( \langle K, s \rangle \) means checking whether every program-fair execution starting at \( \langle K, s \rangle \) eventually fulfills all obligations.
This is the job of the FindFulfillingPairs algorithm, shown in figure 7.14. It expects as input a transformation-simulation relation \( R \) with witness \( Z \). Its goal is to return a subset \( R' \) of \( R \) such that every pair \( \langle \hat{K}, s \rangle \) in \( R' \) satisfies the fulfillment condition.

Because FindFulfillingPairs assumes \( R \) is a transformation-simulation relation, the set \( R \) is closed over transformation-step transitions

\[
\langle \hat{K}, s \rangle \xrightarrow{L,K',Z,O,S,T,A,M} \langle \hat{K}', s' \rangle
\]

Therefore the problem reduces to checking that for every pair \( \langle \hat{K}, s \rangle \in R \) and every obligation \( \langle \eta, po \rangle \) in \( s \), the obligation is fulfilled in every transformation-step execution from that pair. Thus, FindFulfillingPairs checks each obligation one at a time.

The approach exploits a graph-based interpretation of \( R \) and \( Z \), where the pairs in \( R \) are the vertices and the transformation-step transitions between them are the edges. Executions from a pair \( \langle \hat{K}, s \rangle \) that do not fulfill some obligation \( \langle \eta, po \rangle \) in \( s \) can be modeled as the subgraph of vertices reached from \( \langle \hat{K}, s \rangle \) by edges that do not fulfill that obligation. (This approach is adapted from a model-checking algorithm for Fair-CTL [29].) This is called the non-satisfaction graph, defined below.

**Definition.** The non-satisfaction graph from \( \langle \hat{K}, s \rangle \) for the obligation \( \langle \eta, po \rangle \), strategy \( Z \), and transformation \( T \) is the smallest graph \( G = \langle V, E \rangle \) satisfying the following properties:

- \( \langle \hat{K}, s, \eta \rangle \in V \).
- For all \( \langle \hat{K}', s', \eta' \rangle \in V \) and all transformation-step transitions
  \[
  \langle \hat{K}', s' \rangle \xrightarrow{L,K'',Z,O,S,T,A,M} \langle \hat{K}'', s'' \rangle,
  \]
  if \( \langle \eta', po \rangle \notin O \) and \( \langle M(\eta'), po \rangle \in Obls(s'') \), then \( \langle \hat{K}'', s'', M(\eta) \rangle \in V \) and \( E \) contains an edge from \( \langle \hat{K}', s' \rangle \) to \( \langle \hat{K}'', s'' \rangle \) with label \( \langle L, A, M \rangle \).

Because a transformation such as Canonicalize may rename the obligation marker \( \eta \) in each step (as indicated by the use of the substitution \( M \)), each vertex \( \langle \hat{K}', s', \eta' \rangle \) records the marker \( \eta' \) that identifies the obligation in \( s' \). The output pattern \( po \) however, is constant across all vertices in the graph.

Once that graph is constructed for a pair \( \langle \hat{K}, s \rangle \) and obligation \( \langle \eta, po \rangle \), the goal is for FindFulfillingPairs to determine whether any path through it starting at \( \langle \hat{K}, s, \eta \rangle \) defines a program-fair execution. If so, then FindFulfillingPairs does not include \( \langle \hat{K}, s \rangle \) in \( R' \), because that pair it has some program-fair execution that does not fulfill an obligation in \( s \).

A program-fair execution may be finite or infinite. A finite execution is program-fair if its final configuration has no more actors that need to run or messages that need to be received. This state is called quiescence, defined below.

**Definition.** An abstract program configuration \( \bar{K} = \langle \hat{\beta} \mid \bar{\mu} \mid H \rangle \) is quiescent if no actor in \( \bar{K} \) is necessarily enabled, and there exist no \( \bar{a}, H', \) and \( \bar{\nu} \) such that \( \bar{\mu}(\bar{a}@H', \bar{\nu}) = \text{single} \).
Thus, to check for finite program-fair executions from \( \langle \hat{K}, s \rangle \), \text{FindFulfillingPairs} checks whether any vertex \( \langle \hat{K}', s', \eta' \rangle \) in \( G \) contains a quiescent program configuration \( \hat{R}' \). A path from \( \langle \hat{K}, s, \eta \rangle \) to that vertex in \( G \) represents a program-fair execution that does not fulfill the obligation \( \langle \eta, po \rangle \).

An infinite execution manifests as a cycle in \( G \). It is well known that every cycle in a graph \( G \) is contained in a strongly connected component (SCC) of \( G \). Therefore, \text{FindFulfillingPairs} determines whether an infinite program-fair execution exists by searching for an SCC with edges that allow every necessarily enabled actor to run, and every single message to be received. This is called a program-fair SCC, defined below.

**Definition.** An SCC \( C \) is program-fair if for every vertex \( \langle \hat{K}_n, s_n, \eta_n \rangle \) in \( C \), the following properties both hold:

- For every address \( \hat{a} \) such that the actor at \( \hat{a} \) is necessarily enabled in \( \hat{K}_1 \), there exists some path in \( C \) with vertices \( \langle \hat{K}_1, s_1, \eta_1 \rangle \), \( \ldots \), \( \langle \hat{K}_n, s_n, \eta_n \rangle \) and edges labeled \( \langle L_1, A_1, M_1 \rangle \), \( \ldots \), \( \langle L_{n-1}, A_{n-1}, M_{n-1} \rangle \) and some \( \hat{a}' = A_{n-1} \circ \ldots \circ A_1(\hat{a}) \) such that either
  - there is no actor in \( \hat{K}_n \) with address \( \hat{a}' \),
  - the actor at \( \hat{a}' \) is not necessarily enabled in \( \hat{K}_n \), or
  - there is an edge from \( \langle \hat{K}_n, s_n, \eta_n \rangle \) in \( C \) with label \( \langle L_n, A_n, M_n \rangle \) such that \( \hat{a}' \) identifies the active actor for \( L_n \).

- Let \( \hat{K}_1 = \langle \hat{a}, \hat{\mu}, H \rangle \). For all \( \hat{a}, H', \) and \( \hat{\nu} \) such that \( \hat{\mu}(\hat{a}@H', \hat{\nu}) = \text{single} \), there exists some path in \( C \) with vertices \( \langle \hat{K}_1, s_1, \eta_1 \rangle \), \( \ldots \), \( \langle \hat{K}_n, s_n, \eta_n \rangle \) and edges labeled \( \langle L_1, A_1, M_1 \rangle \), \( \ldots \), \( \langle L_{n-1}, A_{n-1}, M_{n-1} \rangle \) and there exist \( A' = A_{n-1} \circ \ldots \circ A_1, M' = M_{n-1} \circ \ldots \circ M_1, \hat{\alpha}' = A(\hat{a}), H'' = M'(H'), \hat{\nu}' = A(M'(\hat{\nu})), \hat{\nu}' \), \( \hat{\nu}' \), \( H'' \), and \( \hat{\nu} \) such that \( \hat{K}_n = \langle \hat{\nu}, \hat{\nu}', H'' \rangle \) and either
  - \( \langle \hat{\alpha}'@H'', \hat{\nu}' \rangle \notin \text{dom}(\hat{\mu}) \),
  - \( \hat{\mu}(\hat{a}@H'', \hat{\nu}') = \text{many} \), or
  - there is an edge from \( \langle \hat{K}_n, s_n, \eta_n \rangle \) in \( C \) with label \( \langle L_n, A_n, M_n \rangle \) such that and \( L_n = \langle \hat{\alpha}': \text{rcv-int}(H'', \hat{\nu}'), \hat{\mu}'' \rangle \) for some \( \hat{\mu}'' \).

\text{FindFulfillingPairs} can then use a standard algorithm to computes the SCCs of \( G \) and check the above properties with a simple traversal of the graph for each necessarily enabled actor and each single message in each vertex in \( G \).

Whenever \text{FindFulfillingPairs} finds such a finite or infinite program-fair execution with the above techniques, it records that fact and skips checking the remaining obligations in \( s \). Outside of that loop, if no such execution was found, then all obligations in \( s \) are always eventually fulfilled, so \text{FindFulfillingPairs} adds \( \langle \hat{K}, s \rangle \) to a set \( R' \). After checking all pairs in \( R \), \text{FindFulfillingPairs} finally returns \( R' \) as the result.
7.7 Correctness and Termination

From a theoretical standpoint, there are two key criteria to be shown for ModelCheck: that its results are a sound approximation of conformance, and that it always terminates. The following two theorems formalize these statements:

**Theorem (ModelCheck Correctness).** For all $P$ and $\Sigma$, if the transformation $T$ used in ModelCheck is conformance-reflecting and ModelCheck$(P, \Sigma)$ returns true, then $P \models \Sigma$.

*Proof.* Let there be $P$ and $\Sigma$ such that ModelCheck$(P, \Sigma)$ returns true, where ModelCheck is using some conformance-reflecting $T$. By the ModelCheck Transformation Conformance theorem (appendix M), $P \models_{TR} \Sigma$. The rest of the proof relies on the chain of refinements to conformance shown in previous chapters.

- By the Transformation Conformance theorem (appendix K), $P \models_{S} \Sigma$.
- By the Summary Conformance theorem (appendix I), $P \models_{A} \Sigma$.
- By the Abstract Conformance theorem (appendix I), $P \models_{PSM} \Sigma$.
- By the PSM Conformance theorem (appendix G), $P \models_{EV} \Sigma$.
- By the Event-Step Conformance theorem (appendix F), $P \models_{D} \Sigma$.
- By the Deterministic-Handler Conformance theorem (appendix E), $P \models_{SH} \Sigma$.
- By the Single-Handler Conformance theorem (appendix D), $P \models_{ER} \Sigma$.
- By the External-Representative Conformance theorem (appendix C), $P \models_{EO} \Sigma$.
- By the Externals-Only Conformance theorem (appendix B), $P \models \Sigma$.

**Theorem (ModelCheck Termination).** For all $P$ and $\Sigma$, if $\vdash_{prog} P$ and the transformation $T$ is termination-guaranteeing, then ModelCheck$(P, \Sigma)$ terminates.

The proof of this theorem is in appendix N. The main obligation is to show that Explore explores only finitely many program-configuration/PSM pairs; the argument is based on the various state-space reductions performed by the abstract interpretation and the various transformations used. The notion of a termination-guaranteeing transformation is defined in that appendix, and the appendix shows that all transformations ModelCheck might use are termination-guaranteeing.
7.8 Tunable Heuristics

The ModelCheck algorithm outlines an approach to check whether a CSA program conforms to an APS specification, but the heuristics it uses are open to change. Indeed, the heuristics described so far represent a first attempt at balancing performance with precision (i.e., the likelihood of finding a conformance proof), but there is room for improvement. This section lists some of the most important heuristics to be modified for tuning the algorithm’s precision, performance, or both.

Heuristics play the biggest role in MatchingSpecSteps, where they are used to “guess” how the original programmer intended to implement the specification. For instance, MatchingSpecSteps assumes that the event that triggers an event handler (i.e., the timeout or received message) corresponds to at most one transition of a PSM: either a state transition whose pattern matches the received message, or a free state transition. This simple approach suffices for the programs and specifications tested so far (see chapter 9), but other programs or specifications might require more sophisticated techniques.

The SimulateOutputs algorithm used by MatchingSpecSteps similarly must guess what aspect of the specification should allow each message sent to the environment. In particular, PsmSimulateOutput uses a complicated scheme for matching a single output: first it tries to match that output against all existing obligation patterns, then against all obligations it can incur without changing its state, then finally against all obligations it can incur with a free transition to any other state. That approach might slow down the model checker if it leads to exploring too many dead ends, or it might overlook a possible conformance proof if it does not explore enough options. Thus, PsmSimulateOutput may require modification depending on the situation.

One final important set of heuristics is the set of rules used to decide whether to assimilate an atomic actor into a collective one. These include things like whether an actor was spawned in the most recent event handler, or whether an actor has a reference to an address marked with an output-monitored marker. Such rules are one approach to deducing which actors might be involved in implementing some specified behavior, but others are possible. Indeed, even the idea that Assimilate converts at most one actor per spawn location is a heuristic; the only hard requirement is that the number of actors left atomic by Assimilate is bounded. Depending on the situation, a programmer might wish to change these rules, perhaps even by tailoring them to match a particular implementation of some specification.

7.9 Related Work

Model Checkers for Actor Languages A variety of model checkers have been developed for actor-based languages. These existing checkers verify different kinds of correctness properties than mine; see the related work section at the end of chapter 3 for a comparison of APS with other specification languages.
As mentioned in chapter 5, Huch [64] built a model checker that can de-
cide LTL properties, but only for Erlang programs with a bounded number of
actors and bounded queues. The etomcrl toolset [11] translates Erlang pro-
grams to $\mu$CRL to use its existing verification tools. The translation involves no
abstraction, though, so model checking terminates only for finite-state programs.
McErlang [49] checks properties specified as safety properties or Büchi automata
written in Erlang by running programs through a custom Erlang runtime, but
it requires users to abstract their programs to finite-state models themselves.
Soter [43] was also mentioned in chapter 5; it focuses on verifying properties
that are checkable in Petri nets.

RMC [100], the Rebeca Model Checker, verifies a Rebeca program against a
given LTL or CTL formula. The model checker explores the state-space of con-
crete actors, so it can handle only programs with a finite state-space. The par-
tial order reduction and symmetry reduction in RMC are similar to some of the
techniques used in my model checker, such as single-handler conformance (ap-
pendix D) and the notion of collective actors.

Other Model Checkers The model checkers SPIN [61], nuSMV [27], and
Murϕ [42] are all commonly used to verify protocols in concurrent programs,
usually against an LTL or CTL formula. Each of them works only for programs
with a finite state space, however, so actor programs that create new actors dy-
amically or send unboundedly many messages require some sort of abstraction
before these tools can be used. Furthermore, that abstraction must be created be-
fore running the tool, so an approach like mine that abstracts the program while
exploring the state space would not be viable in these tools.

Static Analysis for Actors Sagonas et al. have developed static analyses for
actor programs to detect common mistakes such as communication errors [26],
race conditions [25], and type errors [74]. Most of these analyses have been in-
corporated into the multi-purpose Dialyzer tool [73] for Erlang. My work veri-
fies whether a program follows a programmer-defined specification, rather than
searching for fixed, analysis-specific classes of errors.

Run-Time Verification As an alternative to model checking, some actor pro-
grammers use systematic testing approaches to whether their programs are cor-
rect. Arts et al. developed Quviq QuickCheck [12], which randomly generates test
cases for user-specified properties of Erlang programs. Protocols can be tested by
having the tool generate sequences of messages to send. PropEr [87] is a similar
open-source tool for Erlang, but it is geared towards testing sequential code.

The P language [40] comes with a tool for running a P program under many
different schedules. The tool then reports whether executing under any of those
schedules leads to an error such as violating a user-defined assertion or sending
a message to an uninitialized actor. Desai et al. [41] later extended the approach
to use assume-guarantee reasoning, thereby allowing compositional verification.
P# [37] is an embedding of a P-like language into .NET, and it extends P's systematic testing approach to work on programs with components written in .NET languages other than P#.

Francalanza and Seychell wrote a run-time monitoring tool for Erlang called detectEr [47]. Their tool automatically generates run-time monitors for safety properties specified in modal logic. It might be possible to generate similar monitors that check for the safety properties of APS specifications (see the discussion in chapter 10).

Generally, these dynamic techniques are complementary to my static approach. Static techniques cover all possible execution paths at the cost of a loss in precision; dynamic techniques typically have no loss in precision but check only the executed paths.

**Verification of Distributed Systems**  
DISEL [98] is a programming language embedded in the Coq proof assistant designed for implementing distributed systems and verifying their properties. DISEL uses a distributed extension of separation logic to specify correctness properties. Being based on top of a proof assistant, the tool requires more user interaction to prove correctness than my model checker.

IronFleet [56] is a method for building and verifying distributed systems that are too complex to model check. At the lowest level, Hoare-logic-style proofs are used to verify the sequential segments of the program. Then multiple levels of state-machine refinement are used to verify the concurrent interleavings of those sequential segments. Compared to my work, IronFleet trades off automation for scalability to large state spaces.

The Alloy modeling tool [66] has been used to check the correctness of (and find bugs in) distributed systems like the Chord distributed hash table [111]. The approach requires the user to design a model of their system in Alloy (with a bound on, for example, the number of nodes in the system) using relational algebra and first-order predicate logic. The Alloy analyzer can then compile the model to a boolean formula and use a SAT solver to determine whether user-specified invariants hold. The approach is useful for algorithms and systems previously described only in pseudocode, but for existing programs it requires mapping them to Alloy models.
Chapter 8

Optimizations

The ModelCheck algorithm defined in the previous chapter is guaranteed to terminate, but that does not prevent it from taking an excessive amount of time. This chapter introduces four optimizations that help the model checker run faster, making it usable for the evaluation of chapter 9.

The first three optimizations are transformations that ModelCheck applies along with Split, Unmark, Assimilate, and Canonicalize. Thus, in addition to speeding up the model checker, these optimizations also show how transformation conformance serves as a general-purpose framework for reducing the state space on-the-fly while exploring that space.

The final optimization is a simple memoization of a key step of the algorithm, described in section 8.5.

Some of these optimizations are synergetic; for example, the acceleration optimization described in this section can run much faster when memoization is also enabled. To show these effects, the next chapter evaluates each optimization both on its own and in combination with other optimizations.

8.1 Acceleration

8.1.1 Motivation

During state-space exploration, the model checker often encounters many near-identical program configurations. For example, while exploring the stream-processing program’s state space, the model checker might encounter several program configurations that differ only in the current states of a collective processor actor. One configuration might have that collective actor in just the On state, another configuration might have it in both On and Off, a third might have it in all three of On, Off, and Done, etc.

In that set of near-identical configurations, some of the configurations approximate one another. For example, the configuration with a collective processor in both the On and Off states approximates the one whose collective processor is in only the On state, because a collective actor can represent zero or more concrete
actors in each of its states. Figure 8.1 summarizes the approximation relationships between these configurations as a Hasse diagram, with the more approximate configurations appearing higher in the diagram. The subscripts on each configuration list the collective actor’s current states in that configuration, e.g., $\hat{K}_{NF}$ denotes a configuration whose collective processor actor is in the On ($N$) and Off ($F$) states. $\hat{K}_N$ has no collective processor actor.

Suppose that to model check the stream-processing program, it is necessary to show that each of these configurations conforms to the same PSM $s$. A naive model checker would check all eight of them individually; however, seven of those checks are redundant. $\hat{K}_{NFD}$ subsumes the behavior of all of the other configurations, so if $\hat{K}_{NFD}$ conforms to $s$, then all the others do, too. This kind of redundancy is even more pronounced when model checking larger programs, which tend to have even more ways to produce near-identical configurations.

This section introduces a new transformation called Accelerate that helps avoid these kinds of redundant checks. The idea is that when given a pair $\langle \hat{K}, s \rangle$, Accelerate attempts to determine whether the model checker will eventually have to check that some approximation $\hat{K}'$ of $\hat{K}$ also conforms to $s$. If so, then checking whether $\hat{K}$ conforms to $s$ is redundant, so Accelerate transforms $\langle \hat{K}, s \rangle$ into $\langle \hat{K}', s \rangle$. For example, if at some point it is necessary to determine whether $\hat{K}_s$ conforms to a PSM $s$, Accelerate might transform $\langle \hat{K}_s, s \rangle$ into $\langle \hat{K}_{NFD}, s \rangle$ instead, because it is sufficient to check whether $\hat{K}_{NFD}$ conforms to $s$.

The idea is similar to acceleration techniques in the model-checking literature [2, 14, 67], which help speed up the convergence of fixed-point computations; hence the name Accelerate. It is also similar to the idea of widening in abstract interpretation [34].

Acceleration introduces approximation into the given pair $\langle \hat{K}, s \rangle$, so it can sometimes prevent the model checker from proving conformance in cases where
it would otherwise succeed. However, Accelerate is designed to return only pairs that the model checker appears at least likely to visit in the future, thus mitigating most of this loss of precision.

8.1.2 Definition

This subsection introduces the various components of the Accelerate algorithm, with a focus on the ideas behind the algorithm.

The TryTrans Algorithm

Figure 8.2 lists the definition of an algorithm TryTrans, which is called by Accelerate. The purpose of TryTrans is to determine whether exploring a summary transition $K \overset{L}{\rightarrow} K'$ from a pair $⟨K, s⟩$ eventually leads to a pair $⟨K'', s⟩$ such that $K \sqsubseteq K''$. The below description walks through the algorithm step-by-step.

The algorithm mimics the exploration process of the model checker, so given a transition $K \overset{L}{\rightarrow} K'$, it first finds out whether there is a matching transition from the given PSM $s$ (line 3). The point of TryTrans is to find a pair $⟨K'', s⟩$ where $K''$ approximates $K$ but $s$ stays the same. So if there is no matching step $\{s\} \overset{(L,O)}{\rightarrow} S$ that leads back to $s$ again (i.e., $s \in S$), the algorithm indicates failure by returning false.

Otherwise, after finding a matching step from $s$, the model checker would transform each new pair with the transformation $T$. To mimic that step, TryTrans transforms the new pair $⟨K', s⟩$ with a transformation $T_{\text{NoAcc}}$ that includes all transformations used in the model checker except Accelerate itself (line 6). That set of transformations includes at least Unmark, Split, Assimilate, and Canonicalize, but may also include Evict and Detect (defined below), depending on what optimizations are enabled.

The results from $T_{\text{NoAcc}}$ represent the next program-configuration/PSM pairs the model checker would explore, so the algorithm determines whether that transformation yields some pair $⟨K'', s⟩$ with the same PSM $s$. If not, then the algorithm gives up. Otherwise, the algorithm picks one of those pairs arbitrarily. (Most of the time there is at most one such pair, because all others represent the extra PSMs generated by Split whose state definitions differ from those in $s$.)

Next, TryTrans repeats the process all over again to take a second, similar transition from $K''$. The reason is that the first transition might create some new entity (e.g., a new sent message or a new atomic actor) with exactly one copy, meaning that $K''$ does not approximate $K$. Taking a second similar transition might turn that single copy into a zero-or-more representation (such as a message with the quantity many or a collective actor), thereby giving TryTrans a better chance of finding an approximating configuration for $K$.

As an example, imagine an abstract configuration $K$ with no in-flight messages (i.e., $\mu = \emptyset$), and with an actor in the following state:
Function TryTrans($\hat{K}, s, L, \hat{K}'$) =

1. let $\langle \hat{\beta} \mid \mu \rangle H \rangle \hat{\beta} := \hat{K};$
2. if $\exists (O, S) \in \text{MatchingSpecSteps}(s, L)$ such that $s \in S$ then
   return false;
3. if $\exists \hat{K}'', A, M. \langle \hat{K}'', s, A, M \rangle \in T_{\text{NoAcc}}(\hat{K}', s)$ then
   return false;
4. let $\hat{K}'' :=$ an arbitrary $\hat{K}''$ such that $\langle \hat{K}'', s, A, M \rangle \in T_{\text{NoAcc}}(\hat{K}', s);$
5. if $\exists \hat{K}'', L \hat{K}' \rightarrow \hat{K}'''$ repeats $\hat{K} \rightarrow \hat{K}'$ then
   return false;
6. let $\hat{K}''' :=$ an arbitrary $\hat{K}'''$ such that $\hat{K}'' \rightarrow \hat{K}'''$ repeats $\hat{K} \rightarrow \hat{K}'$;
7. if $\exists \hat{K}'', A, M. \langle \hat{K}''', s, A, M \rangle \in T_{\text{NoAcc}}(\hat{K}', s)$ then
   return false;
8. let $\hat{K}'''' :=$ an arbitrary $\hat{K}''''$ such that $\hat{K}''' \rightarrow \hat{K}''''$;
9. if $\exists \exists (\hat{a}, H, \hat{v}), \mu'(\hat{a} \circ H, \hat{v}) = \text{single} \quad \text{and} \quad (\hat{a} \circ H, \hat{v}) \notin \text{dom}(\mu)$ then
   return false;
10. if $\hat{K} \sqsubseteq \text{id}, \text{id} \hat{K}'''' \text{ and } \hat{K} \neq \hat{K}''''$ then
    return $\hat{K}'''$;

Figure 8.2: An algorithm to determine whether repeating a given transition leads to an approximating program configuration. The algorithm is explained step-by-step below, including a definition of “repeats”. 
8.1. ACCELERATION

(\text{define-state} \ (\text{Ready} \ [\text{ping-dest} \ (\text{ Addr} \ (\text{Variant} \ [\text{Ping}]))])) \ (m)
\begin{align*}
&\ (\text{begin}) \\
&\ (\text{send} \ \text{ping-dest} \ (\text{variant} \ \text{Ping})) \\
&\ (\text{goto} \ \text{Ready} \ \text{ping-dest}))
\end{align*}

Now suppose that actor receives a message and runs the above transition. If ping-dest is an internal address \(\hat{a}\), then the message map \(\hat{\mu}'\) in the reached configuration will have \(\hat{\mu}'(\hat{a}@\phi, (\text{variant} \ \text{Ping})) = \text{single}\). Then \(\hat{\mu}'\) does not approximate \(\hat{\mu}\), because \(\hat{\mu}'\) represents only concrete message multisets \(\mu\) with exactly one Ping, but \(\hat{\mu}\) represents only those with zero Pings. However, if that same actor were to take that transition a second time, there would be multiple in-flight Pings, so the message map \(\hat{\mu}''\) in the reached configuration would have \(\hat{\mu}''(\hat{a}@\phi, (\text{variant} \ \text{Ping})) = \text{many}\). The quantity \text{many} represents zero or more concrete Ping messages, so the map \(\hat{\mu}''\) in the configuration reached after two transitions from \(\hat{K}\) does approximate \(\hat{\mu}\).

Defining this notion of similarity between two transitions \(\hat{K}_1 \xrightarrow{L_1} \hat{K}'_1\) and \(\hat{K}_2 \xrightarrow{L_2} \hat{K}'_2\) is subtle. Along with the external effects represented by the labels \(L_1\) and \(L_2\), each transition should also send the same internal messages, spawn the same actors, and reach the same state for the active actor. The following definition formalizes this idea.

\textbf{Definition.} A summary transition \(\hat{K}_1 \xrightarrow{L_1} \hat{K}'_1\) \textit{repeats} a transition \(\hat{K}_2 \xrightarrow{L_2} \hat{K}'_2\) if and only if there exist \(\hat{\beta}_1, \hat{\beta}_2, \hat{\mu}_i, \hat{\mu}'_i, H_i, H'_i, \hat{\rho}_i, \hat{\rho}'_i\) for \(i \in \{1, 2\}\) such that the following conditions hold:

\begin{itemize}
\item \(\hat{K}_1 = \langle \langle \hat{\beta}_1 | \hat{\mu}_1 | H_1 \rangle \rangle^{\hat{\beta}_1}\) and \(\hat{K}_1 = \langle \langle \hat{\beta}'_1 | \hat{\mu}'_1 | H'_1 \rangle \rangle^{\hat{\beta}'_1}\) for \(i \in \{1, 2\}\).
\item \(L_1 = L_2\).
\item There exist \(\hat{a}_1, \ldots, \hat{a}_n\) and \(\hat{b}_1, \ldots, \hat{b}_n\) such that \(\hat{\beta}'_1 = \hat{\beta}_1 \oplus [\hat{a}_1 \rightarrow \hat{b}_1] \oplus \ldots \oplus [\hat{a}_n \rightarrow \hat{b}_n]\) and \(\hat{\beta}'_2 = \hat{\beta}_2 \oplus [\hat{a}_1 \rightarrow \hat{b}_1] \oplus \ldots \oplus [\hat{a}_n \rightarrow \hat{b}_n]\).
\item There exist \((\hat{a}_1@H_1, \hat{v}_1), \ldots, (\hat{a}_n@H_n, \hat{v}_n)\) such that \(\hat{\mu}'_1 = \hat{\mu}_1 \oplus (\hat{a}_1@H_1, \hat{v}_1) \oplus \ldots \oplus (\hat{a}_n@H_n, \hat{v}_n)\) and \(\hat{\mu}'_2 = \hat{\mu}_2 \oplus (\hat{a}_1@H_1, \hat{v}_1) \oplus \ldots \oplus (\hat{a}_n@H_n, \hat{v}_1)\).
\end{itemize}

To implement this idea, \text{TryTrans} first determines whether there is an additional transition \(\hat{K}'' \xrightarrow{L'} \hat{K}''\) that repeats \(\hat{K} \xrightarrow{L} \hat{K}'\) (lines 10–13). In practice, this means checking whether the same event is still enabled in \(\hat{K}''\) and the actor is still in the same state, and then constructing a transition \(\hat{K}'' \xrightarrow{L'} \hat{K}''\) in which that same actor handles the event in the exact same way (i.e., by sending the same messages, spawning the same actors, and going to the same next state). Then \text{TryTrans} transforms the resulting pair \(\langle \hat{K}''', s \rangle\) with \(T_{\text{NoAcc}}\) and finally searches for a result \(\langle \hat{K}''', s, A, M \rangle\) in the transformation results with the same PSM \(s\) (lines 14–17). The discovered pair \(\langle \hat{K}''', s \rangle\) is a pair the model checker would reach after two transitions from the initial pair \(\langle \hat{K}, s \rangle\).
Having found such a pair, TryTrans checks on line 19 whether every single message in $\hat{K}'''$ is also a single message in $\hat{K}$, and rejects the configuration if not (this is necessary to show that Accelerate is conformance-reflecting; see appendix L). Finally, TryTrans determines whether $\hat{K}'''$ is a strict approximation of $\hat{K}$ (line 22). TryTrans uses the identity function as the address- and marker-correspondent functions for $\sqsubseteq$ to ensure that every address and marker in $\hat{K}$ corresponds to the same address and marker in $\hat{K}'''$; this simplifies the conformance-reflection proof for Accelerate.

If such an approximating pair has been found, then TryTrans returns it (line 23). Otherwise, TryTrans failed to find such an approximation, so it returns false instead (line 26).

An Outline of Accelerate

The Accelerate algorithm itself includes some subtle details that improve its performance. To focus on the main idea first, this section presents just an outline of the Accelerate algorithm. The next section then presents the full algorithm and explains the differences.

**Function** AccelerateOutline($\hat{K}, s$) =

1. let seen := $\{\hat{K}\}$;
2. let $W := \text{ProgSteps}(\hat{K})$;
3. while $W \neq \emptyset$ do
4. 1. let $\langle L, \hat{K}' \rangle :=$ an arbitrary element from $W$;
5. 2. $W := W - \{\langle L, \hat{K}' \rangle\}$;
6. 3. if $\hat{K}'' := \text{TryTrans}(\hat{K}, s, L, \hat{K}')$ and $\hat{K}'' \notin \text{seen}$ then
7. 4. seen := seen $\cup \{\hat{K}''\}$;
8. 5. $\hat{K} := \hat{K}''$;
9. 6. $W := \text{ProgStepsFor}(\hat{K}, E)$;
10. end
11. end
12. return $\{\langle \hat{K}, s, \text{id}, \text{id} \rangle\}$

Figure 8.3: An outline of the Accelerate algorithm

Figure 8.3 presents the outlined algorithm. To start, AccelerateOutline initializes a set of seen configurations with $\hat{K}$ and initializes its worklist $W$ with a set of possible summary transitions from $\hat{K}$ (see the definition of ProgSteps in section 7.3). Then within the worklist loop itself, AccelerateOutline uses TryTrans to find out whether any one of those transitions leads to a previously unseen approximating configuration $\hat{K}''$. If so, that configuration is assigned as the current configuration $\hat{K}$, and the worklist is reset with the possible transitions from that configuration to determine if the configuration can be approximated even further.
8.1. ACCELERATION

Once AccelerateOutline runs out of transitions to try (meaning that no summary transition \( \hat{K} \xrightarrow{L} \hat{K}' \)) from the current configuration \( \hat{K} \) leads to some unseen approximating configuration), the loop terminates. AccelerateOutline then returns the tuple \( (\hat{K}, s, id, id) \), which tasks the model checker with showing that \( \hat{K} \) transformation-conforms to \( s \). AccelerateOutline returns the identity function as the mappings between the old and new markers and addresses because the conditions on line 10 of figure 8.2 guarantee that every marker or address in the original given pair corresponds to the same entity in the produced pair.

The Full Accelerate Algorithm

The problem with AccelerateOutline is that \( \text{ProgSteps}(\hat{K}) \) tends to yield a large number of transitions (in the hundreds, if not thousands or more), so it is time-consuming to take every transition multiple times and then apply \( T_{\text{NoAcc}} \) to every result. Often, attempting every possible transition can actually cause Accelerate to slow down the model checker rather than speed it up.

As a result, Accelerate performs some lightweight checks to gauge how likely each transition is to reach an approximating configuration before giving that transition to TryTrans. The criteria used to judge that likelihood are just heuristics, so better ones might be found with further experimentation. The following describes the heuristics used in the current implementation and segues into describing the Accelerate algorithm itself, listed in figure 8.4.

The following discussion requires a couple of formal definitions. For all summary transition labels \( L = (\hat{l}, \hat{\mu}) \), let \( \hat{l} \) be the event for \( L \) (it represents the event that triggers the rest of the transition). Then let \( \hat{l} \) be summary-enabled in a configuration \( \hat{K} \) if and only if there exist some \( L \) and \( \hat{K}' \) such that \( \hat{l} \) is the event for \( L \) and \( \hat{K} \xrightarrow{L} \hat{K}' \).

The first lightweight check uses some heuristics to filter out events that are unlikely to yield an approximating configuration when handled. To explain the idea, suppose that Accelerate produced the configuration \( \hat{K}_{\text{NFD}} \) described above by taking a transition \( \hat{K}_{\text{NF}} \xrightarrow{L} \hat{K}_{\text{NFD}} \) in which the collective processor actor handles a Shutdown message, leading it to the Done state. Later on, the model checker might explore a transition \( \hat{K}_{\text{NFD}} \xrightarrow{L'} \hat{K}'_{\text{NFD}} \) that does not affect the collective actor, e.g., a transition in which an atomic processor actor transitions from Off to On. When applying Accelerate to that new configuration \( \hat{K}'_{\text{NFD}} \), it would be a waste of time for Accelerate to check a transition from \( \hat{K}'_{\text{NFD}} \) to \( \hat{K}'_{\text{NFD}} \) in which the collective processor actor handles a Shutdown message again, because the previous call to Accelerate already took that transition, and the transition \( \hat{K}_{\text{NFD}} \xrightarrow{L'} \hat{K}'_{\text{NFD}} \) does not change how that actor would handle the message.

In light of this, Accelerate only attempts to handle an event if the previous transition was likely to change how that event might be handled. An event identified by a label \( \hat{l} \) is defined to be updated by a transition \( \hat{K} \xrightarrow{L} \hat{K}' \) if and only if at least one of the following holds:
Function Accelerate(\(\hat{K}, s\)) =

\[
\begin{align*}
\text{let } E := & \emptyset; \\
\text{if } \langle \hat{K}, s \rangle \text{ has a previous transition } \hat{K}' \xrightarrow{L} \hat{K}'' & \text{ then} \\
E := & \{ \hat{i} | \hat{i} \text{ is summary-enabled in } \hat{K} \text{ and } \hat{K}' \xrightarrow{L} \hat{K}'' \text{ updates } \hat{i} \}; \\
\text{end} \\
\text{else} & \\
E := & \{ \hat{i} | \hat{i} \text{ is summary-enabled in } \hat{K} \}; \\
\text{end} \\
\text{let } seen := & \{ \hat{K} \}; \\
\text{let } W := & \text{ProgStepsFor}(\hat{K}, E); \\
\text{while } W \neq \emptyset & \text{ do} \\
\text{let } \langle L, \hat{K}' \rangle := & \text{ an arbitrary element from } W; \\
W := & W - \{ \langle L, \hat{K}' \rangle \}; \\
\text{unless IsAccelerateCandidate}(\hat{K}, L, \hat{K}'') & \text{ continue}; \\
\text{end} \\
\text{if let } \hat{K}'' := & \text{TryTrans}(\hat{K}, s, L, \hat{K}'') \text{ and } \hat{K}'' \notin seen \text{ then} \\
seen := & seen \cup \{ \hat{K}'' \}; \\
E := & E \cup \{ \hat{i} | \hat{i} \text{ is summary-enabled in } \hat{K}'' \text{ and } \hat{K} \xrightarrow{L} \hat{K}' \text{ updates } \hat{i} \}; \\
\hat{K} := & \hat{K}''; \\
W := & \text{ProgStepsFor}(\hat{K}, E); \\
\text{end} \\
\text{return } \{ \langle \hat{K}, s, id, id \rangle \} \\
\text{end}
\end{align*}
\]

Figure 8.4: The Accelerate algorithm
8.1. ACCELERATION

- The active actor for $\hat{l}$ is the active actor for $L$.
- The active actor for $\hat{l}$ is one of the actors spawned in $\hat{K} \xrightarrow{L} \hat{K}'$.
- $\hat{l}$ represents receiving an internal message not present in $\hat{K}$, i.e., $\hat{l} = \hat{a} : \text{rcv-int}(H, \hat{u})$ and $\hat{l}$ is not summary-enabled in $\hat{K}$.

Lines 3–8 in figure 8.4 use this idea to filter which summary-enabled events Accelerate will attempt transitions for. The previous transition for a pair $\langle \hat{K}, s \rangle$ is the transition $\hat{K}' \xrightarrow{L} \hat{K}''$ such that some transformation-step transition $\langle \hat{K}', s' \rangle \xrightarrow{L, \hat{K}'', Z, O, S, T, A, M} \langle \hat{K}, s \rangle$ was the first step the model checker took to reach $\langle \hat{K}, s \rangle$ (the model checker's initial explored pair $\langle \hat{K}_{\text{init}}, s_{\text{init}} \rangle$ has no previous transition). Formally this transition should be given as an additional input to every transformation $T$, but because it is used only in Accelerate and because it affects only the performance of Accelerate and not its correctness, this presentation leaves it out; we can think of it instead as a function on the given pair $\langle \hat{K}, s \rangle$ defined as the model checker executes.

Once Accelerate has the set of events $E$ to attempt transitions for, it then uses \texttt{ProgStepsFor} to compute the set of possible summary transitions for those events. The function \texttt{ProgStepsFor} is just like \texttt{ProgSteps} from section 7.3, except that it only returns summary transitions $\hat{K} \xrightarrow{L} \hat{K}'$ in which $E$ contains the event for $L$.

That set of transitions from $\hat{K}$ becomes the algorithm's worklist. For each transition in the list, Accelerate performs a second lightweight check to determine whether to apply \texttt{TryTrans} to that transition. Specifically, the function \texttt{IsAccelerateCandidate} (line 13) returns true for $\langle \hat{K}, L, \hat{K}' \rangle$ if and only if

- all atomic actors spawned during the transition have a corresponding actor in $\hat{K}$ with the same spawn location and current state, and
- at least one of the following holds:
  - At least one new actor is spawned in the transition $\hat{K} \xrightarrow{L} \hat{K}'$.
  - Let $\hat{a}$ identify the active actor for $L$, $\hat{b}$ be the behavior for $\hat{a}$ in $\hat{K}$, and $\hat{b}'$ be the behavior for $\hat{a}$ in $\hat{K}'$. Then $\hat{b}'$ approximates $\hat{b}$.
  - The transition $\hat{K} \xrightarrow{L} \hat{K}'$ sends an internal message that either did not exist in $\hat{K}$, or existed only with abstract quantity single.

If that succeeds and \texttt{TryTrans} returns a new approximating configuration $\hat{K}''$ (line 17), then Accelerate adds to the set of events $E$ any events that might be handled differently after taking the transition $\hat{K} \xrightarrow{L} \hat{K}'$ (line 19). Accelerate then generates the set of transitions from the new $\hat{K}$ based on that updated set of events and updates the worklist; the remainder of the algorithm is similar to AccelerateOutline.
(define-state (Stopped) m
  (case m
    [(Stop) (goto Stopped)]
    [(Start timeout-in-milliseconds)
      (goto Running timeout-in-milliseconds)])
)

(define-state (Running [timeout-in-milliseconds Nat]) m
  (case m
    [(Stop) (goto Stopped)]
    [(Start new-timeout-in-milliseconds)
      (goto Running new-timeout-in-milliseconds)])
    [(timeout timeout-in-milliseconds)
      (begin
        (send expiration-target expiration-message)
        (goto Stopped))])

Listing 8.1: State definitions for a timer actor

8.1.3 Conformance Reflection

Theorem. Accelerate is a conformance-reflecting transformation.

The proof of this theorem is in appendix L. Similar to the conformance-reflection proofs for Unmark and Assimilate, the main idea is to show that the program configuration \( \hat{K} \) returned by Accelerate is an approximation of the given configuration \( \hat{K} \), so transformation conformance for \( (\hat{K}, s) \) implies transformation conformance for \( (\hat{K}, s) \).

8.2 Eviction

8.2.1 Motivation

Sometimes an actor may contribute greatly to the state-space blowup of a program, but precisely modeling its behavior is unnecessary to prove conformance. For example, the TCP implementation described in the next chapter uses separate timer actors to help each TCP session manage its timeouts. Listing 8.1 shows that actor's state definitions: a Start message starts the timer, at the expiration of which the actor sends a given message to a previously provided expiration-target, and a Stop message cancels any active timeout.

The possible combinations of that actor's states and incoming messages contribute to state-space explosion: the actor may be running, it may be stopped, it may have a Start message incoming but no Stop message, it may have many instances of both Start and Stop incoming, etc. None of these differences are important, however, because the session actor that interacts with the timer is written such that it conforms to its specification regardless of what the timer does.
8.2. EVICTION

An optimization called eviction helps the model checker avoid modeling such irrelevant differences. Before running the model checker, the programmer may declare certain syntactic spawn locations in the program as “evictable”. (Thus, this optimization requires a small amount of manual input.) Whenever the model checker spawns an actor at a location marked as evictable (and certain preconditions hold, described below), this optimization removes, or evicts, the actor from the configuration and moves it into the environment instead. Thus, the new configuration models the evicted actor as an unknown external actor so that its current state and incoming messages need not be modeled.

Because an evicted actor is not modeled as precisely as a non-evicted one, eviction causes an overall loss of precision in the model checker. Thus, users of the model checker should be careful to mark an actor’s spawn location as evictable only when that actor’s precise behavior is irrelevant for conformance.

The next section defines some terms necessary for eviction, then defines eviction as a transformation called Evict.

8.2.2 Definition

Before being evicted, an actor might have access to addresses for other internal actors. To account for the messages the evicted actor might have sent to those other actors, eviction leaves behind a set of messages with the quantity many that represent every possible message that can be sent to that each of those actors.

The messages are constructed based on the types at which the evicted actor has access to those other addresses. Those types are defined by replacing all internal addresses in the evicted actor’s behavior with fresh variables, then providing some type environment $\Gamma$ that allows those new behaviors to type-check.

This is formalized in the below definition.

**Definition.** A set $\hat{\rho}' = \{ (\hat{a}'_1 \@ H'_1, \tau_1), \ldots, (\hat{a}'_n \@ H'_n, \tau_n) \}$ is a valid typing for the known internal addresses of the actor at address $\hat{a}$ in a program configuration $\hat{K} = \langle \hat{\beta} \mid \hat{\mu} \mid H \rangle^\delta$ if and only if there exist $\hat{b}_1, \ldots, \hat{b}_m, \hat{b}'_1, \ldots, \hat{b}'_m$, and $x_1, \ldots, x_n$ such that

- $\hat{\beta}(\hat{a}) = \{ \hat{b}_1, \ldots, \hat{b}_m \}$,
- for all $\hat{a}''$ and $H''$, there exists $\tau'$ such that $(\hat{a}'' \@ H'', \tau') \in \hat{\rho}$ if and only if there exists $i \in 1 \ldots m$ such that $\hat{a}'' \@ H''$ appears in $\hat{b}_i$ and $\hat{a}''$ is internal (that is, $\hat{\rho}$ represents the set of marked internal addresses in $\{ \hat{b}_1, \ldots, \hat{b}_m \}$),
- for all $i \in 1 \ldots n$, there does not exist $\tau' \neq \tau_i$ such that $(\hat{a}'_i \@ H'_i, \tau') \in \hat{\rho}$ (that is, $\hat{\rho}$ contains exactly one typing for each marked address),
- $x_1, \ldots, x_n$ do not appear in $\hat{b}_1, \ldots, \hat{b}_m$,
- $\hat{b}_i = \hat{b}'_i[x_1 \leftarrow \hat{a}'_i \@ H'_i] \ldots [x_n \leftarrow \hat{a}'_n \@ H'_n]$ for all $i \in 1 \ldots m$, and
• $\exists \tau', \Gamma. ~ \tau' = \text{ActorType}(\hat{a}), \Gamma = [x_1 \to (\text{Addr } \tau_1), \ldots, x_n \to (\text{Addr } \tau_n)],$ and $\\tau', \Gamma \vdash_{\text{beh}} \hat{b}_j$ for all $j \in 1 \ldots m$.

Eviction is only a valid technique when a number of preconditions hold, defined below.

**Definition.** An actor at address $\hat{a}$ is **evictable** from a program-configuration/PSM pair $\langle K, s \rangle$ (with $K = \langle \hat{\beta} \mid \hat{\mu} \mid H \rangle$ and $s = \langle H', H'', \varphi : \eta, \overline{\Omega}, O \rangle$) if and only if all of the following conditions hold:

1. **Evictable Location** There exist $\ell$ and $n$ such that either $\hat{a} = (\text{addr } \ell \ n)$ or $\hat{a} = (\text{collective-addr } \ell)$, and $\ell$ was declared as an evictable location.

2. **No Monitored Addresses** For all marked addresses $\hat{a}'@H''$ appearing in $\hat{\beta}(\hat{a})$, $H'' \cap H'' = \emptyset$.

3. **No Monitored Receptionists** For all $H''$ and $\tau$ such that $\langle \hat{a}@H'', \tau \rangle \in \hat{\rho}$, $H'' \cap H' = \emptyset$.

4. **Cannot Receive Addresses** For all $\hat{a}'$ such that either $\hat{a}' = \hat{a}$ or $\hat{a}'$ appears in $\hat{\beta}(\hat{a})$, there is no type $\tau$ such that $(\text{Addr } \tau)$ appears in ActorType($\hat{a}'$).

5. **Well-Typed** The program configuration $\hat{K}$ type-checks (i.e., $\vdash_{\text{cfg}} \hat{K}$).

6. **No Spawns** No spawn expression appears in $\hat{\beta}(\hat{a})$.

Precondition 1 requires that the actor is one of the ones marked by the user as evictable in the original program. Preconditions 2 and 3 ensure that messages sent to or from this actor are irrelevant to the specification, because the model checker cannot observe messages an external actor sends to or receives from the rest of the environment. In particular, those preconditions require that the actor's current behavior(s) contain no addresses marked with an output-monitored marker, and that none of its receptionists are marked with an input-monitored marker.

Precondition 4 says that the type of $\hat{a}$ as well as the addresses contained in that actor forbid the actor from sending or receiving further addresses. The well-typedness requirement in precondition 5 ensures that those types are respected. Finally, precondition 6 requires that the evicted actor cannot spawn any other actors. None of these three are strictly necessary, but they simplify both the
8.2. EVICTION

Definition of Evict and its conformance-reflection proof.

Figure 8.5: The Evict algorithm

Figure 8.5 defines the transformation Evict as an algorithm. The algorithm works through each of the internal actors in \( \hat{K} \) one by one and evicts all evictable actors. For each one, the algorithm

- removes that actor from the configuration (lines 7 and 22)
- removes all messages to that configuration (lines 8–13),
- adds new messages with quantity many (by adding each one twice) to account for the messages the evicted actor might have sent, (lines 14–20),
and

- renames all remaining occurrences of the evicted actor’s address to the new external address that identifies the evicted actor (line 22).

The new address $\hat{a}''$ is chosen such that it is the representative address for the type $\text{ActorType}(\hat{a})$, as described in appendix C.

The function $\text{MaxVals}$ defines a “maximal” set of values representing all possible values of type $\tau$. The function is defined and explained in further detail in appendix I. Because an address is evictable only if none of the internal addresses it knows about can receive an address, that set is finite.

The actual implementation of $\text{Evict}$ chooses the smallest possible types for the new receptionists $\hat{\rho}''$ (i.e., the types that accept the smallest set of messages). Eviction over-approximates the set of messages an evicted actor might have sent to the rest of the program, but this strategy at least ensures that the environment can send a message of type $\tau$ to one of those new receptionists only if the evicted actor had access to that receptionist’s address at type $\tau$.

### 8.2.3 Conformance Reflection

**Theorem.** $\text{Evict}$ is a conformance-reflecting transformation.

Appendix L gives a proof of this theorem. The key is to show that for any message the evicted actor would have been able to send to other actors in the program configuration, the new receptionists $\hat{\rho}''$ allow the environment to send the same message.

### 8.3 Dead-Marker Detection

Recall from section 6.4 that the transformation $\text{Split}$ creates PSMs that specify only the outputs to be sent to an address with a particular marker $\eta$. For example, while model checking an implementation of the stream-processing protocol, $\text{Split}$ might create a PSM $\langle \phi, \{\eta_{\text{resp}}\}, \varphi : \eta_{\text{resp}} : \overline{\Phi} : \langle \eta_{\text{resp}}, (\text{variant } \text{Ok}) \rangle \rangle$ requiring only that the program send an Ok response to an address marked with $\eta_{\text{resp}}$. The state definitions $\overline{\Phi}$ created by $\text{Split}$ contain no state definitions, so that PSM can never generate a new obligation. Therefore, after the program fulfills all existing obligations for $\eta_{\text{resp}}$, the residual PSM $\langle \phi, \{\eta_{\text{resp}}\}, \varphi : \eta_{\text{resp}} : \overline{\Phi}, \phi \rangle$ merely specifies that the program may not send further, unwanted messages to an address marked with $\eta_{\text{resp}}$. If $\eta_{\text{resp}}$ no longer appears anywhere in the program configuration (i.e., $\eta_{\text{resp}}$ is dead), then that configuration trivially conforms to the PSM, because the configuration has no address marked with $\eta_{\text{resp}}$ to send a message to.

This section introduces an optimization called dead-marker detection that detects such program-configuration/PSM pairs so that the model checker can stop its exploration at that point. Specifically, whenever the model checker encounters a pair $\langle K, s \rangle$ in which
8.4 Order of Optimizations

When all three of the above optimizations are enabled, the model checker’s full transformation $T$ is defined as the following composition:

$$Detect \circ Accelerate \circ Canonicalize \circ Assimilate \circ Unmark \circ Split \circ Evict$$

Thus, $Evict$ is applied first, and $Accelerate$ and $Detect$ are applied last. When a given optimization is disabled, the transformation $T$ is as above, but with the
particular transformation for that optimization removed from the sequence of compositions.

### 8.5 Memoization

Unlike the previous three optimizations, this last optimization does not involve a transformation; it simply consists of memoizing a key function in the model-checking algorithm. To explain the idea, recall that the Explore algorithm from figure 7.6 included the following line to summarize all abstract event steps \( \hat{K} \xrightarrow{\hat{l}_1, \ldots, \hat{l}_n} \hat{K}' \) from the current program configuration \( \hat{K} \):

\[
\text{nextSteps} := \text{ProgSteps}(\hat{K});
\]

Summarizing all possible abstract event steps from \( \hat{K} \) is an expensive operation. However, most of the cost comes from computing the possible effect (i.e., sent messages, spawned actors, and state changes) of a given handler expression \( \hat{e} \), and often different event steps across different program configurations consist of evaluating the same handler expression. Furthermore, a handler expression’s effects are independent of the context in which the expression evaluates (e.g., actor A’s current state does not affect how actor B handles an incoming message). To save time and avoid recomputing the results every time, the model checker memoizes the computation of each handler expression’s effects in the implementation of ProgSteps.

Along with speeding up the Explore algorithm, this optimization also benefits acceleration. Accelerate often has to compute the effects of evaluating a given handler expressions many times over, so memoizing those effects avoids wasting time on redundant computations.
Chapter 9

Evaluation

My thesis claims that

finite-state, address-passing specifications can be used to automatically verify non-trivial protocols in actor programs.

Thus, I claim that a specification language like APS occupies a “sweet spot” between expressive power and reasoning power: it is expressive enough to describe non-trivial protocols, yet restricted enough to enable automated reasoning.

To test the expressiveness half of the thesis, I evaluated APS against the following criterion:

• Actor programmers occasionally make mistakes that violate conformance to an APS specification.

Whenever such a mistake exists, the ModelCheck Correctness theorem guarantees that the model checker either diverges or reports non-conformance. The more difficult task is for the model checker to prove conformance when the program is free of such mistakes. Therefore, to test the automated reasoning half of the thesis, I evaluated the model checker against the following two criteria:

• The model checker can prove that realistic programs conform to non-trivial specifications.

• The model checker takes a reasonable amount of time to check conformance.

The evaluation consisted of writing several realistic actor programs and non-trivial specifications, recording any conformance-violating bugs uncovered during development (either in the program or the specification), and model checking both the buggy and corrected programs against their specifications. Each check was allowed to run for at most 24 hours, as an initial threshold for a “reasonable amount of time”. These checks both determine how effective the model checker is at proving conformance and also provide initial data on how long it takes to check buggy and bug-free programs.
The evaluation also investigated aspects of the model checker’s performance. Specifically, the evaluation measured the time taken by each step of the model-checking algorithm while checking the bug-free programs, and it measured the impact of optimizations from chapter 8 by running the model checker with different sets of enabled optimizations.

This chapter does not argue that APS specifications or the model checker are ready for real-world use. State-space explosions in more complicated programs sometimes increase the verification time dramatically, and there is plenty of room for more performance engineering as well as changes to APS itself. Instead, the goal of this chapter is to show only that APS-like specifications are worth further exploration as a means of specifying protocols for actor-based systems. A more rigorous performance analysis is left until after further improvements have been made.

The next section describes the evaluated programs and the kinds of properties their associated specifications check. Section 9.2 describes the experimental setup used to evaluate the model checker on those programs and specifications. Sections 9.3 and 9.4 present the results from the effectiveness and performance portions of the evaluation, respectively. Section 9.5 discusses some of the lessons learned, and section 9.6 suggests future research and engineering directions based on the results.

9.1 Evaluated Programs and Specifications

This section describes the CSA programs and APS specifications written for the evaluation. The programs were selected with the intent of representing a wide range of actor-communication patterns that can be expressed in APS. Some of them are reimplementations of real-world Erlang or Akka programs in CSA, while others are artificial examples intended to represent well-known communication patterns. The individual descriptions below say which is which.

As a general rule, each program based on a real-world application implements only the core protocol, as described below, rather than the full application. A more thorough evaluation would evaluate the model checker on fully fleshed out programs, but these core reimplementations suffice for an initial investigation into the model checker’s capabilities.

Some of the example programs have multiple specifications, or multiple versions of the program each with their own specification. The term benchmark is used to refer to a particular program/specification pair.

For ease of implementation and to more closely match how real actor programs are written, these examples are written in a version of CSA that includes syntactic sugar for:

- `let*`;
- `if` and `cond`;
- functions (but not recursive or higher-order functions);
9.1. EVALUATED PROGRAMS AND SPECIFICATIONS

- named definitions for actors, variant types, record types, (transparent) type aliases, and constants; and

- multi-expression bodies for forms such as case and message handlers.

To give the reader a sense of the “flavor” of the evaluated programs, the next two subsections describe the implementation and specification for two of the programs (Raft and Apache Flink) in detail. Brief descriptions for each of the remaining programs and specifications follow.

9.1.1 Raft

Overview

Raft [86] is a consensus protocol, intended to serve as an easier-to-understand replacement for Lamport’s Paxos protocol [71]. The purpose of a consensus protocol is to present a cluster of nodes to the outside world as a single distributed system that continues to operate even when some of its nodes fail. The system accepts client requests from the outside world, and the consensus protocol ensures that each node in the cluster processes the same sequence of requests (which each node records as its log). This ensures that even if a node fails, the other nodes are in the same state as the failed node and can continue processing client requests. The following provides a brief overview of Raft’s operation; see the full paper for a more detailed description.

During normal operation, one node per cluster is in the Leader state while all others are in the Follower state. When the Leader node receives a client request, it sends an AppendEntries message to all other nodes in the cluster, asking whether that request is acceptable given the other requests processed so far. A node can accept the request with an AppendSuccessful response, but might reject it with an AppendRejected response if, for example, some newer Leader node has been elected. If the leader receives AppendSuccessful responses from a majority of the cluster, then it commits the request to its log and instructs all other nodes to do the same.

AppendEntries messages also serve as heartbeat messages to the rest of the cluster. If a certain amount of time elapses before a Follower node receives a heartbeat message, it assumes the leader has failed. That node then transitions to the Candidate state and sends a RequestVote message to all other nodes to start a new election. In a given election cycle, each node votes for the first candidate it hears about, as long as that candidate’s log is up to date (the RequestVote message contains relevant log information). Once a Candidate node receives a majority of votes, it transitions to the Leader state and tells all other nodes to transition to the Follower state so that normal operation may resume.

In the particular implementation upon which I based my version of Raft, each node starts in an Uninitialized state and does not know any other nodes in the cluster. The node waits until it receives a Config message, which contains the addresses of the other nodes, then transitions to the Follower state.
CHAPTER 9. EVALUATION

Uninitialized Follower Candidate Leader receives cluster configuration times out, starts election receives update from newer leader receives update from new leader receives update from newer leader receives majority of votes

Figure 9.1: Summary of the Raft protocol as a state machine

Figure 9.1 summarizes the operation of a Raft node as a state machine. Actions that do not change the state, such as sending an AppendEntries message, are omitted from the diagram.

My Implementation and Specification

My implementation of Raft is based on a Scala/Akka project called akka-raft [6] by Konrad Malawski of the core Akka team. The project is a hobby project for the author rather than a supported Akka module. It is also a work in progress, which is good for this evaluation: it gives the model checker a chance to find the kinds of mistakes that an experienced actor programmer might make, before they have been fixed in a production-ready version of the program.

My implementation focuses on the actor implementing a Raft node and its operation in terms of the core protocol, including messages for adding new client requests to the log and conducting elections. Orthogonal features, such as changing the cluster membership after initialization and persisting accepted requests to disk, were omitted.

This program is the only one of the examples I transliterated into CSA, rather than reimplemented from scratch. Because this was the only non-production-ready real-world program I evaluated, I wanted to include any bugs that the author might have written. I translated sophisticated Scala features into simpler CSA constructs as necessary, but I tried to stay faithful to the structure of the code as much as possible.

Surprisingly, the specification I wrote for Raft has only two states (Uninitialized and Running), rather than the four shown in figure 9.1. This is because an APS specification can only differentiate messages based on their shape, but the transitions between the Follower, Candidate, and Leader states are determined by more than shape alone. For example, when an actor in the Candidate state receives a vote from another node, that actor may or may not transition to the Leader state; it depends on whether that vote grants a majority. The only transition determined by the shape of a message is the transition from Uninitialized
to Follower, triggered by receiving any initial Config message. The other three states are indistinguishable in APS, so my specification merges them together.

The specification I wrote expresses that, once initialized, a Raft actor should have the following behavior:

- Every AppendEntries request receives a single AppendSuccessful or AppendRejected response.
- Every RequestVote request receives a single VoteCandidate or DeclineCandidate response.
- After receiving an AppendRejected message, the actor may send back another AppendEntries request. (This can happen when the AppendRejected indicates the rejecting actor's log is out of date and needs to back-fill old entries.)

Model checking even this weak specification, however, found five conformance violations: four in the program, and one mistake in my own specification. Section 9.3.2 discusses these further.

### 9.1.2 Apache Flink

#### Overview

Apache Flink [9] is an actor-based framework for stateful processing of streams of data, either in streaming or batch modes. The Flink project website suggests using Flink for

- event-driven applications which store their data locally (in memory or on disk) instead of in a remote database;
- real-time data analytics applications which update the results of an analysis as new events are observed; and
- data-pipeline applications which transform data as it is generated, instead of periodically polling a source database for changes to the data.

Flink is a large system comprising many components, but my focus in this evaluation is on the handful of actors that actually manage and execute data-processing jobs submitted by users. The primary actor is the JobManager, which receives jobs from clients, guides their execution through the system, and reports results back to the user. Each job is divided into a number of tasks, which the JobManager distributes to worker actors called TaskManagers. Thus, Flink exemplifies the common master/worker pattern.

When a TaskManager starts, it must register itself with the JobManager by sending a RegisterTaskManager message. The TaskManager then waits until it receives an acknowledgment from the JobManager. If any other messages are received while awaiting the acknowledgment, the TaskManager sends back a rejection response.
Figure 9.2: The process of submitting and executing a job. My reimplementation includes only the JobManager and TaskManager components.

Figure 9.2 illustrates the steps taken to submit and run a job.\(^1\) Upon receiving a SubmitJob request from the client, the JobManager distributes tasks from that job to the TaskManagers with SubmitTask messages. A TaskManager has a fixed number of threads on which it can run tasks, so if no thread is available, the TaskManager rejects the request. Otherwise, it acknowledges the receipt of the task to the JobManager and starts running the task. A task contains many chunks of data to be processed, so the TaskManager periodically requests the next chunk from the JobManager with a RequestNextInputSplit message. Once the task is complete, the TaskManager sends the results back to the JobManager. Tasks can have dependencies on one another, so the completion of one task may enable the JobManager to dispatch additional downstream tasks. Once all of a job’s tasks are completed, the JobManager sends the final result to the client.

The client might also decide to cancel a job after submitting it by sending the JobManager a CancelJob message. If the job is no longer active (e.g., if the job is already complete or was canceled earlier), then the JobManager sends back a failure response. Otherwise, the JobManager tells the TaskManagers to cancel any tasks related to that job and sends a success response back to the client.

\(^1\)Image originally from https://cwiki.apache.org/confluence/display/FLINK/Akka+and+Actors. Reused with permission of the creator, Till Rohrmann.
My Implementation and Specification

My version of the Flink system is a CSA program that models the above protocol with JobManager and TaskManager actors. Additionally, because CSA does not have a notion of threads independent of actors, my program represents each TaskManager thread with an actor called a TaskRunner. Because Flink is such a large and sophisticated framework, I omitted many details of Flink from this model, but the protocol used by the JobManager and TaskManagers is effectively the same.

Flink represents the different jobs a user can submit with objects and higher-order functions. Because CSA lacks those features, my implementation has just a single kind of job: a MapReduce-based job to count the number of occurrences of each word in a given document. A Map task converts each chunk of the document into a dictionary from words to occurrence counts, and a Reduce task combines dictionaries from different chunks. This simple job exercises the main aspects of the Flink protocol, such as running different kinds of tasks and managing dependencies between tasks.

There is one place worth mentioning in which my implementation differs from the actual program. In the actual version of Flink, if a TaskManager does not receive a response to its registration message within a certain period of time, it will resend the message. Unfortunately, a TaskManager with this behavior would conform only to a weak specification that allows transitions between states at any time. This is because each registration message contains a separate copy of the TaskManager's address (distinguished by history marker), but a PSM can monitor messages sent to only one of those copies. As a result, I removed the retry behavior from my Flink implementation rather than weaken the specification so that this example would illustrate the kind of behavior a strong APS specification can describe.

To specify and check different aspects of the protocol, I actually have two different versions of this program. The first includes only a TaskManager actor, and its specification describes its expected interaction with a JobManager in that program’s environment. The second includes both a JobManager and multiple TaskManagers, and it has two different specifications: one that specifies the JobManager’s interaction with a client process, and another that specifies the JobManager’s interaction with TaskManagers. The list below gives a sample of the properties expressed in these specifications.

1. Specification for the TaskManager

   • The TaskManager rejects all tasks until its registration is acknowledged.
   • Once registered, the TaskManager responds to every SubmitTask and CancelTask message with either a success or failure message.

2 An older version of APS was able to handle this issue by treating all copies of the address as “the same thing”, but this approach caused other problems. There may be a modification to APS that would allow this retry behavior, but that is left as future work.
2. Specification for the JobManager, from the TaskManager’s point of view
   • Every request for the next chunk of input receives a response.
   • Upon receiving a registration message from a TaskManager, the JobManager sends an acknowledgment, and thereafter it may send that TaskManager any number of SubmitTask or CancelTask messages.

3. Specification for the JobManager, from the client’s point of view
   • Upon receiving a SubmitJob request, the JobManager can send back to the client any number of success or failure messages. (The abstract interpretation used in the model checker cannot distinguish the different jobs managed by the JobManager, so the model checker is unable to prove that the JobManager sends exactly one response back.)
   • Every CancelJob request gets a success or failure response.

9.1.3 Other Programs
Along with Raft, Flink, and the stream-processing example from chapters 2 and 3, the evaluation included seven other applications. These are described briefly below.

Authentication Server This program models a simple password-based authentication protocol for granting access to a server. The protocol is adapted from a paper on session types in Scala [96]. The specification expresses that the sequence of communication flows as expected: an initial GetSession request receives either an old session or a request to start a new one. In the case of a new session, a username and password are required before being granted a session. Once a session has been obtained, the client can ping the server and expect a response.

Plain Old Telephony Service (POTS) This program is based on an Erlang tutorial for programming finite-state machines commonly found in various sources, such as Armstrong, Virding, and Williams’s book [10]. The single actor models a controller that is responsible for looking up hardware addresses based on telephone numbers, establishing and closing connections between telephones, and sending signals to telephones (e.g., to cause one to ring or play a dial tone). The APS specification for this program effectively encodes the state machine diagram that typically accompanies this tutorial.

Chat Server This example models a simple IRC-like chat system. When a user connects to the server and asks to join a chat room, the server either connects that user to the room if it already exists, or creates that room on the fly if it does not. Every chat room is managed by a separate actor, which is responsible for maintaining the member list and forwarding messages to all members of the room. The specification expresses properties such as a JoinRoom request results
in a response including the address for that room’s actor, and a request for the list of a room’s members always gets a response.

**ENSIME**  ENSIME [44] is a piece of software that adds IDE-like features such as auto-completion and refactoring to text editors for Java and Scala projects. At the top level is a Project actor, which dispatches requests from the text editor to well-known actors depending on the kind of request (e.g., a documentation request, an auto-complete request, etc.). My implementation is simple: the Project actor implements the dispatching logic, but all code in the other actors to actually handle the requests is stubbed out. Each actor just sends back a dummy response. The purpose of this example is to find out how well the model checker can handle a program with several different actors in a static topology.

**TCP**  This program includes a complete (though basic) implementation of TCP [90]. Only packet-parsing is omitted, because CSA does not have the capabilities for parsing byte streams. The program communicates with its environment with IP-layer messages on one end, and application-layer messages on the other. The application-layer interactions are modeled on Akka’s TCP module.\(^3\)

Unfortunately, my original implementation of TCP that could handle both active connections (i.e., initiated by the local machine) and passive connections (i.e., initiated by a remote machine) could not be model checked in a reasonable amount of time due to state-space explosions. As a result, the program is split into two versions: one that includes the logic only for active connections, and another that includes the logic only for passive connections. The specifications for each version describe the expected communication with the application layer.

**TCP Session Controller**  The APS model checker is unable to reason precisely about the sliding-window aspects of TCP, which prevents it from proving interesting properties about the network-level communication in my original TCP implementation. This session-controller example is a component from a hypothetical alternative design in which the the sliding-window aspects of TCP are handled in a different actor than the session-management aspects. The program represents flags such as SYN and ACK in TCP packets with variants rather than numbers so that APS can pattern-match on those flags.

The APS specification for this TCP session-controller component looks much closer to the classical TCP state-machine diagram, with SynSent, SynReceived, Established, Closing, and Closed states. The increased reasoning power gained with this design suggests that subdividing components in this fashion may be useful to actor programmers.

**Spray-Can**  The final example is a model of spray-can, an Akka-based HTTP server from the spray project [104]. The server’s application-level protocol is similar to the one for the Akka TCP module. My reimplementation includes an actor

\(^3\)https://doc.akka.io/docs/akka/current/io-tcp.html
CHAPTER 9. EVALUATION

Table 9.1: Quantitative summary of the evaluation benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Prog. LOC</th>
<th>Spec. LOC</th>
<th>Actors</th>
<th>Evict. Actors</th>
<th>PSMs</th>
<th>Prog. States</th>
<th>Spec. States</th>
</tr>
</thead>
<tbody>
<tr>
<td>auth</td>
<td>170</td>
<td>27</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>chat</td>
<td>111</td>
<td>24</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>ensime</td>
<td>256</td>
<td>46</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>flink-tm</td>
<td>384</td>
<td>47</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>flink-jm-client</td>
<td>726</td>
<td>13</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>flink-jm-tm</td>
<td>726</td>
<td>16</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>pots</td>
<td>281</td>
<td>45</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>raft</td>
<td>1,055</td>
<td>37</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>spray-can</td>
<td>443</td>
<td>40</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>14</td>
<td>7</td>
</tr>
<tr>
<td>tcp-active</td>
<td>1,268</td>
<td>16</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>tcp-passive</td>
<td>1,271</td>
<td>14</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>tcp-session</td>
<td>669</td>
<td>95</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>weather</td>
<td>65</td>
<td>34</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

for the server itself, listener actors for listening on a given port, connection actors, and actors to handle individual requests. I model request-handling by forwarding each request to the application layer (in the environment) and forwarding the application layer’s response back to the requester. The specification describes the expected interactions with the application layer, because that is the complex part of the server’s communication and the easiest part to get wrong.

9.1.4 Quantitative Summary

Table 9.1 provides some quantitative information about the various benchmarks used to evaluate the model checker, including the lines of code for each program and specification, the number of distinct actor definitions, the number of those actor definitions that are marked as evictable (cf. section 8.2), the number of distinct PSMs (including the top-level PSM as well as all distinct fork effects and fork-addr/delayed-fork-addr patterns), and the total number of actor/PSM states across each program/specification, respectively. Each benchmark is referred to by a shorthand name used throughout the rest of this chapter. For the programs, the lines-of-code measurement refers to the program before being desugared into the core CSA language defined in chapter 2. No syntactic sugar was used in writing the specifications, so the lines-of-code measurement refers to lines of APS code as defined in chapter 3.

9.2 Experimental Setup

I ran the model checker on each program/specification pair (with no known conformance-violating bugs) with ten different combinations of optimizations:
9.3. EFFECTIVENESS EVALUATION RESULTS

• all optimizations enabled,
• all optimizations disabled,
• just one optimization enabled (for each of the four listed in chapter 8), and
• all optimizations except one enabled (for each of the four optimizations).

Additionally, for each conformance-violating bug I found while writing the programs and specifications, I model-checked a version of the relevant benchmark that included the bug. All optimizations were enabled for these runs.

The code for the model checker and all of the benchmarks can be found at https://github.com/schuster/aps-conformance-checker/. All experiments were run on machines in Northeastern University’s Discovery cluster running Intel Xeon E5-2690 v3 processors at 2.60 GHz, with 128 GB of RAM. The experiments used Racket version 6.11.4

Given that the current model checker is a research prototype, the results that follow are only rough measurements to get a general idea of the kind of performance one can expect.

9.3 Effectiveness Evaluation Results

9.3.1 Conforming Programs

Table 9.2 summarizes the results from model-checking the conformance-bug-free version of each benchmark with all optimizations enabled. These experiments model the experience users would expect to have when their programs conform to their specifications. As shown in the Result column, the model checker was able to prove conformance in every case except for the passive-connection TCP implementation. In that case, the check hit a self-imposed 24-hour time limit before terminating.

For the other columns in that table, the Visited Pairs column refers to the number of program-configuration/PSM pairs $\langle \hat{K}, s \rangle$ visited during state-space exploration, and the Visited Transitions column refers to the number of summary transitions $\hat{K} \xrightarrow{L} \hat{K}'$ explored from each such pair. The Duration column records how long the model checker ran for each benchmark.

Discussion

Most of the checks completed relatively quickly (within about 10 minutes), but some ran for much longer. The spray-can check took about three hours, and the check for the active-connection version of TCP took almost half a day. Likely this means that there is some aspect of those programs (such as the possible sets of in-flight messages, or the arguments to each actor’s state), that has a wide variety of

---

4Racket 6.12, which is the latest version as of this writing, appears to have a performance regression somewhere in its handling of hash tables. This significantly degraded the model checker’s performance, so the experiments use 6.11 instead.
CHAPTER 9. EVALUATION

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Visited Pairs</th>
<th>Visited Transitions</th>
<th>Duration (hh:mm:ss)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>auth</td>
<td>39</td>
<td>713</td>
<td>0:00:03</td>
<td>success</td>
</tr>
<tr>
<td>chat</td>
<td>86</td>
<td>2,199</td>
<td>0:00:20</td>
<td>success</td>
</tr>
<tr>
<td>ensime</td>
<td>25</td>
<td>1,173</td>
<td>0:00:14</td>
<td>success</td>
</tr>
<tr>
<td>flink-jm-cli</td>
<td>30</td>
<td>1,044</td>
<td>0:00:45</td>
<td>success</td>
</tr>
<tr>
<td>flink-jm-tm</td>
<td>40</td>
<td>15,370</td>
<td>0:10:10</td>
<td>success</td>
</tr>
<tr>
<td>flink-tm</td>
<td>81</td>
<td>1,079</td>
<td>0:00:27</td>
<td>success</td>
</tr>
<tr>
<td>pots</td>
<td>156</td>
<td>2,372</td>
<td>0:00:16</td>
<td>success</td>
</tr>
<tr>
<td>raft</td>
<td>83</td>
<td>1,811</td>
<td>0:03:27</td>
<td>success</td>
</tr>
<tr>
<td>spray-can</td>
<td>2,935</td>
<td>288,209</td>
<td>2:49:56</td>
<td>success</td>
</tr>
<tr>
<td>tcp-active</td>
<td>188</td>
<td>56,784</td>
<td>11:37:41</td>
<td>success</td>
</tr>
<tr>
<td>tcp-passive</td>
<td>57</td>
<td>28,073</td>
<td>&gt; 24 hrs</td>
<td>timeout</td>
</tr>
<tr>
<td>tcp-session</td>
<td>116</td>
<td>3,787</td>
<td>0:00:39</td>
<td>success</td>
</tr>
<tr>
<td>weather</td>
<td>138</td>
<td>2,264</td>
<td>0:00:10</td>
<td>success</td>
</tr>
</tbody>
</table>

Table 9.2: Summarized results from bug-free, all-optimizations-enabled experiments

reachable configurations, even in the abstract interpretation. So while the model checker can often terminate in a reasonable amount of time, the unpredictability of its running time is one of its biggest usability issues.

Surprisingly, even checks for some of the more complicated programs such as Raft, the TCP session controller, and the Flink implementation terminated in the 10-minute range, so program complexity alone appears to be a poor predictor of state-space explosions in the model checker. The number of program-configuration/PSM pairs visited is a similarly poor predictor, seeing as the POTS and active TCP experiments both visited a similar number of pairs, yet the TCP check took several orders of magnitude longer than the POTS check. The number of visited transitions appears to have a stronger correlation with the running time, suggesting that the model checker’s running time is more dependent on the number of transitions between pairs than on the number of visited pairs themselves.

9.3.2 Conformance Bugs Found

Table 9.3 summarizes the conformance-violating bugs I found while developing the evaluated programs and specifications, as well as the time taken to model-check each buggy version. As expected, the model checker failed to prove conformance for each one. All of them are genuine bugs, meaning that either I made a mistake while writing the program or specification, or in the case of Raft, that the code in the original akka-raft project differed from my understanding of the protocol. Thus, it seems that the kinds of bugs that violate an APS specification are indeed a common occurrence when writing actor programs.

The following describes the bugs discovered in each of the benchmarks:
### 9.3. Effectiveness Evaluation Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Bug #</th>
<th>Bug Type</th>
<th>Bug Location</th>
<th>Found by Unit Tests</th>
<th>Duration (hh:mm:ss)</th>
</tr>
</thead>
<tbody>
<tr>
<td>flink-tm</td>
<td>1</td>
<td>liveness</td>
<td>program</td>
<td>yes</td>
<td>0:00:28</td>
</tr>
<tr>
<td>flink-jm-tm</td>
<td>1</td>
<td>safety</td>
<td>program</td>
<td>yes</td>
<td>0:00:02</td>
</tr>
<tr>
<td>raft</td>
<td>1</td>
<td>safety</td>
<td>specification</td>
<td>N/A</td>
<td>0:02:04</td>
</tr>
<tr>
<td>raft</td>
<td>2</td>
<td>liveness</td>
<td>program</td>
<td>yes</td>
<td>0:03:48</td>
</tr>
<tr>
<td>raft</td>
<td>3</td>
<td>liveness</td>
<td>program</td>
<td>yes</td>
<td>0:03:50</td>
</tr>
<tr>
<td>raft</td>
<td>4</td>
<td>liveness</td>
<td>program</td>
<td>yes</td>
<td>0:03:33</td>
</tr>
<tr>
<td>raft</td>
<td>5</td>
<td>safety</td>
<td>program</td>
<td>yes</td>
<td>0:02:31</td>
</tr>
<tr>
<td>spray-can</td>
<td>1</td>
<td>safety</td>
<td>program</td>
<td>yes</td>
<td>2:25:20</td>
</tr>
<tr>
<td>spray-can</td>
<td>2</td>
<td>liveness</td>
<td>program</td>
<td>yes</td>
<td>0:18:18</td>
</tr>
<tr>
<td>tcp-passive</td>
<td>1</td>
<td>safety</td>
<td>program</td>
<td>yes</td>
<td>4:25:36</td>
</tr>
<tr>
<td>tcp-active</td>
<td>1</td>
<td>safety</td>
<td>program</td>
<td>no</td>
<td>10:29:49</td>
</tr>
<tr>
<td>tcp-active</td>
<td>2</td>
<td>safety</td>
<td>specification</td>
<td>N/A</td>
<td>12:14:46</td>
</tr>
<tr>
<td>tcp-active</td>
<td>3</td>
<td>liveness</td>
<td>specification</td>
<td>N/A</td>
<td>11:52:14</td>
</tr>
<tr>
<td>tcp-session</td>
<td>1</td>
<td>safety</td>
<td>program</td>
<td>yes</td>
<td>0:00:31</td>
</tr>
<tr>
<td>tcp-session</td>
<td>2</td>
<td>safety</td>
<td>program</td>
<td>yes</td>
<td>0:00:31</td>
</tr>
<tr>
<td>tcp-session</td>
<td>3</td>
<td>safety</td>
<td>program</td>
<td>yes</td>
<td>0:00:32</td>
</tr>
<tr>
<td>tcp-session</td>
<td>4</td>
<td>safety</td>
<td>specification</td>
<td>N/A</td>
<td>0:00:33</td>
</tr>
</tbody>
</table>

Table 9.3: Summary of conformance bugs detected by the model checker

- **flink-tm #1**: The task manager actor does not send back an acknowledgment when it receives a `SubmitTask` message and has an available actor to run the task.

- **flink-jm-tm #1**: The job manager actor never sends back an acknowledgment to a `RegisterTaskManager` message.

- **raft #1**: The specification does not allow the actor to send an `AppendEntries` message in response to an `AppendRejected` message while in the `Running` state. Such a response is necessary when the actor sending the `AppendRejected` message thinks it is the cluster leader, but has not yet learned this fact.

- **raft #2**: When the Raft actor receives an `AppendEntries` message from an actor that thinks it is the leader but no longer is, that receiving actor sends back a response only if the `AppendEntries` message contains at least one new entry. It should respond to all received `AppendEntries` messages, regardless of the contained entries.

- **raft #3**: Identical to the previous bug, except this behavior applies to messages from actors that do not believe themselves to be the cluster leader.

- **raft #4**: If a Raft actor receives an `AppendEntries` message while it thinks a leader election is occurring, and the sender of that message is no longer
the cluster leader, then the receiving actor ignores the message rather than sending back the appropriate AppendRejected response.

- **raft #5**: If a Raft actor in the Leader state receives an AppendEntries message from a previous leader, that actor sends back its own AppendEntries message rather than an AppendRejected response.

- **spray-can #1**: When the HTTP-connection actor is in the Running state and receives a request to send status messages to some new address instead of a previous one, the actor continues to send those messages to the previous address.

- **spray-can #2**: The actor that should time out if binding to a socket fails is marked as evictable. Evicting that actor adds too much imprecision to the abstract program, making it impossible to prove that all bind requests receive a response. (The mistake was not in the program itself, but in the decision to mark the actor as evictable during model checking.)

- **tcp-passive #1**: After sending a SYN/ACK message, if the TCP session actor times out while waiting for the corresponding ACK, it does not send the application layer a message to notify it of the failure.

- **tcp-active #1**: The TCP session actor sends a ConfirmedClosed message back to the client whenever it reaches the Closing state; however, it should send that message only when it reaches the Closing state by a certain sequence of state transitions.

- **tcp-active #2**: The specification does not allow the session actor to send a Closed message back to the application-layer actor while waiting to register with the TCP session manager. This can happen if the actor times out while waiting on its registration to go through.

- **tcp-active #3**: The specification does not allow the session actor to transition from a particular Closing state to the Closed state without notifying the application layer. This should be allowed in certain situations.

- **tcp-session #1**: While in the Established state, if the TCP session actor receives a SYN packet, then it processes the packet like a normal data packet rather than aborting the connection as required by the TCP protocol.

- **tcp-session #2**: If the TCP session actor receives a RST packet while in the Closed state, then it tries to abort the connection even though RST packets should not trigger an abort at that stage.

- **tcp-session #3**: A session actor in the Closed state aborts a connection *only* in response to RST packets, rather than in response to all packets *other than* RST packets. This bug is the result of accidentally flipping a boolean test while attempting to fix the previous bug.
9.3. EFFECTIVENESS EVALUATION RESULTS

- **tcp-session #4**: The specification does not allow the session actor to close the connection immediately if it receives a packet with both the ACK and RST flags set, which should be allowed by the TCP protocol.

Discussion

**Advantage of Exhaustive Search**  Most of the bugs originating in a program (rather than in a specification) were found before model checking that program, e.g., during unit testing. Bug number 1 for active-connection TCP program, however, was found only by model-checking—the unit test suite did not catch the bug, because the particular execution path that manifests the bug was not tested. This underscores the value of an exhaustive-search technique such as model-checking compared with hand-written tests: a test can check the behavior only for a single path through the code, but an exhaustive search checks all possible paths. Or to quote Dijkstra [21], “Testing shows the presence, not the absence of bugs.”

**Necessity of Both Safety and Liveness**  Table 9.3 records every bug as violating either the safety aspect of APS (no output may be sent without an obligation), or the liveness aspect (every obligation must eventually be fulfilled by a sent message). The violated aspect is determined by whether the model checker failed during the state-space-exploration or obligation-checking phase of the algorithm. Table 9.3 lists a handful of instances of each, suggesting that both kinds of errors are common in actor programs. Much of the complexity of APS’ semantics stems from the desire to catch both kinds of errors—in fact, I have often contemplated focusing on just one kind to simplify the semantics. These results confirm that this complexity is needed.

**Buggy Specifications**  Table 9.3 also categorizes each bug according to whether the root cause was found in the program or the specification, as determined by a manual inspection of the error. A handful of the discovered bugs came from the specification. One might doubt whether discovering such bugs is valuable (“My program was right all along!”), but they tend to indicate a gap in the specification-writer’s understanding of the protocol, which is worth correcting. In cases where the programmer and specification-writer are two different people, this discrepancy is even more important to clear up.

**Model Checking as a Debugging Aid**  Every conformance violation was detected in less time than or about the same amount of time that it took to verify the violation-free version of that benchmark. In one extreme case, the model checker was able to find a bug for the passive-connection version of TCP in about four and a half hours, even though the model checker was unable to finish checking the conforming version of that program before exceeding the 24-hour time limit. Model checking buggy programs sometimes takes less time because the state-space-exploration algorithm stops when it finds a transition of the program that has no corresponding PSM transition. This suggests that even when a program is too complex for full verification, model checking might still be useful as a


### 9.4 Performance Evaluation Results

#### 9.4.1 Step-Wise Running Time Breakdown

Table 9.4 shows the time taken in each step of ModelCheck for each benchmark, distinguishing between the two separate calls to Prune. Generally, the state-space exploration in Explore takes the vast majority of the time. This is unsurprising, since that step is the only one that has to traverse the AST of each visited program configuration, both during abstract evaluation and when applying the transformation $T$. All other steps treat the program configurations as opaque tokens.

#### 9.4.2 Impact of Optimizations

Table 9.5 shows the amount of time the model checker took in each benchmark for each evaluated combination of enabled optimizations. In the column headers, $\emptyset$ stands for no enabled optimizations, “A” stands for just acceleration enabled, “A/E/M” stands for acceleration, eviction, and memoization enabled, etc. An entry of “$> 24$ hrs” indicates that the experiment reached the 24-hour time limit before finishing (those cells are marked in gray). No optimization caused the model checker to fail on any of these combinations: it always either succeeded or timed out.

The charts in figure 9.3 summarize this information. Each chart shows how much an optimization sped up the model checker for each benchmark, relative to

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Explore (hh:mm:ss)</th>
<th>Prune 1 (hh:mm:ss)</th>
<th>FindFulfillingPairs (hh:mm:ss)</th>
<th>Prune 2 (hh:mm:ss)</th>
<th>Total (hh:mm:ss)</th>
</tr>
</thead>
<tbody>
<tr>
<td>auth</td>
<td>0:00:03</td>
<td>0:00:00</td>
<td>0:00:00</td>
<td>0:00:00</td>
<td>0:00:03</td>
</tr>
<tr>
<td>chat</td>
<td>0:00:12</td>
<td>0:00:00</td>
<td>0:00:08</td>
<td>0:00:00</td>
<td>0:00:20</td>
</tr>
<tr>
<td>ensime</td>
<td>0:00:13</td>
<td>0:00:00</td>
<td>0:00:01</td>
<td>0:00:00</td>
<td>0:00:14</td>
</tr>
<tr>
<td>flink-jm-cli</td>
<td>0:00:42</td>
<td>0:00:00</td>
<td>0:00:02</td>
<td>0:00:01</td>
<td>0:00:45</td>
</tr>
<tr>
<td>flink-jm-tm</td>
<td>0:08:20</td>
<td>0:00:01</td>
<td>0:01:47</td>
<td>0:00:02</td>
<td>0:10:10</td>
</tr>
<tr>
<td>flink-tm</td>
<td>0:00:23</td>
<td>0:00:00</td>
<td>0:00:03</td>
<td>0:00:01</td>
<td>0:00:27</td>
</tr>
<tr>
<td>pots</td>
<td>0:00:10</td>
<td>0:00:00</td>
<td>0:00:06</td>
<td>0:00:00</td>
<td>0:00:16</td>
</tr>
<tr>
<td>raft</td>
<td>0:03:05</td>
<td>0:00:00</td>
<td>0:00:21</td>
<td>0:00:01</td>
<td>0:03:27</td>
</tr>
<tr>
<td>spray-can</td>
<td>2:01:47</td>
<td>0:00:23</td>
<td>0:46:41</td>
<td>0:01:05</td>
<td>2:49:56</td>
</tr>
<tr>
<td>tcp-active</td>
<td>10:45:23</td>
<td>0:00:34</td>
<td>0:49:25</td>
<td>0:02:19</td>
<td>11:37:41</td>
</tr>
<tr>
<td>tcp-passive</td>
<td>$&gt; 24$ hrs</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>$&gt; 24$ hrs</td>
</tr>
<tr>
<td>tcp-session</td>
<td>0:00:32</td>
<td>0:00:01</td>
<td>0:00:06</td>
<td>0:00:00</td>
<td>0:00:39</td>
</tr>
<tr>
<td>weather</td>
<td>0:00:09</td>
<td>0:00:00</td>
<td>0:00:01</td>
<td>0:00:00</td>
<td>0:00:10</td>
</tr>
</tbody>
</table>

Table 9.4: Timings from bug-free, all-optimizations-enabled experiments

debbuging aid. Future work could focus on this use by having the model checker present the user with potential reasons for non-conformance.

9.4 Performance Evaluation Results

9.4.1 Step-Wise Running Time Breakdown

Table 9.4 shows the time taken in each step of ModelCheck for each benchmark, distinguishing between the two separate calls to Prune. Generally, the state-space exploration in Explore takes the vast majority of the time. This is unsurprising, since that step is the only one that has to traverse the AST of each visited program configuration, both during abstract evaluation and when applying the transformation $T$. All other steps treat the program configurations as opaque tokens.

9.4.2 Impact of Optimizations

Table 9.5 shows the amount of time the model checker took in each benchmark for each evaluated combination of enabled optimizations. In the column headers, $\emptyset$ stands for no enabled optimizations, “A” stands for just acceleration enabled, “A/E/M” stands for acceleration, eviction, and memoization enabled, etc. An entry of “$> 24$ hrs” indicates that the experiment reached the 24-hour time limit before finishing (those cells are marked in gray). No optimization caused the model checker to fail on any of these combinations: it always either succeeded or timed out.

The charts in figure 9.3 summarize this information. Each chart shows how much an optimization sped up the model checker for each benchmark, relative to
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>auth</td>
<td>10:42:17</td>
<td>0:00:05</td>
<td>10:37:07</td>
<td>9:44:55</td>
<td>10:21:54</td>
<td>9:41:42</td>
<td>0:00:04</td>
<td>0:00:04</td>
<td>0:00:03</td>
<td>0:00:03</td>
</tr>
<tr>
<td>chat</td>
<td>0:13:46</td>
<td>0:00:20</td>
<td>0:13:42</td>
<td>0:13:47</td>
<td>0:13:46</td>
<td>0:13:38</td>
<td>0:00:19</td>
<td>0:00:19</td>
<td>0:00:19</td>
<td>0:00:20</td>
</tr>
<tr>
<td>ensime</td>
<td>&gt; 24 hrs</td>
<td>0:00:14</td>
<td>&gt; 24 hrs</td>
<td>&gt; 24 hrs</td>
<td>&gt; 24 hrs</td>
<td>&gt; 24 hrs</td>
<td>0:00:14</td>
<td>0:00:14</td>
<td>0:00:13</td>
<td>0:00:14</td>
</tr>
<tr>
<td>flink-jm-cli</td>
<td>&gt; 24 hrs</td>
<td>0:01:09</td>
<td>&gt; 24 hrs</td>
<td>&gt; 24 hrs</td>
<td>&gt; 24 hrs</td>
<td>&gt; 24 hrs</td>
<td>0:00:44</td>
<td>0:00:49</td>
<td>0:01:05</td>
<td>0:04:45</td>
</tr>
<tr>
<td>flink-jm-tm</td>
<td>&gt; 24 hrs</td>
<td>0:54:23</td>
<td>&gt; 24 hrs</td>
<td>&gt; 24 hrs</td>
<td>&gt; 24 hrs</td>
<td>&gt; 24 hrs</td>
<td>0:10:09</td>
<td>0:10:29</td>
<td>0:53:32</td>
<td>0:10:10</td>
</tr>
<tr>
<td>flink-tm</td>
<td>&gt; 24 hrs</td>
<td>0:00:28</td>
<td>&gt; 24 hrs</td>
<td>&gt; 24 hrs</td>
<td>&gt; 24 hrs</td>
<td>&gt; 24 hrs</td>
<td>0:00:27</td>
<td>0:00:29</td>
<td>0:00:26</td>
<td>0:00:27</td>
</tr>
<tr>
<td>pots</td>
<td>0:00:14</td>
<td>0:00:16</td>
<td>0:00:13</td>
<td>0:00:13</td>
<td>0:00:14</td>
<td>0:00:13</td>
<td>0:00:14</td>
<td>0:00:16</td>
<td>0:00:16</td>
<td>0:00:16</td>
</tr>
<tr>
<td>raft</td>
<td>0:08:57</td>
<td>0:28:21</td>
<td>0:08:58</td>
<td>0:08:17</td>
<td>0:06:29</td>
<td>0:06:08</td>
<td>0:03:24</td>
<td>0:03:42</td>
<td>0:25:03</td>
<td>0:03:27</td>
</tr>
<tr>
<td>tcp-passive</td>
<td>&gt; 24 hrs</td>
<td>&gt; 24 hrs</td>
<td>&gt; 24 hrs</td>
<td>&gt; 24 hrs</td>
<td>&gt; 24 hrs</td>
<td>&gt; 24 hrs</td>
<td>&gt; 24 hrs</td>
<td>&gt; 24 hrs</td>
<td>&gt; 24 hrs</td>
<td></td>
</tr>
<tr>
<td>tcp-session</td>
<td>8:52:44</td>
<td>0:33:38</td>
<td>0:00:32</td>
<td>8:53:30</td>
<td>8:50:03</td>
<td>0:00:35</td>
<td>0:32:44</td>
<td>0:00:39</td>
<td>0:00:39</td>
<td></td>
</tr>
<tr>
<td>weather</td>
<td>0:00:42</td>
<td>0:00:11</td>
<td>0:00:40</td>
<td>0:00:37</td>
<td>0:00:40</td>
<td>0:00:37</td>
<td>0:00:10</td>
<td>0:00:10</td>
<td>0:00:10</td>
<td></td>
</tr>
</tbody>
</table>

Table 9.5: Time taken (hh:mm:ss) to model check bug-free benchmarks with various optimizations enabled (A = acceleration, E = eviction, D = dead-marker detection, M = memoization)
CHAPTER 9. EVALUATION

Figure 9.3: Speedups for each optimization, relative to enabling all but that optimization
running the model checker on that benchmark with all optimizations except that one. For example, the first bar in the “Speedups for Acceleration” chart shows that the check for the authentication-server example is about 10,000 times faster when all optimizations are enabled, compared to when acceleration is disabled but all other optimizations are turned on. A speedup of $\infty$ indicates that disabling the given optimization caused the check to time out, but adding it back in allowed the check to complete within the allotted time. No combination of optimizations was able to finish checking the passive TCP implementation, so the speedup charts omit that benchmark.

Discussion

**Acceleration** is certainly the most impactful optimization. Figure 9.3 shows that several checks are unable to complete at all without acceleration, and it gives a large speedup to a few others. The first two columns in table 9.5 also show that just acceleration alone can decrease the time taken to model check several benchmarks from over 24 hours to less than an hour, and less than a minute for three benchmarks.

The only benchmark for which acceleration adds a significant slowdown is Raft, when all other optimizations are disabled (compare the “∅” and “A” columns). This is possibly because memoization is disabled in that case, so acceleration may be forcing the model checker to evaluate the possible transitions from each program configuration much more often than when acceleration is disabled.

**Eviction** also has a significant impact in the benchmarks where it is used (i.e., spray-can and all of the TCP benchmarks), other than perhaps the passive TCP benchmark which did not terminate under any combination of optimizations. With just eviction alone, the model checker can check conformance for the TCP session controller in under a minute rather than almost 9 hours. Figure 9.3 also reveals that eviction is necessary to finish checking the active TCP implementation and spray-can within the 24 hour limit.

**Dead-marker detection** seems to have a moderate impact on most of these experiments. It never causes more than about a 30% speed-up, but it lends a modest improvement in running time to several benchmarks.

**Memoization** has a more mixed impact. There are a few benchmarks for which disabling memoization causes a significant slowdown, but several others in which it has no significant impact at all.

### 9.5 Lessons Learned

This section summarizes a few of the lessons learned from building and evaluating the model checker.

**Utility of Simple Specifications** Compared to a full protocol specification one might find in an academic paper or RFC, an APS specification is extremely lim-
Chapter 9. Evaluation

Section 9.1 describes how APS is unable to distinguish even between most of the states of a relatively simple protocol like Raft, much less the full state machine for TCP. Still, checking the evaluated programs against even these simple specifications uncovered several bugs.

Importance of Acceleration  
The results from section 9.4.2 show that a technique such as acceleration is critical to finishing checks in a reasonable amount of time. Without acceleration, over half of the checks were unable to complete before timing out, and adding just acceleration alone without any other optimizations enabled four more checks to terminate.

Mixed Benefits from FSM-Based Actors  
Some of the restricted structure of actors in CSA appears to have made building the model checker easier. In particular, the enforced termination and lack of recursion inside event handlers make it possible to reason about each handler as if it runs from start to finish without being preempted by another actor (see the description of single-handler conformance in appendix D).

I was unable to find strong evidence that the separate named states for actors give similar benefits, however. In particular, it is not clear whether an explicitly named state results in more reasoning power than simply representing an actor’s “state” with an additional variant parameter.

Untamed State-Space Explosions  
For certain programs, the model checker still runs into state-space explosions that slow it down, even with all optimizations enabled. These problems are why the TCP implementation had to be split into active-only and passive-only implementations, and why the model checker was unable to prove conformance even for the passive-only implementation. Part of the problem might be due to the many ways to evaluate a given handler expression, especially those involving for/fold loops, since the evaluation for those has not been heavily optimized. As mentioned above, predicting when and why these explosions occur is also a problem to be addressed.

Room for Better Abstractions  
Although many of the checks completed quickly, there is still room for better abstraction even for the simpler benchmarks. For example, the model checker visited 138 different program-configuration/PSM pairs while checking the stream-processing weather-station example, (see table 9.2), while the manual proof in chapter 4 needed to reason about only 4 different cases: one for each state in the specification. Thus, there is still a wide gap between the capabilities of a human prover and the model checker, which future work could potentially narrow.

Limits of Expressiveness for APS  
As mentioned in section 9.1.2, my initial version of the Flink system does not conform to one of the specifications, because of issues around treating two copies of the same address as different entities. The
general problem appears to be that history markers make too many distinctions between each copy of an address.

9.6 Future Performance Work

This evaluation suggests a few avenues for improvements to the model checker's performance. For one, table 9.4 shows that the vast majority of the model checker's time is spent exploring the state space to build a simulation relation. That code is currently single-threaded, but could easily be parallelized to take advantage of multicore machines, with multiple threads processing program-configuration/PSM pairs from the worklist concurrently.

Another possibility would be to change the internal representation of program configurations. The model checker currently uses a substitution-based evaluation rather than an environment-based one, which greatly increases the size of each program configuration. That in turn slows down transformations such as Assimilate and Canonicalize that must traverse the entire configuration's AST. Switching to an environment-based evaluation method would likely yield some easy performance wins.

Finally, interleaving the simulating-finding and pruning phases of the model checker may improve the running time when conformance violations are found. Currently, if the state-space-exploration phase finds a non-conforming program-configuration/PSM pair that would eventually cause the whole check to fail, the model checker will continue exploring the state space even though no further work will change the end result. If the model checker instead performed the backward-chaining aspect of the pruning algorithm as soon as each non-conforming pair was found, it might result in returning an answer to the user sooner.
Chapter 10

Conclusion

10.1 Summing Up

To recap, my thesis is that

finite-state, address-passing specifications can be used to automatically verify non-trivial protocols in actor programs.

The specification language APS represents one possible implementation of finite-state, address-passing specifications. The language organizes an actor-protocol specification as a collection of finite-state machines, each describing a program's expected communication from a different point of view. The patterns in the language can bind addresses to names,\(^1\) thus giving these machines the ability to describe how addresses sent in messages should be used (i.e., the expected address-passing behavior).

I showed in the previous chapter that APS can describe non-trivial protocols, meaning protocols that are easily mis-implemented. Thirteen conformance-violating bugs were found in programs while developing the evaluated programs, including some from the original akka-raft project from which the Raft example was ported. The bugs represent issues such as forgetting to send a reply to a message, or handling a received message incorrectly in a given state.

I also showed that APS is restricted enough to automatically verify these protocols in actor programs—specifically, CSA programs. The refinements to conformance defined in appendices B–G, as well as abstract conformance, summary conformance, and transformation conformance all take advantage of the restricted nature of APS to define techniques that make it easier to prove conformance. These led to the ModelCheck algorithm, which can prove conformance for an infinite-state program by exploring only a finite number of (abstract) states. As discussed in the previous chapter, the model checker implementing this algorithm was able to verify conformance for all but one benchmark in less than 24

\(^{1}\)Technically, the patterns bind history markers that identify particular copies of those addresses, as explained in chapter 3.
hours, and in about 10 minutes or less in most cases. Future work may be able to improve performance in problematic cases.

In total, this dissertation supports my thesis with the following contributions:

1. APS, a specification language for describing actor protocols with address-passing, finite-state machines, and an associated notion of conformance,

2. a series of refinements to conformance to make proving conformance easier, including
   - externals-only conformance (appendix B),
   - external-representative conformance (appendix C),
   - single-handler conformance (appendix D),
   - deterministic-handler conformance (appendix E),
   - event-step conformance (appendix F),
   - PSM conformance (appendix G),
   - abstract conformance (chapter 5),
   - summary conformance (chapter 5), and
   - transformation conformance (chapter 6),

3. an abstract interpretation for CSA,

4. a state-space-reduction technique \(\text{i.e., transformation conformance}\),

5. a sound (but not complete) algorithm, \textit{ModelCheck}, for automatically verifying whether a program conforms to its specification,

6. a set of optimizations for \textit{ModelCheck}, and

7. an empirical validation of \textit{ModelCheck}'s precision.

\textbf{10.2 Future Work}

To conclude this dissertation, I present some ideas for future work inspired by developing and writing about this work.

\textbf{Compositional Reasoning} Arguably the biggest shortcoming of APS is that there is no obvious way to reason about specifications compositionally. For example, it would be useful to replace each of a program’s components with an APS specification that each one satisfies, then reason about the interactions between those specifications to show that the overall program satisfies some top-level specification. This kind of compositionality would make specifications much more useful, and likely reduce the state-space-explosion problem when model checking. Unfortunately, it is not clear how to make this happen.
The lack of compositional reasoning is partly due to allowing the environment to send any possible message at any time, rather than mandating that the environment must follow some particular protocol. Designing a replacement to APS around assume-guarantee-style reasoning [1, 8], which reasons about a “contract” between a program and its environment, could potentially resolve this issue.

**Non-Linear Use of Monitored Addresses** An APS specification places a kind of linearity requirement on the usage of each monitored address, in that a conforming program will send a message to a given monitored address if and only if the specification has a corresponding obligation for that message. This allows the model checker to detect both safety and liveness violations (as shown in table 9.3), but it also complicates the semantics. Much of the machinery from chapter 3 exists to ensure that PSMs use addresses linearly, and without the linearity requirement, the three different kinds of “fork” mechanisms in APS could likely be reduced to one.

It would be worth exploring whether an APS-like language that drops either the “must fulfill every obligation” or “cannot send without an obligation” condition is strong enough to be useful as a verification or debugging aid. The former condition is probably the more important one, given that the most common specification for a message-passing system is usually “every request receives a response”, but exploring a version of APS that drops either condition could be fruitful.

Note that a tool that checks only for liveness violations might also be able to catch bugs currently categorized as safety violations. For example, a buggy program might respond to a request with a NotOk message instead of an expected Ok message. The model checker currently detects this as a safety violation (the wrong message was sent), but if the program does not also send a Ok message, then the bug also violates a liveness property (the expected message was never sent).

**Mailbox Types** The mailbox types of de'Liguoro and Padovani [38] are a promising approach for detecting communication errors with a substructural type system. As mentioned in chapter 3, every channel type is parameterized over a regular expression of types, describing the messages expected to be sent to or received from that channel. These types are not as expressive as APS protocol specifications (e.g., they cannot specify that the response to a GetMean message in the running example should be sent to some mdest address received in a previous message), but they would likely be able to catch many of the kinds of errors actor programmers tend to make.

It would be worthwhile to find out whether this approach can be adapted from their calculus to the actor model. The major challenges would be finding a way to typecheck the (implicit) points where an actor receives a message, and determining whether a substructural type system is practical enough for writing actor programs.
CHAPTER 10. CONCLUSION

Structured Programming for Actors  In some ways, the basic actor model is similar to the goto-based, “unstructured” programming of the early days of software development. The only control forms at the inter-actor level are send and spawn, and neither of those tell the reader anything about what the receiving/spawned actor will do from that point on. Tracing the control flow between various actors can be like trying to read spaghetti code, because the shape of the program does not correspond to the shape of the protocol.

A more radical change to address this issue would be to modify the actor model even further to add more structure to the inter-actor control flow. The Akka framework, for example, comes with a notion of futures built on top of the message-passing infrastructure to express asynchronous request/response patterns. The Syndicate language [50] exemplifies a more extreme change, in which actors are structured into nested groups to model the nested nature of conversations.

Removal of History Markers  As explained in chapter 3, history markers distinguish different copies of a given address so that copies being used for two different conversations are not confused with each other. Markers have no correspondence with objects the programmer works with, however, and their usage complicates the semantics of both CSA and APS. The following are some initial thoughts on how to define the semantics of APS without history markers, organized in terms of the scenarios from chapter 3 that motivated them in the first place.

Scenario 1 from section 3.4.1 deals with what happens when the environment sends one of the program’s own addresses as a callback address (in which case the environment is unable to observe the response). To handle this scenario without markers, there could be a transition rule that allows a PSM to fulfill any obligation to an internal address during a → step, instead of exposing some of the message-sends to internal addresses as observed events. Effectively, this would correspond to the PSM “guessing” when an obligation is fulfilled, even though the environment would not be able to observe the actual message-send that fulfills it. The program would still have to use observable output events to fulfill the obligations for external addresses, however, so this approach would still catch programs that use addresses from the environment incorrectly. A notion of conformance similar to externals-only conformance from appendix B would allow conformance proofs to reason only about the external-address cases and avoid the “guessing” steps altogether.

Scenario 2 from section 3.4.3 deals with what happens when the environment sends the same address to the program in two separate messages. Instead of distinguishing these two copies with history markers, a PSM could potentially record the set of unmonitored external addresses that the program has access to, so that messages could be sent to such addresses without the need to fulfill an obligation. It would then likely be sufficient to reason only about the executions in which every externally provided address is fresh, thus removing the need to keep track of the unmonitored-address set in conformance proofs.
Finally, APS is designed to model an observer interacting with a program in an environment where there are other, unknown entities also interacting with the program concurrently. Rather than modeling those other entities as part of the environment, it may be simpler to compose the specified program with a non-deterministic process that models them, then run that composed system in an environment containing only the observer. Then every event reaching the environment would be observed, rather than just those sent to or received at addresses the observer knows about.

More Hints The model checker relies on many different heuristics to determine how a program might correspond to its specification, such as those used to decide which actors to assimilate, or which PSM transitions should be attempted in order to match a given event step. Currently, the only “hint” the programmer can give to the model checker about this correspondence is the set of actors marked as evictable for the Evict transformation. It may be helpful to introduce more hints that guide the model checker’s state-space exploration, although such hints come with the trade-off of requiring more human interaction.

Run-Time Verification This dissertation uses APS specifications only for static verification with a model checker or by manual proofs; indeed, this is the purpose for which APS was designed. Given the success enjoyed by other systematic testing tools such as QuickCheck and PropEr, however, it may also be useful to check conformance-like properties at run time. For instance, an APS specification could be applied to a program like a contract, and an error could be thrown whenever the program sends a message not allowed by the specification.

Such a system would be able to verify only safety properties, e.g., the program never sends an unexpected message. However, liveness properties could be approximated by limiting either the amount of time that can elapse or the number of events the program can handle before each expected message must be sent. Another challenge would be to handle the inherent non-determinism of APS specifications at run-time.
Bibliography


formal model for dynamic systems. In Proceedings of the International

Flat acceleration in symbolic model checking. In International Sym-
posium on Automated Technology for Verification and Analysis, pages 474–

full-duplex transmission over half-duplex links. Communications of the

and verification of dynamic communication systems. In International
Conference on Application of Concurrency to System Design (ACSD), 2006.

[17] Philip A. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen
Thelin. Orleans: Distributed virtual actors for programmability and scal-

[18] Gérard Berry and Georges Gonthier. The ESTEREL synchronous program-
ing language: Design, semantics, implementation. Science of Computer

of the ACM (JACM), 30(2), 1983.

type equality and subtyping. In International Conference on Typed
Lambda-Calculi and Applications (TLCA), 1997.


[22] Luís Caires and Hugo Torres Vieira. Conversation types. In Proceedings of

[23] Luca Cardelli. Amber. In Combinators and Functional Programming Lan-

Proceedings of the International Conference on Concurrency Theory (CON-
CUR), pages 211–228, 2009.

[25] Maria Christakis and Konstantinos Sagonas. Static detection of race con-
ditions in Erlang. In Proceedings of the International Symposium on Prac-


Appendix A

Proof of the Maximal Instantiation Theorem

**Theorem** (Maximal Instantiation). For all CSA programs $P$ and APS specifications $\Sigma$, $P \vdash \Sigma$ if and only if there exists some maximal instantiation $\tilde{K}$ and $S$ of $P$ and $\Sigma$ such that $\tilde{K} \models S$.

**Proof.** The left-to-right direction is trivial. For the right-to-left direction, let $\langle \tilde{K}_{\text{init}}, S_{\text{init}} \rangle$ be any pair instantiable from $P$ and $\Sigma$, and let $\langle \tilde{K}'_{\text{init}}, S'_{\text{init}} \rangle$ be a maximal instantiation of $P$ and $\Sigma$. Without loss of generality, we can assume that the history markers in each pair are the same, because history markers are uninterpreted symbols and therefore renaming them has no effect on the possible transition sequences from each configuration, modulo renaming. Thus, $S_{\text{init}} = S'_{\text{init}}$.

The proof requires formalizing a relationship between the program configurations as they evolve. Let $\sigma$ stand for a total function from marked addresses to marked addresses, and define the application of $\sigma$ to actor-behavior maps $\tilde{\beta}$ and message multisets $\tilde{\mu}$ as the obvious element-wise, component-wise application. Then let $\tilde{K}' = \langle \tilde{\beta}', \tilde{\mu}', H' \rangle^{\tilde{\rho}'}$ be a maximal representative of $\tilde{K} = \langle \tilde{\beta} \mid \tilde{\mu} \mid H \rangle^{\tilde{\rho}}$ if and only if there exists some $\sigma$ such that

- for all $a, a', H,$ and $H'$, if $\sigma(a@H) \neq a@H$, then $a$ and $a'$ are external, $H = H'$, and $\text{ActorType}(a') < \text{ActorType}(a)$.
- $\sigma(\tilde{\beta}) = \tilde{\beta}'$ and $\sigma(\tilde{\mu}) = \tilde{\mu}'$,
- $H = H'$, and
- for all $\langle a@H'', \tau \rangle \in \tilde{\rho}$, there exists $\tau'$ such that $\langle a@H'', \tau' \rangle \in \tilde{\rho}'$ and $\tau < : \tau'$.

The final bullet of the definition allows the receptionists on the “maximal” configuration to have supertypes of those on the other configuration, to handle cases such as the (Variant [A] [B]) example above.
APPENDIX A. PROOF OF THE MAXIMAL INSTANTIATION THEOREM

To show that $\bar{K}'_{init}$ is a maximal representative of $\bar{K}_{init}$, define $\sigma$ as the identity function except that it maps each of the marked addresses provided as externals in the creation of $\bar{K}_{init}$ to each of the corresponding marked addresses provided as externals in the creation of $\bar{K}'_{init}$.

Next, we will show that any maximal representation can simulate the transitions of any configuration it represents. Let there be $\bar{K}$, $\bar{K}'$, $\bar{K}''$, and $\bar{l}$ such that $\bar{K}'$ is a maximal representative of $\bar{K}$ via some function $\sigma$ and $\bar{K} \xrightarrow{\bar{l}} \bar{K}''$. Define the application of $\sigma$ to a specification transition label $\lambda$ be the obvious component-wise application to its components. By case analysis on the rule enabling the transition $\bar{K} \xrightarrow{\bar{l}} \bar{K}''$, we can see that there exist some $\bar{l}'$ and $\bar{K}'''$ such that $\bar{K}' \xrightarrow{\bar{l}'} \bar{K}'''$, $\sigma(\bar{l}) = \bar{l}'$, and $\bar{K}'''$ is a maximal representative of $\bar{K}''$.

- If the transition is enabled by the M-ReceiveExternal rule, then let $\langle a@H, \tau \rangle$ be the receptionist on $\bar{K}$ that receives the message $\bar{v}$. By the definition of a maximal representative, there exists $\bar{\tau}'$ such that $\langle a@H, \bar{\tau}' \rangle$ is a receptionist on $\bar{K}'$ and $\bar{\tau} < \bar{\tau}'$. By the definition of the M-ReceiveExternal rule, $\phi, \phi \vdash \bar{v} : \bar{\tau}$, so by subsumption, $\phi, \phi \vdash \bar{v} : \bar{\tau}'$. Therefore, $\bar{K}'$ can receive $\bar{v}$ on $a@H$ with a transition $\bar{K}' \xrightarrow{\bar{\tau}'} \bar{K}'''$. Because all external addresses in $\bar{v}$ must have no markers by the definition of this rule, $\sigma(\bar{v}) = \bar{v}$, and therefore $\sigma(\bar{l}) = \bar{l}'$. The resulting actor-behavior maps $\bar{\beta}$ and $\bar{\beta}'$ differ only by $\sigma$, the message multisets are unchanged, the used-marker sets on each component are updated the same way, and this rule does not change the receptionists on either configuration. Therefore, $\bar{K}'''$ is a maximal representative of $\bar{K}''$.

- If the transition is enabled by the M-SendExternal rule, let $a@H$ be the marked address to which the sent message $\bar{v}$ was sent. By the definition of a maximal representative, $\bar{K}'$ must have a corresponding actor in a context ready to send a message $\bar{v}'$ to a marked external address $a@H$ such that $\sigma(\bar{v}) = \bar{v}'$, $\sigma(a@H) = a'@H$, and $\text{ActorType}(a') = \text{ActorType}(a)$. Let $H'$ be the set of used markers in $\bar{K}$. By the definition of this rule, there exist some $H''$ and $\bar{v}''$ such that $\langle \bar{v}'', H'' \rangle \in \text{Markings}(\bar{v}, H')$. Then by the definition of Markings, it is easy to see that there exists some $\bar{v}'''$ such that $\langle \bar{v}'', H'' \rangle \in \text{Markings}(\bar{v}', H')$. Let $\sigma'$ be the function like $\sigma$, except that it maps the new marked addresses in $\bar{v}'$ to those in $\bar{v}''$ (because the markers are fresh, this does not affect change the mapping for any previously existing markers in $\bar{K}$. Then $\bar{K}'$ can take a transition $\bar{K}' \xrightarrow{\bar{l}'} \bar{K}''$ to send $\text{ival}'$ to $a'@H$, and $\sigma'(\bar{l}) = \bar{l}'$. As in the previous case, the resulting actor-behavior maps $\bar{\beta}$ and $\bar{\beta}'$ in $\bar{K}''$ and $\bar{K}'''$ differ only by $\sigma$, and the message multisets are unchanged. The used-marker component for both $\bar{K}''$ and $\bar{K}'''$ is $H''$. Because $\text{ActorType}(a') = \text{ActorType}(a)$, it is easy to see by the definition of $\sigma'$ and IntAddrTypes that for each $\langle a@H'', \tau \rangle \in \text{IntAddrTypes}(\bar{v}'', \text{ActorType}(a))$, there exists $\tau'$ such that $\langle a@H'', \tau' \rangle \in \text{IntAddrTypes}(\bar{v}'', \text{ActorType}(a'))$ and $\tau < \tau'$; this covers the new receptionists in $\bar{K}''$ and $\bar{K}'''$. Therefore, $\bar{K}'''$ is a
maximal representative of $\bar{K}''$.

- The cases for M-RECEIVEINTERNAL and M-SENDINTERNAL are similar to the above cases. The remaining cases involve no communication and are therefore simple.

Now, let $R'$ be the conformance-dense relation that proves $\bar{K}_{init} \vdash S'_{init}$, and define $R$ as follows.

$$R = \{ ⟨\bar{K}, S⟩ | ∃\bar{K}' \text{ is a maximal representative of } \bar{K} \text{ and } ⟨\bar{K}', S⟩ ∈ R' \}$$

We have that $⟨\bar{K}_{init}, S_{init}⟩ ∈ R$; it remains to show that $R$ is conformance-dense. Let $⟨\bar{K}_1, S_1⟩$ be a member of $R$, and let $\bar{K}_1 \xrightarrow{I_1} \bar{K}_2 \xrightarrow{I_2} \ldots$ be a fair, non-stuck program execution. By the definition of $R$, there exists some $\bar{K}'$ such that $\bar{K}'_1$ is a maximal representative of $\bar{K}_1$ and $⟨\bar{K}'_1, S⟩ ∈ R'$. By the above simulation result, there exists some simulating execution $\bar{K}'_1 \xrightarrow{l_1'} \bar{K}'_2 \xrightarrow{l_2'} \ldots$ with the same number of transitions, and some functions $σ_1, \ldots$ such that $σ_i(⌊l_i⌋) = ⌊l_i'⌋$ for each corresponding pair of labels in the executions, and $\bar{K}'_i$ is a maximal representative of $\bar{K}_i$ for each corresponding pair of configurations.

Because the actor-behavior maps and message multisets in each pair of configurations are identical up to the renaming by some $σ$, $\bar{K}_1 \xrightarrow{l_1} \bar{K}_2 \xrightarrow{l_2} \ldots$ is also a fair, non-stuck execution. Therefore, because $R'$ is conformance-dense, there exists a fair specification execution $S_1 \xrightarrow{[l_1]} S_2 \xrightarrow{[l_2]} \ldots$ with the same length as $\bar{K}_1 \xrightarrow{l_1} \bar{K}_2 \xrightarrow{l_2} \ldots$ such that $⟨\bar{K}'_i, S_i⟩ ∈ R$ for all $\bar{K}_i$ and $S_i$ in the respective executions. Then because each $σ_i(α@H) = α'@H'$ implies $H = H'$ for all $α@H$ and $α'@H'$, there also exists a fair specification execution $S_1 \xrightarrow{[l_1]} S_2 \xrightarrow{[l_2]} \ldots$ with the same length as $\bar{K}_1 \xrightarrow{l_1} \bar{K}_2 \xrightarrow{l_2} \ldots$ such that $⟨\bar{K}_i, S_i⟩ ∈ R$ for all $\bar{K}_i$ and $S_i$ in the respective executions. Therefore, $R'$ is conformance-dense, which implies that $\bar{K}_{init} \vdash S_{init}$, and therefore $P |- Σ$.

□
APPENDIX A. PROOF OF THE MAXIMAL INSTANTIATION THEOREM
Appendix B

Externals-Only Conformance

Conformance is defined partially in terms of the messages a program sends to addresses received from the environment, but it complicates conformance proofs when those are *internal* addresses. For instance, recall the scenario from section 3.4.3 in which a program is expected to respond to a message (variant \texttt{SendGetMean }a) by sending a \texttt{GetMean} message to \texttt{a}. When \texttt{a} is external, conformance requires only that the program send a \texttt{GetMean} back to the environment through \texttt{a}. When \texttt{a} is *internal*, though, that \texttt{GetMean} response will be sent to an internal actor, so conformance additionally requires that whatever actions the receiving actor takes to handle the \texttt{GetMean} are allowed by the specification.

However, it should be sufficient to show conformance only for the external-address cases and avoid the complexities of received internal addresses altogether. In CSA, the only way an actor can use an address is by sending a message to it (addresses cannot even be compared for equality). As a result, an actor’s communication behavior is parametric with respect to the addresses in its received messages. That is, whenever an actor receives a message containing an address, the actor will send that address the same messages regardless of which address it is. So if we know that a \texttt{SendGetMean} message containing an *external* address results in a \texttt{GetMean} message to that address, we should be able to infer that the same happens for a \texttt{SendGetMean} containing an *internal* address, as well.

At this point the careful reader might ask, “But can’t a program determine whether an address is internal based on whether an internal actor receives messages sent to it?” Well, if the environment can cause a message to be sent to some address \texttt{a} by including that address in some other message, then the environment could also send that message directly to \texttt{a}. So in a scenario like the one above, it is impossible for the actor receiving a \texttt{GetMean} message to determine whether that message came from the manager that received a \texttt{SendGetMean} message or from the environment directly.
Externals-only conformance is a refinement of the standard APS conformance relation that captures this idea. This notion of conformance is based on a restricted version of the CSA transition relation that does not allow messages from the environment to contain internal addresses.

**Definition.** The relation $\xrightarrow{\text{EO}}$ is defined such that $\xrightarrow{\text{EO}}$ if and only if
- $\xrightarrow{\text{EO}}$, and
- if there exist $a$, $H$, $\bar{v}$, and $\tau$ such that $\bar{H} = a \cdot \text{rcv-ext}(H, \bar{v}, \tau)$, then $\bar{v}$ contains no internal addresses.

The definition of externals-only conformance is then identical to standard conformance, except that it requires the specification to simulate only externals-only steps rather than steps.

**Definition.** A relation $R$ on program/specification-configuration pairs is externals-only-conformance-dense if and only if for all $\langle \bar{K}_1, S_1 \rangle \in R$ and all fair, non-stuck executions $\bar{K}_1 \xrightarrow{\text{EO}} \bar{K}_2 \xrightarrow{\text{EO}} \cdots$ there exists a fair specification execution $S_1 \xrightarrow{\text{EO}} S_2 \xrightarrow{\text{EO}} \cdots$ with the same length such that $\langle \bar{K}_i, S_i \rangle \in R$ for all $\bar{K}_i$ and $S_i$ in the respective executions.

**Definition.** A marked program configuration $\bar{K}$ externals-only conforms to a specification configuration $S$, written $\bar{K} \vdash_{\text{EO}} S$, if there exists an externals-only-conformance-dense relation $R$ such that $\langle \bar{K}, S \rangle \in R$. As with standard conformance, $\bar{P} \vdash_{\text{EO}} \Sigma$ if there exists some maximal instantiation $\langle \bar{K}, S \rangle$ of $\bar{P}$ and $\Sigma$ such that $\bar{K} \vdash_{\text{EO}} S$.

The remainder of this appendix provides the definitions, lemmas, and proofs used to show that externals-only conformance is equivalent to standard conformance.

### B.1 Definitions for Proofs

The proof that externals-only conformance implies standard conformance exploits the fact that a CSA program’s behavior is (in a sense) parametric in terms of the addresses provided to it from the environment. No expression can distinguish one address from another (e.g., the equality operator $\equiv$ is not defined for addresses), so a handler expression always takes the same actions, parameterized over the set of addresses bound to its free variables. APS specifications are similarly parametric. Therefore, whenever $\bar{P} \vdash_{\text{EO}} \Sigma$, the specification executions used to show that fact can be adapted to additionally show $\bar{P} \vdash \Sigma$. The proof therefore requires three relationships between the various components.

1. A relationship between two similar program/specification-configuration pairs: one pair in which the program can take any step, and another in which the program takes only externals-only steps.
2. A notion of the transition label $\tilde{l}'$ for an externals-only step $\tilde{K}' \xrightarrow{\text{EO}} \tilde{K}_2$ simulating the transition identified by the label $\tilde{l}$ of a standard step $\tilde{K} \xrightarrow{l} \tilde{K}_2$.

3. A notion of compatibility between a standard program configuration and a specification configuration to ensure that any monitored actions the program configuration can take are also possible in its externals-only variant.

### B.1.1 Externals-Only Variation

The first relationship is based on the existence of a (partial) substitution $\sigma$ that maps each marked address $a@H$ in the standard program configuration to a marked address of externals-only configuration. Such a substitution is max-preserving if and only if for all $a@H$, $a'@H'$, and $\eta$ such that $\sigma(a@H) = a'@H'$ and $\max(H) = \eta$, $\max(H') = \eta$. The application $\sigma(\tilde{K})$ of the substitution $\sigma$ to a program configuration $\tilde{K}$ is defined as the obvious element-wise, component-wise, point-wise application to its parts. The application is undefined if $\sigma(a@H)$ is undefined for any marked address $a@H$ in $\tilde{K}$. This substitution then leads to the definition of an externals-only variation.

**Definition.** A program configuration $\tilde{K}'$ is an externals-only variation of another configuration $\tilde{K}$ via the substitution $\sigma$ if and only if there exist $\tilde{\beta}$, $\tilde{\mu}$, $\tilde{\mu}'$, $H$, $\tilde{\rho}$, $\tilde{\rho}'$, and $\tilde{\rho}''$ such that the following conditions hold.

- **Max Preservation** $\sigma$ is max-preserving.
- **Matching Programs** $\sigma(\tilde{K}) = \langle \tilde{\beta} \mid \tilde{\mu} \cup \tilde{\mu}' \mid H \rangle^{\tilde{\beta} \cup \tilde{\rho}'}$ and $\tilde{K}' = \langle \tilde{\beta} \mid \tilde{\mu} \mid H \rangle^{\tilde{\rho} \cup \tilde{\rho}''}$.
- **Well-Typed** $\vdash_{\text{cfg}} \tilde{K}$ and $\vdash_{\text{cfg}} \tilde{K}'$.
- **Extra Receptionists** For all $\langle a@H', \tau \rangle \in \tilde{\rho}'$, $a$ is an external address and $\tau < : \text{ActorType}(a)$.
- **Extra Messages** For all $\langle a@H', \tilde{v} \rangle \in \tilde{\mu}'$, $a$ is an external address and $\text{IntAddrTypes}(\tilde{v}, \text{ActorType}(a)) \subseteq \tilde{\rho} \cup \tilde{\rho}''$.
- **Internal Addresses** For all internal marked addresses $a@H'$ appearing in $\tilde{\beta}$ or $\tilde{\mu}$, $H' = \emptyset$, and for all internal marked addresses $a@H'$ appearing in $\tilde{\beta} \cup \tilde{\rho}''$, $|H'| = 1$.
- **External Addresses** For all external marked addresses $a@H'$ appearing in $\tilde{\beta}$ or $\tilde{\mu}$, $|H'| = 1$.
- **No Unused Mappings** For all $a@H' \in \text{dom}(\sigma)$, $H' \subseteq H$, and if $H' = \emptyset$ then $a \in \text{dom}(\tilde{\beta})$.
- **No External to Internal Mappings** For all $a@H'$ and $a'@H''$, if $a$ is an external address and $\sigma(a@H') = a'@H''$, then $a'$ is an external address.
Internal to Internal Mappings  For all $a@H'$ and $a'@H''$ such that $\sigma(a@H') = a'@H''$, if $a$ and $a'$ are both internal, then $a@H = a'@H'$.

Internal to External Mappings  For all $a@H'$ and $a'@H''$ such that $\sigma(a@H') = a'@H''$, if $a$ is internal and $a'$ is external, then there exists a receptionist $(a@H'''', \tau) \in \hat{\rho} \cup \hat{\rho}''$ such that

- $H''' = \{\min(H')\}$ and
- $\text{ActorType}(a') < \tau$.

External to External Mappings  For all $a@H'$ and $a'@H''$ such that $\sigma(a@H') = a'@H''$, if $a$ and $a'$ are both external, then $a = a'$.

The extra receptionists in the above definition are a result of the standard configuration sending an internal address back to the environment rather than an external one, which can happen when the environment sent that internal address to the program rather than an external one in some earlier step. Similarly, the extra messages are created when the standard configuration sends a message to an internal address rather than an external one. The conditions on the various mappings in $\sigma$ ensure that the externals-only configuration can account for these differences.

B.1.2 EO-Simulation

Next is the definition of one transition label $\hat{l}'$ $EO$-simulating another. Roughly, this means that the label represents the same actor taking the same kind of action. Furthermore, when the action is some sort of communication, the markers on the destination address for the externals-only label must be a particular subset of those on the standard label (later conditions will ensure that those are exactly the markers monitored by the specification configuration). The formal definition follows.

**Definition.** A transition label $\hat{l}'$ $EO$-simulates another label $\hat{l}$ via $\sigma$ if and only if the following conditions hold.

- $\sigma$ is max-preserving.
- For all $a@H$ and $a'@H'$ such that $\sigma(a@H) = a'@H'$, if $a$ is internal, then $a@H = a'@H'$ and $|H| \leq 1$.
- $\hat{l}$ and $\hat{l}'$ have the same active actor $a$.
- If $\hat{l} = a: \text{rcv}\text{-int}(H, \hat{v})$, then either $\sigma(a@H)$ is an external marked address or $\hat{l}' = a: \text{rcv}\text{-int}(H, \hat{v}')$ where $\sigma(\hat{v}) = \hat{v}'$.
- If $[\hat{l}] = \bullet$, then $[\hat{l}'] = \bullet$.
- If $[\hat{l}] = a@H?!\hat{v}$, then $[\hat{l}'] = a'@H'?!\hat{v}'$, $H' = \{\min(H)\}$, and $\sigma(\hat{v}) = \hat{v}'$.
- If $[\hat{l}] = a@H!\hat{v}$, then $[\hat{l}'] = a'@H'!\hat{v}'$, $H' = \{\max(H)\}$, and $\sigma(\hat{v}) = \hat{v}'$. 
B.2. EO INPUT-PATTERN-MATCHING LEMMA

B.1.3 EO-Compatibility

Finally, when an externals-only transition \( \tilde{K}_1 \overset{I}{\rightarrow} \tilde{K}_2 \) simulates some standard transition \( \tilde{K}_1 \overset{I}{\rightarrow} \tilde{K}_2 \), the markers on the addresses involved might differ. The following condition on specification configurations ensures that the two transitions have the same monitored markers, even if they differ otherwise.

The definition first requires some definitions to name the sets of markers that a PSM monitors. For a PSM \( s = \left\langle H, H', \varphi : \bar{\eta}, \bar{\Phi}, O \right\rangle \), section 3.6.5 defined \( \text{OutMon}(s) \) as that PSM’s set of output-monitored markers, i.e., \( \text{OutMon}(s) = H' \). Similarly, \( \text{InMon}(s) \) is defined that PSM’s set of input-monitored markers, i.e., \( \text{InMon}(s) = H \).

With those helper functions in hand, we can now define the notion of a program configuration being EO-compatible with a PSM.

Definition. A program configuration \( \tilde{K} = \left\langle \tilde{\mu}, \tilde{\rho} \right| H \rangle \) is EO-compatible with a PSM if and only if the following conditions hold.

1. For all \( a \in H \) appearing in \( \tilde{K} \), either \( H = \emptyset \) or \( H \cap \text{InMon}(s) \subseteq \{ \text{min}(H) \} \).
2. For all \( a \in H \) appearing in \( \tilde{K} \), either \( H = \emptyset \) or \( H \cap \text{OutMon}(s) \subseteq \{ \text{max}(H) \} \).
3. For all \( \langle a \in H, \tilde{\varphi} \rangle \in \tilde{\mu} \) such that \( H \neq \emptyset \), and for all \( a' \in H' \) appearing in \( \tilde{\varphi} \), \( H' \cap \text{OutMon}(s) = \emptyset \).
4. For all \( \langle a \in H, \tau \rangle \in \tilde{\rho}, H \cap \text{OutMon}(s) = \emptyset \).

Intuitively, this definition states the following.

1. Only the minimal marker on a given address may be input-monitored.
2. Only the maximal marker on a given address may be output-monitored.
3. For all marked addresses in the “extra” messages in \( \tilde{K} \) (as indicated by \( H \neq \emptyset \)), none of the markers on contained addresses are output-monitored.
4. None of the markers on receptionists in \( \tilde{K} \) are output-monitored.

The definition of EO-compatibility naturally extends to specification configurations as follows.

Definition. A program configuration \( \tilde{K} \) is EO-compatible with a specification \( S \) if and only if for all \( s \in S, \tilde{K} \) is EO-compatible with \( s \).

B.2 EO Input-Pattern-Matching Lemma

Lemma (Externals-Only Input-Pattern-Matching). For all \( \tilde{v}, \tilde{v}', \sigma, \pi, \eta_1, \ldots, \eta_n, \) and \( \eta'_1, \ldots, \eta'_n, \) if \( \tilde{v}' \sim \pi \triangleright [x_1 \rightarrow \eta'_1, \ldots, x_n \rightarrow \eta'_n], \) \( \sigma(\tilde{v}) = \tilde{v}', \) and \( \sigma \) is max-preserving, then \( \tilde{v} \sim \pi \triangleright [x_1 \rightarrow \eta_1, \ldots, x_n \rightarrow \eta_n] \).
APPENDIX B. EXTERNALS-ONLY CONFORMANCE

Proof. The proof is by structural induction on $pi$. All cases other than $pi = x$ are trivial. When $pi = x$, by the definition of the corresponding pattern-matching rule there exist $a$, $H$, and $\eta$ such that $v' = a@H$, $\max(H) = \eta$, and $[x_1 \rightsquigarrow \eta_1, \ldots, x_n \rightsquigarrow \eta_n] = [x \rightsquigarrow \eta]$.

By the definition of $\sigma$, there must also exist $a'$ and $H'$ such that $v = a'@H'$. Then because $\sigma$ is max-preserving, we also know that $\max(H') = \eta$. Therefore, $\bar{v} \sim x \leftrightarrow [x \rightsquigarrow \eta]$, which completes the proof.

B.3 EO Output-Pattern-Matching Lemma

Lemma (Externals-Only Output-Pattern-Matching). For all $\bar{v}$, $\bar{v}'$, $\sigma$, $po$, $H$, and $S$, if

- $\bar{v}' \sim po \triangleright H, S$,
- $\sigma(\bar{v}) = \bar{v}'$,
- $\sigma$ is max-preserving, and
- for all $a@H'$ and $a'@H''$ such that $\sigma(a@H') = a'@H''$, if $a' is internal then $a is also internal,

then $\bar{v} \sim po \triangleright H, S$.

Proof. The proof is by structural induction on $po$. All cases other than $po = \text{self-addr}$ and $po = (\text{delayed-fork-addr } (\text{goto } \phi) \ \Phi)$ are trivial.

In the case of $po = \text{self-addr}$, by the definition of the corresponding pattern-matching rule there exist some $a$, $H'$, and $\eta$ such that $v' = a@H'$, $a$ is internal, $\max(H') = \eta$, $H = \{\eta\}$, and $S = \emptyset$. By the definition of $\sigma$, there must also exist $a'$ and $H''$ such that $\bar{v} = a'@H''$. Then by the definition of this lemma, we know that $a'$ is internal. Furthermore, because $\sigma$ is max-preserving, $\max(H'') = \eta$. Therefore we have $a'@H'' \sim po \triangleright \{\eta\}, c$, which completes this case.

The case for $po = (\text{delayed-fork-addr } (\text{goto } \phi) \ \Phi)$ is similar.

B.4 Externals-Only PSM Input Lemma

Lemma (Externals-Only PSM Input). For all $\bar{K}_1$, $\bar{K}_2$, $\bar{l}$, $\bar{l}'$, $a$, $H$, $\bar{v}$, $s_1$, $s_2$, $O$, and $S$, if

- $\bar{l} = a@H?\bar{v}$,
- $\bar{K}_1 \quad \bar{l} \quad \bar{K}_2$,
- $\bar{l}'$ EO-simulates $\bar{l}$,
- $\bar{K}_1$ is EO-compatible with $s_1$, and
- $s_1 \quad \bar{l}', O, S \quad s_2$. 

B.4. EXTERNALS-ONLY PSM INPUT LEMMA

Then \( s_1 \xrightarrow{[\hat{I}]_{O,S}} s_2 \), \( K_2 \) is EO-compatible with \( s_2 \) and every PSM \( s' \in S \).

Proof: By the definition of EO-simulation, there exist \( \sigma \), \( a' \), \( H' \), and \( \hat{v}' \) such that \( \sigma \) is max-preserving, \( \hat{v}' = a'@H'?\hat{v}' \), \( H' = \{ \min(H) \} \), and \( \sigma(\hat{v}) = \hat{v}' \). The rules P-MONITOREDRECEIVE and P-UNMONITOREDRECEIVE are the only transition rules that allow a transition with the label \( a'@H'?\hat{v}' \). We prove the lemma for each case separately.

Case: P-MONITOREDRECEIVE

By the definition of this rule, we know InMon\((s_1) \cap H' \neq \emptyset \). Then because \( H' = \{ \min(H) \} \), InMon\((s_1) \cap H \neq \emptyset \).

By the definition of this rule, we have \( \hat{v}' \sim pi > [x_1' - \eta_1', \ldots, x_m' - \eta_m'] \) for some \( pi, x_1', \ldots, x_m' \), and \( \eta_1', \ldots, \eta_m' \). Then by the Externals-Only Input-Pattern-Matching lemma, we also have \( \hat{v} \sim pi > [x_1 - \eta_1', \ldots, x_m' - \eta_m'] \). The remaining parts of the rule are identical for both \([\hat{I}] \) and \([\hat{I}] \), so we have \( s_1 \xrightarrow{[\hat{I}]_{O,S}} s_2 \).

It remains to show that \( K_2 \) is EO-compatible with \( s_2 \) and every PSM \( s' \in S \). Let \( s'' \) be a member of \( \{ s_2 \} \cup S \). By the definition of the P-MONITOREDRECEIVE rule, InMon\((s'') \subseteq \) InMon\((s_1) \) and OutMon\((s'') \subseteq \) OutMon\((s_1) \cup \{ \eta_1', \ldots, \eta_m' \} \). We will show that \( K_2 \) is EO-compatible with \( s'' \).

For the first two conditions for EO-compatibility, let \( a''@H'' \) be a marked address appearing in \( K_2 \). If \( a''@H'' \) appeared in \( K_1 \), then we already know that \( H'' \cap InMon(s_1) \subseteq \{ \min(H'') \} \) and \( H'' \cap OutMon(s_1) \subseteq \{ \max(H'') \} \). By the definition of the transition \( K_1 \xrightarrow{I} K_2 \), none of the markers \( \eta_1', \ldots, \eta_m' \) appear in \( K_1 \) (i.e., they are fresh), so \( H'' \cap InMon(s'') \subseteq \{ \min(H'') \} \) and \( H'' \cap OutMon(s'') \subseteq \{ \max(H'') \} \).

Otherwise, \( a''@H'' \) must be a new marked address created during the transition \( K_1 \xrightarrow{I} K_2 \), with \( \max(H'') \) a fresh marker not appearing in \( K_1 \). If the transition is an instance of the M-RECEIVEEXTERNAL rule, then either \( a'' \) is external and therefore \( |H''| = 1 \) (in which case the first two EO-compatibility properties hold), or there is a receptionist \( \langle a''@H'', \tau \rangle \) on \( K_1 \) such that \( H'' = H'' - \{ \max(H'') \} \). In that latter case, if \( H'' = \emptyset \), then the first two EO-compatibility properties hold automatically. If \( H'' \neq \emptyset \), then \( \min(H'') = \max(H'') \), and because \( \max(H'') \) is fresh, \( H'' \cap InMon(s'') \subseteq \{ \min(H'') \} \). Because \( K_1 \) is EO-compatible with \( s_1 \), we also know that \( H'' \cap OutMon(s_1) = \emptyset \). Therefore, \( H'' \cap OutMon(s'') \subseteq \{ \max(H'') \} \).

For the third condition for EO-compatibility, let \( \langle a''@H'', \hat{v}'' \rangle \) be an in-flight message in \( K_2 \) such that \( H'' \neq \emptyset \), and let \( a''@H''' \) be a marked address appearing in \( \hat{v}'' \). The transition \( K_1 \xrightarrow{I} K_2 \) must be an instance of either the M-RECEIVEEXTERNAL or M-RECEIVEINTERNAL rule, so by the definition of those rules, \( \langle a''@H'', \hat{v}'' \rangle \) is also an in-flight message in \( K_1 \). Because \( K_1 \) is EO-compatible with \( s_1 \), we know that \( H''' \cap OutMon(s_1) = \emptyset \). Then because none of the markers \( \eta_1', \ldots, \eta_m' \) appear in \( K_1 \), \( H''' \cap OutMon(s'') = \emptyset \).

Finally, let \( \langle a''@H'', \tau \rangle \) be a receptionist on \( K_2 \). Again by the M-RECEIVEEXTERNAL and M-RECEIVEINTERNAL rules, \( \langle a''@H'', \tau \rangle \) must also
be a receptionist on $K_1$, so $H'' \cap \text{OutMon}(s_1) = \emptyset$. Then because none of the markers $\eta'_1, \ldots, \eta'_m$ appear in $K_1$, $H'' \cap \text{OutMon}(s'') = \emptyset$.

**Case: P-UNMONITOREDRECEIVE**

In this case, we know $\text{InMon}(s_1) \cap H' = \emptyset$. By the definition of the M-RECEIVEEXTERNAL or M-RECEIVEINTERNAL rule that enables the $K_1 \xrightarrow{I} K_2$ transition, the address $a@H$ must appear in $K$. Then because $K_1$ is EO-compatible with $s_1$, $H \cap \text{InMon}(s_1) \subseteq \{\min(H)\}$. Then because $H' = \{\min(H)\}$, $H \cap \text{InMon}(s_1) = \emptyset$, and therefore $s_1 \xrightarrow{[I],O,S} s_2$ via P-UNMONITOREDRECEIVE.

It remains to show that $K_2$ is EO-compatible with $s_2$ and every PSM $s' \in S$. By the definition of the P-UNMONITOREDRECEIVE rule, $s_2 = s$ and $S = \emptyset$, so it suffices to show that $K_2$ is EO-compatible with $s_2$. The proof is similar to the proof for the P-REMONITOREDRECEIVE case above, except that $\text{OutMon}(s_2) = \text{OutMon}(s_1)$ (i.e., there are no new output-monitored markers in $s_2$). □

**B.5 Externals-Only Specification Input Lemma**

**Lemma** (Externals-Only Specification Input). For all $K_1$, $K_2$, $I$, $I'$, $a$, $H$, $\bar{v}$, $S_1$, $S_2$, and $O$, if

- $[I] = a@H?\bar{v}$,
- $K_1 \xrightarrow{I} K_2$,
- $I'$ EO-simulates $I$,
- $K_1$ is EO-compatible with $S_1$, and
- $S_1 \xrightarrow{[I'],O} S_2$,

then $S_1 \xrightarrow{[I],O} S_2$ and $K_2$ is EO-compatible with $S_2$.

**Proof.** By the S-SENDORRECEIVE rule for the configuration transition relation, there exist $s_1, \ldots, s_n, O_1, \ldots, O_n, S'_1, \ldots, S'_n$, and $s'_1, \ldots, s'_n$ such that

- $S_1 = \{s_1, \ldots, s_n\}$,
- $s_i \xrightarrow{[I],O_i,S_i} s'_i$ for all $i \in 1 \ldots n$,
- $O = O_1 \uplus \ldots \uplus O_n$, and
- $S_2 = \{s'_1, \ldots, s'_n\} \cup S'_1 \cup \ldots \cup S'_n$.

By the Externals-Only PSM Input lemma, for all $i \in 1 \ldots n$, $s_i \xrightarrow{[I],O_i,S_i} s'_i$, $K_2$ is EO-compatible with $s'_i$, and $K_2$ is EO-compatible with every PSM $s'' \in S'_i$.

Therefore, by the S-SENDORRECEIVE rule, $S_1 \xrightarrow{[I],O} S_2$ and $K_2$ is EO-compatible with $S_2$. □
B.6 Externals-Only PSM Output Lemma

Lemma (Externals-Only PSM Output). For all $\bar{K}_1$, $\bar{K}_2$, $\bar{l}$, $\bar{l}'$, $a$, $H$, $\bar{v}$, $s_1$, $s_2$, $O$, and $S$, if

- $[\bar{l}] = a@H\bar{v}$,
- $\bar{K}_1 \xrightarrow{\bar{l}} \bar{K}_2$,
- $\bar{l}'$ EO-simulates $\bar{l}$,
- $\bar{K}_1$ is EO-compatible with $s_1$, and
- $s_1 \xrightarrow{[\bar{l}]} O.S \rightarrow s_2$,

then $s_1 \xrightarrow{[\bar{l}]} O.S \rightarrow s_2$, $\bar{K}_2$ is EO-compatible with $s_2$, and $\bar{K}_2$ is EO-compatible with every PSM $s' \in S$.

Proof. By the definition of EO-simulation, there exist $\sigma$, $a'$, $H'$, and $\bar{v}'$ such that

- $\sigma$ is max-preserving,
- for all $a''@H''$ and $a''@H'''$ such that $\sigma(a''@H'') = a''@H'''$, if $a''$ is internal, then $a''@H'' = a''@H'''$ and $|H''| \leq 1$,
- $\bar{l}' = a'H'\bar{v}'$,
- $H' = \{\text{max}(H)\}$, and
- $\sigma(\bar{v}) = \bar{v}'$.

The only PSM-transition rule that allows a transition with the label $a'H'\bar{v}'$ is P-SEND. By the definition of that rule, there exist $\eta_1', \ldots, \eta_n'$, $p_{o_1}, \ldots, p_{o_n}$, $H_1''', \ldots, H_n'''$, and $S_1, \ldots, S_n$ such that $InMon(s_1) \cap H' = \{\eta_1', \ldots, \eta_n'\}$ and $\bar{v}' \sim p_{o_i} H'''_i, S_i$ for all $i \in 1 \ldots n$.

The transition $\bar{K}_1 \xrightarrow{\bar{l}} \bar{K}_2$ must be a use of either the M-SENDEXTERNAL or M-SENDINTERNAL rule, and in either case $a@H$ must appear in $\bar{K}_1$. Then because $\bar{K}_1$ is EO-compatible with $s_1$, $H \cap InMon(s_1) \subseteq \{\text{max}(H)\}$, and therefore $H \cap InMon(s_1) = \{\eta_1', \ldots, \eta_n'\}$. Also, by the Externals-Only Output-Pattern-Matching lemma, $\bar{v} \sim p_{o_i} H'''_i, S_i$ for all $i \in 1 \ldots n$. The remaining parts of the transition rule are identical for both $[\bar{l}]$ and $[\bar{l}']$, so we have $s_1 \xrightarrow{[\bar{l}]} O.S \rightarrow s_2$.

It remains to show that $\bar{K}_2$ is EO-compatible with $s_2$ and every PSM $s' \in S$. Let $s''$ be a member of $(s_2) \cup S$, and let $H_{new}$ be the set of markers that appear in $\bar{K}_2$ but not $\bar{K}_1$. By the definition of the M-SENDEXTERNAL or M-SENDINTERNAL rule that enables the transition $\bar{K}_1 \xrightarrow{\bar{l}} \bar{K}_2$, and the definition of the P-SEND rule, $InMon(s'') \subseteq InMon(s_1) \cup H_{new}$ and $OutMon(s'') \subseteq OutMon(s_1)$. We will show that $\bar{K}_2$ is EO-compatible with $s''$. 
For the first two conditions for EO-compatibility, let $a''@H''$ be a marked address appearing in $\bar{K}_2$. If $a''@H''$ appeared in $\bar{K}_1$, then we already know that $H'' \cap \text{InMon}(s_1) \subseteq \{\min(H'')\}$ and $H'' \cap \text{OutMon}(s_1) \subseteq \{\max(H'')\}$. None of the markers in $H_{\text{new}}$ appear in $\bar{K}_1$, so we therefore know that $H'' \cap \text{InMon}(s'') \subseteq \{\min(H'')\}$ and $H'' \cap \text{OutMon}(s'') \subseteq \{\max(H'')\}$.

If $a''@H''$ does not appear in $\bar{K}_2$, then by the definition of the M-SENDEXTERNAL and M-SENDINTERNAL rules, it must be one of the freshly marked addresses in $\bar{v}$. Therefore there exists some $H''' = H'' \setminus \{\max(H'')\}$ such that $a''@H'''$ appears in $\bar{K}_1$. By the P-SEND rule, $H'' \cap \text{OutMon}(s_1) = \emptyset$, so we also have $H'' \cap \text{OutMon}(s'') = \emptyset$.

For the input-monitored markers, we know there exists some $a'''$ and $H'''$ such that $\sigma(a''@H'') = a'''@H'''$ and $\max(H'') = \max(H'''')$. If $\max(H'''') \in \text{InMon}(s'')$, then by the input-pattern-matching rules, $a'''$ is internal. Then by the EO-simulation properties, $H''' = H''''$ and $|H'''| = 1$, and therefore $H'' \cap \text{InMon}(s''') \subseteq \{\min(H'''')\}$ because $|H'''| = 1$. On the other hand, if $\max(H'''') \notin \text{InMon}(s'')$, then $\max(H'''') \notin \text{InMon}(s'')$ because $\max(H'') = \max(H'''')$. We know that $a'''@H'''$ appears in $\bar{K}_1$, and by EO-compatibility, either $H''' = \emptyset$ or $H''' \cap \text{InMon}(s_1) \subseteq \{\min(H'''')\}$. Then we also have that $H''' \cap \text{InMon}(s_1) \subseteq \{\min(H'''')\}$. Because $H'''' = H''' - \max(H''')$, either $H'''' = \emptyset$ or $\min(H'''') = \max(H''')$. We know that $H'''' \cap H_{\text{new}} = \emptyset$ and $\max(H''') \notin \text{InMon}(s'')$, so $H'''' \cap \text{InMon}(s'') \subseteq \{\min(H'''')\}$.

For the third EO-compatibility property, let $\langle a''@H'', \bar{v}'' \rangle$ be an in-flight message in $\bar{K}_2$ such that $H'' \neq \emptyset$, and let $a'''@H'''$ be a marked address appearing in $\bar{v}''$. If $\langle a''@H'', \bar{v}'' \rangle$ is also an in-flight message in $\bar{K}_1$, then we know that $H'''' \cap \text{OutMon}(s_1) \subseteq \{\max(H'''')\}$, and therefore $H'''' \cap \text{OutMon}(s'') \subseteq \{\max(H'''')\}$.

Otherwise, $\bar{v}''$ is the message sent in the transition $\bar{K}_1 \xrightarrow{I} \bar{K}_2$. In that case, by the P-SEND rule, $H'''' \cap \text{OutMon}(s_1) = \emptyset$. Therefore, $H'''' \cap \text{OutMon}(s'') = \emptyset$.

Finally, for the fourth EO-compatibility property, let $\langle a''@H'', r \rangle$ be a receptionist on $\bar{K}_2$. If $\langle a''@H'', r \rangle$ is also a receptionist on $\bar{K}_1$, then because $\bar{K}_1$ is EO-compatible with $s_1$, $H'' \cap \text{OutMon}(s_1) = \emptyset$, and therefore $H'' \cap \text{OutMon}(s'') = \emptyset$.

Otherwise, $\langle a''@H'', r \rangle$ must be a receptionist created in the transition $\bar{K}_1 \xrightarrow{\bar{l}'} \bar{K}_2$, meaning that $a''@H''$ appears in the sent message. In that case, by the P-SEND rule, $H'' \cap \text{OutMon}(s_1) = \emptyset$, so $H'' \cap \text{OutMon}(s'') = \emptyset$.

\begin{flushright}
$\square$
\end{flushright}

### B.7 Externals-Only Specification Output Lemma

**Lemma** (Externals-Only Specification Output). For all $\bar{K}_1, \bar{K}_2, \bar{I}, S_1, S_2, \text{ and } O$, if

- $\bar{I} = a@H\bar{v}$,
- $\bar{K}_1 \xrightarrow{I} \bar{K}_2$,
- $\bar{l}'$ EO-simulates $\bar{l}$,
- $\bar{K}_1$ is EO-compatible with $S_1$, and

then $\bar{K}_1 \xrightarrow{\bar{l}'} \bar{K}_2$. 


B.8. EXTERNALS-ONLY SILENT STEP LEMMA

• \( S_1 \xrightarrow{[\hat{l}].O} S_2 \).

then \( S_1 \xrightarrow{[\hat{l}].O} S_2 \) and \( \bar{K}_2 \) is EO-compatible with \( S_2 \).


\[ \square \]

B.8 Externals-Only Silent Step Lemma

Lemma (Externals-Only Silent Step). For all \( \bar{K}, S_1, \) and \( S_2 \), if

• \( \bar{K} \) is EO-compatible with \( S_1 \), and

• \( S_1 \xrightarrow{.,\emptyset} S_2 \),

then \( \bar{K} \) is EO-compatible with \( S_2 \).

Proof. By the definition of the \( \xrightarrow{\cdot} \) relation, there exist \( S'_1, \ldots, S'_n \) such that \( S'_1 = S_1, S'_n = S_2 \), and \( S'_1 \xrightarrow{.,\emptyset} \cdots \xrightarrow{.,\emptyset} S'_n \). We will show that for all \( i \in 1 \ldots n \), \( \bar{K} \) is EO-compatible with \( S'_i \). The proof is by induction on \( i \).

In the base case, \( S'_1 = S_1 \), and we already have that \( \bar{K} \) is EO-compatible with \( S_1 \). In the inductive case, we know that \( \bar{K} \) is EO-compatible with \( S'_i \) and that \( S'_i \xrightarrow{.,\emptyset} S'_{i+1} \); we must show that \( \bar{K} \) is EO-compatible with \( S'_{i+1} \). A transition of the form \( S'_i \xrightarrow{.,\emptyset} S'_{i+1} \) cannot change the sets of input-monitored or output-monitored markers in the configuration, so we have that \( \bar{K} \) is EO-compatible with \( S'_{i+1} \). \( \square \)

B.9 EO Specification Simulation Lemma

Lemma (Externals-Only Specification Simulation). For all \( \bar{K}_1, \bar{K}_2, \bar{l}, \bar{l}', S_1, S_2, \) and \( O \), if

• \( \bar{K}_1 \xrightarrow{\bar{l}} \bar{K}_2 \),

• \( \bar{l}' \) EO-simulates \( \bar{l} \),

• \( \bar{K}_1 \) is EO-compatible with \( S_1 \), and

• \( S_1 \xrightarrow{[\hat{l}].O} S_2 \),

then \( S_1 \xrightarrow{[\hat{l}].O} S_2 \) and \( \bar{K}_2 \) is EO-compatible with \( S_2 \).

Proof. The proof is by cases on the shape of \( [\hat{l}] \).
Case: \([\vec{l}] = \bullet\)

By the definition of EO-simulation, \([\vec{l}'] = \bullet\), and so we have \(S'_1 \xrightarrow{[\vec{l}],\sigma} \ldots \xrightarrow{[\vec{l}],\sigma} S'_n\). It remains to show that \(\vec{K}_2\) is EO-compatible with \(S_2\).

By the Externals-Only Silent Step lemma, \(\vec{K}_1\) is EO-compatible with \(S_2\); we must show that \(\vec{K}_2\) is similarly EO-compatible with \(S_2\). Because \([\vec{l}] = \bullet\), we know that

- the only new marked addresses created by the transition must be of the form \(a@\emptyset\) (they are addresses created by the M-SPAWN rule),
- \(H = \emptyset\) for any new in-flight message \(<aH,v>\), and
- the transition creates no new receptionists.

As a result of these facts and because \(\vec{K}_1\) is EO-compatible with \(S_2\), \(\vec{K}_2\) is also EO-compatible with \(S_2\).

**B.9.1 Case: \([\vec{l}] = a@H?\vec{v}\)**

By the definition of the \(\longrightarrow\) relation, there exist \(S_3\) and \(S_4\) such that \(S_1 \xrightarrow{[\vec{l}],\cdot} S_3 \xrightarrow{a@H?O} S_4 \xrightarrow{\cdot} S_2\). By the Externals-Only Silent Step lemma, \(\vec{K}_1\) is EO-compatible with \(S_3\). By the Externals-Only Specification Input lemma, \(S_3 \xrightarrow{[\vec{l}],O} S_4\) and \(\vec{K}_2\) is EO-compatible with \(S_4\). Then by another use of the Externals-Only Silent Step lemma, \(\vec{K}_2\) is EO-compatible with \(S_2\). Therefore, \(S_1 \xrightarrow{[\vec{l}],O} S_2\) and \(\vec{K}_2\) is EO-compatible with \(S_2\).

**B.9.2 Case: \([\vec{l}] = a@H!\vec{v}\)**

The proof for this case is identical to the proof for the previous case, except that it uses the Externals-Only Specification Output lemma instead of the the Externals-Only Specification Input lemma.

**B.10 Externals-Only Simulation Lemma**

**Lemma** (Externals-Only Simulation). For all \(\vec{K}_1, \vec{K}_2, \vec{K}'_1, \sigma\), and \(\vec{l}\), if \(\vec{K}'_1\) is an externals-only variation of \(\vec{K}_1\) via \(\sigma\) and \(\vec{K}_1 \xrightarrow{\vec{l}} \vec{K}_2\), then there exist \(\vec{l}'\), \(\vec{K}'_2\), and \(\sigma'\) such that

- \(\vec{K}'_1 \xrightarrow{\vec{l}'} \vec{K}'_2\),
- \(\vec{l}'\) EO-simulates \(\vec{l}\) via \(\sigma \cup \sigma'\), and
- \(\vec{K}'_2\) is an externals-only variation of \(\vec{K}_2\) via \(\sigma \cup \sigma'\).
Proof. The proof proceeds by case analysis on the transition rule that enables the step \( \bar{K}_1 \xrightarrow{l} \bar{K}_2 \). For each case, let \( a \) be the address of the active actor in this step, and let its behavior be \( b \). By the Internal to Internal Mappings, Well-Typed, and Matching Programs properties for an externals-only variation, there must exist an actor in \( \bar{K}_1' \) with address \( a \) and behavior \( \sigma(b) \). By the Max Preservation property, we also know that \( \sigma \) is max-preserving.

**Cases: M-GOTO, M-TIMEOUT, M-FUNC**

Because the actor at address \( a \) in \( \bar{K}_1' \) has behavior \( \sigma(b) \), it can take a similar step by the same transition rule, \( \bar{K}_1' \xrightarrow{\ell} \bar{K}_2' \) with \( \ell' = \sigma(l) \). The labels \( l \) and \( \ell' \) have the same active actor, \( l \) is not a \texttt{rcv-int} label, and \( [l] = [\ell'] = * \). Let \( \sigma' = \emptyset \); we have that \( \sigma \sqcup \sigma' \) is max-preserving, so \( \ell' \) EO-simulates \( l \) via \( \sigma \sqcup \sigma' \).

Next, we must show that \( \bar{K}_2' \) is an externals-only variation of \( \bar{K}_2 \) via \( \sigma \sqcup \sigma' \).

Let \( \sigma' \) be the behavior of the actor at \( a \) in \( \bar{K}_2 \). Each of these transition rules is completely parametric in terms of the addresses contained in the actor's behavior, so the behavior of the actor at \( a \) in \( \bar{K}_2' \) is \( \sigma(b') \). None of these transition rules change any other part of the configuration, so Matching Programs holds.

Well-Typed holds by the Type Preservation lemma (appendix J). The remaining properties hold for the same reasons they hold for \( \bar{K}_1' \) and \( \bar{K}_1 \), because no other parts of the configurations change.

**Case: M-SPAWN**

In this case, \( \ell = a : \texttt{spawn}(a') \). Let \( b' \) be the spawned actor's behavior in \( \bar{K}_2 \).

By Matching Programs, Well-Typed, and Internal to Internal Mappings, there is no actor in \( \bar{K}_1' \) with address \( a' \). Then because the actor at address \( a \) in \( \bar{K}_1' \) has behavior \( \sigma(b) \), it can take a similar step \( \bar{K}_1' \xrightarrow{\ell} \bar{K}_2' \) by the same transition rule, spawning an actor with address \( a' \) so that \( \ell' = l \).

Let \( \sigma' = \{ a' @ \emptyset \rightarrow a @ \emptyset \} \). By No Unused Mappings, \( a' @ \emptyset \notin \text{dom} (\sigma) \), so \( \sigma \) and \( \sigma' \) have disjoint domains. Let \( b'' \) be the behavior of the actor at \( a \) in \( \bar{K}_2 \). Then by the construction of the transition \( \bar{K}_1' \xrightarrow{\ell} \bar{K}_2' \), the behavior of the actor at \( a \) in \( \bar{K}_2' \) is \( (\sigma \sqcup \sigma')(b'') \), and the behavior of the actor at \( a' \) in \( \bar{K}_2' \) is \( (\sigma \sqcup \sigma')(b') \).

Because \( \text{dom}(\sigma') = \{ a' @ \emptyset \} \), \( \sigma' \) is max-preserving. We also know that \( \ell \) and \( \ell' \) have the same active actor \( a \), and \( [\ell] = [\ell'] = * \), so \( \ell' \) EO-simulates \( l \) via \( \sigma \sqcup \sigma' \).

Next, we must show that \( \bar{K}_2' \) is an externals-only variation of \( \bar{K}_2 \). We have already shown that \( \sigma \sqcup \sigma' \) is max-preserving. The only changed parts of the configurations \( \bar{K}_2 \) and \( \bar{K}_2' \) are the new behavior for \( a \) and the presence of the new actor at \( a' \). We have already shown the correspondence for those changes by \( \sigma \sqcup \sigma' \), so Matching Programs holds. Well-Typed holds by the Type Preservation lemma. The only new marked internal address in this transition is \( a' @ \emptyset \), which obeys the Internal Addresses condition. The only new mapping in \( \sigma \sqcup \sigma' \) is the one for the new actor at \( a' \), so No Unused Mappings holds. That mapping for
the new actor also respects the Internal to Internal Mappings property. All other properties hold for the same reason they did for $K'_2$ and $K_1$.

**Case: M-RECEIVEEXTERNAL**

In this case, there exist $H$, $\bar{v}$, and $\tau$ such that $\bar{l} = a : \text{rcv-ext}(H, \bar{v}, \tau)$. To find a simulating step from $K'_1$, we must find a corresponding receptionist $\langle a'@H', \tau' \rangle$ and construct a corresponding message $\bar{v}'$.

For the receptionist, if $\sigma(a@H)$ is internal, then by Extra Receptionists, $K'$ has the same receptionist $\langle a@H, \tau \rangle$ used for the transition $K_1 \xrightarrow{l} K_2$. By Internal to Internal Mappings and and Internal Addresses, $|H| = 1$, so $H = \{\min(H)\}$. In that case, let $a' = a$, $H' = H$, and $\tau' = \tau$. Otherwise, by Internal to External Mappings, there exists a receptionist $\langle a'@H', \tau' \rangle$ on $K'$ such that $H' = \{\min(H)\}$ and $\tau \vdash \tau' <: .$

To construct the message $\bar{v}'$, by the definition of M-RECEIVEEXTERNAL, every internal marked address $a''@H''$ in $\bar{v}$ must correspond to some receptionist $\langle a''@H'', \tau'' \rangle$ where the type $\tau''$ is acceptable at that position in $\bar{v}$. By the Extra Receptionists, Internal to Internal Mappings, and Internal to External Mappings properties, either that same receptionist is present in $K'_1$, or there exists some receptionist with type $\tau'''$ in $K'_1$ such that $\tau'' <: \tau'''$. Construct a value $\bar{v}'$ by replacing each such $a''@H''$ in $\bar{v}$ with a marked address $a'''@H'''$, where $a'''$ is an external address such that $\text{ActorType}(a''')$ is the type of the corresponding receptionist on $K'_1$, and $H''' = \{\max(H')\}$. By the definition of this transition rule, $\emptyset, \emptyset \vdash \bar{v} : \tau$, so by subsumption we have $\emptyset, \emptyset \vdash \bar{v}' : \tau$, and therefore $\emptyset, \emptyset \vdash \bar{v}' : \tau'$.

To construct the transition itself, because the behavior of the actor at $a$ in $K'$ is $\sigma'(\bar{v})$, its expression must be of the form $(\text{receive } x e tc)$ (i.e., ready to receive a message). Thus, there is a step $K'_1 \xrightarrow{l'} EO K'_2$ via M-RECEIVEEXTERNAL, where $l' = a : \text{rcv-ext}(H', \bar{v}', \tau')$.

Let $\sigma'$ be the substitution that maps every marked address in $\bar{v}$ to the marked address at the corresponding position in $\bar{v}'$. Every marked address in $\bar{v}$ has a fresh marker by the definition of the M-RECEIVEEXTERNAL rule, so $\sigma$ and $\sigma'$ have disjoint domains. Also, $\sigma'$ is max-preserving by the construction of $\bar{v}'$, so $\sigma \cup \sigma'$ is max-preserving. We also have that $\bar{l}$ and $l'$ have the same active actor, $|\bar{l}| = a@H!\bar{v}$, $|l'| = a'@H'?\bar{v}'$, $H' = \{\min(H)\}$, and $(\sigma \cup \sigma')(\bar{v}) = \bar{v}'$. Therefore, $l'$ EO-simulates $\bar{l}$ via $\sigma \cup \sigma'$.

It remains to show that $K'_2$ is an externals-only variation of $K_2$ via $\sigma \cup \sigma'$. We have already shown that $\sigma \cup \sigma'$ is max-preserving. The only part of the configurations that change are the behaviors for the actors at $a$, and the used-marker sets. Because the previous behaviors in $K_1$ and $K'_1$ correspond by $\sigma$, and $(\sigma \cup \sigma')(\bar{v}) = \bar{v}'$, the new behaviors correspond via $\sigma \cup \sigma'$ (because each new behavior just substitutes $\bar{v}$ or $\bar{v}'$ into the corresponding handler expression). The same new markers are added to $\bar{v}$ and $\bar{v}'$, so the used-marker sets for $K_2$ and $K'_2$ are identical. Therefore, Matching Programs holds.

Well-Typed holds by the Type Preservation lemma.
No new internal marked addresses appear in $K_2'$, so Internal Addresses still holds. Every new external address in $K_2'$ has exactly one marker on it, by the above construction of $\tilde{v}'$, so External Addresses holds.

Every mapping in $\sigma'$ is for some address $a''@H''$ with at least one marker in the used-marker set of $K_2$ (because a new marker was applied to every address in the message), so No Unused Mappings holds. Every such mapping is either an internal-to-external or external-to-external mapping. For each internal-to-external mapping $\sigma(a''@H'') = a''@H''$, the required receptionist exists by construction of $a''@H''$, so Internal to External Mappings holds. For each external-to-external mapping $\sigma(a''@H'') = a''@H''$, $a''@H'' = a''@H''$, and $|H''| = 1$ by the definition of M-ReceiveExternal, so External to External Mappings holds. All other properties hold for the same reason they did for $K_1'$ and $K_1$.

**Case: M-ReceiveInternal**

In this case, there exist $H, \tilde{v},$ and $\tilde{v}'$ such that $\tilde{l} = a : rcv-int(H, \tilde{v})$ and $\langle \sigma(a@H), \sigma(\tilde{v}') \rangle$ is an in-flight message in $K_1$ that triggered this transition. We must consider two cases: one in which $K_1'$ has an in-flight message $\langle \sigma(a@H), \sigma(\tilde{v}') \rangle$, and another in which it does not.

**Sub-Case 1**

In this case, we assume $K_1'$ has an in-flight message $\langle \sigma(a@H), \sigma(\tilde{v}') \rangle$. By Well-Typed, $\sigma(a@H)$ is internal, and therefore by Internal to Internal Mappings, $\sigma(a@H) = a@H$. Then by Matching Programs, $K_1'$ can take a similar M-ReceiveInternal step, $K_1' \xrightarrow{\tilde{l}'} \text{EO} K_2'$, where $\tilde{l}' = a : rcv-int(H, \tilde{v}'')$.

In this case, by Internal Addresses, $H = \emptyset$, so no new markers are created, and $\tilde{v} = \tilde{v}'$. Let $\sigma' = \sigma$; then $\sigma$ and $\sigma'$ have disjoint domains and $\sigma \cup \sigma'$ is max-preserving. Also because $H = \emptyset$, $|l| = |l'| = 1$. Finally, $l$ and $\tilde{l}'$ have the same active actor, and $\tilde{l}' = a : rcv-int(H, \tilde{v}'')$ where $\langle \sigma \cup \sigma'(\tilde{v}) = \tilde{v}''. \rangle$. Therefore, $\tilde{l}'$ EO-simulates $l$ via $\sigma \cup \sigma'$.

To show that $K_2'$ is an externals-only variation of $K_2$ via $\sigma \cup \sigma'$, we have already shown that $\sigma \cup \sigma'$ is max-preserving, and Well-Typed holds by the Type Preservation lemma. The only parts of each program configuration that changed were the behavior for the actor at $a$ and the removal of the message $\langle a@H, \tilde{v}' \rangle$. Because both configurations received the message, and because both the actor behavior and message in each configuration correspond by $\sigma \cup \sigma'$, Matching Programs holds. The remaining properties hold for the same reasons they hold for $K_1'$ and $K_1$, therefore $K_2'$ is an externals-only variation of $K_2$ via $\sigma \cup \sigma'$.

**Sub-Case 2**

In this case, $K_1'$ does not have an in-flight message $\langle \sigma(a@H), \sigma(\tilde{v}') \rangle$. Instead, $K_1'$ will simulate this step with an external receive step. This requires finding a suitable receptionist on $K_1'$ and constructing a suitable message $\tilde{v}'$.
By Extra Messages, \(\sigma(a@H)\) is an external marked address \(a’@H’\) and \(\text{IntAddrTypes}(\sigma(\bar{v})), \text{ActorType}(a’))\) is a subset of the receptionists on \(\bar{K}_1\). This in turn implies that \(\emptyset, \emptyset \vdash \sigma(\bar{v}) : \text{ActorType}(a’).\)

For the receptionist, by Internal to External Mappings, there exists a receptionist \(\langle a@H’, \tau \rangle\) on \(\bar{K}_1\) such that \(H’ = \{\min(H)\}\) and \(\text{ActorType}(a’)<:\tau.\)

Construct the message \(\bar{v}''\) similarly to the message for the M-RECEIVEEXTERNAL case above, replacing each internal address in \(\bar{v}\) with an external address whose type is based on the type of the internal address’ corresponding receptionist. Additionally, replace each external address \(a''@H''\) in \(\bar{v}\) with \(a''@\{\max(H'')\}\). By subsumption, \(\emptyset, \emptyset \vdash \bar{v}'':\text{ActorType}(a’),\) and therefore \(\emptyset, \emptyset \vdash \bar{v}'':\tau.\)

Because the behavior of the actor at \(a\) in \(\bar{K}_1\) is \(\sigma(\bar{b})\), its expression must be of the form (receive \(x\ e\) tc) (i.e., ready to receive a message). Thus, there is a step \(\bar{K}_1 \xrightarrow{\bar{l}} \bar{K}_2\) via M-RECEIVEEXTERNAL, where \(\bar{l} = a : \text{rcv-ext}(H'',\bar{v}'',\tau).\)

Let \(\sigma’\) be the substitution that maps every marked address in \(\bar{v}\) to the marked address at the corresponding position in \(\bar{v}''.\) Every marked address in \(\bar{v}\) has a fresh marker by the definition of the M-RECEIVEINTERNAL rule, so \(\sigma\) and \(\sigma’\) have disjoint domains. Also, \(\sigma’\) is max-preserving by the construction of \(\bar{v}'',\) so \(\sigma \cup \sigma’\) is max-preserving. We also have that \(\bar{l}\) and \(\bar{l}'\) have the same active actor, \((\sigma \cup \sigma’)(a@H)\) is an external marked address, \([\bar{l}] = a@H?\bar{v}, [\bar{l}’] = a’@H’?\bar{v}', H’ = \{\min(H)\}\), and \((\sigma \cup \sigma’)(\bar{v}) = \bar{v}'.\) Therefore, \(\bar{l}' EO\)-simulates \(\bar{l}\) via \(\sigma \cup \sigma’.\)

The proof that \(\bar{K}_2\) is an externals-only variation of \(\bar{K}_2\) via \(\sigma \cup \sigma’\) is nearly identical to the proof for the M-RECEIVEEXTERNAL case above. The only difference is that we must also account for the new external-to-external mappings as created above; these respect the External to External Mappings property by construction.

**Case: M-SENDEXTERNAL**

In this case, there exist \(a’, H, H_{used}\) and \(\bar{v}\) such that \(\bar{l} = a : \text{send-ext}(a’@H,\bar{v}),\) \(a’\) is external, and \(H_{used}\) is the used-markers component of \(\bar{K}_1.\) By the definition of this rule, there must exist \(\bar{v}', E,\) and \(H_{used}'\) such that \(\langle \bar{v}, H_{used}' \rangle \in \text{Markings}(\bar{v}', H_{used}),\) and the behavior for the actor at \(a\) in \(\bar{K}_1\) is \(E[\langle \text{send}\ a’@H\ \bar{v}' \rangle].\)

By Internal to Internal Mappings and Matching Programs, the actor at \(a\) in \(\bar{K}_1\) must be \(E'[\langle \text{send}\ a''@H'\ \bar{v}' \rangle]\) for some \(E', a'', H',\) and \(\bar{v}'\), where \(\sigma(E) = E',\) \(\sigma(a’@H) = a''@H',\) and \(\sigma(\bar{v}) = \bar{v}'.\) Also, the used-markers component of \(\bar{K}_1\) is \(H_{used}.\) Furthermore, by No External to Internal Mappings, \(a''\) must be external. Because \(\sigma(\bar{v}) = \bar{v}'',\) \(\bar{v}'\) can be marked using the exact same set of markers, so there exists \(\bar{v}''\) such that \(\langle \bar{v}'', H_{used}' \rangle \in \text{Markings}(\bar{v}'', H_{used}).\)

Therefore, \(\bar{K}_1\) can take a similar M-SENDEXTERNAL step, \(\bar{K}_1 \xrightarrow{\bar{l}} \bar{K}_2,\) where \(\bar{l} = a : \text{send-ext}(a’@H,\bar{v}'').\)

Let \(\sigma’\) be the function mapping every marked address in \(\bar{v}\) to the marked address at the corresponding position in \(\bar{v}''.\) Because each address in \(\bar{v}\) received a
B.10. EXTERNALS-ONLY SIMULATION LEMMA

fresh marker, the domains of $\sigma$ and $\sigma'$ are disjoint. Because those new markers on $\bar{v}$ and $\bar{v}''$ are the maximal markers on each address, $\alpha'$ is max-preserving, and therefore so is $\sigma \cup \sigma'$. The labels $\bar{l}$ and $\bar{l}'$ have the same active actor by construction. By Max Preservation and External Addresses, $H' = (\max(H))$. Finally, $\bar{l}$ is not a cv-int label, $[\bar{l}] = \alpha'@H\bar{v}$, $[\bar{l}'] = \alpha''@H'\bar{v}''$, and $(\sigma \cup \sigma')(\bar{v}) = \bar{v}''$, so $\bar{l}'$ EO-simulates $\bar{l}$ via $\sigma \cup \sigma'$.

It remains to show that $\bar{K}_2'$ is an externals-only variation of $\bar{K}_2$ via $\sigma \cup \sigma'$. We have already shown that $\sigma \cup \sigma'$ is max-preserving. The only changes in $\bar{K}_2$ and $\bar{K}_2'$ are the possible addition of new receptionists (which Matching Programs allows), and the evaluated send expression in the behavior of the actor at $a$ in each configuration is replaced with (variant Unit). Then we have $(\sigma \cup \sigma')\bar{E}((\text{variant Unit})) = \bar{E}'((\text{variant Unit}))$, so Matching Programs holds.

Well-Typed holds by the Type Preservation lemma.

Let $\langle a''@H'', \tau \rangle$ be one of the new receptionists on $\bar{K}_2$ such that $(\sigma \cup \sigma')(a''@H'') = a''@H''$ is an external address (and therefore not a receptionist on $\bar{K}_2'$). That receptionist must be in IntAddrTypes($\bar{v}$, ActorType($a''$)) and $a''$ must be internal. By the definition of this transition rule and of IntAddrTypes, $\phi, \phi \vdash \bar{v} : \tau$ and $\phi, \phi \vdash \bar{v}'' : \tau$. Therefore $\phi, \phi \vdash a''@H'' : (\text{Addr } \tau)$. That implies that $\tau \ll<: \text{ActorType}(a''\rangle)$, so Extra Receptionists holds.

No new internal messages were created in this transition, so Extra Messages holds.

For Internal Addresses, the only new marked internal addresses in $\bar{K}_2'$ are the new receptionists. Each of these is just a marked address from $\bar{v}''$ with a single new marker added. By Internal Addresses, each such marked address in $\bar{v}''$ has zero markers, so each such new receptionist has exactly one marker, and Internal Addresses holds. No new marked external addresses appear in $\bar{K}_2'$, so External Addresses still holds.

All new mappings in $\sigma'$ just add a new marker to an existing mapping in $\sigma$, so No Unused Mappings and No External to Internal Mappings hold, as well. For the remaining mapping properties, let there be $a''@H''$ and $a'''@H''$ such that $\alpha'(a''@H'') = a'''@H''$ (the mappings in $\sigma$ still satisfy the necessary properties, so we need not discuss those cases further). By Matching Programs and the definition of $\sigma'$, $\sigma(a''@H'' - \{\max(H'')\}) = a'''@H'' - \{\max(H'')\}$ and $\max(H'') = \max(H'')$. If $a''$ and $a'''$ are both internal, then by Internal to Internal Mappings $a'' = a'''$ and $H'' - \{\max(H'')\} = H'' - \{\max(H'')\}$. Therefore $H'' = H''$ and Internal to Internal Mappings holds. If $a''$ is internal but $a'''$ is external, then the necessary receptionist for Internal to External Mappings is the same as the one for the mapping from $a''@H'' - \{\max(H'')\}$ to $a'''@H'' - \{\max(H'')\}$; therefore Internal to External Mappings holds. If $a''$ and $a'''$ are both external, then by External to External Mappings for $\bar{K}_1$ and $\bar{K}_1$, $a'' = a'''$. The marked address $a''@H''$ does not appear in $\bar{K}_2$ (it appears only in the message, which escapes to the environment), so External to External Mappings holds. Therefore, $\bar{K}_2'$ is an externals-only variation of $\bar{K}_2$ via $\sigma \cup \sigma'$. 
**Case: M-SendInternal**

In this case, there exist $a'$, $H$, and $\bar{v}$ such that $\bar{I} = a : \text{send-int}(a'@H, \bar{v})$, and let $a''@H'$ be the marked address such that $\sigma(a'@H) = a''@H'$.

We must consider two cases: one in which $a''$ is an internal address, and one in which it is external.

**Sub-Case 1**

In this case, $a''$ is an internal address. Then by Internal to Internal Mappings and Internal Addresses, $a'' = a'$ and $H = H' = \emptyset$. Then by the definition of this rule, we know that the actor at $a$ in $K_1$ has behavior expression $E(\langle \text{send} a'@H \; \bar{v} \rangle)$, and by $\sigma(\bar{v}) = \bar{v}'$, we know that the actor at $a$ in $K_1'$ has behavior expression $E(\langle \text{send} a''@H' \; \bar{v}' \rangle)$, where $\sigma(\bar{v}) = \bar{v}'$. Then $K_1'$ can take a step with this same rule, $K_1' \xrightarrow{\text{EO}} K_2'$, where $\bar{I}' = a : \text{send-int}(a'@H, \bar{v}')$.

Let $\sigma' = \emptyset$; clearly $\sigma$ and $\sigma'$ have disjoint domains and $\sigma \cup \sigma'$ is max-preserving. The labels $\bar{I}$ and $\bar{I}'$ have the same active actor, $\bar{I}$ is not a $\text{rcv-int}$ label, and $|\bar{I}| = |\bar{I}'| = \bullet$. Therefore, $\bar{I}'$ EO-simulates $\bar{I}$ via $\sigma \cup \sigma'$.

It remains to show that $K_2'$ is an externals-only variation of $K_2$ via $\sigma \cup \sigma'$. We have already shown that $\sigma \cup \sigma'$ is max-preserving. The argument for Matching Programs is similar to the one for the M-SendExternal case above, except that there are no new receptionists, and $\sigma \cup \sigma'$ provides a correspondence for the new in-flight message (i.e., $(\sigma \cup \sigma')(\langle a'@H, \bar{v} \rangle) = \langle a''@H, \bar{v}' \rangle$). Well-Typed holds by the Type Preservation lemma. The argument for the remaining properties is the same as the one for showing that $K_1'$ is an externals-only variation of $K_1$ via $\sigma$.

**Sub-Case 2**

In this case, $a''$ is an external address. Let $H_{\text{used}}$ be the used-marker component of $K_1$ and $K_1'$ (it must be the same for both, by Matching Programs). By Internal to External Mappings, $H \neq \emptyset$, so there exist $\bar{v}'$ and $H'_{\text{used}}$ such that $\bar{b} = E(\langle \text{send} a'@H \; \bar{v}' \rangle)$, $(\bar{v}', H'_{\text{used}}) \in \text{Markings}(\bar{v}', H_{\text{used}})$, and $H'_{\text{used}}$ is the used-marker component of $K_2$. By Matching Programs, there exist $E'$ and $\bar{v}''$ such that $\sigma(E) = E'$, $\sigma(\bar{v}) = \bar{v}''$, and $\bar{b}' = E(\langle \text{send} a''@H' \; \bar{v}'' \rangle)$. Because $\sigma(\bar{v}) = \bar{v}''$, $\bar{v}''$ can be marked using the exact same set of markers, so there exists $\bar{v}'''$ such that $(\bar{v}', H_{\text{used}}) \in \text{Markings}(\bar{v}'', H_{\text{used}})$. Therefore, $K_1'$ can simulate that transition with an external send by the M-SendExternal rule, $K_1' \xrightarrow{\text{EO}} K_2'$, where $\bar{I}' = a : \text{send-ext}(a''@H', \bar{v}'')$.

Let $\sigma'$ be the function mapping every marked address in $\bar{v}$ to the marked address at the corresponding position in $\bar{v}''$. Because each address in $\bar{v}$ received a fresh marker, the domains of $\sigma$ and $\sigma'$ are disjoint. Because those new markers on $\bar{v}$ and $\bar{v}''$ are the maximal markers on each address, $\sigma'$ is max-preserving, and therefore so is $\sigma \cup \sigma'$. The labels $\bar{I}$ and $\bar{I}'$ have the same active actor by construction. The label $\bar{I}$ is a $\text{rcv-int}$ label and $(\sigma \cup \sigma')(a'@H)$ is an external address.
B.11. EXTERNALS-ONLY CONFORMANCE THEOREM

Finally, $\bar{I} = a'@H'\tilde{v}$, $\bar{I} = a''@H''\tilde{v}'', H' = (\max(H))$ by Max Preservation and External Addresses, and $(\sigma \cup \sigma')(\tilde{v}) = \tilde{v}'''$. Therefore, $l'$ EO-simulates $l$ via $\sigma \cup \sigma'$.

It remains to show that $K'_2$ is an externals-only variation of $K_2$. We have already shown that $\sigma \cup \sigma'$ is max-preserving. The only changes to $K_2$ and $K'_2$ are a new in-flight message in $K_2$, possibly new receptionist in $K'_2$, and the send expression in the behavior of the actor at $a$ in both configurations is replaced by (variant Unit). All of these are allowed by Matching Programs, so that property holds. Well-Typed holds by the Type Preservation lemma.

$K''$ does not have any receptionists not in $K$, so Extra Receptionists holds.

The configuration $K_2$ has one new in-flight message $\langle a'*H, \tilde{v} \rangle$, such that $(\sigma \cup \sigma')(a'*H) = a''@H'$ and $(\sigma \cup \sigma')(\tilde{v}) = \tilde{v}'''$. We know that $a''$ is external, and by the definition of M-SENDEXTERNAL, every member of IntAddrTypes$(\tilde{v}''', \text{ActorType}(a''))$ is a receptionist on $K'_2$. Therefore, Extra Messages holds.

No new marked external addresses appear in $K'_2$, so External Addresses still holds. The only new internal marked addresses appearing in $K'_2$ are the new receptionists. By the definition of M-SENDEXTERNAL, the marked address for each new receptionist must be of the form $a''@\eta \cup H''$, where $a''$ is internal and $a''@H''$ appears in the behavior of the actor at $a$ in $K_1$. By Internal Addresses, $H'' = \phi$, so each new receptionist has exactly one marker and Internal Addresses holds.

Because each new mapping in $\sigma'$ is constructed from just adding the same marker to both marked addresses in some existing mapping from $\sigma$, the mapping properties all hold as in the M-SENDEXTERNAL case above.

B.11. EXTERNALS-ONLY CONFORMANCE THEOREM

**Theorem** (Externals-Only Conformance). For all $P$ and $\Sigma$ such that $\vdash_{\text{prog}} P$, $P \models_{\text{EO}} \Sigma$ if and only if $P \models \Sigma$.

**Proof.** Showing the $P \models \Sigma$ implies $P \models_{\text{EO}} \Sigma$ is trivial: every fair, non-stuck, externals-only execution is also a fair, non-stuck execution of \( \rightarrow \) steps, so the specification executions used to show $P \models \Sigma$ can be reused to show $P \models_{\text{EO}} \Sigma$.

For the other direction, let there be $P$ and $\Sigma$ such that $\vdash_{\text{prog}} P$ and $P \models_{\text{EO}} \Sigma$. By the definition of $\models_{\text{EO}}$, there exists some maximal instantiation $\langle K, S \rangle$ of $P$ and $\Sigma$ such that $K \models_{\text{EO}} S$. By the Maximal Instantiation theorem, it suffices to show that $K \models S$ in order to prove that $P \models \Sigma$.

Define a relation $R$ as the set of pairs of the form $\langle K', S' \rangle$ such that there exists some $K''$ and $\sigma$ such that

- $K''$ is an externals-only variation of $K'$ via $\sigma$,
- $K'$ is EO-compatible with $S$, and
- $K'' \models_{\text{EO}} S'$.


By the Instantiation Type Preservation lemma (see appendix J), $\vdash_{cfg} \tilde{K}$. Let $\sigma$ be the identity function over the set of marked addresses appearing in $\tilde{K}$. Every marked address in $\tilde{K}$ has at most one marker (by the definition of instantiation), so we have that $\tilde{K}$ is an externals-only variation of itself via $\sigma$. Instantiation also ensures that no receptionist is output-monitored, so we have that $\tilde{K}$ is EO-compatible with $S$. Therefore, $(\tilde{K}, S) \in R$. It remains to show that $R$ is conformance-dense, which would imply $\tilde{K} \vdash S$.

Let $(\tilde{K}_1, S_1)$ be a member of $R$, and let $\tilde{K}_1 \xrightarrow{l_1} \ldots$ be a fair, non-stuck execution. We must show that there exists a fair specification execution $S_1 \xrightarrow{l_1} \ldots$ with the same length such that $(\tilde{K}_i, S_i) \in R$ for all $\tilde{K}_i$ and $S_i$ in the respective executions. The idea is to

1. use the Externals-Only Simulation lemma to show that an externals-only variation $\tilde{K}_1'$ of $\tilde{K}_1$ can simulate that execution,

2. use the definition of $\models_{EO}$ to get some fair specification execution $S_1 \xrightarrow{l_1} \ldots$ that can simulate the execution $\tilde{K}_1' \xrightarrow{l_1}_{EO} \ldots$, and

3. use the Externals-Only Specification Simulation lemma to adapt that specification execution to simulate $\tilde{K}_1 \xrightarrow{l_1} \ldots$.

By the definition of $R$, there exists some $\tilde{K}_1'$ that is an externals-only variation of $\tilde{K}_1$ via some $\sigma_1$. By repeated uses of the Externals-Only Simulation lemma, there exists an execution $\tilde{K}_1' \xrightarrow{l_1}_{EO} \ldots$ with the same length as $\tilde{K}_1 \xrightarrow{l_1} \ldots$ and a corresponding sequence of substitutions $\sigma_2, \ldots$ such that for all labels $l_i$ and $l_i'$ and configurations $\tilde{K}_i$ and $\tilde{K}_i'$ in each respective execution, $l_i'$ EO-simulates $l_i$ via $\sigma_1 \cup \ldots \cup \sigma_i$ and $\tilde{K}_i'$ is an externals-only variation of $\tilde{K}_i$ via $\sigma_1 \cup \ldots \cup \sigma_i$.

Because each $\tilde{K}_i'$ is an externals-only variation of $\tilde{K}_i$, and because externals-only variation requires that the actor behaviors differ only by a substitution $\sigma$, an actor in $\tilde{K}_i'$ is stuck only if the corresponding actor in $\tilde{K}_i$ is. Therefore, because $\tilde{K}_1 \xrightarrow{l_1} \ldots$ is non-stuck, $\tilde{K}_1' \xrightarrow{l_1}_{EO} \ldots$ is non-stuck as well.

We also must show that $\tilde{K}_1' \xrightarrow{l_1}_{EO} \ldots$ is fair. First, let $\tilde{K}_i'$ be a configuration in that execution, and let $a$ identify some actor with an enabled non-rcv-ext step from $\tilde{K}_i'$. By the definition of an externals-only variation, there exists an actor in $\tilde{K}_i$ with address $a$ and an enabled non-rcv-ext step from $\tilde{K}_i$. Because $\tilde{K}_1 \xrightarrow{l_1} \ldots$ is fair, there exists some step $\tilde{K}_{i+j} \xrightarrow{l_{i+j}} \tilde{K}_{i+j+1}$ in that execution where $a$ identifies the active actor for $\tilde{l}_{i+j}$. Then for the step $\tilde{K}_{i+j} \xrightarrow{l_{i+j}} \tilde{K}_{i+j+1}$ in the other execution, $a$ identifies the active actor for $\tilde{l}_{i+j}$ because $\tilde{l}_{i+j}'$ EO-simulates $\tilde{l}_{i+j}$ via $\sigma_1 \cup \ldots \cup \sigma_{i+j}$.
Second, let $\tilde{K}'_i$ be a configuration in $\tilde{K}'_i \xrightarrow{\tilde{l}_i} \ldots$, and let $\langle a@H, \tilde{v} \rangle$ be an inflight message in $\tilde{K}'_i$. Let $\sigma' = \sigma_1 \cup \ldots \cup \sigma_i$. By the definition of an externals-only variation, there exist some $a'@H'$ and $\tilde{v}'$ such that $\sigma'(a'@H') = a@H$, $\sigma'(\tilde{v}') = \tilde{v}$, and $\langle a'@H', \tilde{v}' \rangle$ is an inflight message in $\tilde{K}_i$. Because $\tilde{K}_1 \xrightarrow{\tilde{l}_1} \ldots$ is fair, there exists some step $\tilde{K}_{i+j} \xrightarrow{a'.rcv-int(H', \tilde{v}')} \tilde{K}_{i+j+1}$ in that execution. Then because $\tilde{l}_{i+j} \xrightarrow{\sigma' \cup \sigma_{i+j+1}} \tilde{l}_{i+j+1}$, $\tilde{l}_{i+j} = a : rcv-int(H, \tilde{v})$.

Then because $\tilde{K}'_1 \models_{EO} S_1$ and $\tilde{K}'_1 \xrightarrow{\tilde{l}'_1} \ldots$ is fair, there exists a fair specification execution $S_1 \xrightarrow{[l'_1], O_1} S_2 \xrightarrow{[l'_2], O_2} \ldots$ such that $\tilde{K}'_1 \models_{EO} S_i$ for all $\tilde{K}'_i$ and $S_i$ of the respective executions. Then by repeated uses of the Externals-Only Specification Simulation lemma, there also exists a specification execution $S_1 \xrightarrow{[l'_1], O_1} S_2 \xrightarrow{[l'_2], O_2} \ldots$ that uses the labels from the standard program execution, and for which each $\tilde{K}_i$ is EO-compatible with $S_i$. Because it contains the same specification configurations and fulfills the same obligations, that latter execution is also fair. Furthermore, we have that each corresponding pair $\langle \tilde{K}_i, S_i \rangle$ is in $R$. Therefore, $R$ is conformance-dense, which completes the proof. $\square$
Appendix C

External-Representative Conformance

Even with the externals-only restriction in effect, the environment can send a program unboundedly many variations on the same message just by changing the contained addresses. For example, if a program expects to receive Ping messages that include the response address for a Pong, then the environment could send (variant Ping a₁), (variant Ping a₂), (variant Ping a₃), or number of similar variations. This may not be an obstacle for a human prover who can quantify over all possible addresses, but it poses a problem to an automated model checker that tests its given program against all receivable messages.

However, as mentioned in the previous section, a CSA actor has no way of distinguishing one address from another: all it can do with an address is send a message to it. Therefore, as long as that program sends the appropriate response in response to the message (variant Ping a₁), then we should be able to assume that it would similarly respond appropriately to (variant Ping a₂) and (variant Ping a₃). This section introduces a notion of conformance, called external-representative conformance, that allows conformance proofs to make these kinds of assumptions.

The main idea is to formalize this notion of similarity in terms of equivalence classes of addresses, then consider only the received messages in which each contained address is the representative for the class of addresses a well-typed message may contain at that point. The representatives are used specifically for the received external addresses; hence the name external-representative conformance.

There is one catch to selecting the representatives. Say some program can receive an address that itself expects addresses to which A or B messages may be sent (i.e., the program expects an address of type (Addr (Addr (Variant [A] [B])))). By the subtyping rules in section 2.4.2, an address a with the type (Addr (Addr (Variant [A])) is a valid message to send that program. However, the type of a says that if the program sends to a
\( \text{ExtRepMsgs}(\tau) = \)

Case: \( \tau = (\text{Addr} \ \tau') \)
\[ \{a'@H\} \text{ where ExtRepAddr}(\tau') = a' \]

Case: \( \tau = (\text{Record} \ [r_1 \ \tau'_1] \ldots [r_n \ \tau'_n]) \)
\[ \{(\text{record} \ [r_1 \ \bar{v}_1] \ldots [r_n \ \bar{v}_n]) \mid \bar{v}_i \in \text{ExtRepMsgs}(\tau') \text{ for all } i \in 1\ldots n\} \]

Case: \( \tau = (\text{Variant} \ [t_1 \ \tau'_{1,1} \ldots \tau'_{1,m}] \ldots [t_n \ \tau'_{n,1} \ldots \tau'_{n,m}]) \)
\[ \bigcup_{i \in 1\ldots n} \{(\text{variant} \ t_i \ \bar{v}_1 \ldots \bar{v}_m) \mid \bar{v}_j \in \text{ExtRepMsgs}(\tau'_{i,j}) \text{ for } j \in 1\ldots m\} \]

Case: \( \tau = (\text{rec} \ X \ \tau') \)
\[ \{(\text{fold} \ \tau \ \bar{v}) \mid \bar{v} \in \text{ExtRepMsgs}(\tau'[X \leftarrow \tau])\} \]

Case: \( \tau = (\text{List} \ \tau') \)
\[ \{(\text{list} \ \bar{v}_1 \ldots \bar{v}_n) \mid \bar{v}_i \in \text{ExtRepMsgs}(\tau') \text{ for all } i \in 1\ldots n\} \]

Case: \( \tau = (\text{Dict} \ \tau' \ \tau'') \)
\[ \left\{(\text{dict} \ [\bar{v}_1 \ \bar{v}'_1] \ldots [\bar{v}_n \ \bar{v}'_n]) \mid \bar{v}'_i \in \text{ExtRepMsgs}(\tau') \text{ and } \bar{v}'_i \in \text{ExtRepMsgs}(\tau'') \text{ for all } i \in 1\ldots n\right\} \]

Case: \( \tau = \text{Nat} \)
\[ \{n \mid n \in \mathbb{N}\} \]

Case: \( \tau = \text{String} \)
\[ \{\text{str} \mid \text{str} \in \text{String}\} \]

Figure C.1: The representative of a message \( \bar{v} \) with type \( \tau \)

The function \( \text{ExtRepMsgs} \), given in figure C.1, defines the set of messages of a given type \( \tau \) that contain only representative addresses. The definition assumes the existence of a function \( \text{ExtRepAddr} \) that, for each type \( \tau \), defines the representative address \( a = \text{ExtRepAddr}(\tau) \) such that \( \text{ActorType}(a) = \tau \) and \( a \) is external. The addresses can be chosen arbitrarily as long as they obey these conditions, so the exact definition of \( \text{ExtRepAddr} \) is irrelevant and therefore omitted. \( \text{ExtRepMsgs} \) defines the representative of an address as its \( \text{ExtRepAddr} \) representative; the representatives for all other cases are the obvious component-wise definitions.

The function \( \text{ExtRepMsgs} \) entails a new transition relation \( \rightarrow_{\text{ER}} \) that is like the externals-only transition relation \( \rightarrow_{\text{EO}} \), but accepts only representative messages from the environment.

**Definition.** The relation \( \rightarrow_{\text{ER}} \) is defined such that for all \( \bar{K} = \langle \langle \bar{\bar{v}} \mid \bar{\mu} \mid H \rangle \rangle^{\bar{\bar{v}}} \) and \( \bar{K}' = \langle \langle \bar{\bar{v}}' \mid \bar{\mu}' \mid H' \rangle \rangle^{\bar{\bar{v}}'} \), \( \bar{K} \rightarrow_{\text{ER}} \bar{K}' \) if and only if
C.1. DEFINITIONS

As with externals-only conformance, the only difference in the definition of external-representative conformance is the use of this new transition relation. The formal definition follows.

**Definition.** A relation $R$ is external-representative-conformance-dense if and only if for all $\langle ˘\mathbf{K}_1, S_1 \rangle \in R$ and all fair, non-stuck executions $\vec{K}_1 \xrightarrow{I_1}^{\text{ER}} \vec{K}_2 \xrightarrow{I_2}^{\text{ER}} \ldots$ there exists a fair specification execution $S_1 \xrightarrow{[I_1]} S_2 \xrightarrow{[I_2]} \ldots$ with the same length such that $\langle ˘\mathbf{K}_i, S_i \rangle \in R$ for all $\vec{K}_i$ and $S_i$ in the respective executions.

**Definition.** A marked program configuration $\vec{K}$ external-representative conforms to a specification configuration $S$, written $\vec{K} \models^{\text{ER}} S$, if there exists an external-representative-conformance-dense relation $R$ such that $\langle ˘\mathbf{K}, S \rangle \in R$. As with previous notions of conformance, $P \models^{\text{ER}} \Sigma$ if there exists some maximal instantiation $\langle ˘\mathbf{K}, S \rangle$ of $P$ and $\Sigma$ such that $\vec{K} \models^{\text{ER}} S$.

The remainder of this appendix provides the definitions, lemmas, and theorems needed to show that external-representative conformance is equivalent to externals-only conformance.

### C.1 Definitions

As in the proof of the Externals-Only Conformance theorem in the previous appendix, the proof relies on showing a correspondence between the marked addresses in two program configurations. As in that previous appendix, let $\sigma$ stand for a partial function from marked addresses to marked addresses. The application $\sigma(˘\mathbf{K})$ of the substitution $\sigma$ to an arbitrary term in a program configuration is defined as the obvious element-wise, component-wise, point-wise application to its parts, and the application is undefined if $\sigma(a@H)$ is undefined for any marked address $a@H$ in that term.

A substitution $\sigma$ is an ER-substitution if and only if for all $a, a', H, \text{ and } H'$, if $\sigma(a@H) = a'@H'$ then

- if $a$ is internal then $a = a'$,
- if $a$ is external then $a'$ is external and $\text{ActorType}(a') < \text{ActorType}(a)$, and
- $H = H'$.

A program configuration $\vec{K}'$ is an ER-variation of $\vec{K}$ via $\sigma$ if and only if there exist $\vec{K}', \vec{\beta}', \vec{\beta}, \vec{\rho}', H, \vec{\rho},$ and $\vec{\rho}'$ such that the following conditions hold.

- $\sigma$ is an ER-substitution.
The argument for the \( \gamma \)
Proof.

\[ K = \left[ \left. \beta \right| \mu \right] H \]
and
\[ K' = \left[ \left. \beta' \right| \mu' \right] H \].

For all \( a@H' \in \text{dom}(\sigma) \), either \( a \in \text{dom}(\beta) \) or \( H' \subseteq H \).

\[ \sigma(\beta) = \beta' \text{ and } \sigma(\mu) = \mu'. \]

For all \( \langle a@H', \tau \rangle \in \rho \), there exists \( \tau' \) such that \( \langle a@H', \tau' \rangle \in \beta' \) and \( \tau <: \tau' \).

A transition label \( \tilde{I} \) \( \text{ER-simulates} \) \( I \) via \( \sigma \) if and only if the following conditions hold.

\[ \sigma \text{ is an ER-substitution}. \]
\[ \tilde{I} \text{ and } I' \text{ have the same active actor}. \]
\[ \text{If there exist } a, H \text{ and } \tilde{v} \text{ such that } \tilde{I} = a : \text{rcv-int}(H, \tilde{v}), \text{ then there exists } \tilde{v}' \text{ such that } \sigma(\tilde{v}) = \tilde{v}' \text{ and } \tilde{I}' = a : \text{rcv-int}(H, \tilde{v}'). \]
\[ \sigma([I]) = [I']. \]

C.2 External-Representative Message Lemma

Lemma (External-Representative Message). For all \( \tau \) and \( \tilde{v} \) such that \( \varphi, \varphi \vdash \tilde{v} : \tau \), \( \tilde{v} \) contains no internal addresses, and every marked address in \( \tilde{v} \) is distinct, there exist \( \tilde{v}' \) and \( \sigma \) such that \( \tilde{v}' \in \text{ExtRepMsgs}(\tau) \), \( \sigma \) is an ER-substitution, and \( \sigma(\tilde{v}) = \tilde{v}' \).

Proof. The proof is by structural induction on \( \tilde{v} \). Most cases are straightforward; the argument for the \( \tilde{v} = a@H \) case is given here.

By the type rules, there exists some \( \tau' \) such that \( \tau = (\text{Addr } \tau') \) and \( \tau' <: \text{ActorType}(a) \). Let \( a' = \text{ExtRepAddr}(\tau') \), let \( \tilde{v}' = a'@H \), and let \( \sigma = [a@H \mapsto a'@H] \). By the definition of ExtRepAddr, \( \text{ActorType}(a') = \tau' \), so \( \text{ActorType}(a') <: \text{ActorType}(a) \). Then \( \tilde{v}' \in \text{ExtRepMsgs}(\tau) \), \( \sigma \) is an ER-substitution, and \( \sigma(\tilde{v}) = \tilde{v}' \).

C.3 ER Receptionist Lemma

Lemma (External-Representative Receptionist). For all \( \tilde{v}, \tilde{v}', \tau, \tau' \), and \( \sigma \), if \( \sigma \) is an ER-substitution, \( \sigma(\tilde{v}) = \tilde{v}' \), \( \tau' <: \tau \), and \( \text{IntAddrTypes}(\tilde{v}, \tau) \) and \( \text{IntAddrTypes}(\tilde{v}', \tau') \) are defined, then for all \( \langle a@H, \tau'' \rangle \in \text{IntAddrTypes}(\tilde{v}, \tau) \), there exists \( \tau''' \) such that \( \langle a@H, \tau''' \rangle \in \text{IntAddrTypes}(\tilde{v}', \tau') \) and \( \tau'' <: \tau''' \).

Proof. The proof is by structural induction on \( \tilde{v} \). Most cases are straightforward, the case for \( \tilde{v} = a@H \) is given here.

When \( \tilde{v} = a@H \), there must exist some \( \tau'' \) such that \( \tau = (\text{Addr } \tau'') \). If \( a \) is external, then because \( \sigma \) is an ER-substitution, \( \tilde{v}' \) is also a marked external address, and therefore \( \text{IntAddrTypes}(\tilde{v}, \tau) = \text{IntAddrTypes}(\tilde{v}', \tau') = \varnothing \), in which case we're done.
Otherwise, $a$ is internal and therefore $\text{IntAddrTypes}(\vec{v}, \tau) = \{\langle a@H, \tau'' \rangle \}$. Because $\sigma$ is an ER-substitution, $\vec{v}' = \sigma(\vec{v}) = a@H$. There also exists $\tau'''$ such that $\vec{v}' = (\text{Addr} \ \tau''')$, and therefore $\text{IntAddrTypes}(\vec{v}', \tau') = \{\langle a@H, \tau''' \rangle \}$. Because $(\text{Addr} \ \tau''') <: (\text{Addr} \ \tau'')$, by the subtyping rules we have $\tau'' <: \tau'''$, which completes the proof. 

\section{C.4 External-Representative Simulation Lemma}

\textbf{Lemma} (External-Representative Simulation). For all $\vec{K}_1$, $\vec{K}_2$, $\vec{K}_1'$, $\sigma$, and $l$, if $\vec{K}_1'$ is an ER-variation of $\vec{K}_1$ via $\sigma$ and $\vec{K}_1 \xrightarrow{l} \vec{K}_2$, then there exist $\vec{l}'$, $\vec{K}_2''$, and $\sigma'$ such that
\begin{itemize}
  \item $\vec{K}_1' \xrightarrow{\vec{l}'} \vec{K}_2''$
  \item $\vec{l}'$ ER-simulates $\vec{l}$ via $\sigma \circ \sigma'$, and
  \item $\vec{K}_2''$ is an ER-variation of $\vec{K}_2$ via $\sigma \circ \sigma'$.
\end{itemize}

\textbf{Proof}. The proof is by case analysis on the transition rule that enables the step $\vec{K}_1 \xrightarrow{l} \vec{K}_2$. The argument for most cases is straightforward; the arguments for the $\text{M-ReceiveExternal}$ and $\text{M-ReceiveInternal}$ cases are below.

\textbf{Case: M-ReceiveExternal}

In this case, there exist $a$, $H$, $\vec{v}$, and $\tau$ such that $\vec{l} = a : \text{rcv-ext}(H, \vec{v}, \tau)$, $a$ is internal, and $\emptyset \vdash \vec{v} : \tau$. Because this is an externals-only transition, $\vec{v}$ contains no internal addresses, and the marked addresses in $\vec{v}$ must be distinct because each one received a fresh marker in this transition. Then by the External-Representative Message lemma, there exist $\vec{v}'$ and $\sigma'$ such that $\vec{v}' \in \text{ExtRepMsgs}(\tau)$, $\sigma'$ is an ER-substitution, and $\sigma'(\vec{v}) = \vec{v}'$. Because $\sigma$ is an ER-substitution, we know that $\sigma(a@H) = a@H$. Then because the actor-behavior maps for $\vec{K}_1$ and $\vec{K}_1'$ differ only by the substitution $\sigma$, the configuration $\vec{K}_1'$ can take a similar transition $\vec{K}_1' \xrightarrow{\vec{l}'} \vec{K}_2''$, where $\vec{l}' = a : \text{rcv-ext}(H, \vec{v}', \tau)$.

Because all of the markers on $\vec{v}$ are fresh and $\sigma'$ contains mappings only for the marked addresses in $\vec{v}$, $\sigma$ and $\sigma'$ have disjoint domains, so $\sigma \circ \sigma'$ is defined and is an ER-substitution. The labels $\vec{l}$ and $\vec{l}'$ have the same active actor and they are not $\text{rcv-int}$ labels. We also have that $(\sigma \circ \sigma' )(a@H) = a@H$, $(\sigma \circ \sigma')(\vec{v}) = \vec{v}'$, $[\vec{l}] = a@H?\vec{v}$, and $[\vec{l}'] = a@H?\vec{v}'$. Therefore, $\vec{l}'$ ER-simulates $\vec{l}$ via $\sigma \circ \sigma'$.

Finally, we must show that $\vec{K}_2''$ is an ER-variation of $\vec{K}_2$ via $\sigma \circ \sigma'$. We already know that $\sigma \circ \sigma'$ is an ER-substitution. The only changes to $\vec{K}_2$ and $\vec{K}_2''$ are the behaviors for the receiving actor, which must correspond through $\sigma \circ \sigma'$, and the used-marker components, which must be the same since the same markers were used to mark $\vec{v}$ and $\vec{v}'$; therefore $\vec{K}_2$ and $\vec{K}_2''$ correspond as expected. Every mapping in $\sigma'$ corresponds to some freshly marked address in $\vec{v}$, so the markers on those addresses are members of the used-marker component on $\vec{K}_2$. Finally, this
transition creates no new receptionists, so the property on the receptionists holds for the same reason it does for $K_1$ and $K_1'$.

**Case: M-SendExternal**

In this case, there exist $a$, $a'$, $H$, and $\bar{v}$ such that $l = a : \text{send-ext}(a'@H, \bar{v})$ and $a'$ is external. Also, there exist $H_{\text{used}}$, $H'_{\text{used}}$, $\bar{E}$, and $\bar{v}'$ such that the behavior of the actor at $a$ in $K_1$ is $\bar{E}(\text{send } a'@H \bar{v}')$, $H_{\text{used}}$ and $H'_{\text{used}}$ are the used-marker components of $K_1$ and $K_2$, respectively, and $\langle \bar{v}, H'_{\text{used}} \rangle \in \text{Markings}(\bar{v}', H_{\text{used}})$.

Because the behavior of the actor at $a$ in $K_1'$ differs from that in $K_1$ only by the substitution $\sigma$ (by the definition of an ER-variation), there exist $\bar{E}'$, $a''$, and $\bar{v}''$ such that the behavior of the actor at $a$ in $K_1'$ is $\bar{E}'(\text{send } a''@H \bar{v}'')$, $\sigma(a'@H) = mlita''H$ (because $\sigma$ is an ER-substitution), and $\sigma(\bar{v}') = \bar{v}''$. Then $\bar{v}''$ can be marked in the same way as $\bar{v}'$ to produce $\bar{v}'''$ (i.e., $\langle \bar{v}'', H'_{\text{used}} \rangle \in \text{Markings}(\bar{v}'', H_{\text{used}})$). Therefore, the configuration $K_1'$ can take a similar step $K_1' \xrightarrow{\sigma} K_2'$, where $l = a' : \text{send-ext}(a''@H, \bar{v}''')$.

Let $\sigma'$ be the function that maps every marked address in $\bar{v}$ to the corresponding marked address in $\bar{v}'''$. Because every address in $\bar{v}$ has a fresh marker, the domain of $\sigma'$ is $\bar{v}$ and $\sigma'$ are disjoint. Because every mapping in $\sigma'$ is just adding a fresh marker to both sides of a mapping from $\sigma$, $\sigma'$ is an ER-substitution and so is $\sigma \cup \sigma'$. The labels $l$ and $l'$ have the same active actor and are not rcv-int labels.

We also have that $(\sigma \cup \sigma')(a'@H) = a''@H$ (because $\sigma$ is an ER-substitution), and $(\bar{v}) = a'@H \bar{v}$, and $\langle l \rangle = a''@H \bar{v}'$. Therefore, $l'$ ER-simulates $l$ via $\sigma \cup \sigma'$.

Finally, we must show that $K_1'$ is an ER-variation of $K_2$ via $\sigma \cup \sigma'$. We already know that $\sigma \cup \sigma'$ is an ER-substitution. The only changes to $K_2$ and $K_2'$ are that the evaluated send expressions in each configuration are both replaced with (variant Unit) (which holds the $(\sigma \cup \sigma')$-correspondence for the actor-behavior maps for those configurations, and each configuration might have new receptionists.

The new receptionists for $K_2$ and $K_2'$ are defined by $\text{IntAddrTypes}(\bar{v}, \text{ActorType}(a'))$ and $\text{IntAddrTypes}(\bar{v}'', \text{ActorType}(a''))$, respectively. Because $\sigma$ is an ER-substitution, we know that $\text{ActorType}(a'') <: \text{ActorType}(a')$, so by the External-Representative Receptionist lemma, for all $\langle a'''@H', \tau \rangle \in \text{IntAddrTypes}(\bar{v}, \text{ActorType}(a'))$, there exists $\tau'$ such that $\langle a'''@H', \tau' \rangle \in \text{IntAddrTypes}(\bar{v}'', \text{ActorType}(a''))$ and $\tau <: \tau'$.

Finally, as in the previous case, every mapping in $\sigma'$ corresponds to some freshly marked address in $\bar{v}$, so the markers on those addresses are members of the used-marker component on $K_2$.

\[\square\]

### C.5 ER Specification Simulation Lemma

**Lemma** (External-Representative Specification Simulation). For all $S$, $S'$, $l$, $l'$, $O$, and $\sigma$, if $S \xrightarrow{\langle l \rangle, O} S'$ and $l'$ ER-simulates $l$ via $\sigma$, then $S \xrightarrow{\langle l' \rangle, O} S'$.
C.6. EXTERNAL-REPRESENTATIVE CONFORMANCE THEOREM

**Proof.** The proof is straightforward. By the definition of ER-simulation, \( \sigma([\bar{l}]) = [\bar{l}'] \) and \( \sigma \) is an ER-substitution. Therefore, the only difference between \([\bar{l}]\) and \([\bar{l}']\) is that for every marked address \( a@H \) appearing in \([\bar{l}]\), there exists some \( a' \) such that \( a'@H \) appears in the same relative position of \([\bar{l}']\), and \( a \) and \( a' \) are either both internal or both external. Any other differences between \( a \) and \( a' \) are irrelevant to the transitions of a specification configuration, so we have that \( S \xrightarrow{[\bar{l}]_{\sigma}} S' \).

\[ \square \]

C.6 External-Representative Conformance Theorem

**Theorem (External-Representative Conformance).** For all \( P \) and \( \Sigma \), \( P \models_{\text{ER}} \Sigma \) if and only if \( P \models_{\text{EO}} \Sigma \).

**Proof.** Analogously to the Externals-Only Conformance theorem, it is trivial to show that \( P \models_{\text{EO}} \Sigma \) implies \( P \models_{\text{ER}} \Sigma \). Every fair, non-stuck external-representative execution is also a fair, non-stuck externals-only execution, so the specification executions used to show \( P \models_{\text{EO}} \Sigma \) can be reused to show \( P \models_{\text{ER}} \Sigma \).

For the other direction, let there be \( P \) and \( \Sigma \) such that \( P \models_{\text{ER}} \Sigma \). By the definition of \( \models_{\text{ER}} \), there exists some maximal instantiation \( \langle \bar{K}_{\text{init}}, S_{\text{init}} \rangle \) of \( P \) and \( \Sigma \) such that \( \bar{K}_{\text{init}} \models_{\text{ER}} S_{\text{init}} \). It remains to show that \( \bar{K}_{\text{init}} \models_{\text{EO}} S_{\text{init}} \).

By the definition of \( \models_{\text{ER}} \), there exists some external-representative-conformance-dense relation \( R_{\text{ER}} \) such that \( \langle \bar{K}_{\text{init}}, S_{\text{init}} \rangle \in R_{\text{ER}} \). Define a relation \( R_{\text{EO}} \) such that \( \langle \bar{K}, S \rangle \in R_{\text{EO}} \) if and only if there exist \( \bar{K}' \) and \( \sigma \) such that \( \bar{K}' \) is an ER-variation of \( \bar{K} \) via \( \sigma \) and \( \langle \bar{K}', S \rangle \in R_{\text{ER}} \).

Let \( \sigma_{\text{init}} \) be the identity function restricted to the marked addresses appearing in \( \bar{K}_{\text{init}} \). It is easy to see that \( \sigma_{\text{init}} \) is an ER-substitution and \( \bar{K}_{\text{init}} \) is an ER-variation of itself via \( \sigma_{\text{init}} \). Therefore \( \langle \bar{K}_{\text{init}}, S_{\text{init}} \rangle \in R_{\text{EO}} \). It remains to show that \( R_{\text{EO}} \) is externals-only-conformance-dense.

Let \( \langle \bar{K}_1, S_1 \rangle \) be a member of \( R_{\text{EO}} \), and let \( \bar{K}_1 \xrightarrow{[\bar{l}]} \bar{K}_2 \xrightarrow{[\bar{l}]} \ldots \) be a fair, non-stuck specification execution with the same length such that \( \langle \bar{K}_i, S_i \rangle \in R_{\text{EO}} \) for all \( \bar{K}_i \) and \( S_i \) in the respective executions. The strategy is similar to the one for externals-only conformance in the previous appendix: we will show that a fair, non-stuck external-representative simulation of this execution is possible, retrieve a simulating specification execution from the proof that \( R_{\text{ER}} \) is external-representative-conformance-dense, and adapt that specification execution to simulate the \( \rightarrow_{\text{EO}} \) execution.

By the definition of \( R_{\text{EO}} \), there exist \( \bar{K}'_1 \) and \( \sigma_1 \) such that \( \bar{K}'_1 \) is an ER-variation of \( \bar{K}_1 \) via \( \sigma_1 \). By repeated uses of the External-Representative Simulation lemma, there exists an execution \( \bar{K}'_1 \xrightarrow{[\bar{l}]} \bar{K}'_2 \xrightarrow{[\bar{l}]} \ldots \) of the same length as \( \bar{K}_1 \xrightarrow{[\bar{l}]} \bar{K}_2 \xrightarrow{[\bar{l}]} \ldots \) and substitutions \( \sigma_2, \sigma_3, \ldots \) such that
APPENDIX C. EXTERNAL-REPRESENTATIVE CONFORMANCE

• $\tilde{I}_i'$ ER-simulates $\tilde{I}_i$ via $\sigma_1 \uplus \ldots \uplus \sigma_{i+1}$ for all corresponding labels $\tilde{I}_i$ and $\tilde{I}_i'$ in the two executions, and

• $\tilde{K}_i'$ is an ER-variation of $\tilde{K}_i$ via $\sigma_1 \uplus \ldots \uplus \sigma_i$ for all corresponding configurations $\tilde{K}_i$ and $\tilde{K}_i'$ in the two executions.

We know that each $\tilde{K}_i$ in the first execution is non-stuck. Because the corresponding $\tilde{K}_i'$ differs only by some ER-substitution, $\tilde{K}_i'$ is also non-stuck, and therefore the execution $\tilde{K}_1' \xrightarrow{\tilde{I}_1'} \text{ER} \ldots$ is non-stuck.

To show that $\tilde{K}_1' \xrightarrow{\tilde{I}_1'} \text{ER} \ldots$ is fair, first let $\tilde{K}_i'$ be a configuration in the execution, and let $a$ identify some actor with an enabled non-rcv-ext step from $\tilde{K}_i'$. There must be an actor with address $a$ in $\tilde{K}_i$ with an enabled non-rcv-ext step, and because $\tilde{K}_1 \xrightarrow{\tilde{I}_1} \text{EO} \ldots$ is fair, there must exist some future step $\tilde{K}_{i+j} \xrightarrow{\tilde{I}_{i+j}} \text{EO} \tilde{K}_{i+j+1}$ in the execution with $a$ identifying the active actor for $\tilde{I}_{i+j}$. Then by the definition of ER-simulation, $a$ also identifies the active actor for $\tilde{I}_{i+j}'$.

Second, let $\tilde{K}_i'$ be a configuration in $\tilde{K}_1' \xrightarrow{\tilde{I}_1'} \text{ER} \ldots$, and let $\langle a@H, \tilde{v} \rangle$ be an in-flight message in $\tilde{K}_i'$. By the definition of an ER-variation, there exists some $\tilde{v}'$ such that $\langle a@H, \tilde{v}' \rangle$ is an in-flight message in the corresponding configuration $\tilde{K}_i$. Because $\tilde{K}_1 \xrightarrow{\tilde{I}_1} \ldots$ is fair, there exists some step $\tilde{K}_{i+j} \xrightarrow{a : \text{rcv-int}(H, \tilde{v}')} \tilde{K}_{i+j+1}$ in that execution. Then by the definition of ER-simulation, $\tilde{I}_{i+j}' = a : \text{rcv-int}(H, \tilde{v})$.

Therefore $\tilde{K}_1' \xrightarrow{\tilde{I}_1'} \text{ER} \ldots$ is fair.

By the definition of $R_{\text{EO}}$, $\langle \tilde{K}_i', S_i \rangle \in R_{\text{ER}}$. Then because $R_{\text{ER}}$ is external-representative-conformance-dense, there exists some fair specification execution $S_1 \xrightarrow{[\tilde{I}_i], O_1} \ldots$ such that $\langle \tilde{K}_i', S_i \rangle \in R_{\text{EO}}$ for all $\tilde{K}_i'$ and $S_i$ in the respective executions. Then by repeated uses of the External-Representative Specification Simulation lemma, there also exists a specification execution $S_1 \xrightarrow{[\tilde{I}_i], O_1} \ldots$ of the same length. Because it contains the same specification configurations and fulfills the same obligations, that latter execution is also fair. Furthermore, we have that each corresponding pair $\langle \tilde{K}_i, S_i \rangle$ is in $R_{\text{EO}}$. Therefore, $R_{\text{EO}}$ is externals-only-conformance-dense, which completes the proof. \qed
Appendix D

Single-Handler Conformance

CSA actors non-deterministically interleave their transition steps with each other, leading to many different possible executions. Once an actor starts handling an event, however, no other actor can change its execution until it finishes its current handler. Therefore many of these executions are merely different interleavings of actors handling the same sequence of events, and are thus equivalent with respect to conformance.

This next technique takes advantage of these equivalences so that conformance proofs can avoid reasoning about most of these interleavings. The idea is to consider only those executions in which all handler expressions terminate before another actor starts handling a new event. The resulting semantics executes only one actor at a time (plus the initialization for any actors it spawns while handling an event), so the resulting notion of conformance is called single-handler conformance.

To ensure that the initialization of a newly spawned actor cannot arbitrarily interleave with the execution of the actor that spawned it, the single-handler imposes a prioritization among the running actors. Specifically, whenever an actor spawns a child, the child’s handler expression runs to completion before control returns to the spawning actor. An ordering $\prec$ on the syntactic locations from which actors are spawned encodes this prioritization.

Definition. The ordering $\prec$ is defined such that $\ell < \ell'$ if and only if

- the spawn expression referred to by $\ell'$ is lexically nested inside the spawn expression referred to by $\ell$, or

- neither spawn expression is nested inside the other, and the expression for $\ell'$ appears earlier than the one for $\ell$ in the original program.

Intuitively, if $\ell < \ell'$, then an actor spawned at $\ell'$ takes priority over an actor spawned at $\ell$ in the single-handler semantics (greater locations get higher priority).
APPENDIX D. SINGLE-HANDLER CONFORMANCE

This ordering extends to addresses, as well. An actor (identified by address \((\text{addr } \ell \ n)\)) takes priority over another if either its spawn location \(\ell\) has a higher priority, or it shares a spawn location with the other actor but has a higher identifier \(n\). Formally, \((\text{addr } \ell \ n) < (\text{addr } \ell' \ n')\) if and only if \(\ell < \ell'\) or \(\ell = \ell'\) and \(n < n'\).

A new relation, the single-handler transition relation \(\rightarrow_{\text{SH}}\), uses this ordering to enforce the prioritized execution of actors. Specifically, if any actors in a configuration are handling an event (i.e., a timeout or a received message), only the highest-priority such actor can take a step, and an actor can start handling an event only if no other actors are in the midst of handling their own event.

**Definition.** The relation \(\rightarrow_{\text{SH}}\) is defined such that \(\tilde{K} \overset{l}{\rightarrow}_{\text{SH}} \tilde{K}'\) if and only if

- \(\tilde{K} \overset{l}{\rightarrow}_{\text{ER}} \tilde{K}'\),
- if \(l\) is a handler-continuation label with active actor identified by \(a\), then there is no \(a'\) such that \(a < a'\) and the actor at \(a'\) in \(\tilde{K}\) is handling an event, and
- if \(l\) is a handler-start label, then no actor in \(\tilde{K}\) is handling an event.

Recall from section 2.5.4 that a handler-start label is a transition label indicating an actor starting to execute some event handler (by either receiving a message or handling a timeout), and a handler-continuation label is any other kind of label (representing the further execution of an event handler).

As with externals-only and external-representative conformance, single-handler conformance is identical to standard conformance with the exception of the program transition relation used.

**Definition.** A relation \(R\) is single-handler-conformance-dense if for all \(\langle \tilde{K}_1, S_1 \rangle \in R\) and all fair, non-stuck single-handler executions \(\tilde{K}_1 \overset{l_1}{\rightarrow}_{\text{SH}} \tilde{K}_2 \overset{l_2}{\rightarrow}_{\text{SH}} \cdots\), there exists a fair specification execution \(S_1 \overset{l_1}{\rightarrow} S_2 \overset{l_2}{\rightarrow} \cdots\) with the same length such that \(\langle \tilde{K}_i, S_i \rangle \in R\) for all \(\tilde{K}_i\) and \(S_i\) in the respective executions.

**Definition.** A marked program configuration \(\tilde{K}\) single-handler conforms to a specification configuration \(S\), written \(\tilde{K} \models_{\text{SH}} S\), if and only if there exists a single-handler-conformance-dense relation \(R\) such that \(\langle \tilde{K}, S \rangle \in R\). As with standard conformance, \(P \models_{\text{SH}} \Sigma\) if and only if there exists some maximal instantiation \(\langle \tilde{K}, S \rangle\) of \(P\) and \(\Sigma\) such that \(\tilde{K} \models_{\text{SH}} S\).

Single-handler conformance is equivalent to external-representative conformance, as stated in the following theorem.

**Theorem (Single-Handler Conformance).** For all \(P\) and \(\Sigma\), \(P \models_{\text{SH}} \Sigma\) if and only if \(P \models_{\text{ER}} \Sigma\).
D.1  Definitions for the Proof

The proof that single-handler conformance implies external-representative conformance requires some auxiliary definitions. First, a specification execution $S_1 \rightarrow_{\lambda_1} \ldots$ single-handler-simulates a program execution $\tilde{K}_1 \rightarrow_{\tilde{l}_1} \ldots$ if the two executions have the same number of steps, $\lambda_i = |\tilde{l}_i|$ for all labels $\lambda_i$ and $\tilde{l}_i$ in the two executions, and $\tilde{K}_i \models_{SH} S_i$ for all $\tilde{K}_i$ and $S_i$ in the two executions.

Second, we formalize a correspondence between the labels of an execution and a reordered version of that execution. A function $\theta$ is an index map from an execution $\tilde{K}_1 \rightarrow_{\tilde{l}_1} \ldots$ to an execution $\tilde{K}_1' \rightarrow_{\tilde{l}_1'} \ldots$ if the two executions have the same number of steps, $\theta$ is a bijection over the set of label indices in $\tilde{K}_1 \rightarrow_{\tilde{l}_1} \ldots$, and $\tilde{l}_i = \tilde{l}'_j$ whenever $\theta(i) = j$. Additionally, such an index map is handler-swap-safe if for all $i$ and $j$ such that $i < j$ and $\theta(i) > \theta(j)$, $\tilde{l}_j$ is not a rcv-ext or rcv-int label, and the active actors for $\tilde{l}_i$ and $\tilde{l}_j$ are distinct.

Finally, the proof uses partial address-to-address functions (denoted $\sigma_A$) and partial marker-to-marker functions (denoted $\sigma_M$) to show that two program configurations or specification configurations are isomorphic, differing only in the choice of address or substitutions (which are otherwise uninterpreted tokens). The application $\sigma_A(K)$ of such an address substitution $\sigma_A$ to a program configuration $K$ is defined as the obvious element-wise, component-wise, point-wise application to its parts, and similarly for a marker-substitution function $\sigma_M$.

The application is undefined if $\sigma_A(a)$ or $\sigma_M(\eta)$ is undefined for any address $a$ or marker $\eta$ appearing in $K$. An address-substitution function $\sigma_A$ is location-preserving if and only if for all $(addr \ell n) \in \text{dom}(\sigma_A)$ there exists $n'$ such that $\sigma_A((addr \ell n)) = (addr \ell n')$.

The remainder of this appendix proves the necessary lemmas in bottom-up fashion, followed by the main theorem.

D.2 Handler Step Commutativity Lemma

Lemma (Handler Step Commutativity). For all $K$, $K'$, $K''$, $\tilde{l}$, and $\tilde{l}'$, if

- $K \rightarrow_{ER} K'$ and $K' \rightarrow_{ER} K''$,
- the active actors for $\tilde{l}$ and $\tilde{l}'$ are distinct, and
- $K \rightarrow_{ER} K''$

then $K'' \rightarrow_{ER} K''$.

Proof. Because of the “shared nothing” approach of the actor model, one CSA actor cannot take a step that prevents an actor from taking a step it was previously able to take, up to the choice of new markers in a send or receive step, or
the choice of address for a spawn step. In this case, because $K' \xrightarrow{l'} \text{ER} K''$ and because the label for spawn steps identifies the new address, we know that any step with label $l'$ does not assign an address that was already assigned by the $l$ step. Similarly, we know that any new markers assigned by the $l$ and $l'$ are disjoint, because Markings requires all assigned markers to be fresh and the label includes all new assigned markers. Therefore, $K'''$ is able to take a step labeled with $l$.

It remains to show that such a step can reach the configuration $K''$. Effectively, we must show that the effects of the $l$ and $l'$ steps commute. Those possible effects are

- reducing the active actor’s behavior expression,
- spawning new actors,
- adding new in-flight messages,
- removing existing in-flight messages,
- adding new receptionists, and
- adding new markers to the configuration’s set $H$ of used markers.

Because the two transition steps have distinct active actors, they each reduce a different actor’s behavior expression, so it is easy to see how those two effects commute. The other effects involves only adding or removing elements from unordered collections, so those effects obviously commute, as well. Therefore, $K''' \xrightarrow{l} \text{ER} K''$. $\Box$

D.3 Program Fair Suffix Lemma

**Lemma** (Program Fair Suffix). For all external-representative executions $K_1 \xrightarrow{l_1} \ldots$, if the suffix $K_k \xrightarrow{l_k} \ldots$ of that execution is fair for some $k > 0$, then $K_1 \xrightarrow{l_1} \ldots$ is fair.

**Proof.** The proof is by induction on $k$. In the base case, if $k = 1$, then the suffix includes the entire execution and we’re done. Otherwise, by the induction hypothesis, the execution $K_2 \xrightarrow{l_2} \ldots$ is fair, and it remains to show that all actors with work to do in $K_1$ eventually transition and all in-flight messages in $K_1$ are eventually received.

Let $a$ be the address of some actor with an enabled transition step from $K_1$. If the actor at $a$ in $K_2$ also has an enabled transition step, then because the suffix from $K_2$ is fair, there must be some future step of the execution with $a$ as its active actor. Otherwise, the only way the actor at $a$ would not have an enabled step from $K_2$ is if the step labeled with $l_1$ transitioned that actor, so $a$ would be the active actor for that step.
Similarly, let \( \langle a@H, \bar{v} \rangle \) be an in-flight message in \( \bar{K}_1 \). If that message also exists in \( \bar{K}_2 \), then by the fairness of the suffix from that point, some future step must receive that message. Otherwise, the only way for that message to not exist in \( \bar{K}_2 \) is for the step labeled with \( l_1 \) to receive that message. Therefore, all actors with work to do in \( \bar{K}_1 \) eventually transition, and all in-flight messages in \( \bar{K}_1 \) are eventually received.

\[ \square \]

D.4 Handler Termination Lemma

This lemma formalizes the idea that in the single-handler semantics, the highest-priority actor handling an event will eventually finish its handler expression (including the initializers for any actors it spawns), as long as it does not get stuck.

As a shorthand in this lemma, let \( \text{Handling}(\bar{K}) \) be the set of addresses \( \{a_1, \ldots, a_n\} \) such that the actor at \( a_i \) is handling an event in \( \bar{K} \).

**Lemma** (Handler Termination). Let there be a fair single-handler execution \( \bar{K}_1 \xrightarrow{l_1} \bar{K}_2 \cdots \) in which \( a \) is the maximal address (according to \( \prec \)) in \( \text{Handling}(\bar{K}_1) \). Then there exists a configuration \( \bar{K}_n \) in that execution such that either the actor at \( a \) is awaiting an event in \( \bar{K}_n \) and \( \text{Handling}(\bar{K}_n) \subset \text{Handling}(\bar{K}_1) \), or \( \bar{K}_n \) is stuck.

**Proof.** The proof is by induction on the number of levels of nested \textit{spawn} expressions in the handler expression. In the base case, the handler expression for the actor at \( a \) in \( \bar{K}_1 \) contains no \textit{spawn} expressions. Then it is easy to see that the syntax of the remaining CSA expressions restricts them to terminating expressions (note that the only loops are bounded \textit{for/fold} loops). Then by the definition of \( \xrightarrow{\text{SH}} \), there exists some \( n \) such that the actor at \( a \) in \( \bar{K}_n \) either has finished its handler and is awaiting an event (and no other actor has started handling an event), or that actor is stuck.

In the inductive case, the expression contains \( i + 1 \) levels of nested \textit{spawn} expressions, and the actor may spawn some number of children during evaluation of the handler expression. Each spawned actor must come from a \textit{spawn} expression lexically nested inside this actor, so by the definition of \( \prec \), the spawned actor takes priority and the single-handler semantics starts evaluating the child’s handler expression instead as soon as it is spawned. However, by the induction hypothesis we know that the child’s handler expression eventually terminates or gets stuck. If it gets stuck, then the entire configuration is stuck. Otherwise, it terminates and the parent resumes execution. Therefore the original parent actor eventually finishes its own handler expression or becomes stuck, by an argument similar to the base case above. At that point, all other actors that started handling an event during that sequence (\textit{i.e.}, the spawned actors) also terminated, so \( \text{Handling}(\bar{K}_n) \subset \text{Handling}(\bar{K}_1) \). \[ \square \]
D.5 PSM Output Commutativity Lemma

Lemma (PSM Output Commutativity). For all $s_1, S_2, S_3, a, H, \delta$, and $\lambda$, if

1. $s_1 \xrightarrow{a@H} S_2 \xrightarrow{\lambda} S_3$, and

2. if there exist $a', H'$, and $\delta'$ such that $\lambda = a'@H'\delta'$, then
   - there does not exist some $a''@H''$ in $\delta$ such that $\max(H'') \in H'$, and
   - there does not exist some $a''@H''$ in $\delta'$ such that $\max(H'') \in H$,

then there exists $S_4$ such that $s_1 \xrightarrow{a@H} S_4 \xrightarrow{\lambda} S_3$.

Proof. By the definition of $s_1 \xrightarrow{a@H} S_2$, there exist $s_2, O_1$, and $S'_2$ such that $S_2 = \{s_2\} \cup S'_2$ and $s_1 \xrightarrow{a@H; O_1} S'_2 \xrightarrow{a} s_2$. Then by the definition of the second step $S_2 \xrightarrow{\lambda} S_3$, there also exist $S'_3$ and $S''_3$ such that $S_3 = \{s_3\} \cup S''_3$, $s_2 \xrightarrow{\lambda} S'_3$, and $S'_3 \xrightarrow{\lambda} S''_3$. We will show how the two transitions commute on these different components.

Because the transition $s_1 \xrightarrow{a@H} S_2$ is an output step, by the definition of the PSM transition relation, that transition is an instance of the P-SEND rule. Therefore, there exist $H_{in}$, $H_{out}$, $H''$, $\bar{\delta}$, $\Phi$, and $O$ such that $s_1 = (H_{in}, H_{out}, \Phi; \bar{\delta}, \Phi, O \cup O_1)$ and $s_2 = (H_{in}, H_{out}, \Phi; \bar{\delta}, \Phi, O)$. Formally, say that a PSM $s$ differs from $s'$ by $H$ and $O$ if and only if the two PSMs are identical except that $\text{InMon}(s') = \text{InMon}(s) \cup H$ and $s$ has the additional obligations $O$ relative to $s'$. Thus, $s_1$ differs from $s_2$ by $H''$ and $O_1$.

Next, we show that $s_1$ can simulate the transitions from $s_2$. By the definition of the transition $s_2 \xrightarrow{a@H} S'_2$, there exist $s_3, \ldots, s_{n+m+1}, S''_n, \ldots, S''_{n+m}$, and $S''_m$, such that

- $s_2 \xrightarrow{\Phi, S''_n, \ldots, \Phi, S''_{n+m}} s_{n+1}$
- $s_{n+1} \xrightarrow{\lambda, O_2, S''_{n+1}} s_{n+2}$
- $s_{n+2} \xrightarrow{\Phi, S''_{n+2}} s_{n+3}$
- $s_{n+3} \xrightarrow{\Phi, S''_{n+3}} s_{n+4}$
- $s_{n+m+1} \xrightarrow{\lambda} S''_m$

for all $i \in 2, \ldots, n$, $S''_i \xrightarrow{\lambda} S''_{i+1}$,

for all $i \in n+1, \ldots, n+m$, $S''_i \xrightarrow{\lambda} S''_{i+1}$, and

$S''_3 = \{s_{n+m+1}\} \cup S''_2 \cup \ldots \cup S''_{n+m}$.

Each $s_i \xrightarrow{\Phi, S''_i} s_{i+1}$ transition must be an instance of the P-FREE transition rule. Such a step does not depend on the input-monitored markers or obligations in $s_i$ to be enabled, so there exist $s'_2, \ldots, s'_{n+1}$ such that

- $s'_2 \xrightarrow{\Phi, S''_2} \ldots \xrightarrow{\Phi, S''_n} s'_{n+1}$
- $s'_{n+1} = s_1$, and
• for all \( i \in 2 \ldots n + 1 \), \( s_i' \) differs from \( s_i \) by \( H'' \) and \( O_1 \).

Also, for each new input-monitored marker \( \eta \in H'' \), it must be the case that
\( \eta = \max(H''') \) for some \( a''@H''' \) in \( \breve{v} \) (by the definition of P-SEND and the output-pattern-matching rules). Therefore, by the preconditions to this lemma, if \( \lambda = a''@H'''?\breve{v}' \), then \( H' \cap H'' = \emptyset \). That means that those extra input-monitored markers do not affect how \( s_{n+1} \) transitions with the label \( \lambda \), so there exists \( s_{n+2}' \) such that
\[
\frac{\lambda, O_2, S''_{n+1}}{s_{n+2}'} \text{ and } s_{n+2}' \neq s_{n+2} \text{ by } H'' \text{ and } O_1. \]
By a similar argument to above, there also exist \( s_{n+3}', \ldots, s_{n+m+1}' \) such that
\[
\frac{s_{n+2}'}{\ldots} \text{ and } \frac{\text{ for all } i \in n + 3 \ldots n + m + 1, s_i' \text{ differs from } s_i \text{ by } H'' \text{ and } O_1.}
\]

We already know that \( S''_{n+1} \xrightarrow{\lambda} S'''_{n+1} \) for all \( i \in 2 \ldots n \) and \( S''_{n+1} \xrightarrow{a} S'''_{n+1} \) for all \( i \in n + 1, \ldots, n + m, \) so we have \( \{s_1\} \xrightarrow{\lambda} \{s_{n+1}\} \cup S''_{n+1} \cup \ldots \cup S'''_{n+1} \). Let \( S_4 = \{s_{n+m+1}\} \cup S''_{n+1} \cup \ldots \cup S'''_{n+1} \); then we have \( \{s_1\} \xrightarrow{\lambda} S_4 \).

Next, we must show that each of those resulting PSMs in \( S_4 \) can take a transition labeled with \( a@H;!\breve{v} \). Because \( s_{n+m+1}' \) differs from \( s_{n+m+1} \) by \( H'' \) and \( O_1 \), a P-SEND transition similar to the one from \( s_1 \) is possible, so that
\[
\frac{a@H;O_1, S_2'}{s_{n+m+1}'} \text{ and } \frac{S_2'}{s_{n+m+1}} \text{ by } \frac{S_2'}{s_{n+m+1}} \text{ are the same because they are defined by matching the same patterns (from } O_1 \text{) against the same value } \breve{v}. \text{ As a result, we have } \frac{\{s_{n+m+1}\}}{a@H;O_1, S_2'} \text{ and } S_2' \text{ by } \frac{\{s_{n+m+1}\} \cup S_2'}{s_{n+m+1}}.
\]

For the forked PSMs, let \( i \in 2 \ldots n + m \), and let \( s'' \) be a member of \( S_i''' \). We will show that \( \text{OutMon}(s'') \cap H = \emptyset \) by cases on the rule that enabled the transition that forked the PSMs \( S_i''' \).

• **P-UNMONITOREDRECEIVE:** This rule does not fork any PSMs, so it does not apply.

• **P-MONITOREDRECEIVE:** In this case, the transition was on a label of the form \( a'@H'?!\breve{v}' \). Then for all \( \eta \in \text{OutMon}(s''), \eta = \max(H''') \) for some \( a''@H''' \) in \( \breve{v}' \) (by the definition of this rule and the input-pattern-matching rules). Therefore, by the preconditions to this lemma, \( \eta \notin H \), so \( \text{OutMon}(s'') \cap H = \emptyset \).

• **P-FREETRANSITION:** By the definition of this rule, \( \text{OutMon}(s'') = \emptyset \).

• **P-SEND:** The only PSMs forked during a transition using this rule come from delayed-fork-addr patterns. Such a PSM has no output-monitored markers, so \( \text{OutMon}(s'') = \emptyset \).
Then because $\text{OutMon}(s'') \cap H = \emptyset$, by the P-Send rule, $s'' \xrightarrow{a@H^\ell, \varphi, \varphi} s'''$. Therefore, we have $S''_i \xrightarrow{a@H^\ell} S'''_i$ for all $i \in 2 \ldots n + m$. We know that $S'_3 = (s_{n+m+1}) \cup S''_2 \cup \ldots \cup S''_{n+m}$, so in total we have $S'_4 \xrightarrow{a@H^\ell} S'_3 \cup S'_2$.

It remains to show that $S'_2 \xrightarrow{\varphi} S'_3$. We already know that $S'_2 \xrightarrow{\lambda} S''_3$ and that the PSMs $S'_2$ were created in the transition $s_1 \xrightarrow{a@H^\ell, O_1, S'_2} s_2$, which is an instance of the P-Send rule. If $\lambda = \varphi$, then we’re done. Otherwise, there exist $S_5$ and $S_6$ such that $S'_2 \xrightarrow{\lambda} S_5 \xrightarrow{\varphi} S_6 \xrightarrow{\varphi} S''_3$. We will show by cases on $\lambda$ that $S_5 = S_6$.

In the first case, let there be $a'$, $H'$, and $\varphi'$ such that $\lambda = a'@H'@\varphi'$. By the definition of the P-Send rule, each PSM $s'' \in S'_2$ was created from a delayed-fork-addr pattern, and thus $\text{OutMon}(s'') = \emptyset$. A transition labeled with $\varphi$ does not add new monitored markers, so $\text{OutMon}(s'') = \emptyset$ for all $s'' \in S_5$. Therefore $s'' \xrightarrow{a'@H'@\varphi', \varphi, \varphi} s'''$ by another use of the P-Send rule for all $s'' \in S_5$, and therefore $S_5 = S_6$.

In the remaining case, there exist $a'$, $H'$, and $\varphi'$ such that $\lambda = a'@H'@\varphi'$. Let $\eta$ be a member of $\text{InMon}(s'')$. By the definition of the P-Send rule and the output-pattern-matching rules, there exists some $a''@H''$ in $\varphi$ such that $\eta = \text{max}(H'\eta)$. By the preconditions to this lemma, $\eta \notin H'$, so $\text{InMon}(s'') \cap H = \emptyset$. Again, a transition labeled with $\varphi$ does not add new monitored markers, so $\text{InMon}(s'') = \emptyset$ for all $s'' \in S_5$. Therefore, $s'' \xrightarrow{\lambda, \varphi, \varphi} s'''$ by the P-UnmonitoredReceive rule for all $s'' \in S_5$, and therefore $S_5 = S_6$.

As a result of the above cases, we have $S'_2 \xrightarrow{\lambda} S_5 \xrightarrow{\varphi} S''_3$. Therefore, by the definition of $\xrightarrow{\lambda}$, $S'_2 \xrightarrow{\lambda} S''_3$. Also, by the definition of $\xrightarrow{\varphi}$, we have $S'_3 \xrightarrow{\varphi} S'_3$. In total, we have that

- $\{s_1\} \xrightarrow{\lambda} S_4$,
- $S_4 \xrightarrow{a@H^\ell} S'_3 \cup S'_2$,
- $S'_3 \xrightarrow{\varphi} S'_3$,
- $S'_2 \xrightarrow{\varphi} S''_3$,
- $S_3 = S'_3 \cup S''_3$.

That gives us the following.

$$\{s_1\} \xrightarrow{\lambda} S_4 \xrightarrow{a@H^\ell} (S'_3 \cup S'_2) \xrightarrow{\varphi} S'_3$$

Then by collapsing the $\xrightarrow{\varphi}$ transition into the previous transition, we have the following, which completes the proof.

$$\{s_1\} \xrightarrow{\lambda} S_4 \xrightarrow{a@H^\ell} S_3$$
D.6 Specification Output Commutativity Lemma

**Lemma** (Specification Output Commutativity). For all $S_1, S_2, S_3, a, H, \bar{v},$ and $\lambda$, if

1. $S_1 \xrightarrow{a@H!\bar{v}} S_2 \xrightarrow{\lambda} S_3$, and
2. if there exist $a', H'$, and $\bar{v}'$ such that $\lambda = a'@H'?\bar{v}'$, then
   - there does not exist some $a''@H''$ in $\bar{v}$ such that $\max(H'') \in H'$, and
   - there does not exist some $a''@H''$ in $\bar{v}'$ such that $\max(H'') \in H$,

then there exists $S_4$ such that $S_1 \xrightarrow{\lambda} S_4 \xrightarrow{a@H!\bar{v}} S_3$.

**Proof.** Let $\{s_{1,1}, \ldots, s_{1,n}\} = S_1$. By the definition of the $\rightarrow$ and $\Rightarrow$ relations, there exist $S_{2,1}, \ldots, S_{2,n}$ and $S_{3,1}, \ldots, S_{3,n}$ such that

- $S_2 = S_{2,1} \cup \ldots \cup S_{2,n}$,
- $S_3 = S_{3,1} \cup \ldots \cup S_{3,n}$, and
- for all $i \in 1 \ldots n$, $\{s_i\} \xrightarrow{a@H!\bar{v}} S_{2,i} \xrightarrow{\lambda} S_{3,i}$.

By the PSM Output Commutativity lemma, there exist $S_{4,1}, \ldots, S_{4,n}$ such that for all $i \in 1 \ldots n$, $\{s_i\} \xrightarrow{\lambda} S_{4,i} \xrightarrow{a@H!\bar{v}} S_{3,i}$. Let $S_4 = S_{4,1} \cup \ldots \cup S_{4,n}$. Then by the definition of the $\Rightarrow$ relation, $S_1 \xrightarrow{\lambda} S_4 \xrightarrow{a@H!\bar{v}} S_3$. \qed

D.7 Specification Step Commutativity Lemma

**Lemma** (Specification Step Commutativity). For all $S_1, S_2, S_3, \lambda_1$, and $\lambda_2$, if

- $S_1 \xrightarrow{\lambda_1} S_2 \xrightarrow{\lambda_2} S_3$,
- either $\lambda_1 = \bullet$ or there exist $a, H, \text{ and } \bar{v}$ such that $\lambda_1 = a@H!\bar{v}$, and
- if there exist $a, a', H, H', \bar{v},$ and $\bar{v}'$ such that $\lambda_1 = a@H!\bar{v}$ and $\lambda_2 = a'@H'?\bar{v}'$, then
  - there does not exist some $a''@H''$ in $\bar{v}$ such that $\max(H'') \in H'$, and
  - there does not exist some $a''@H''$ in $\bar{v}'$ such that $\max(H'') \in H$,

then there exists $S_4$ such that $S_1 \xrightarrow{\lambda_2} S_4 \xrightarrow{\lambda_1} S_3$.

The lemma’s final precondition says that neither step defines a new marker used by the other step, which gives the two steps a notion of independence from one another.
Proof. First, consider the case where $\lambda_1 = \cdot$. In that case, we have $S_1 \xrightarrow{\lambda_2} S_2 \xrightarrow{\cdot} S_3$. By the definition of the specification multi-step relation $\rightarrow$, the steps can be written equivalently as $S_1 \xrightarrow{\lambda_2} S_3 \xrightarrow{\cdot} S_3$. Let $S_4 = S_3$ and we’re done.

Otherwise, there are some $a$, $H$, and $\hat{v}$ such that $\lambda = a@H!\hat{v}$. By expanding the step $S_1 \xrightarrow{a@H!\hat{v}} S_2$, we have the following for some $S_5$ and $S_6$.

$$S_1 \xrightarrow{\cdot} S_5 \xrightarrow{a@H!\hat{v}} S_6 \xrightarrow{\lambda_2} S_3$$

By collapsing the latter $\cdot$ step into the $\lambda_2$ step, we get the following.

$$S_1 \xrightarrow{a@H!\hat{v}} S_5 \xrightarrow{\lambda_2} S_6 \xrightarrow{\cdot} S_3$$

Then by the Specification Output Commutativity lemma, there exists some $S_4$ such that the following holds.

$$S_1 \xrightarrow{\lambda_2} S_5 \xrightarrow{a@H!\hat{v}} S_4 \xrightarrow{\lambda_2} S_3$$

Finally, by collapsing the step labeled with $\cdot$ into the following step, we have the following.

$$S_1 \xrightarrow{\lambda_2} S_4 \xrightarrow{a@H!\hat{v}} S_3$$

\[\square\]

D.8 Specification Fair Suffix Lemma

Lemma (Specification Fair Suffix). For all specification executions $S_1 \xrightarrow{\lambda_1, O_1} \ldots$, if the suffix $S_k \xrightarrow{\lambda_k, O_k} \ldots$ of that execution is fair for some $k > 0$, then $S_1 \xrightarrow{\lambda_1, O_1} \ldots$ is fair.

Proof. The proof is by induction on $k$. In the base case, if $k = 1$, then the suffix includes the entire execution and we’re done. Otherwise, by the induction hypothesis, the execution $S_2 \xrightarrow{\lambda_2, O_2} \ldots$ is fair, and it remains to show that all obligations in $S_1$ are eventually fulfilled.

Let there be $\eta$ and $po$ such that $\langle \eta, po \rangle \in \text{Obls}(S_1)$. We must show that some future step fulfills that obligation. If $\langle \eta, po \rangle \in \text{Obls}(S_2)$, then the fairness for the suffix from $S_2$ guarantees that some future step fulfills the obligation. Otherwise, the only way $S_2$ would not contain the obligation $\langle \eta, po \rangle$ is that an instance of the P-SEND rule fulfilled that obligation during the first step of the execution, so that $S_1 \xrightarrow{\cdot, \cdot} S_1 \xrightarrow{\lambda_1, O_1} \cdot, \cdot S_2$ and $\langle \eta, po \rangle \in O$. Then every obligation in $\text{Obls}(S_1)$ is eventually fulfilled, so $S_1 \xrightarrow{\lambda_1} \ldots$ is fair. \[\square\]
D.9 Single-Handler Isomorphism Lemma

Lemma (Single-Handler Isomorphism). For all $\vec{K}$, $\vec{K}'$, $S$, $S'$, $\sigma_A$, and $\sigma_M$, if

- $\sigma_A$ is location-preserving and one-to-one,
- $\sigma_M$ is one-to-one,
- $\sigma_A(\sigma_M(\vec{K})) = \vec{K}'$,
- $\sigma_M(S) = S'$, and
- $\vec{K} \models_{\text{SH}} S$,

then $\vec{K}' \models_{\text{SH}} S'$.

Proof. By the definition of $\models_{\text{SH}}$, there exists a single-handler-conformance-dense relation $R$ such that $\langle \vec{K}, S \rangle \in R$. Let $R'$ be the relation defined of all pairs $\langle \vec{K}', S' \rangle$ for which there exist $\vec{K}_1$, $S_1$, $\sigma_A'$, and $\sigma_M'$ such that

- $\sigma_A'$ is location-preserving and one-to-one,
- $\sigma_M'$ is one-to-one,
- $\sigma_A'(\sigma_M'(\vec{K}_1)) = \vec{K}'_1$,
- $\sigma_M'(S_1) = S'_1$, and
- $\langle \vec{K}_1, S_1 \rangle \in R$.

Clearly $\langle \vec{K}', S' \rangle \in R'$. It remains to show that $R'$ is single-handler-conformance-dense. Let $\langle \vec{K}', S' \rangle$ be a member of $R'$, let $\vec{K}_1$, $S_1$, $\sigma_A'$, and $\sigma_M'$ be the associated configurations and functions given by $R'$, and let $\vec{K}'_1 \xrightarrow{i_1} \vec{K}' \models_{\text{SH}} \ldots$ be a fair, non-stuck execution. Because the specific addresses and markers in the configurations are uninterpreted tokens, it is easy to see that the choice of specific address or marker does not matter (as long as the address substitution is location-preserving).

Therefore, there is a fair, non-stuck execution $\vec{K}_1 \xrightarrow{i_1} \vec{K}_1 \xrightarrow{i_1} \models_{\text{SH}} \ldots$ that is like $\vec{K}'_1 \xrightarrow{i_1} \models_{\text{SH}} \ldots$ except that it might allocate different addresses and markers at each step, so that there are one-to-one, location-preserving substitutions $\sigma_A', \sigma_M', \ldots$ such that $\sigma_A'(\sigma_M'(\vec{K}_1)) = \vec{K}'_1$ and $\sigma_M'(S_1) = S'_1$ for each corresponding pair of configurations and labels in the two executions.

Because $R$ is single-handler-conformance-dense, there exists some fair execution $S_1 \xrightarrow{[i_1]} \ldots$ that single-handler simulates $\vec{K}_1 \xrightarrow{i_1} \models_{\text{SH}} \ldots$. Again the choice of markers is irrelevant to specification-configuration transitions, so there is also a fair execution $S'_1 \xrightarrow{[i_1]} \ldots$ such that $\sigma_M'(S_1) = S'_1$ for all corresponding pairs of configurations in the two executions. Therefore, each pair $\langle \vec{K}', S' \rangle$ is a member of $R'$, and therefore $R'$ is single-handler-conformance-dense. \qed
D.10 Single-Handler Prefix Lemma

Lemma (Single-Handler Prefix). For all \( \bar{K}_1, \bar{K}_2, S_1, S_2, \) and \( \bar{l} \), if

1. \( \bar{K}_2 \models_{\text{SH}} S_2 \),
2. \( \bar{K}_1 \xrightarrow{\bar{l}}_{\text{SH}} \bar{K}_2 \),
3. \( S_1 \xrightarrow{\bar{l}} S_2 \), and
4. \( \bar{l} \) is a handler-continuation label,

then \( \bar{K}_1 \models_{\text{SH}} S_1 \).

Proof. Let \( R = (\bar{K}_1, S_1) \cup \models_{\text{SH}} \). It is sufficient to show that \( R \) is single-handler-conformance-dense.

Let \( (\bar{K}_1', S_1') \) be a member of \( R \), and let \( \bar{K}_1' \xrightarrow{l'}_{\text{SH}} \ldots \) be a fair, non-stuck execution. If \( \bar{K}_1' \models_{\text{SH}} S_1' \), then finding the matching specification execution is trivial.

Otherwise, \( \bar{K}_1' = \bar{K}_1 \) and \( S_1' = S_1 \). Because

1. \( \bar{K}_1 \xrightarrow{l}_{\text{SH}} \bar{K}_2 \),
2. \( \bar{l} \) is a handler-continuation label,
3. the single-handler semantics transitions only the highest-priority running actor, and
4. handler steps are deterministic up to the choice of new markers and addresses,

there exists some one-to-one, location-preserving address substitution \( \sigma_A \) and one-to-one marker substitution \( \sigma_M \) such that \( \sigma_A(\sigma_M(\bar{K}_2)) = \bar{K}_2' \) and \( \sigma_A(\sigma_M(\bar{l})) = \bar{l}' \). Furthermore, there exists \( S_2' \) such that \( S_1' \xrightarrow{\bar{l}_1'} S_1' \xrightarrow{\bar{l}_2'} S_2' \) and \( \sigma_M(S_1) = S_1' \).

By the Single-Handler Isomorphism lemma, \( \bar{K}_2' \models_{\text{SH}} S_2' \). Therefore there exists a fair specification execution \( S_2' \xrightarrow{\bar{l}_3'} \ldots \) that single-handler-simulates the program execution from \( \bar{K}_2' \). Then by the Specification Fair Suffix lemma, \( S_1' \xrightarrow{\bar{l}_1'} \ldots \) is also fair. Also, for all corresponding \( \bar{K}_1' \) and \( S_1' \) in the two executions, \( (\bar{K}_1', S_1') \in R \). Therefore, \( R \) is single-handler-conformance-dense. \( \square \)

D.11 Handler Continuation Lemma

Lemma (Handler Continuation). For all \( \bar{K}_1, \bar{K}_2, S_1, S_2, \) and \( \bar{l} \), if

1. \( \bar{K}_2 \models_{\text{SH}} S_2 \),


• $\tilde{K}_1 \xrightarrow{\tilde{l}}_{ER} \tilde{K}_2$,
• $S_1 \xrightarrow{[\tilde{l}]} S_2$, and
• \(\tilde{l}\) is a handler-continuation label,
then $\tilde{K}_1 \models_{SH} S_1$.

**Proof.** Let $R = \{\langle \tilde{K}_1, S_1 \rangle\} \cup \models_{SH}$. We will show that the relation $R$ is single-handler-conformance-dense.

Let $\langle \tilde{K}'_1, S'_1 \rangle$ be a member of $R$, and let $\tilde{K}'_1 \xrightarrow{[\tilde{l}]}_{SH} \cdots$ be a fair, non-stuck execution. We must show that there exists a fair specification execution $S'_1 \xrightarrow{[\tilde{l}]}_{SH} \cdots$ of the same length such that $\langle \tilde{K}'_i, S'_i \rangle \in R$ for all corresponding $\tilde{K}'_i$ and $S'_i$ in the respective executions.

If $\langle \tilde{K}'_1, S'_1 \rangle \in \models_{SH}$, then the proof is trivial by the definition of $\models_{SH}$. Otherwise, $\tilde{K}'_1 = \tilde{K}_1$ and $S'_1 = S_1$. The general strategy is to show that this execution eventually converges with some execution starting from a configuration isomorphic to $\tilde{K}_2$, then adapt a specification execution from a configuration isomorphic to $S_2$.

The following steps detail this approach, and the diagrams in figure D.1 illustrate the steps. For readability, the diagram abbreviates sequences of transitions such as $\tilde{l}'_1 \xrightarrow{[\tilde{l}]} \cdots \xrightarrow{[\tilde{l}]} \tilde{l}'_n$ as a single arrow with multiple labels, as in $\tilde{l}'_1 \xrightarrow{[\tilde{l}]} \cdots \xrightarrow{[\tilde{l}]} \tilde{l}'_n$.

1. Let $a$ be the address of the active actor for $\tilde{l}$. By the Handler Termination lemma and the fairness/non-stuck-ness of $\tilde{K}'_1 \xrightarrow{[\tilde{l}]}_{SH} \cdots$, each actor with a higher priority than $a$ in $\tilde{K}'_1$ will eventually finish its handler expression.

Therefore there exists some $n$ such that $\tilde{K}'_1 \xrightarrow{[\tilde{l}]}_{SH} \cdots \xrightarrow{[\tilde{l}]} \tilde{K}'_{n+1}$ and no actor with a higher priority than the one at $a$ is handling an event in $\tilde{K}'_{n+1}$.

2. Because $\tilde{l}$ is a handler-continuation label, the actor at $a$ must be handling an event in $\tilde{K}'_1$, and because $a$ does not identify the active actor for any label in $\tilde{l}'_1, \ldots, \tilde{l}'_n$, the actor at $a$ is still handling an event in $\tilde{K}'_{n+1}$. Because the execution is fair and that actor has the highest priority at that point, there must exist a step $\tilde{K}'_{n+1} \xrightarrow{[\tilde{l}]}_{SH} \tilde{K}'_{n+2}$ in that execution such that $a$ identifies the active actor for $\tilde{l}'_{n+1}$.

3. Because $\tilde{l}$ is a handler-continuation label and $a$ does not identify the active actor for any label in $\tilde{l}'_1, \ldots, \tilde{l}'_n$, the label $\tilde{l}'_{n+1}$ must be identical to $\tilde{l}$ except for the choice of a new address in the case of a spawn label or the choice of new markers in the case of a send-int or send-ext label. Those choices do not affect whether the transition is enabled, so because $\tilde{K}_1 \xrightarrow{[\tilde{l}]}_{ER} \tilde{K}_2$,

there exists some $\tilde{K}''_2$ such that $\tilde{K}'_1 \xrightarrow{[\tilde{l}]}_{ER} \tilde{K}''_2$. Furthermore, there exist a one-to-one, location-preserving address-substitution function $\sigma_A$ and a
Figure D.1: Illustration of proof steps for Handler Continuation
one-to-one marker substitution function $\sigma_M$ such that $\sigma_A(\sigma_M(\bar{I})) = \bar{I}'$ and $\sigma_A(\sigma_M(\bar{K}_2)) = \bar{K}''_2$.

4. Because the actor with address $a$ is the highest priority actor in $\bar{K}'_{n+1}$, by the Handler Termination lemma that actor will eventually complete its handler. Therefore because the execution is fair, it contains some steps

$$\bar{K}'_{n+2} \xrightarrow{SH} \cdots \xrightarrow{SH} \bar{K}'_{n+m+1}$$

such that the actor at $a$ is not handling an event in $\bar{K}'_{n+m+1}$.

5. Next, we show that there is a similar sequence of single-handler steps that can reach $\bar{K}'_{n+m+1}$ from $\bar{K}'_n$. The prefix of $\bar{I}'_{n+1}, \ldots, \bar{I}'_n$ whose labels have an active address $a'$ such that $a < a'$ will run before the remainder of the handler for the actor at $a$, so there exists some $k \leq n$ such that the steps first $k$ steps can execute sequentially. Then the reached configuration can take the steps labeled with $\bar{I}'_{n+2}, \ldots, \bar{I}'_{n+m}$ to finish the handler for the actor at $a$. The single-handler semantics will then allow the remaining steps with labels $\bar{I}'_{k+1}, \ldots, \bar{I}'_n$ to run only after that handler terminates. Finally, because the sequence $\bar{I}'_{n+1}, \bar{I}'_1, \bar{I}'_{k+1}, \ldots, \bar{I}'_{n+2}, \bar{I}'_{n+3}, \ldots, \bar{I}'_{n+m}, \bar{I}'_{n+1}, \ldots, \bar{I}'_n$ is a permutation of the sequence $\bar{I}'_1, \ldots, \bar{I}'_{n+m}$, and the permutation preserves the relative order of labels with the same active actor, by the Handler Step Commutativity lemma, the reached configuration after taking those steps is $\bar{K}'_{n+m+1}$.

6. As mentioned above, there exist a one-to-one, location-preserving address-substitution function $\sigma_A$ and a one-to-one marker substitution function $\sigma_M$ such that $\sigma_A(\sigma_M(\bar{I})) = \bar{I}'_{n+1}$. The address used for a spawn label is irrelevant for specification steps, and the specific markers chosen do not affect whether a step is enabled or not. So because we have $S^{1} \xrightarrow{[\bar{I}]} S^{2}$, there exists $S^{2''}$ such that $S^{1} \xrightarrow{[\bar{I}]} S^{2''} \xrightarrow{\bar{I}''_{n+1}} S^{2''}$ and $\sigma_A(\sigma_M(S_2)) = S''_2$.

7. By the Single-Handler Isomorphism lemma, $\bar{K}'_1 \equiv_{SH} \bar{K}'_2$. The execution $\bar{K}'_1 \xrightarrow{\bar{I}'_1} \cdots$ is fair and non-stuck, so its suffix starting from $\bar{K}'_{n+m+1}$ is also fair and non-stuck. Let $\bar{K}'_{3} \xrightarrow{\bar{I}'_{3}} \cdots$ be the execution defined by appending $\bar{K}'_{n+m+1}$ to the end of $\bar{K}'_2 \xrightarrow{\bar{I}'_1} \cdots \xrightarrow{\bar{I}'_k} \cdots \xrightarrow{\bar{I}'_{n+m}} \cdots \xrightarrow{\bar{I}'_n} \cdots$. A program configuration cannot become unstuck by transitions once it is stuck, so that execution is not stuck. By the Program Fair Suffix lemma, that execution is fair. Therefore, there exists a fair specification execution $S^{2''}_2 \xrightarrow{\bar{I}'_1} \cdots \xrightarrow{\bar{I}'_k} \cdots \xrightarrow{\bar{I}'_{n+m}} \cdots \xrightarrow{\bar{I}'_n} \cdots S^{2''}_{n+m+1}$ that single-handlersimulates $\bar{K}'_2 \xrightarrow{\bar{I}'_1} \cdots$.
8. By the Specification Step Commutativity lemma, that specification execution can be rearranged, yielding an execution $S'_1 \rightarrow^{[l]_1} \ldots$, where $S'_1 = S_4$. Because the suffix from $S'_n + m + 1$ is fair, by the Specification Fair Suffix lemma, $S'_1 \rightarrow^{[l]_1} \ldots$ is fair.

9. It remains to show that $K'_i \models_{SH} S'_i$ for all $K'_i$ and $S'_i$ in each execution. For $i > n + m + 1$, this is given by step 7. For all other pairs $\langle K'_i, S'_i \rangle$ such that $0 < i \leq n + m + 1$, the proof is by induction on the number of steps from $K'_n + m + 1$, i.e., on $(n + m + 1) - i$. The base case of $i = n + m + 1$ is also given by step 7. In the inductive case, by the induction hypothesis, $K'_{i+1} \models_{SH} S'_{i+1}$. Then by the Single-Handler Prefix lemma, $K'_i \models_{SH} S'_i$.

\[\Box\]

### D.12 Rearrange Lemma

**Lemma (Rearrange).** For all fair external-representative executions $K_1 \rightarrow^{l_1}_{ER} \ldots$, there exists a fair single-handler execution $K'_1 \rightarrow^{l_1}_{SH} \ldots$ and $\theta$ such that $K'_1 = K_1$ and $\theta$ is a handler-swap-safe index map from $K_1 \rightarrow^{l_1}_{ER} \ldots$ to $K'_1 \rightarrow^{l_1}_{SH} \ldots$.

**Proof.** In every fair external-representative execution, if a given step $K_i \rightarrow^{l_i}_{ER} K_{i+1}$ is not also a single-handler step, then fairness guarantees that the move representing the next single-handler step must occur later in the execution. We construct the single-handler execution by first swapping that step with the steps before it to move it into place. Then we do the same with the second such step, and the third, and so on until the entire execution consists of single-handler steps.

Formally, let $D$ be the set of pairs $\langle K''_1 \rightarrow^{l''_1}_{ER} \ldots, \theta \rangle$ such that $K''_1 = K_1$, $K''_1 \rightarrow^{l''_1}_{ER} \ldots$ is fair and $\theta$ is a handler-swap-safe index map from $K_1 \rightarrow^{l_1}_{ER} \ldots$ to $K''_1 \rightarrow^{l''_1}_{ER} \ldots$. We define a function $\text{Rearrange} : D \rightarrow D$ be a function over such pairs to rearrange the execution into a valid single-handler execution one step at a time.

For a given pair $\langle K''_1 \rightarrow^{l''_1}_{ER} \ldots, \theta \rangle$, if every step $K''_i \rightarrow^{l''_i}_{ER} K''_{i+1}$ in the given execution is also a single-handler step (i.e., $K''_i \rightarrow^{l''_i}_{SH} K''_{i+1}$), then define $\text{Rearrange}(K''_1 \rightarrow^{l''_1}_{ER} \ldots, \theta) = \langle K''_1 \rightarrow^{l''_1}_{ER} \ldots, \theta \rangle$. 
D.12. REARRANGE LEMMA

Otherwise, let \( \tilde{K}''_i \xrightarrow{l''_i} \tilde{K}''_{i+1} \) be the first step in the execution that is not a single-handler step. Let \( a \) be the maximal address according to \( < \) such that an actor with address \( a \) in \( \tilde{K}''_i \) is handling an event (there must be such an address if this step is not a single-handler step). Because the execution is fair, there exists some \( j > i \) such that \( l''_j \) is the first label after \( l''_i \) in which \( a \) is the active actor; this is the step that should come first in a single-handler execution. Because the actor at \( a \) is handling an event, \( l''_j \) must be a handler-continuation label, and because no intermediate step has that actor as the active actor, a step with that label is possible from \( \tilde{K}''_i \). By repeated uses of the Handler Step Commutativity lemma, let \( \tilde{K}''_i l''_i \ldots \) be the execution created after repeatedly swapping that step with the previous steps until it is the \( i \)th step of the execution. Let \( \theta' = \text{id}[i \mapsto i + 1, i + 1 \mapsto i + 2, \ldots, j - 1 \mapsto j, j \mapsto i] \) be the index map from \( \tilde{K}''_i \xrightarrow{l''_i} \tilde{K}''_{i+1} \) to \( \tilde{K}''_i \xrightarrow{l''_j} \ldots \) then the mapping from \( \tilde{K}_1 \xrightarrow{l_1} \ldots \) to \( \tilde{K}'_1 \xrightarrow{l''_1} \ldots \) is provided by \( \theta' \circ \theta \). Because the label \( l''_j \) is a handler-continuation label and was swapped only with labels with different active actors, \( \theta' \) is handler-swap-safe, and therefore so is \( \theta' \circ \theta \). Define \( \text{Rearrange}(\tilde{K}_1 l_1 \ldots, \theta) = \left( \tilde{K}'_1 \xrightarrow{l''_1} \tilde{K}'_2 \xrightarrow{l''_2} \ldots, \theta' \circ \theta \right) \).

No swap in the above definition modifies the initial configuration, so \( \tilde{K}''_1 = \tilde{K}_1 \). By the definition of fair executions, the execution suffix starting at \( \tilde{K}''_{j+1} \) is fair, and so by the Program Fair Suffix lemma, \( \tilde{K}''_1 \xrightarrow{l''_1} \tilde{K}''_{i+1} \) is a fair execution. Therefore \( \text{Rearrange} \) always returns an element of \( D \).

If \( \tilde{K}_1 l_1 \ldots \) is a finite execution with \( n \) steps, then let \( \left( \tilde{K}_1 l_1 \ldots, \theta \right) = \text{Rearrange}^n(\tilde{K}_1 l_1 \ldots, \text{id}_n) \), where \( \text{id}_n \) is the identity function over the domain \( \{1, \ldots, n\} \). Otherwise, define \( \left( \tilde{K}_1 l_1 \ldots, \theta \right) \) to be the limit as \( n \) approaches infinity of \( \text{Rearrange}^n(\tilde{K}_1 l_1 \ldots, \text{id}) \), where \( \text{id} \) is the identity function over the natural numbers.

In the limit, the execution \( \tilde{K}_1 l_1 \ldots \) is well-defined because the first \( n \) steps are defined after \( n \) applications of \( \text{Rearrange} \). To prove that the index map \( \theta \) is well-defined, we must show that for any \( k > 0 \), the final value of \( \theta(k) \) is defined after some finite number of applications of \( \text{Rearrange} \). Let \( n \) be the number of actors handling an event in \( \tilde{K}_1 \), and let \( m \) be the number of handler-start labels in \( l_1, \ldots, l_{h-1} \); then there are at most \( n + m \) handler expressions that must finish before a single-handler step labeled with \( l_h \) is enabled. The labels swapped in front of \( l_h \) during the rearrangement never start new handler expressions (except for actors spawned while handling an event), so every execution from an iteration of \( \text{Rearrange} \) consists of steps from at most \( n + m \) handlers and their spawned actors, followed by a step with label \( l_h \). By the Handler Termination lemma, each
such handler (and its spawned actors) eventually terminates in finitely many single-handler steps, so there exists some iteration of Rearrange in which all higher-priority handlers terminate before the $l_k$ step. Then Rearrange moves any preceding steps of that handler into place, followed by the $l_k$ step. After that point, the $l_k$ step never moves, so $\theta(k)$ is defined in finitely many steps.

By the definition of Rearrange, every step in $\hat{K}_1^l \mathbin{\xrightarrow{l_k}} \hat{\tau}_{\text{ER}} \ldots$ is a single-handler step, i.e., $\hat{K}_1^l \mathbin{\xrightarrow{l_k}} \hat{\tau}_{\text{SH}} \ldots$ is a valid execution. Furthermore, by the definition of $D$, $\hat{K}_1 = \hat{K}_1'$, the execution is fair, and $\theta$ is a handler-swap-safe index map from $\hat{K}_1 \mathbin{\xrightarrow{l_1}} \hat{\tau}_{\text{ER}} \ldots$ to $\hat{K}_1^l \mathbin{\xrightarrow{l_k}} \hat{\tau}_{\text{SH}} \ldots$, which completes the proof. \hfill $\Box$

D.13 Unrearrange Lemma

After rearranging an external-representative execution into a single-handler execution and finding a simulating specification execution, it remains to show how to rearrange that specification execution back into the original order from the external-representative execution. This lemma shows how this is possible.

**Lemma (Unrearrange).** For all external-representative executions $\hat{K}_1 \mathbin{\xrightarrow{l_1}} \hat{\tau}_{\text{ER}} \ldots$, single-handler executions $\hat{K}_1^l \mathbin{\xrightarrow{l_k}} \hat{\tau}_{\text{SH}} \ldots$, specification executions $S'_1 \mathbin{\xrightarrow{\lambda'_1}} \ldots$, and $\theta$, if

- $\hat{K}_1^l \mathbin{\xrightarrow{\lambda'_1}} \hat{\tau}_{\text{SH}} \ldots$ is fair,
- $S'_1 \mathbin{\xrightarrow{\lambda'_1}} \ldots$ is fair,
- $\hat{K}_1 = \hat{K}_1'$,
- $S'_1 \mathbin{\xrightarrow{\lambda'_1}} \ldots$ single-handler-simulates $\hat{K}_1^l \mathbin{\xrightarrow{l_k}} \hat{\tau}_{\text{SH}} \ldots$, and
- $\theta$ is a handler-swap-safe index map from $\hat{K}_1 \mathbin{\xrightarrow{l_1}} \hat{\tau}_{\text{ER}} \ldots$ to $\hat{K}_1^l \mathbin{\xrightarrow{l_k}} \hat{\tau}_{\text{SH}} \ldots$,

then there exists a fair specification execution $S_1 \mathbin{\xrightarrow{\lambda_1}} \ldots$ with $S_1 = S'_1$ that single-handler-simulates $\hat{K}_1 \mathbin{\xrightarrow{l_1}} \hat{\tau}_{\text{ER}} \ldots$.

**Proof.** To construct $S_1 \mathbin{\xrightarrow{\lambda_1}} \ldots$, the idea is to gradually reorder the transitions of $\hat{K}_1^l \mathbin{\xrightarrow{l_k}} \hat{\tau}_{\text{SH}} \ldots$ back into $\hat{K}_1 \mathbin{\xrightarrow{l_1}} \hat{\tau}_{\text{ER}} \ldots$, while at the same time reordering the transitions of $S'_1 \mathbin{\xrightarrow{\lambda'_1}} \ldots$ in the exact same way. Thus, at every intermediate step, the current program execution has a simulating specification execution.

Formally, let $D$ be the set of 3-tuples $\left(\hat{K}_1^l \mathbin{\xrightarrow{l_k}} \hat{\tau}_{\text{ER}} \ldots, S'_1 \mathbin{\xrightarrow{\lambda'_1}} \ldots, \theta\right)$ such that
D.13. UNREARRANGE LEMMA

- $\hat{K}_1'' = \hat{K}_1'$,
- $S_1'' = S_1$,
- $K_1'' \xrightarrow{\lambda_1''} \ldots$ and $S_1'' \xrightarrow{\lambda_1''} \ldots$ are both fair,
- $S_1'' \xrightarrow{\lambda_j''} \ldots$ single-handler-simulates $K_1'' \xrightarrow{\lambda_1''} \text{SH} \ldots$, and
- $\theta$ is a handler-swap-safe index map from $\hat{K}_1 \xrightarrow{\lambda_1} \ldots$ to $\hat{K}_1'' \xrightarrow{\lambda_1''} \ldots$.

A function Unrearrange is responsible for taking a member of $D$ with $n$ steps of the program execution in the expected order and turning it into a tuple with $n + 1$ steps in the expected order. Thus, applying Unrearrange repeatedly to any member of $D$ eventually reorders the program execution into $\hat{K}_1 \xrightarrow{\lambda_1} \ldots$, while adjusting the associated specification execution to match.

D.13.1 Definition of Unrearrange

Formally, let $\text{Unrearrange} : D \rightarrow D$ be defined as follows. If $\theta$ is the identity function, then the execution must already be in the proper order, so $\text{Unrearrange}(\hat{K}_1'' \xrightarrow{\lambda_1''} \ldots, S_1'' \xrightarrow{\lambda_j''} \ldots) = \left( \hat{K}_1'' \xrightarrow{\lambda_1''} \ldots, S_1'' \xrightarrow{\lambda_j''} \ldots \right)$, otherwise, we must reorder the transitions in $\hat{K}_1'' \xrightarrow{\lambda_1''} \ldots$, and $S_1'' \xrightarrow{\lambda_j''} \ldots$, as described below.

Let $i$ be the least index such that $\theta(i) \neq i$. Then $(l_1'', \ldots, l_{i-1}'') = (\hat{l}_1, \ldots, \hat{l}_{i-1})$, and because transition labels uniquely determine the configuration reached from a given configuration, $(\hat{K}_1'', \ldots, \hat{K}_i'') = (\hat{K}_1, \ldots, \hat{K}_i)$. Let $j = \theta(i)$; it must be the case that $j > i$. By the definition of $\theta$, we know that $\hat{l}_i = l_j''$ and therefore a transition labeled with $l_j''$ is possible from $\hat{K}_1''$. Furthermore, because $\theta$ is handler-swap-safe, there is no label in $l_i'', \ldots, l_{j-1}''$ whose active actor is the active actor from $l_j''$, so a transition labeled with $l_j''$ is possible from each configuration in $\hat{K}_i'', \ldots, \hat{K}_{j-1}''$, as well. Then by repeated uses of the Handler Step Commutativity lemma, let $\hat{K}_1'' \xrightarrow{\lambda_1''} \ldots$ be the execution created after repeatedly swapping the step $\hat{K}_j'' \xrightarrow{\lambda_j''} \hat{K}_{j+1}''$ with the previous steps until the $i$th step of the execution is labeled with $l_j''$. Let $\theta' = id[i + 1 \mapsto i, \ldots, j \mapsto j - 1, i \mapsto f]$. Then $\theta$ is a handler-swap-safe index map from $\hat{K}_1'' \xrightarrow{\lambda_1''} \ldots$ to $\hat{K}_1'' \xrightarrow{\lambda_1''} \ldots$, and therefore $\theta' \circ \theta$ is a handler-swap-safe index map from $\hat{K}_1 \xrightarrow{\lambda_1} \ldots$ to $\hat{K}_1'' \xrightarrow{\lambda_1''} \ldots$.

For the specification execution, because $\theta$ is handler-swap-safe and the specification execution single-handler-simulates the program execution, the steps labeled by $\lambda_1', \ldots, \lambda_{j-1}'$ are all non-receive steps. Additionally, because we have one
program execution in which a step labeled with \( l''_i \) precedes the step with \( l''_j \), and another in which the \( l''_j \) step precedes the \( l''_i \) step, we know that neither step defines a new marker used by the other step. This satisfies the preconditions for the Specification Step Commutativity lemma, so by repeated uses of that lemma, let \( S''_1 \xrightarrow{\lambda''_i} \ldots \rightarrow S''_i \rightarrow S''_{i+1} \) with the previous steps until the \( i \)th step of the execution is labeled with \( \lambda''_j \).

We must show that these constructed executions and index map satisfy the conditions to be a member of \( D \). The swaps do not change the initial configuration of either execution, so \( \tilde{K}'''_1 = \tilde{K}'_1 \) and \( S''_1 = S_1 \). By the definition of fair executions, the execution suffix starting at \( \tilde{K}'''_{j+1} \) is fair, and so by the Program Fair Suffix lemma, \( \tilde{K}'''_1 \rightarrow \ldots \rightarrow S''_1 \rightarrow \ldots \) is fair. Similarly, the specification execution suffix starting at \( S''_{j+1} \) is fair, so by the Specification Fair Suffix lemma, \( \tilde{K}'''_1 \rightarrow \ldots \rightarrow \theta'' \circ \theta'' \) is a member of \( D \), and \( Unrearrange \) returns that tuple as its result.

**D.13.2 Use of Unrearrange**

If \( \tilde{K}'_1 \xrightarrow{l'_1} \ldots \) is a finite execution with \( n \) steps, then let \( \langle \tilde{K}'_1 \xrightarrow{l'_1} \ldots \xrightarrow{\lambda_1} \ldots , S_1 \xrightarrow{\lambda_1} \ldots , \theta'' \rangle = Unrearrange^n(\tilde{K}'_1 \xrightarrow{l'_1} \ldots \xrightarrow{\lambda_1} \ldots , \theta) \). Otherwise, define \( \langle \tilde{K}'_1 \xrightarrow{l'_1} \ldots \xrightarrow{\lambda_1} \ldots , \theta'' \rangle \) to be the limit as \( n \) approaches infinity of \( Unrearrange^n(\tilde{K}'_1 \xrightarrow{l'_1} \ldots \xrightarrow{\lambda_1} \ldots , \theta) \). The limit is well-defined because the \( i \)th step of each execution and the final value of \( \theta''(i) \) are all defined after the \( i \)th iteration of \( Unrearrange \).

In either case, \( \theta'' \) is the identity function, Then because \( \tilde{K}'''_1 = \tilde{K}'_1 \) and program transition labels uniquely determine the next configuration, \( \tilde{K}'''_i = \tilde{K}_i \) for all corresponding configurations \( \tilde{K}'''_i \) and \( \tilde{K}_i \) in the two executions, and \( l''_i = l_i \) for all corresponding labels \( l''_i \) and \( l_i \) in the two executions. Then by the definition of \( D \), \( S'_1 = S_1 \), and \( S_1 \xrightarrow{\lambda_1} \ldots \) is fair and single-handler-simulates \( \tilde{K}_1 \xrightarrow{l_1} \ldots \), which completes the proof.
D.14 Single-Handler Conformance Theorem

**Theorem** (Single-Handler Conformance). For all $P$ and $\Sigma$, $P \models_{SH} \Sigma$ if and only if $P \models_{ER} \Sigma$

**Proof.** Proving the $P \models_{ER} \Sigma \implies P \models_{SH} \Sigma$ direction is trivial. Every fair, non-stuck single-handler execution is also a fair, non-stuck external-representative execution, so the specification executions used to show $P \models_{ER} \Sigma$ can be reused to show $P \models_{SH} \Sigma$.

For the other direction, let there be $P$ and $\Sigma$ such that $P \models_{SH} \Sigma$. By the definition of $\models_{SH}$, there exists some maximal instantiation $\langle \vec{K}, S \rangle$ of $P$ and $\Sigma$ such that $\vec{K} \models_{SH} S$. It remains to show that $\vec{K} \models_{ER} S$, which we accomplish by showing that the conformance relation $\models_{SH}$ is itself external-representative-conformance-dense.

Let $\langle \vec{K}_1, S_1 \rangle$ be a member of $\models_{SH}$, and let $\vec{K}_1 \xrightarrow{I_1} \cdots$ be a fair, non-stuck execution. By the Rearrange lemma, there exists some fair single-handler execution $\vec{K}'_1 \xrightarrow{I'_1} \cdots$ with $\vec{K}'_1 = \vec{K}_1$ and a handler-swap-safe index map $\theta$ from $\vec{K}_1 \xrightarrow{I_1} \cdots$ to that execution. Then because $\vec{K}_1 \models_{SH} S_1$, there exists a fair specification execution $S'_1 \xrightarrow{\vec{j}_1} \cdots$ that single-handler-simulates $\vec{K}'_1 \xrightarrow{I'_1} \cdots$. Then by the Unrearrange lemma, there exists a fair specification execution $S_1 \xrightarrow{\vec{j}_1} \cdots$ that single-handler-simulates $\vec{K}_1 \xrightarrow{I_1} \cdots$. Therefore, $\models_{SH}$ is external-representative-conformance-dense. \qed
Appendix E

Deterministic-Handler Conformance

Handler expressions have just two sources of non-determinism: the addresses chosen for new actors in spawn steps, and the markers applied in send steps. Both addresses and markers are opaque tokens, however, so as long as a fresh token is chosen at each step, the choice of token cannot affect a program’s execution in a way that changes its conformance to a specification.

This appendix introduces deterministic-handler conformance, which considers only executions in which transition steps select new markers and addresses deterministically. This helps to reduce the state space that the model checker developed in chapter 7 must explore.

The first component of this deterministic-selection scheme is the Mark function for marking values deterministically, defined in figure E.1. The only significant difference from the non-deterministic Markings function from chapter 3 is that Mark uses the minimal unused marker to mark each address (see the first case in figure E.1).

A new program transition relation \(\longrightarrow_R\) encodes the remainder of the determinism. The relation is based on the single-handler transition relation \(\longrightarrow_{\text{SH}}\), but it gives each new spawned actor an address with the minimal identifier \(n\) greater than all other identifiers for actors spawned at that location, and it uses Mark to mark sent and received messages. This relation encompasses all restrictions placed on concrete configurations for the various proof techniques, so it is called the restricted transition relation (hence the letter “R” denoting it).

Definition. The relation \(\longrightarrow_R\) is defined such that \(\tilde{K} \xrightarrow{l} R \tilde{K}'\) if and only if there exist \(\tilde{\beta}, \tilde{\mu}, H,\) and \(\tilde{\rho}\) such that \(\tilde{K} = \langle\langle \tilde{\beta} \mid \tilde{\mu} \mid H \rangle\rangle^{\tilde{\rho}}\) and all of the following conditions hold

- \(\tilde{K} \xrightarrow{\text{SH}} \tilde{K}'\)
\[ \text{Mark}(\bar{v}, H) = \begin{cases} 
\text{Case } \bar{v} = a@H': \\
\langle a@H' \cup \{\eta\}, H \cup \{\eta\} \rangle \\
\text{where } \eta = \min\{\eta' \mid \eta' \in H \text{ and } \eta' > \eta'' \text{ for all } \eta'' \in H'\} 
\text{Case } \bar{v} = n \text{ or } \bar{v} = \text{str}: \\
\langle \bar{v}, H \rangle 
\text{Case } \bar{v} = (\text{variant } t \bar{v}_1 \ldots \bar{v}_m): \\
\langle (\text{variant } t \bar{v}_1' \ldots \bar{v}_m'), H_{n+1} \rangle \\
\text{where } \langle \bar{v}_i', H_{i+1} \rangle = \text{Mark}(\bar{v}_i, H_i) \text{ and } H_1 = H 
\text{Case } \bar{v} = (\text{record } [r_1 \bar{v}_1] \ldots [r_n \bar{v}_n]): \\
\langle (\text{record } [r_1 \bar{v}_1'] \ldots [r_n \bar{v}_n']), H_{n+1} \rangle \\
\text{where } \langle \bar{v}_i', H_{i+1} \rangle = \text{Mark}(\bar{v}_i, H_i) \text{ and } H_1 = H 
\text{Case } \bar{v} = (\text{fold } \tau \bar{v}'): \\
\langle (\text{fold } \tau \bar{v}''), H' \rangle \text{ where } \langle \bar{v}'', H' \rangle = \text{Mark}(\bar{v}', H) 
\text{Case } \bar{v} = (\text{list } \bar{v}_1 \ldots \bar{v}_n): \\
\langle (\text{list } \bar{v}_1' \ldots \bar{v}_n'), H_{n+1} \rangle \\
\text{where } \langle \bar{v}_i', H_{i+1} \rangle = \text{Mark}(\bar{v}_i, H_i) \text{ and } H_1 = H 
\text{Case } \bar{v} = (\text{dict } [\bar{v}_1 \bar{v}_1'] \ldots [\bar{v}_n \bar{v}_n']): \\
\langle (\text{dict } [\bar{v}_1'' \bar{v}_1'''] \ldots [\bar{v}_n'' \bar{v}_n'''']), H_{n+1} \rangle \\
\text{where } \langle \bar{v}_i''', H_i' \rangle = \text{Mark}(\bar{v}_i', H_i), \langle \bar{v}_i'''', H_{i+1} \rangle = \text{Mark}(\bar{v}_i', H_i'), \\
\text{and } H_1 = H 
\end{cases} \]

Figure E.1: Deterministic marking of values with Mark
• If \( \bar{l} = a: \text{spawn}(\text{addr} \ell n) \) for some \( a, \ell, \) and \( n \), then \( n = \min(\{n'| \forall (\text{addr} \ell n') \in \text{dom}(\bar{\beta}). n' > n''\}) \).

• If \( \bar{l} \) is one of
  - \( a: \text{rcv-ext}(H', \bar{\nu}, \tau) \),
  - \( a: \text{rcv-int}(H', \bar{\nu}) \),
  - \( a': \text{send-ext}(a@H', \bar{\nu}) \), or
  - \( a': \text{send-int}(a@H', \bar{\nu}) \)

for some \( a, a', H', \bar{\nu}, \) and \( \tau \) such that \( H' \neq \emptyset \), then there exist \( \bar{\nu}' \) and \( H'' \) such that \( \text{Mark}(\bar{\nu}', H') = \langle \bar{\nu}, H'' \rangle \).

Except for the use of the new transition relation, deterministic-handler conformance is defined just like the previous notions of conformance.

**Definition.** A relation \( R \) is deterministic-handler-conformance-dense if for all \( \langle K_1, S_1 \rangle \in R \) and all fair, non-stuck deterministic-handler executions \( K_1 \xrightarrow{L_1} K_2 \xrightarrow{L_2} \ldots \), there exists a fair specification execution \( S_1 \xrightarrow{L_1} S_2 \xrightarrow{L_2} \ldots \) with the same length such that \( \langle K_i, S_i \rangle \in R \) for all \( K_i \) and \( S_i \) in the respective executions.

**Definition.** A marked program configuration \( K \) deterministic-handler conforms to a specification configuration \( S \), written \( K \models_D S \), if there exists a deterministic-handler-conformance-dense relation \( R \) such that \( \langle K, S \rangle \in R \). As with previous relations, \( P \models_D \Sigma \) if there exists some maximal instantiation \( \langle K, S \rangle \) of \( P \) and \( \Sigma \) such that \( K \models_D S \).

Deterministic-handler conformance is equivalent to single-handler conformance, as stated in the following theorem.

**Theorem** (Deterministic-Handler Conformance). For all \( P \) and \( \Sigma \), \( P \models_D \Sigma \) if and only if \( P \models_{SH} \Sigma \).

**Proof.** The \( P \models_{SH} \Sigma \Rightarrow P \models_D \Sigma \) direction is trivial. Every fair, non-stuck deterministic-handler execution is also a fair, non-stuck single-handler execution, so the specification executions used to show \( P \models_{SH} \Sigma \) can be reused to show \( P \models_D \Sigma \).

For the other direction, let there be \( P \) and \( \Sigma \) such that \( P \models_D \Sigma \). By the definition of \( \models_D \), there exists some maximal instantiation \( \langle K_{init}, S_{init} \rangle \) of \( P \) and \( \Sigma \) such that \( K_{init} \models_D S_{init} \). It remains to show that \( K_{init} \models_{SH} S_{init} \), which we accomplish by defining a single-handler-conformance-dense relation \( R \) that contains \( \langle K_{init}, S_{init} \rangle \).

As in the previous appendix, let \( \sigma_A \) stand for a partial address-to-address function, and let \( \sigma_M \) stand for a partial marker-to-marker function. The application \( \sigma_A(\bar{K}) \) of such an address substitution \( \sigma_A \) to a program configuration \( \bar{K} \) is defined as the obvious element-wise, component-wise, point-wise application to
its parts, and similarly for a marker substitution \( \sigma_M \). The application is undefined if \( \sigma_A(a) \) or \( \sigma_M(\eta) \) is undefined for any address \( a \) or marker \( \eta \) appearing in \( K \). The same is true for the application of \( \sigma_M \) to a specification configuration \( S \).

An address-substitution function \( \sigma_A \) is location-preserving if and only if for all \((\text{addr} \ \ell \ n) \in \text{dom}(\sigma_A)\) there exists \( n' \) such that \( \sigma_A((\text{addr} \ \ell \ n)) = (\text{addr} \ \ell \ n') \).

Let \( R \) be the set of pairs \( \langle K, S \rangle \) such that there exist \( K', S', \sigma_A, \) and \( \sigma_M \) such that

- \( \sigma_A \) is location-preserving and one-to-one,
- \( \sigma_M \) is one-to-one,
- \( \sigma_A(\sigma_M(K)) = K' \),
- \( \sigma_M(S) = S' \), and
- \( \bar{K}' \models_D S' \).

Clearly \( \langle K, S \rangle \in R \), with the identity function as both the address substitution \( \sigma_A \) and the marker substitution \( \sigma_M \). It remains to show that \( R \) is single-handler-conformance-dense.

Let \( \langle K_1, S_1 \rangle \) be a member of \( R \), and let \( K_1 \xrightarrow{l_1} \cdots \xrightarrow{l_k} S_1 \) be a fair, non-stuck single-handler execution; we must provide some simulating fair specification execution. Let there be \( K'_1, S'_1, \sigma_A, \) and \( \sigma_M \) that prove the membership of \( \langle K_1, S_1 \rangle \) in \( R \). Without loss of generality, assume that \( \sigma_A \) is restricted to the set of addresses appearing in \( K_1 \), and that \( \sigma_M \) is restricted to the set of markers appearing in \( K_1 \) and \( S_1 \). Then because the \( \xrightarrow{R} \) relation is just like \( \xrightarrow{SH} \) except that it allocates new addresses and markers deterministically, it is easy to see that there exists a similar fair deterministic-handler execution \( K'_1 \xrightarrow{l'_1} \cdots \xrightarrow{l'_k} S'_1 \), where the only differences between the two executions are the initial differences defined by \( \sigma_A \) and \( \sigma_M \) and the new address and markers chosen in each step. The condition that \( \sigma_A \) be location-preserving ensures that any transition rules that depend on an address being internal or external work the same way in both executions.

Thus, there exist new substitutions \( \sigma'_A \) and \( \sigma'_M \) that account for these new address and markers, where \( \text{dom}(\sigma_A) \cap \text{dom}(\sigma'_A) = \emptyset \) and \( \text{dom}(\sigma_M) \cap \text{dom}(\sigma'_M) = \emptyset \), \( \sigma'_A \) and \( \sigma'_M \) are both one-to-one, and \( \sigma_A \) is location-preserving. Then for all corresponding configurations \( K_i \) and \( K'_i \) in the two executions, we have \( (\sigma_A \circ \sigma'_A)(\sigma_M \circ \sigma'_M)(K_i) = K'_i \). Similarly, for all corresponding labels \( l_i \) and \( l'_i \) in the two executions, we have \( (\sigma_A \circ \sigma'_A)(\sigma_M \circ \sigma'_M)(l_i) = l'_i \).

Because \( K'_1 \models_D S'_1 \), there exists a fair specification execution \( S_1 \xrightarrow{l_1} \cdots \) with the same length such that \( K'_i \models_D S'_i \) for all \( K'_i \) and \( S'_i \) in the respective executions. Then because \( \sigma'_M \) accounts for all new markers introduced by the labels of the execution, there also exists a similar fair specification execution \( S_1 \xrightarrow{l_1} \cdots \) such that \( (\sigma_M \circ \sigma'_M)(S_i) = S'_i \) for all corresponding configurations \( S_i \) and \( S'_i \) in the two executions.
Therefore, for each corresponding pair \( \tilde{K}_i \) and \( S_i \), there exist some \( \tilde{K}'_i \) and \( S'_i \) such that

- \( \sigma_A \cup \sigma'_A \) is location-preserving and one-to-one,
- \( \sigma_M \cup \sigma'_M \) is one-to-one,
- \((\sigma_A \cup \sigma'_A)(\sigma_M \cup \sigma'_M)(\tilde{K}_i) = \tilde{K}'_i\),
- \((\sigma_M \cup \sigma'_M)(S_i) = S'_i\), and
- \( \tilde{K}'_i \models_D S'_i \).

Therefore \( (\tilde{K}_i, S_i) \in R \), and therefore \( R \) is single-handler-conformance-dense. \( \square \)
APPENDIX E. DETERMINISTIC-HANDLER CONFORMANCE
Appendix F

Event-Step Conformance

All notions of conformance defined so far say that for a specification execution $S_1 \xrightarrow{[f_1]} \ldots$ to simulate a program execution $K_1 \xrightarrow{l_1} \ldots$, every corresponding pair of configurations $\langle K_i, S_i \rangle$ must be in an appropriate conformance relation $R$. Writing a proof of conformance for such a relation would be incredibly tedious, and would place a heavy burden of proof on any automated model checker.

As it turns out, however, it is sufficient to show membership only for those pairs corresponding to the moments just after one event handler finishes, but before another one begins. Intuitively, this is because given the restrictions included in the $\xrightarrow{R}$ transition relation, the only remaining non-determinism is the choice of which event to handle after the previous event handler terminates. Therefore, it is enough for the specification to show only that it has a matching sequence of transitions for each complete event-handler execution. A new conformance relation defined in this section, called event-step conformance, takes advantage of this idea.

Event-step conformance first requires a new transition relation $\rightarrow$, called the event-step relation. A single step $K \xrightarrow{l_1, \ldots, l_n} K'$ of the relation represents $K$ taking some sequence of steps labeled $l_1, \ldots, l_n$ to handle some event and reach a new configuration $K'$.

**Definition.** The relation $\rightarrow$ is defined such that $K_1 \xrightarrow{l_1, \ldots, l_n} K_{n+1}$ if and only if there exist $K_2, \ldots, K_n$ such that

- $K_1 \xrightarrow{l_1} \ldots \xrightarrow{l_n} K_{n+1}$,
- $n > 0$,
- some actor in $K_i$ is handling an event for all $i \in 2 \ldots n$, and
- no actor in $K_{n+1}$ is handling an event.

An event-step execution is a sequence $K_1 \xrightarrow{l_{1_1}, \ldots, l_{1_{n_1}}} K_2 \xrightarrow{l_{2_1}, \ldots, l_{2_{n_2}}} \ldots$ of such steps, and such an execution is stuck if any configuration $K_i$ in the execu-
tion is stuck (since the reached configuration must have no actor handling an event, this effectively means that only the zero-step execution starting with a stuck configuration is stuck). The transition labels uniquely identify the “next” configuration for each individual restricted transition, so an event-step execution is fair if and only if the restricted-transition execution it denotes is fair.

A similar relation is defined for specification configurations, which merely combines multiple \( \rightarrow \) transitions into one.

**Definition.** The relation \( \rightarrow \) is defined such that \( S_1 \xrightarrow{\lambda_1, O_1} \ldots \xrightarrow{\lambda_n, O_n} S_{n+1} \) if and only if there exist \( S_2, \ldots, S_n \) such that \( S_1 \xrightarrow{\lambda_1, O_1} \ldots \xrightarrow{\lambda_n, O_n} S_n \rightarrow S_{n+1} \).

As with the \( \rightarrow \) relation, the obligations \( O_i \) can be dropped where irrelevant, so that \( S_1 \xrightarrow{\lambda_1, O_1} \ldots \xrightarrow{\lambda_n, O_n} S_{n+1} \) is written as \( S_1 \xrightarrow{\lambda_1, \ldots, \lambda_n} S_{n+1} \). An execution \( S_1 \xrightarrow{\lambda_1, O_1} \ldots \xrightarrow{\lambda_n, O_n} S_{n+1} \) of such transitions is fair if and only if for all \( S_i \) in the execution and all \( \langle \eta, po \rangle \in \text{Obls}(S_i) \), there exists a step \( S_i \xrightarrow{\lambda_{i+1}, O_{i+1}, 1} \ldots \xrightarrow{\lambda_{n+1}, O_{n+1}, 1} S_{i+j+1} \) in the execution and some \( k \leq n_{i+j} \) such that \( \langle \eta, po \rangle \in O_{i+j,k} \).

Event-step conformance is then defined in terms of event steps rather than individual transition steps.

**Definition.** A relation \( R \) is event-step-conformance-dense if and only if for all \( \langle \tilde{K}_1, S_1 \rangle \in R \) and all fair, non-stuck executions \( \tilde{K}_1 \xrightarrow{\lambda_1, \ldots, \lambda_n} \ldots \), there exists a fair specification execution \( S_1 \xrightarrow{\lambda_1, \ldots, \lambda_n} \ldots \) with the same length such that for all \( \tilde{K}_i \) and \( S_i \) in the respective executions, \( \langle \tilde{K}_i, S_i \rangle \in R \).

**Definition.** A marked program configuration \( \tilde{K} \) event-step conforms to a specification configuration \( S \), written \( \tilde{K} \models_{\text{EV}} S \), if and only if there exists an event-step-conformance-dense relation \( R \) such that \( \langle \tilde{K}, S \rangle \in R \). Furthermore, \( P \models_{\text{EV}} \Sigma \) if and only if there exists some maximal instantiation \( \langle \tilde{K}, S \rangle \) of \( P \) and \( \Sigma \) such that \( \tilde{K} \models_{\text{EV}} S \).

Event-step conformance is equivalent to deterministic-handler conformance, as stated in the following theorem.

**Theorem** (Event-Step Conformance). For all \( P \) and \( \Sigma \), \( P \models_{\text{EV}} \Sigma \) if and only if \( P \models_{\text{D}} \Sigma \).

**Proof.** The proof is divided into two parts, one for each direction of the if-and-only-if.

**Part 1:** \( P \models_{\text{D}} \Sigma \implies P \models_{\text{EV}} \Sigma \)

By the definition of \( \models_{\text{D}} \), there must be some \( \tilde{K}, S \), and \( R \) such that \( \tilde{K} \) and \( S \) are a maximal instantiation of \( P \) and \( \Sigma \), and \( R \) is a deterministic-handler-conformance-dense relation such that \( \langle \tilde{K}, S \rangle \in R \). We will show that \( R \) is event-step-conformance-dense, and that therefore \( P \models_{\text{EV}} \Sigma \).
Let \( \langle \tilde{K}_1, S_1 \rangle \) be a member of \( R \), and let \( \tilde{K}_1 \xrightarrow{I_1} \ldots \) be a fair, non-stuck event-step execution. By the definition of \( \xrightarrow{R} \), we can rewrite this as \( \tilde{K}_{1,1} \xrightarrow{I_{1,1}} \tilde{K}_{1,2} \xrightarrow{I_{1,2}} \ldots \xrightarrow{I_{1,n}} \tilde{K}_{1,1} \xrightarrow{I_{2,1}} \ldots \), where \( \tilde{K}_{i,1} = \tilde{K}_i \) for all \( \tilde{K}_i \) in the event-step execution. Then because \( \langle \tilde{K}_1, S_1 \rangle \in R \), there exists a fair specification execution \( S_1 \xrightarrow{\lfloor I_{1,1} \ldots I_{1,n} \rfloor} \ldots \) with the same length such that \( \langle \tilde{K}_i, S_i \rangle \in R \) for all \( \tilde{K}_i \) and \( S_i \) in the respective executions. By the definition of a specification event step, we can rewrite that execution as a fair execution \( S_{1,1} \xrightarrow{I_{1,1}} S_{1,2} \xrightarrow{I_{1,2}} \ldots \xrightarrow{I_{1,n}} \ldots \xrightarrow{I_{2,1}} \ldots \), where \( S_{i,1} = S_i \) for all \( S_i \) in the specification event-step execution. Therefore, \( R \) is event-step conformance-dense, and \( \Sigma \models_{EV} \Sigma \).

**Part 2:** \( \Sigma \models_{EV} \Sigma \implies \Sigma \models_{D} \Sigma \)

Let there be \( \Sigma \) such that \( \Sigma \models_{EV} \Sigma \). By the definition of \( \models_{EV} \), there exists some maximal instantiation \( \tilde{K} \) and \( S \) of \( \Sigma \) such that \( \tilde{K} \models_{EV} S \). We will show that \( \Sigma \models_{EV} \Sigma \) is deterministic-handler-conformance-dense, and that therefore \( \tilde{K} \models_{D} S \).

Let \( \langle \tilde{K}_i, S_i \rangle \) be a member of \( \models_{EV} \), and let \( \tilde{K}_1 \xrightarrow{I_1} \ldots \) be a fair, non-stuck execution. We must provide some fair specification execution \( S_1 \xrightarrow{I_1} \ldots \) with the same length such that for all \( \tilde{K}_i \) and \( S_i \) in the respective executions, \( \tilde{K}_i \models_{EV} S_i \).

For every configuration \( \tilde{K}_i \) in that execution, there are only a finite number of actors handling an event in \( \tilde{K}_i \). By the Handler Termination lemma from appendix D, every event handler eventually terminates, and the \( \xrightarrow{R} \) relation does not allow another actor to start handling an event unless no other actor is handling an event. Therefore, because the execution \( \tilde{K}_1 \xrightarrow{I_1} \ldots \) is fair, all running event handlers in the execution eventually terminate, so for each such \( \tilde{K}_i \) there exists some \( \tilde{K}_{i+1} \) in the execution such that no actor in \( \tilde{K}_{i+1} \) is handling an event. Therefore, the execution can be rewritten as a fair event-step execution \( \tilde{K}_1 \xrightarrow{I_{1,1} \ldots I_{1,n}} \ldots \) such that \( \tilde{K}_1 = \tilde{K}_i \), the overall sequence of labels is the same, and the configurations \( \tilde{K}_2, \tilde{K}_3, \ldots \) are the configurations from \( \tilde{K}_2 \xrightarrow{I_2} \ldots \) such that no actor in \( \tilde{K}_i \) is handling an event.

Because \( \tilde{K}_1 \models_{EV} S_1 \), there exists a fair specification execution \( S'_1 \xrightarrow{\langle I_{1,1}, O_{1,1} \rangle \ldots \langle I_{1,n}, O_{1,n} \rangle} \ldots \) with the same length as \( \tilde{K}_1 \xrightarrow{I_{1,1} \ldots I_{1,n}} \ldots \) such that \( S'_1 = S_1 \) and for all \( \tilde{K}'_j \) and \( S'_j \) in the respective executions, \( \tilde{K}'_j \models_{EV} S'_j \).

Let \( S_1 \xrightarrow{I_1, O_1} \ldots \) be some execution that \( S'_1 \xrightarrow{\langle I_{1,1}, O_{1,1} \rangle \ldots \langle I_{1,n}, O_{1,n} \rangle} \ldots \) denotes. To show that that execution is fair, let \( S_i \) be a configuration in that execution, and let \( \langle \eta, po \rangle \) be an obligation in \( Obls(S_i) \). We must show that the obligation is eventually fulfilled. Let \( S_{i+j} \) be the first configuration after \( S_i \) that corresponds to some \( S'_{i+j} \) in the event-step execution. If \( \langle \eta, po \rangle \in Obls(S_{i+j}) \), then because the event-step execution is fair, we know that obligation is eventually fulfilled. Oth-
erwise, the only way that $\langle \eta, \text{po} \rangle$ would not be in \( \text{Obls}(S_{i+j}) \) is that it was fulfilled in some earlier transition (i.e., $\langle \eta, \text{po} \rangle \in O_{i+k}$ for some $k < j$). Therefore, every obligation in $S_1 \xrightarrow{i_1, O_1} \ldots$ is eventually fulfilled, and therefore $S_1 \xrightarrow{i_1, O_1} \ldots$ is fair.

Finally, let $K_i$ and $S_i$ be some corresponding pair of configurations from $K_1 \xrightarrow{l_1 \in R} \ldots$ and $S_1 \xrightarrow{l_1 \in O_1} \ldots$, respectively. We must show that $K_i \equiv_{\text{EV}} S_i$. If no actor in $K_i$ is handling an event, then the configuration $K_i$ corresponds to some $K'_j$ in $K'_1 \xrightarrow{i_{1,1}, \ldots, l_{1,n}} \ldots$, and therefore we already know that $K_i \equiv_{\text{EV}} S_i$.

Otherwise, let $K''_1 \xrightarrow{l'_{1,1}, \ldots, l'_{1,n}} \ldots$ be a fair, non-stuck execution such that $K''_1 = K_i$. Because at least one actor in $K''_1$ is handling an event and the execution is fair, it contains at least one step $K''_1 \xrightarrow{l''_{1,1} \ldots l''_{1,n}} K''_2$. Then there exist some $K''_1, \ldots, K''_{n+1}$ such that $K''_1 = K''_1'$, $K''_{n+1} = K''_2'$, $K''_1$ $\xrightarrow{l''_{1,1} \ldots l''_{1,n}} \ldots$ $\xrightarrow{R} K''_{n+1}$: there is at least one actor handling an event in $K''_i$ for all $i \in 1 \ldots n$, and no actor is handling an event in $K''_{n+1}$. Then because the $\xrightarrow{R}$ relation completely determines the next transition for any configuration with actors handling an event, the configurations $K''_1, \ldots, K''_{n+1}$ are the same as $K_i, \ldots, K_{i+n}$ and the labels $l''_{1,1}, \ldots, l''_{1,n}$ are the same as $l_{1,1}, \ldots, l_{1,n-1}$. Then let $S''_j = S_{i+j-1}$ for all $j \in 2 \ldots n + 1$, and we have $S''_1 \xrightarrow{l''_{1,1}} \cdots \xrightarrow{l''_{1,n}} S''_{n+1}$, and therefore $S''_1 \xrightarrow{l''_{1,1} \cdots l''_{1,n}} S''_{n+1}$. Because no actor in $K''_{n+1}$ is handling an event, we know that $K''_{n+1} \equiv_{\text{EV}} S''_{n+1}$. Let $S''_2 = S''_{n+1}$; then we know that there exists some fair specification execution $S''_2 \xrightarrow{l''_{2,1} \cdots l''_{2,n}} \ldots$ with the same length as $K''_1 \xrightarrow{l''_{1,1} \cdots l''_{1,n}} \ldots$ such that $K''_i \equiv_{\text{EV}} S''_i$ for all corresponding pairs $K''_i$ and $S''_i$ in those executions. Then because its suffix starting from $S''_2$ is fair, the execution $S''_1 \xrightarrow{l''_{1,1} \cdots l''_{1,n}} \ldots$ (where $S''_1 = S'_1$) is fair. Therefore, $K_i \equiv_{\text{EV}} S_i$. \qed
Appendix G

PSM Conformance

Although conformance is most easily defined in terms of a collection of PSMs, it simplifies conformance proofs to consider just a single PSM at a time. Indeed, after a PSM is forked from its parent, it has no way of interacting with any other existing PSMs in the configuration, so it acts as an independent specification of the program’s behavior. This next technique, called PSM conformance, splits a specification configuration into its constituent PSMs after simulating each event step, then checks that the program conforms to each individual PSM rather than the specification configuration as a whole.

A PSM-conformance-dense relation, relates a program configuration to a single PSM, rather than to an entire specification configuration. Its definition below is otherwise similar to an event-step-conformance-dense relation.

Definition. A relation $R$ is **PSM-conformance-dense** if and only if for all $\langle \tilde{K}, s \rangle \in R$ and all fair, non-stuck executions $\tilde{K} \xrightarrow{I_{1,1}, \ldots, I_{1,n}} \ldots$, there exists a fair specification execution $S_1 \xrightarrow{I_{1,1}, \ldots, I_{1,n}} \ldots$ with the same length such that $S_1 = \{s\}$ and for all $\tilde{K}$ and $S_1$ in the respective executions and all $s' \in S_1$, $\langle \tilde{K}, s' \rangle \in R$.

PSM-conformance is then defined on top of this relation similar to other notions of conformance.

Definition. A marked program configuration $\tilde{K}$ **PSM-conforms** to a PSM $s$, written $\tilde{K} \models_{PSM} s$, if and only if there exists a PSM-conformance-dense relation $R$ such that $\langle \tilde{K}, s \rangle \in R$, and a program $P$ **PSM-conforms** to a specification $\Sigma$, written $P \models_{PSM} \Sigma$, if and only if there exists some maximal instantiation $\langle \tilde{K}, \{s\} \rangle$ of $P$ and $\Sigma$ such that $\tilde{K} \models_{PSM} s$.

The remainder of this appendix provides the definitions, lemmas, and proofs used to show that PSM conformance is equivalent to event-step conformance.
G.1 Definitions

For a PSM \( s = (H, H', \eta, \phi, \Phi) \), section 3.6.5 defined \( \text{OutMon}(s) \) as that PSM’s set of output-monitored markers, i.e., \( \text{OutMon}(s) = H' \). Similarly, \( \text{InMon}(s) \) is defined as that PSM’s set of input-monitored markers, i.e., \( \text{InMon}(s) = H \). The function \( \text{Mon}(s) \) defines all of a PSM’s monitored markers, i.e., \( \text{Mon}(s) = \text{OutMon}(s) \cup \text{InMon}(s) \).

We say that all PSMs in a specification configuration \( S \) monitor distinct markers if and only if for all \( s \in S \) and \( s' \in S \) such that \( s \neq s' \), \( \text{Mon}(s) \cap \text{Mon}(s') = \emptyset \).

It is sometimes useful to talk about the markers monitored by all of the PSMs of an entire specification configuration, so we also define

- \( \text{OutMon}(S) = \bigcup_{s \in S} \text{OutMon}(s) \),
- \( \text{InMon}(S) = \bigcup_{s \in S} \text{InMon}(s) \), and
- \( \text{Mon}(S) = \bigcup_{s \in S} \text{Mon}(s) \).

A PSM \( (H, H', \eta_1, \ldots, \eta_n, \phi, \langle \eta'_1, \rho_1 \rangle, \ldots, \langle \eta'_m, \rho_m \rangle) \) is well-formed if and only if \( \{\eta_1, \ldots, \eta_n, \eta'_1, \ldots, \eta'_m\} \subseteq H' \), i.e., it output-monitors all of its state arguments and obligation destinations. A specification configuration \( S \) is well-formed if and only if all of its constituent PSMs are well-formed.

The function \( \text{Matchable} \) defines the markers from a label \( \lambda \) that can possibly be matched by some pattern. It is defined as follows.

\[
\text{Matchable}(\lambda) =
\begin{cases}
\{ \eta \mid \exists \pi, \eta, \eta' \in \eta \text{ and } \pi \rightarrow [x \mapsto \eta'] \text{ and } \eta \in \{\eta'\} \} & \text{if } \lambda = a'@H'?\emptyset \\
\{ \eta \mid \exists \rho, H, S. \emptyset \sim \rho \triangleright H, S \text{ and } \eta \in H \cup \text{Mon}(S) \} & \text{if } \lambda = a'@H'!\emptyset \\
\emptyset & \text{if } \lambda = \lambda \\
\end{cases}
\]

G.2 Concrete Specification Well-Formed Preservation

Lemma (Concrete Specification Well-Formed Preservation). For all \( S, S', \) and \( \lambda \), if \( S \) is well-formed and \( S \xrightarrow{\lambda} S' \), then \( S' \) is well-formed.

Proof. The proof is nearly identical to the proof of the Specification Well-Formed Preservation lemma (appendix K), except using concrete pattern-matching rules rather than abstract ones. \( \square \)

G.3 Concrete Distinct-Marker Preservation

Lemma (Concrete Distinct-Marker Preservation). For all \( S, S', \) and \( \lambda \), if

- \( S \xrightarrow{\lambda} S' \),
• S is well-formed,
• all of the PSMs in S monitor distinct markers,
• if there exist a, H, and ˘v such that λ = a@H?˘v or λ = a@H!˘v, then |H| = 1 and no marker in Matchable(λ) appears more than once in ˘v,
• Matchable(λ) ∩ Mon(S) = ∅
then all of the PSMs in S′ monitor distinct markers.


G.4 PSM Conformance Theorem

Theorem (PSM Conformance). For all P and Σ, P ⊨PSM Σ if and only if P ⊨EV Σ.

Proof. The proof is divided into two parts, one for each direction of the if-and-only-if.

Part 1: P ⊨PSM Σ ⇒ P ⊨EV Σ

By the definition of ⊨PSM, there exists some maximal instantiation ˘K and S = {s} of P and Σ and some PSM-conformance-dense relation RPSM such that ˘K,s ∈ RPSM. Let REV be defined as follows.

\[ R_{EV} = \{ (˘K',S') | (˘K',s') ∈ R_{PSM} \text{ for all } s' ∈ S' \} \]

Clearly ˘K,S ∈ R EV. It remains to show that R EV is event-step-conformance-dense.

Let ˘K1,S1 be a member of R EV, where S1 = {s1, ..., sn}, and let ˘K1 \[ i_1 \cdots i_m \] ... be a fair, non-stuck event-step execution. By the definition of a PSM-conformance-dense relation, for all i ∈ 1...n, there exists a fair specification execution S' i \[ i_{1,1} \cdots i_{1,m} \] ... with the same length such that S' i,1 = {si}, and for all ˘K j and S' j in the respective executions and all s' ∈ S' i,j, ˘K j,s' ∈ RPSM. Because the presence of a PSM does not affect the transitions of another PSM in the same configuration (see the specification configuration transition rules in section 3.6.7), we can combine those executions of the individual PSMs s1, ..., sn into a single simulating execution S \[ i_{1,1} \cdots i_{1,m} \] ... such that S j = \[ \cup_{i \in 1...n} S' i,j \]. Every PSM in that execution still fulfills all of its obligations, so the execution is fair. Furthermore, for all ˘K j and S j in the respective executions and all s' ∈ S j, ˘K j,s' ∈ RPSM. Therefore, ˘K j,S j ∈ R EV by the definition of R EV. Therefore, R EV is event-step-conformance-dense.
APPENDIX G. PSM CONFORMANCE

296

Part 2: \( P \models_{EV} \Sigma \implies P \models_{PSM} \Sigma \)

The proof for this direction is similar to the opposite direction, except we require some additional conditions to ensure that the obtained specification execution is fair.

By the definition of \( \models_{EV} \), there exists some maximal instantiation \( \bar{K} \) and \( S = \{s\} \) of \( P \) and \( \Sigma \) and some PSM-conformance-dense relation \( R_{EV} \) such that \( \langle \bar{K}, S \rangle \in R_{EV} \). Let \( R_{PSM} \) be defined as the set of pairs \( \langle \bar{K}, S \rangle \) such that there exists some \( S \) such that

- \( s \in S \),
- \( \langle \bar{K}, S \rangle \in R_{EV} \),
- \( \bar{K} \) is an externals-only configuration,
- \( S \) is well-formed, and
- all of the PSMs in \( S \) monitor distinct markers.

The configuration \( \bar{K} \) is an externals-only configuration by the definition of \( \text{Inst} \), so \( \langle \bar{K}, S \rangle \in R_{PSM} \). It remains to show that \( R_{PSM} \) is PSM-conformance-dense.

Let \( \langle \bar{K}_1, s' \rangle \) be a member of \( R_{PSM} \), and let \( \bar{K}_1 \xrightarrow{I_{1,1},\ldots,I_{1,m}} \ldots \) be a fair, non-stuck event-step execution. Let \( S_1 = \{s''_1,\ldots,s''_n\} \) be a specification configuration that proves the membership of \( \langle \bar{K}_1, s' \rangle \) in \( R_{PSM} \). By the definition of an event-step-conformance-dense relation, there exists a fair specification execution \( S_1 \xrightarrow{[I_{1,1},\ldots,I_{1,m}]} \ldots \) with the same length as \( \bar{K}_1 \xrightarrow{I_{1,1},\ldots,I_{1,m}} \ldots \) such that for all \( \bar{K}_j \) and \( S_j \) in the respective executions, \( \langle \bar{K}_j, S_j \rangle \in R_{EV} \).

Because the presence of a PSM does not affect the transitions of another PSM in the same configuration (see the specification configuration transition rules in section 3.6.7), the execution \( S_1 \xrightarrow{[I_{1,1},\ldots,I_{1,m}]} \ldots \) can be split into \( n \)-many executions of the form \( S'_{1,i} \xrightarrow{[I_{1,1},\ldots,I_{1,m}]} \ldots \) for \( i \in 1\ldots n \), where \( S'_{1,i} = \{s''_i\} \) for all \( i \in 1\ldots n \) and \( S_j = \bigcup_{i \in 1\ldots n} S'_{i,j} \) for all \( S_j \) in the first execution.

Let there be \( k \in 1\ldots n \) such that \( s' = s''_k \). We must show that \( S'_{k,1} \xrightarrow{[I_{1,1},\ldots,I_{1,m}]} \ldots \) is fair, using the extra conditions required by \( R_{PSM} \). Because \( \bar{K}_1 \) is an externals-only configuration, by the Externals-Only Preservation lemma (appendix I), every configuration \( \bar{K}_j \) in \( \bar{K}_1 \xrightarrow{I_{1,1},\ldots,I_{1,n}} \ldots \) is externals-only as well. Therefore for every receptionist or external address \( a@H \) appearing in each \( \bar{K}_j \), \( |H| = 1 \). Furthermore, if any of the labels \( I_j \) is sending or receiving a message \( \bar{v} \) to or from the environment, then every matchable marker on \( \bar{v} \) is a fresh marker by the definition of the program transition rules. Finally, because \( S_1 \) is well-formed, by the Concrete Specification Well-Formed Preservation lemma, each \( S_j \) in \( S_1 \xrightarrow{[I_{1,1},\ldots,I_{1,n}]} \ldots \) is well-formed. Therefore, by the Concrete Distinct-Marker Preservation lemma, for every \( S_j \) in \( S_1 \xrightarrow{[I_{1,1},\ldots,I_{1,n}]} \ldots \), all of the PSMs in \( S_j \)
monitor distinct markers. A well-formed specification configuration can only fulfill obligations for an obligation it monitors, so for any $S'_{k,j}$ in $S'_{k,1} \overset{[l_{1,1} \ldots l_{1,m}]}{\rightarrow} \ldots$ and any $\langle \eta, po \rangle \in Obls(S'_{k,j})$, because that obligation is eventually fulfilled in $S_1 \overset{[l_{1,1} \ldots l_{1,m}]}{\rightarrow} \ldots$, it must also be fulfilled in $S'_{k,1} \overset{[l_{1,1} \ldots l_{1,m}]}{\rightarrow} \ldots$, so the latter execution is fair.

Finally, we must show that for all corresponding pairs $\bar{K}_j$ and $S'_{k,j}$ in those executions, and all $s'' \in S'_{k,j}$, $\langle \bar{K}_j, s'' \rangle \in R_{PSM}$. We know that $S'_{k,j} \subseteq S_j$, so $s'' \in S_j$. We have already shown that $\langle \bar{K}_j, S_j \rangle \in R_{EV}$, $\bar{K}_j$ is an externals-only configuration, $S_j$ is well-formed, and all of the PSMs in $S_j$ monitor distinct markers. Therefore, $\langle \bar{K}_j, s'' \rangle \in R_{PSM}$ and $R_{PSM}$ is PSM-conformance-dense. \qed
Appendix H

Type System for Abstract CSA

The type system for abstract program configurations is largely similar to the one for concrete configurations. This appendix lists the various type rules. The main difference from the concrete type system is the introduction of a notion of depth. Appendix N explains how the depth is used to prove termination for the ModelCheck algorithm. The symbol \(d\) ranges over depths, which are merely natural numbers. For all of the judgments defined here, the depth can be dropped where irrelevant; e.g., \(\vdash_{cfg} \overset{K}{\hat{\rho}}\) means there exists some \(d\) such that \(\vdash_{cfg} \overset{K}{\hat{\rho}} : d\).

H.1 Abstract Program Configurations

In addition to passing the expected depth \(d\) to all lower-level judgments, this rule also checks that every receptionist type and the type of every address appearing in the configuration has a depth bounded by \(d\). This helps to ensure that only a bounded number of receptionist types and external addresses can appear in any execution starting from this configuration.

\[
\forall \langle \hat{a}@H', \tau \rangle \in \hat{\beta}, \hat{a} \in \text{dom}(\hat{\beta}) \text{ and } \phi, \phi \vdash \hat{a}@H' : (\text{Addr } \tau), d \text{ and } \text{Depth}(\tau) \leq d
\]

\[
\forall \hat{a} \text{ appearing in } \hat{\beta} \text{ or } \hat{\mu}. \text{Depth}(\text{ActorType}(\hat{a})) \leq d \quad \forall \hat{a} \in \text{dom}(\hat{\beta}). \hat{a} \text{ is internal}
\]

\[
\forall \hat{a} \in \text{dom}(\hat{\beta}). \exists \tau. \text{ActorType}(\hat{a}) = \tau \text{ and } \phi, \phi \vdash_{\text{beh}} \hat{b} : d \text{ for all } \hat{b} \in \text{Pok}(\hat{a})
\]

\[
\forall \langle \hat{a}@H', \hat{v} \rangle \in \text{dom}(\hat{\mu}). \exists \phi, \phi \vdash \hat{a}@H' : (\text{Addr } \tau), d \text{ and } \phi, \phi \vdash \hat{v} : \tau, d
\]

\[
\vdash_{cfg} \langle \langle \hat{\beta} \mid \hat{\mu} \mid H \rangle \rangle^{\hat{\beta}} : d
\]
H.2 Abstract Behaviors

\[ \hat{Q}_i = (\text{define-state } q_i \ [x_{i,1} \ t'_{i,1}] \ \ldots \ [x_{i,m} \ t'_{i,m}] \ x'_{i} \ e'_i \ \hat{f}c_i) \) for all \( i \in 1 \ldots n \)

\( q_1, \ldots, q_n \) are distinct \( \quad \text{Depth}(t'_{i,j}) \leq d \) for all \( i \in 1 \ldots n, j \in 1 \ldots m \)

\( \Theta = [q_1 \rightarrow (t'_{1,1} \ldots t'_{1,m}), \ldots, q_n \rightarrow (t'_{n,1} \ldots t'_{n,m})] \)

\( \Gamma, \Theta \vdash \hat{\epsilon} : \bot, d \quad \Gamma, \Theta \vdash_{\text{state}} \hat{Q}_i : d \) for all \( i \in 1 \ldots n \)

\( \Gamma, \Theta \vdash_{\text{beh}} \langle \hat{Q}_1 \ldots \hat{Q}_n, \hat{\epsilon} \rangle : d \)

Q_i = (\text{define-state } q_i \ [x_{i,1} \ t'_{i,1}] \ \ldots \ [x_{i,m} \ t'_{i,m}] \ x'_{i} \ e'_i \ \hat{f}c_i) \) for all \( i \in 1 \ldots n \)

\( q_1, \ldots, q_n \) are distinct \( \quad \text{Depth}(t'_{i,j}) \leq d \) for all \( i \in 1 \ldots n, j \in 1 \ldots m \)

\( \Theta = [q_1 \rightarrow (t'_{1,1} \ldots t'_{1,m}), \ldots, q_n \rightarrow (t'_{n,1} \ldots t'_{n,m})] \)

\( \Gamma[x'' \rightarrow \tau], \Theta \vdash \hat{\epsilon} : \bot, d \quad \Gamma, \Theta \vdash_{\text{tc}} f : d \quad \Gamma, \Theta \vdash_{\text{state}} Q_i : d \) for all \( i \in 1 \ldots n \)

\( \Gamma, \Theta \vdash_{\text{beh}} \langle Q_1 \ldots Q_n, (\text{receive } x'' \hat{\epsilon} f c) \rangle : d \)

H.3 Abstract State Definitions

\( \text{Depth}(r_i) \leq d \) for all \( i \in 1 \ldots n \quad \Gamma(\text{self}) = (\text{Addr } \tau') \quad \text{Depth}(\tau') \leq d \)

\( \Gamma' = \Gamma[x_1 \rightarrow r_1, \ldots, x_n \rightarrow r_n] \quad \Gamma'[x' \rightarrow r'], \Theta \vdash \hat{\epsilon} : \bot, d \quad \Gamma', \Theta \vdash_{\text{tc}} f : d \)

\( \Gamma, \Theta \vdash_{\text{state}} (\text{define-state } (q \ [x_1 \ r_1] \ \ldots \ [x_n \ r_n]) \ x' \ e' \ f c) : d \)

H.4 Abstract Timeout Clauses

\( \Gamma, \Theta \vdash_{\text{tc}} \text{no-timeout} : d \quad \Gamma, \Theta \vdash_{\text{tc}} [\text{timeout } ov \hat{\epsilon}] : d \)

H.5 Abstract Expressions

H.5.1 Effectful Expressions

\( \Theta(q) = r_1, \ldots, r_n \quad \Gamma, \Theta \vdash \hat{\epsilon}_i : \tau_i, d \) for all \( i \in 1 \ldots n \)

\( \Gamma, \Theta \vdash (\text{goto } q \ \hat{\epsilon}_1 \ \ldots \ \hat{\epsilon}_n) : \bot, d + 1 \)

Q_i = (\text{define-state } q_i \ [x_{i,1} \ t'_{i,1}] \ \ldots \ [x_{i,m} \ t'_{i,m}] \ x'_{i} \ e'_i \ \hat{f}c_i) \) for all \( i \in 1 \ldots n \)

\( q_1, \ldots, q_n \) are distinct \( \quad \text{Depth}(\tau) \leq d \)

\( \Gamma' = \Gamma[\text{self} \rightarrow (\text{Addr } \tau)] \quad \Theta' = [q_1 \rightarrow (r_{1,1} \ldots r_{1,m}), \ldots, q_n \rightarrow (r_{n,1} \ldots r_{n,m})] \)

\( \Gamma', \Theta' \vdash \hat{\epsilon} : \bot, d \quad \Gamma', \Theta' \vdash_{\text{state}} Q_i : d \) for all \( i \in 1 \ldots n \)

\( \Gamma, \Theta \vdash (\text{spawn' } \tau \hat{\epsilon} Q_1 \ \ldots \ Q_n) : (\text{Addr } \tau), d + 1 \)

\( \Gamma, \Theta \vdash \hat{\epsilon}_1 : (\text{Addr } \tau), d \quad \Gamma, \Theta \vdash \hat{\epsilon}_2 : \tau, d \)

\( \Gamma, \Theta \vdash (\text{send } \hat{\epsilon}_1 \ \hat{\epsilon}_2) : (\text{Variant } [\text{Unit}]), d + 1 \)
H.5.2 Other Expressions

\[ \Gamma(x) = \tau \quad \text{Depth}(\tau) = d \]
\[ \Gamma, \Theta \vdash x : \tau, d \]
\[ \text{ActorType}(\hat{a}) = \tau \]
\[ \Gamma, \Theta \vdash \hat{a} \circ H : (\text{Addr} \ \tau), 1 \]
\[ \Gamma, \Theta \vdash \text{abs-nat} : \text{Nat}, 1 \]
\[ \Gamma, \Theta \vdash \hat{e}_i : \tau_i, d \text{ for all } i \in 1 \ldots n \]
\[ \Gamma, \Theta \vdash \hat{e} : \tau, d + 1 \]
\[ \Gamma, \Theta \vdash \text{(record } [r_1 \hat{e}_1] \ldots [r_n \hat{e}_n]) : (\text{Record} \ [r_1 \tau_1] \ldots [r_n \tau_n]), d + 1 \]
\[ \Gamma, \Theta \vdash \hat{e} : (\text{Record} \ [r_1 \tau_1] \ldots [r_n \tau_n]), d \quad i \in 1 \ldots n \]
\[ \Gamma, \Theta \vdash (: \hat{e} \ r_i) : \tau_i, d + 1 \]
\[ \Gamma, \Theta \vdash \hat{e}_i : \tau_i, d \text{ for all } i \in 1 \ldots n \]
\[ \Gamma, \Theta \vdash (\text{variant } t \hat{e}_1 \ldots \hat{e}_n) : (\text{Variant} \ [t \tau_1 \ldots \tau_n]), d + 1 \]
\[ \Gamma, \Theta \vdash \hat{e} : (\text{Variant} \ [t_1 \tau_{1,1} \ldots \tau_{1,m}] \ldots [t_n \tau_{n,1} \ldots \tau_{n,m}]), d \]
\[ \Gamma[x_{i,1} \leftarrow t_{i,1}, \ldots, x_{i,m} \leftarrow t_{i,m},] \Theta \vdash \hat{e}'_i : \tau', d \text{ for all } i \in 1 \ldots n \]
\[ \Gamma, \Theta \vdash (\text{case } \hat{e} \ [(t_1 x_{1,1} \ldots x_{1,m}) \hat{e}'_1] \ldots [(t_n x_{n,1} \ldots x_{n,m}) \hat{e}'_n]) : \tau', d + 1 \]
\[ \text{Depth}(\tau) \leq d \]
\[ \tau = (\text{rec } X \tau') \quad \Gamma, \Theta \vdash \hat{e} : \tau[X \leftarrow \tau], d \]
\[ \Gamma, \Theta \vdash (\text{fold } \hat{e}) : \tau, d + 1 \]
\[ \text{Depth}(\tau) \leq d \]
\[ \tau = (\text{rec } X \tau') \quad \Gamma, \Theta \vdash \hat{e} : \tau, d \]
\[ \Gamma, \Theta \vdash (\text{unfold } \tau \hat{e}) : \tau[X \leftarrow \tau], d + 1 \]
\[ \Gamma, \Theta \vdash \hat{e}_i : \tau, d \text{ for all } i \in 1 \ldots n \]
\[ \Gamma, \Theta \vdash (\text{list } \{\hat{e}_1, \ldots, \hat{e}_n\}) : (\text{List} \ \tau), d + 1 \]
\[ \Gamma, \Theta \vdash (\text{dict } \{\hat{e}_1, \ldots, \hat{e}_m\}) : (\text{Dict} \ \tau \ \tau'), d + 1 \]
\[ \Gamma, \Theta \vdash \hat{e}_i : \tau \quad \Gamma, \Theta \vdash \hat{e}'_i : \tau', d \text{ for all } i \in 1 \ldots m \]
\[ \Gamma, \Theta \vdash (\text{for/fold } [x \hat{e}] [x' \hat{e}']) : \tau, d + 1 \]
\[ \Gamma, \Theta \vdash (\text{for/fold } [x \hat{e}] [x' \hat{e}']) : \tau, d + 1 \]

H.5.3 Type and Depth Subsumption

A judgment of the form \( \Gamma, \Theta \vdash \hat{e} : \tau, d \) says that depth of \( \hat{e} \) is no more than \( d \). Therefore, the below depth-subsumption rule says that such a judgment can be
proved if one can prove that $\hat{e}$ is bounded by a smaller depth $d'$. The standard type-subsumption rule is also given.

$$
\frac{\Gamma, \Theta \vdash \hat{e} : \tau, d' \quad d' < d}{\Gamma, \Theta \vdash \hat{e} : \tau, d}
$$

$$
\frac{\Gamma, \Theta \vdash \hat{e} : \tau, d}{\Gamma, \Theta \vdash \hat{e} : \tau, d'}
$$

H.6 Labels

The type/depth-checking rules for labels are used in the termination proof in appendix N to help show that a given configuration has a limited number of next steps it can take. The rules merely check the depth of sent and received values.

$$
\phi, \phi \vdash \hat{v} : \tau, d \\
\quad \vdash \hat{a} : \text{rcv-ext}(H, \hat{v}, \tau) : d
$$

$$
\text{ActorType}(\hat{a}') = \tau \\
\phi, \phi \vdash \hat{v} : \tau, d \\
\quad \vdash \hat{a} : \text{rcv-int}(H, \hat{v}) : d
$$

$$
\text{ActorType}(\hat{a}') = \tau \\
\phi, \phi \vdash \hat{v} : \tau, d \\
\quad \vdash \hat{a} : \text{send-ext}(\hat{a}'@H, \hat{v}) : d
$$

$$
\vdash \hat{a} : \text{timeout} : d \\
\quad \vdash \hat{a} : \text{func} : d \\
\quad \vdash \hat{a} : \text{goto} : d \\
\quad \vdash \hat{a} : \text{spawn} : d
$$

H.7 Type Depth

Depth for types is defined in terms of the maximum lexical depth of all types reachable from a given type by unfolding; this accounts for unfolded recursive types that are larger than their folded counterparts.

**Definition.** The lexical depth of a type $\tau$, written $\text{LexDepth}(\tau)$, is defined as follows.

$$
\text{LexDepth}(\tau) =
\begin{cases}
1 & \text{Case } \tau \in \{\text{Nat}, \text{String}, \bot\} : \\
1 + \text{LexDepth}(\tau') & \text{Case } \tau = (\text{Variant} \ [t_1 \tau_{1,1} \ldots \tau_{1,m}] \ldots [t_n \tau_{n,1} \ldots \tau_{n,m}]) : \\
1 + \max(\text{LexDepth}(\tau_{1,1}), \ldots, \text{LexDepth}(\tau_{n,m})) & \text{Case } \tau = (\text{Record} \ [r_1 \tau_1] \ldots [r_n \tau_n]) : \\
1 + \max(\text{LexDepth}(\tau_1), \ldots, \text{LexDepth}(\tau_n)) & \text{Case } \tau = (\text{Addr } \tau') : \\
1 + \text{LexDepth}(\tau') & \text{Case } \tau = (\text{rec } X \tau') : \\
1 + \text{LexDepth}(\tau') & \text{Case } \tau = (\text{List } \tau') : \\
1 + \text{LexDepth}(\tau') & \text{Case } \tau = (\text{Dict } \tau' \tau'') : \\
1 + \max(\text{LexDepth}(\tau'), \text{LexDepth}(\tau'')) & \text{Case } \tau = (\text{Dict } \tau' \tau'') : 
\end{cases}
$$
Definition. The reachable types from a given type, written $\text{Reachable}(\tau)$, is defined as follows.

\[
\text{Reachable}(\tau) = \\
\begin{cases} 
\{\tau\} & \text{Case } \tau \in \{\text{Nat, String, } \bot\} \\
\{\tau\} \cup \text{Reachable}(\tau') & \text{Case } \tau = (\text{Variant } [t_1 \tau_{1,1} \ldots t_{1,m}] \ldots [t_n \tau_{n,1} \ldots t_{n,m}]) \\
\{\tau\} \cup \text{Reachable}(\tau_1) \cup \ldots \cup \text{Reachable}(\tau_m) & \text{Case } \tau = (\text{Record } [r_1 \tau_1] \ldots [r_n \tau_n]) \\
\{\tau\} \cup \text{Reachable}(\tau') & \text{Case } \tau = (\text{Addr } \tau') \\
\{\tau\} \cup \text{Reachable}(\tau'') & \text{Case } \tau = (\text{rec } X \tau) \\
\{\tau\} \cup \{\tau''[X \leftarrow \tau'] | \tau'' \in \text{Reachable}(\tau')\} & \text{Case } \tau = \text{List } \tau' \\
\{\tau\} \cup \text{Reachable}(\tau') & \text{Case } \tau = \text{Dict } \tau' \tau'' \\
\{\tau\} \cup \text{Reachable}(\tau') \cup \text{Reachable}(\tau'') & \text{Case } \tau = \text{Dict } \tau' \tau'' \\
\end{cases} 
\]

Intuitively, the types reachable from a given type are those that are either component types of a given type, or an unfolding of the current type in the case of a type $(\text{rec } X \tau)$. By structural induction on $\tau$, it is easy to see that $\text{Reachable}(\tau)$ is both well-defined and finite for all types $\tau$. This approach to bounding the set of reachable types by the “subexpressions” of those types comes from Brandt and Henglein [20].

Definition. The depth of a type, written $\text{Depth}(\tau)$, is the maximal lexical depth of all types reachable from $\tau$. That is, $\text{Depth}(\tau)$ is the greatest $d$ such that $\text{LexDepth}(\tau') = d$ for some $\tau' \in \text{Reachable}(\tau)$.

Because $\text{Reachable}(\tau)$ is finite for all $\tau$, $\text{Depth}(\tau)$ is well-defined.
Appendix I

Proofs for Abstract Conformance

The Abstract Conformance theorem says, roughly, that if an abstraction of a program conforms to an abstraction of a specification, then the concrete program also conforms to the concrete specification. This appendix first proves some lemmas needed for the that theorem, then proves the theorem itself.

I.1 Definitions

Define $\text{ExtAddr}$ as the set of all abstract external addresses, i.e., $\text{ExtAddr} = \{\hat{a} | \hat{a} \text{ is external}\}$.

An abstract program configuration $\hat{K} = \langle \hat{\beta} \mid \hat{\mu} \mid H \rangle^{\hat{\rho}}$ is well-formed if and only if

1. for all $\hat{a}$ appearing in $\hat{\beta}$, $\hat{\mu}$, or $\hat{\rho}$, if $\hat{a}$ is internal then $\hat{a} \in \text{dom}(\hat{\beta})$,
2. for all $\hat{a} \not@ H'$ appearing in $\hat{\beta}$, $\hat{\mu}$, or $\hat{\rho}$, $H' \subseteq H$, and
3. $\hat{\mu}$ is fully merged.

An abstract program configuration $\hat{K} = \langle \hat{\beta} \mid \hat{\mu} \mid H \rangle^{\hat{\rho}}$ is an externals-only configuration if and only if

- for all $\hat{a} @ H'$ appearing in either $\hat{\beta}$ or $\hat{\mu}$, $H' = \emptyset$ if $a$ is internal, and $|H'| \leq 1$ otherwise, and
- for all $\langle a @ H', r \rangle \in \hat{\rho}$, $|H'| \leq 1$.

Note that this differs from the analogous definition for a concrete configuration by requiring that some addresses have at most one marker, rather than exactly one. This accounts for the fact that abstraction can remove markers.
The function $\text{Used}$ returns the used-marker component of a program configuration. That is, for a concrete program configuration $\hat{K} = \langle \hat{\beta} \mid \hat{\mu} \mid H \rangle$, $\text{Used}(\hat{K}) = H$. Similarly, for an abstract program configuration $\hat{K} = \langle \hat{\beta} \mid \hat{\mu} \mid H \rangle$, $\text{Used}(\hat{K}) = H$.

The application of a marker-correspondence function $M : \text{HistMark} \rightarrow \text{HistMark}$ to a PSM $s$, i.e., $M(s)$, is defined as the obvious component-wise, element-wise application to its parts. Similarly the application to a specification configuration $S$, i.e., $M(S)$, is defined as the element-wise application to each of its member PSMs. In either case, the application is defined only when $\text{dom}(M)$ contains all of the markers appearing in $s$ or $S$, respectively.

An abstract message map $\hat{\mu}$ is fully merged if and only if for all $\hat{a}$, $H$, $\hat{v}$, and $\hat{v}'$ such that $\langle \hat{a} @ H, \hat{v} \rangle \in \text{dom}(\hat{\mu})$ and $\langle \hat{a} @ H, \hat{v}' \rangle \in \text{dom}(\hat{\mu}')$, either $\hat{v} = \hat{v}'$ or $\text{Merge}(\hat{v}, \hat{v}')$ is not defined.

**I.1.1 Restricted Abstract-Transition Relation**

The restricted abstract-transition relation $\rightarrow_{\text{RA}}$ adapts the restrictions of the concrete restricted transition relation $\rightarrow_{\text{R}}$ to abstract configurations. Specifically, those restrictions are the following.

- The environment never sends the program a message containing an internal address.
- Each address in a message from the environment is the “representative” message from its equivalence class, as described in appendix C.
- Only one actor (plus the children it spawns while handling an event) run at a time.
- New addresses and history markers are allocated using a deterministic scheme.

It also adds one additional restriction to help reduce the state space.

- Every message received from the environment must be a “maximal” value for its type.

This section introduces some definitions that adapt these restrictions to an abstract context, then formally defines that relation.

**Definition.** The definition of handler-start and handler-continuation labels is similar to that for concrete labels: abstract $\text{rcv-ext}$, $\text{rcv-int}$, and $\text{timeout}$ labels are all handler-start labels; all other abstract transition labels are handler-continuation labels.

The single-handler ordering $<$ on addresses is extended to abstract addresses, where a collective actor takes priority over an atomic actor spawned from the
same location. Formally, let the function \( \text{AddrLoc} \) be defined as follows.

\[
\text{AddrLoc}(\tilde{a}) = \begin{cases} 
\ell & \text{if } \tilde{a} = (\text{addr } \ell \ n) \\
\ell' & \text{if } \tilde{a} = (\text{collective-addr } \ell')
\end{cases}
\]

**Definition.** The relation \(<\) is defined for abstract addresses such that \( \tilde{a} < \tilde{a}' \) if and only if any of the following hold.

- \( \text{AddrLoc}(\tilde{a}) < \text{AddrLoc}(\tilde{a}') \)
- \( \tilde{a} = (\text{addr } \ell \ n), \tilde{a}' = (\text{addr } \ell \ n'), \text{ and } n < n' \).
- \( \text{AddrLoc}(\tilde{a}) = \text{AddrLoc}(\tilde{a}'), \tilde{a} \) is atomic, and \( \tilde{a}' \) is collective.

Figure I.1 adapts the deterministic-marking function \( \text{Mark} \) from appendix E. Note that to match the abstract interpretation, the function does not mark values inside abstract lists and dictionaries.

\[
\text{Mark}(\tilde{v}, H) = \begin{cases} 
\langle a@H' \cup \{\eta\}, H \cup \{\eta\} \rangle & \text{Case } \tilde{v} = a@H' \\
\langle \text{variant } t \tilde{v}_1 \ldots \tilde{v}_m, H_{n+1} \rangle & \text{Case } \tilde{v} = (\text{variant } t \tilde{v}_1 \ldots \tilde{v}_m) \\
\langle \text{record } [r_1 \tilde{v}_1] \ldots [r_n \tilde{v}_n], H_{n+1} \rangle & \text{Case } \tilde{v} = (\text{record } [r_1 \tilde{v}_1] \ldots [r_n \tilde{v}_n]) \\
\langle \text{fold } \tau \tilde{v}', H' \rangle & \text{Case } \tilde{v} = (\text{fold } \tau \tilde{v}') \\
\langle \text{dict } \{\tilde{v}_1', \ldots, \tilde{v}_n', \tilde{v}_1'', \ldots, \tilde{v}_m''\}, H \rangle & \text{Case } \tilde{v} = \{\tilde{v}_1', \ldots, \tilde{v}_n', \tilde{v}_1'', \ldots, \tilde{v}_m''\}
\end{cases}
\]

Figure I.1: Deterministic marking of abstract values with \( \text{Mark} \)

The function \( \text{MaxVals} \) in figure I.2 inductively defines the set of maximal values that the environment may send for each type. The \textbf{List} and \textbf{Dict} cases simply construct a single value containing all maximal values of the contained type. The \((\text{Addr } \tau')\) case defines the maximal value in terms of the representative address for \( \tau' \). The other cases are defined as all possible combinations of their maximal-value components.
MaxVals(\(\tau\)) =
\[
\begin{cases}
  \text{Case } \tau = (\text{List } \tau') : \\
  \{\text{list MaxVals}(\tau')\}
  \\
  \text{Case } \tau = (\text{Dict } \tau' \tau'') : \\
  \{\text{dict MaxVals}(\tau') \text{ MaxVals}(\tau'')\}
  \\
  \text{Case } \tau = (\text{Addr } \tau') : \\
  \{(\text{collective-addr } \ell) @ \emptyset \mid \exists n. \text{ExtRepAddr}(\tau') = (\text{addr } \ell n)\}
  \\
  \text{Case } \tau = (\text{Variant } [t_1 \tau'_{1,1} \ldots \tau'_{1,m}] \ldots [t_n \tau'_{n,1} \ldots \tau'_{n,m}]) : \\
\quad \bigcup_{i \in 1 \ldots n} \{(\text{variant } t_i \hat{\nu}_1 \ldots \hat{\nu}_m) \mid \hat{\nu}_j \in \text{MaxVals}(\tau'_{i,j}) \text{ for } j \in 1 \ldots m\}
  \\
  \text{Case } \tau = (\text{Record } [r_1 \hat{\nu}_1] \ldots [r_n \hat{\nu}_n]) : \\
  \{(\text{record } [r_1 \hat{\nu}_1] \ldots [r_n \hat{\nu}_n]) \mid \hat{\nu}_i \in \text{MaxVals}(\tau'_i) \text{ for } i \in 1 \ldots n\}
  \\
  \text{Case } \tau = (\text{rec } X \tau') : \\
  \{(\text{fold } \tau \hat{\nu}) \mid \hat{\nu} \in \text{MaxVals}(\tau'\langle X \leftarrow \tau\rangle)\}
  \\
  \text{Case } \tau = \text{Nat}: \\
  \{\text{abs-nat}\}
  \\
  \text{Case } \tau = \text{String}: \\
  \{\text{abs-string}\}
\end{cases}
\]

Figure I.2: Maximal values for each type

**Definition.** The restricted abstract-transition relation \(\rightarrow_{\text{RA}}\) is defined such that for all \(\hat{K} = \langle \hat{\beta} \mid \hat{\mu} \mid H \rangle\hat{\beta}\) and \(\hat{K}' = \langle \langle \hat{\beta}' \mid \hat{\mu}' \mid H' \rangle \rangle\hat{\beta}'\), \(\hat{K} \stackrel{l}{\rightarrow} \hat{K}'\) if and only if the following conditions hold.

1. \(\hat{K} \stackrel{l}{\rightarrow} \hat{K}'\).

2. If \(\hat{l}\) is a handler-continuation label with active actor identified by \(\hat{a}\), then there is no \(\hat{a}'\) such that \(\hat{a} < \hat{a}'\) and the actor at \(\hat{a}'\) in \(\hat{K}\) is handling an event.

3. If \(\hat{l}\) is a handler-start label, then no actor (atomic or collective) in \(\hat{K}\) is handling an event.

4. If \(\hat{l} = \hat{\alpha}: \text{spawn}(\langle \text{addr } \ell n \rangle)\) for some \(\hat{a}, \ell,\) and \(n\), then there do not exist \(\hat{K}''\) and \(n' < n\) such that \(\hat{K} \xrightarrow{\hat{\alpha}: \text{spawn}(\langle \text{addr } \ell n' \rangle)} \hat{K}''\).

5. If \(\hat{l} = \hat{\alpha}: \text{rcv-ext}(H'', \hat{\nu}, \tau)\) for some \(H'', \hat{\nu},\) and \(\tau\), then there exists \(\hat{\nu}'\) such that \(\text{Mark}(\hat{\nu}', \tau) = \langle \hat{\nu}, H' \rangle\) and \(\hat{\nu}' \in \text{MaxVals}(\tau)\).

6. If \(\hat{l} = \hat{\alpha}: \text{send-ext}(\hat{a}@H'', \hat{\nu})\) for some \(\hat{a}, \hat{a}', H'', \) and \(\hat{\nu}\), then either \(H = H'\) or there exists \(\hat{\nu}'\) such that \(\text{Mark}(\hat{\nu}, H') = \langle \hat{\nu}', H \rangle\).
I.2 Value Approximation Lemma

Lemma (Value Approximation). For all \( \hat{v} \) and \( \hat{e} \) such that \( \hat{v} \sqsupseteq A, M \hat{e} \) for some \( A \) and \( M \), \( \hat{e} \) is a value.

Proof. The proof is by structural induction on \( \hat{v} \). The case for a list is shown below. The case for a dictionary is similar, and the remaining cases are straightforward.

Case: \( \hat{v} = (\text{list } \hat{v}_1 \ldots \hat{v}_n) \)

By the definition of \( \sqsupseteq \), there exist \( \hat{o}v_1, \ldots, \hat{o}v_m \) such that \( \hat{e} = (\text{list } \{\hat{o}v_1, \ldots, \hat{o}v_m\}) \). An abstract value has no free variables, so by the definition of \( \sqsupseteq \) for lists, \( \hat{e} \) has no free variables. Therefore, every \( \hat{o}v_i \) in \( \hat{e} \) is a value, so \( \hat{e} \) is a value. □

Corollary I.2.1. For all \( \tilde{v} \) and \( \hat{e} \) such that \( |\tilde{v}| \sqsupseteq A, M \hat{e} \) for some \( A \) and \( M \), \( \hat{e} \) is a value.

Proof. By a straightforward structural induction on \( \tilde{v} \), \( |\tilde{v}| \) is a value. Then by the Value Approximation lemma, \( \hat{e} \) is a value. □

I.3 Abstract Substitution Lemma

This lemma shows that substitution preserves the \( \sqsupseteq \)-ordering of the abstract interpretation.

Lemma (Abstract Substitution). For all \( \hat{e}, \hat{e}', \hat{e}' \mathcal{R} x A, \) and \( M \), if

- \( |\hat{e}| \sqsupseteq A, M \hat{e} \),
- \( |\hat{e}'| \sqsupseteq A, M \hat{e}' \), and
- either there exists \( \tilde{E} \) such that \( \hat{e} = \tilde{E}[x] \) or there exists \( \tilde{v} \) such that \( \hat{e}' = \tilde{v} \),

then \( |\hat{e}[x \leftarrow \hat{e}']| \sqsupseteq A, M \hat{e}[x \leftarrow \hat{e}'] \).

Proof. The proof is by structural induction on \( \hat{e} \). The case for a list is shown below. The case for a dictionary is similar, and the remaining cases are straightforward.

Case: \( \hat{e} = (\text{list } \hat{o}v_1 \ldots \hat{o}v_n) \)

A list expression cannot be a context \( \tilde{E} \), so there must exist \( \hat{v} \) such that \( \hat{e}' = \hat{v} \), and by corollary I.2.1 to the Value Approximation lemma there exists some \( \hat{v} \) such that \( \hat{e}' = \hat{v} \). By the definitions of \( |\cdot| \) and \( \sqsupseteq \), there exist \( \hat{o}v_1, \ldots, \hat{o}v_m \) such that \( \hat{e} = (\text{list } \{\hat{o}v_1, \ldots, \hat{o}v_m\}) \) such that for all \( i \in 1 \ldots n \), there exists \( j \in 1 \ldots m \) such that \( |\hat{o}v_i| \sqsupseteq A, M \hat{o}v_j \). Then by the induction hypothesis, for all \( i \in 1 \ldots n \) there exists \( j \in 1 \ldots m \) such that \( |\hat{o}v_i[x \leftarrow \hat{v}]| \sqsupseteq A, M (\text{list } \{\hat{o}v_j[x \leftarrow \hat{v}]\}) \).

By the definition of substitution, we have
• $e[x \rightarrow v] = (\text{list } \delta v_1[x \rightarrow v] \ldots \delta v_n[x \rightarrow v])$ and

• $e[x \rightarrow v] = (\text{list } \cup_{j \in 1 \ldots m} \{\delta v_j[x \rightarrow v]\})$.

Because we are only substituting values into the expressions, $e[x \rightarrow v]$ and $\tilde{e}[x \rightarrow \tilde{v}]$ contain only open values, so they are each valid list expressions. Furthermore, $|\tilde{e}| = (\text{list } \cup_{i \in 1 \ldots n} \{|\delta v_i[x \rightarrow \tilde{v}]|\})$. Therefore, by the definition of $\sqsubseteq$ for lists, we have $|\tilde{e}[x \rightarrow \tilde{v}]| \sqsubseteq_{A,M} \tilde{e}[x \rightarrow \tilde{v}]$.

\[\blacksquare\]

**Corollary I.3.1.** For all $\tilde{E}, \tilde{e}, \tilde{E}, \tilde{v}, A, \text{ and } M$, if $|\tilde{E}| \sqsubseteq_{A,M} \tilde{E}$ and $|\tilde{e}| \sqsubseteq_{A,M} \tilde{e}$, then $|\tilde{E}[\tilde{v}]| \sqsubseteq_{A,M} \tilde{E}[\tilde{v}]$.

**Proof.** Convert $\tilde{E}$ and $\tilde{E}$ into expressions $\tilde{e}'$ and $\tilde{e}''$ by replacing the hole in each context with a variable $x'$ that does not appear in either $\tilde{E}$, $\tilde{E}$, or $\tilde{e}$. Then we have that $\tilde{E}[\tilde{e}'] = \tilde{e}''[x' \rightarrow \tilde{e}'']$, $\tilde{E}[\tilde{e}'] = \tilde{e}''[x' \rightarrow \tilde{e}'']$, and $|\tilde{e}''| \sqsubseteq_{A,M} \tilde{e}''$. Furthermore, a context $\tilde{E}$ cannot have a hole inside a list or dict expression, so $\tilde{e}'[x \rightarrow \tilde{e}]$ is a valid expression. Then by the Abstract Substitution lemma, $|\tilde{e}'[x \rightarrow \tilde{e}]| \sqsubseteq_{A,M} \tilde{e}'[x \rightarrow \tilde{e}]$, so $|\tilde{E}[\tilde{v}]| \sqsubseteq_{A,M} \tilde{E}[\tilde{v}]$.

\[\blacksquare\]

### I.4 Abstract Context Lemma

**Lemma (Abstract Context).** For all $\tilde{E}, \tilde{e}, \tilde{E}, A, \text{ and } M$, if $|\tilde{E}| \sqsubseteq_{A,M} \tilde{E}$, then there exist $\tilde{E}'$ and $\tilde{e}''$ such that $\tilde{e}' = \tilde{E}'[\tilde{e}''], \tilde{E} \sqsubseteq_{A,M} \tilde{E}', \text{ and } \tilde{e} \sqsubseteq_{A,M} \tilde{e}''$.

**Proof.** Straightforward structural induction on $\tilde{E}$.

\[\blacksquare\]

**Corollary I.4.1.** For all $\tilde{E}, \tilde{e}, \tilde{E}, A, \text{ and } M$, if $|\tilde{E}[\tilde{v}]| \sqsubseteq_{A,M} \tilde{e}$, then there exist $\tilde{E}$ and $\tilde{e}'$ such that $\tilde{e} = \tilde{E}[\tilde{e}'], |\tilde{E}| \sqsubseteq_{A,M} \tilde{E}$, and $|\tilde{e}| \sqsubseteq_{A,M} \tilde{e}'$.

**Proof.** Straightforward structural induction on $\tilde{E}$.

\[\blacksquare\]

### I.5 Well-Formed Preservation Lemma

**Lemma (Well-Formed Preservation).** For all $\tilde{K}, I$, and $\tilde{K}' = \langle \langle \tilde{\beta}' \mid \tilde{\beta}' \mid \tilde{H}' \rangle \rangle$, if $\tilde{K}$ is well-formed and $\tilde{K} \xrightarrow{I} \tilde{K}'$, then $\tilde{K}'$ is well-formed and every marker appearing in $I$ is a member of $\tilde{H}'$.

**Proof.** First, we show the requirement that every internal address in $\tilde{K}'$ is in $\text{dom}(\tilde{\beta}')$. Once an address is added to $\text{dom}(\tilde{\beta})$, it is never removed. The only rule of the transition relation enabling the step $\tilde{K} \xrightarrow{I} \tilde{K}'$ that introduces a new internal address is M-SPAWN. That rule creates a mapping for the new address in $\tilde{\beta}$, so the condition is satisfied.

The remainder of the proof is by cases on the rule of the transition relation enabling $\tilde{K} \xrightarrow{I} \tilde{K}'$. Only the rules M-RECEIVEINTERNAL, M-RECEIVEEXTERNAL,
M-SendInternal, and M-SendExternal create new markers, change the set $H'$ of used markers in $K'$, add new receptionists, or add new external addresses, so we can ignore the other rules.

There exist $\hat{\beta}$, $\hat{\mu}$, $H$, and $\hat{\rho}$ such that $\hat{K} = \langle \hat{\beta}, \hat{\mu}, H \rangle^{\hat{\rho}}$. In each of the four rules above, either no new markers are created and $H = H'$, (in the case of an unmarked destination address in M-ReceiveInternal and M-SendInternal), or there exist $\hat{v}$ and $\hat{v}'$ such that $\langle \hat{v}', H' \rangle \in \text{Markings}(\hat{v}, H)$. By the definition of Markings, $H'$ contains all new markers created in that call. Therefore, every marker appearing in $l$ and $K'$ is an element of $H'$.

The only rule that creates fresh occurrences of external addresses is M-ReceiveExternal. In that case, the external addresses come from a received message $\hat{v}$. The rule uses Markings to mark every address in $\hat{v}$ with a fresh marker before substituting the message into the configuration, so every external address in $K'$ in that case is marked with at least one external marker. None of the other rules remove markers on existing external addresses, so in every case, every external address in $K'$ is marked with at least one external marker.

The only rule that changes the set of receptionists is M-SendExternal. In that case, every new receptionist’s marked address $\hat{a}@H''$ comes from the marked message $\hat{v}'$, which was marked by Markings. Because $\hat{K}$ is well-formed, every marker appearing in $\hat{K}$ is an element of $H$. Every address marked with Markings gets a fresh marker not in $H$, so every new receptionist has a unique marked address. Therefore, in every case, every receptionist has a unique marked address.

\(\square\)

**I.6 Abstract Well-Formed Preservation Lemma**

**Lemma** (Abstract Well-Formed Preservation). For all $\hat{K}, \hat{l},$ and $\hat{K'}$, if $\hat{K}$ is well-formed and $\hat{K} \xrightarrow{\hat{l}} \hat{K'}$, then $\hat{K'}$ is well-formed.

**Proof:** Similar to the proof of the Well-Formed Preservation lemma. We must additionally show that the message map $\hat{\mu}'$ in $\hat{K'}$ is fully merged. We know by the definition of a well-formed configuration that the message-map $\hat{\mu}$ in $\hat{K}$ is well-formed. If $\hat{K} \xrightarrow{\hat{l}} \hat{K'}$ is enabled by the A-SendInternal rule, then the result holds by the Fully Merged Preservation lemma. If the transition is enabled by the A-ReceiveInternal rule, then either $\hat{\mu} = \hat{\mu}'$, or there exist some $\hat{a}, H$, and $\hat{v}$ such that $\hat{\mu} = \hat{\mu}' \cup \langle \hat{a}@H, \hat{v} \rangle \rightarrow \text{single}$. In the latter case we’re done; in the former case, it suffices to know that $\hat{\mu}$ is fully merged. \(\square\)

**I.7 Extra Markers Lemma**

**Lemma** (Extra Markers). For all $\hat{v}, \hat{v}', H, H', \hat{v}, A,$ and $M$, if

- $\text{dom}(M) \subseteq H$. 

APPENDIX I. PROOFS FOR ABSTRACT CONFORMANCE

• \(|\hat{v}| \subseteq_{A,M} \hat{v}\), and

• \(\text{Mark}(\hat{v}, H) = \langle \hat{v}', H' \rangle\),

then \(|\hat{v}'| \subseteq_{A,M} \hat{v}\).

Proof. The proof is by structural induction on \(\hat{v}\). The case for addresses is shown below; others are straightforward.

Case: \(\hat{v} = \hat{a}@H'\)

By the definition of \(\text{Mark}\), there exists some \(\eta \notin H\) such that \(\hat{v}' = \hat{a}@H' \cup \{\eta\}\). Also, by the definitions of \(|·|\) and \(\sqsubseteq\), there exist \(\hat{a}\) and \(H''\) such that \(\hat{v} = \hat{a}@H''\), \(|a| \subseteq_{A} \hat{a}\), and \(H' \subseteq_{M} H''\).

Because \(H' \subseteq_{M} H''\), there exist \(H'_1\) and \(H'_2\) such that \(H' = H'_1 \cup H'_2\), \(M(H'_1) = H''\) and \(H'_2 \cap \text{dom}(M) = \emptyset\). Because \(\eta \notin H\) and \(\text{dom}(M) \subseteq H\), we also have \((H'_2 \cup \eta) \cap \text{dom}(M) = \emptyset\). Therefore \(H' \cup \{\eta\} \subseteq_{M} H''\), so \(\hat{a}@H' \cup \{\eta\} \subseteq_{A,M} \hat{a}@H''\).

\(\Box\)

I.8 Deterministic Marking Lemma

Lemma (Deterministic Marking). For all \(\hat{v}, \hat{v}', H, H', \text{ and } H'\), if \(\text{Mark}(\hat{v}, H) = \langle \hat{v}', H' \rangle\), then \(\langle \hat{v}', H' \rangle \in \text{Markings}(\hat{v}, H)\).

Proof. Straightforward structural induction on \(\hat{v}\).

I.9 Marker Soundness Lemma

Lemma (Marker Soundness). For all \(\hat{v}, \hat{v}', \hat{H}_C, H'_C, H_A, A, \text{ and } M\) such that

• \(\text{Mark}(\hat{v}, H_C) = \langle \hat{v}', H'_C \rangle\),

• \(|\hat{v}| \subseteq_{A,M} \hat{v}\),

• \(H_C \subseteq_{M} H_A\),

• \(H_C\) includes every marker appearing in \(\hat{v}\),

• \(M\) is one-to-one, and

• \(\text{dom}(M) \subseteq C\),

there exist \(\hat{v}', H'_A, \text{ and } M'\) such that

• \(\text{Mark}(\hat{v}, H_A) = \langle \hat{v}', H'_A \rangle\),

• \(|\hat{v}'| \subseteq_{A,M=M'} H'\),

• \(H_C \subseteq_{M} H'_C \subseteq_{M} H_A \subseteq_{M} H'_A\),

\(\Box\)
1.9. MARKER SOUNDNESS LEMMA

- $M \cup M'$ is one-to-one, and
- $\text{dom}(M') \subseteq H'_C$

Proof. The proof is by structural induction on $\vec{v}$. The cases for addresses, variants, lists, numbers, and strings are given below. The cases for records and fold-expressions are similar to the one for variants, and the case for dictionaries is similar to the one for lists.

Case: $\vec{v} = a \circ H''_C$

By the definition of $\text{Mark}$, there exists $\eta_C$ such that $\vec{v}' = a \circ H''_C \cup \{\eta_C\}$, $H'_C = \{\eta_C\}$, and $\eta_C \notin H_C$.

By the definition of $|\cdot|$ and $\sqsubseteq$, there exist $\hat{a}$ and $H''_{\hat{a}}$ such that $\vec{v} = \hat{a} \circ H''_{\hat{a}}$, $|a| \sqsubseteq_{\hat{a}} \hat{a}$, and $H'_C \sqsubseteq M H''_{\hat{a}}$. Then by the definition of $\text{Mark}$ for abstract values, there also exist $\vec{v}', H'_A$, and $\eta_A$ such that $\text{Mark}(\vec{v}, H_A) = \langle \vec{v}', H_A \cup H''_{\hat{a}} \rangle$, $\vec{v}' = \hat{a} \circ H''_{\hat{a}} \cup \{\eta_A\}$, $H'_A = \{\eta_A\}$, and $\eta_A \notin H_A$.

Let $M' = [\eta_C - \eta_A]$. Because $\text{dom}(M) \subseteq H_C$, $\eta_C \notin \text{dom}(H_C)$, so the disjoint union $M \cup M'$ is defined. Then we have that $H''_C \cup \{\eta_C\} \sqsubseteq_{M \cup M'} H''_{\hat{a}} \cup \{\eta_A\}$. Therefore, we also have $|\vec{v}'| \sqsubseteq_{M \cup M'} \vec{v}'$. Similarly, we have $H_C \cup \{\eta_C\} \sqsubseteq_{M \cup M'} H_A \cup \{\eta_A\}$, so $H_C \sqsubseteq H'_C \sqsubseteq_{M \cup M'} H_A \sqsubseteq_{M \cup M'} H''_{\hat{a}}$.

Also, because $\text{dom}(M) \subseteq H_C$ and $H_C \sqsubseteq M H_A$, $\text{rng}(M) = H_A$ and therefore $\eta_A \notin \text{rng}(M)$. As a result, $M \cup M'$ is one-to-one. Finally, $\text{dom}(M') = \{\eta_C\} = H'_C$.

Case: $\vec{v} = (\text{variant } t \; \vec{v}_1' \ldots \vec{v}_n')$

By the definition of $|\cdot|$ and $\sqsubseteq$, there exist $\vec{v}_1', \ldots, \vec{v}_n'$ such that $\vec{v} = (\text{variant } t \; \vec{v}_1' \ldots \vec{v}_n')$ and $|\vec{v}_i'| \sqsubseteq_{A,M} \vec{v}_i'$ for all $i \in 1 \ldots n$. Let $H''_{C,1} = H_C$. By the definition of $\text{Mark}$, there exist $\vec{v}_1'', \ldots, \vec{v}_n''$ and $H''_{C,2}, \ldots, H''_{C,n+1}$ such that

- $\cup_{i \in 2 \ldots n+1} H''_{C,i} = H'_C$,
- $\text{Mark}(\vec{v}_i', \cup_{j \in 1 \ldots i} H''_{C,j}) = \langle \vec{v}_i'', \cup_{j \in 1 \ldots i+1} H''_{C,j} \rangle$ for all $i \in 1 \ldots n$,
- and $\vec{v}' = (\text{variant } t \; \vec{v}_1'' \ldots \vec{v}_n'')$.

Let $H''_{A,1} = H_A$. By repeated uses of the induction hypothesis for each $i \in 1 \ldots n$, there also exist $\vec{v}_1'', \ldots, \vec{v}_n''$, $H''_{A,2}, \ldots, H''_{A,n+1}$, and $M''_1, \ldots, M''_n$ such that for all $i \in 1 \ldots n$,

- $\text{Mark}(\vec{v}_i', \cup_{j \in 1 \ldots i} H''_{A,j}) = \langle \vec{v}_i'', \cup_{j \in 1 \ldots i+1} H''_{A,j} \rangle$,
- $|\vec{v}_i''| \sqsubseteq_{A,M \cup M''_{1 \ldots i} \cup \ldots \cup M''_{i+1 \ldots n}} \vec{v}_i''$,
- $\cup_{j \in 1 \ldots i+1} H''_{C,j} \sqsubseteq_{M \cup M''_{1 \ldots i} \cup \ldots \cup M''_{i+1 \ldots n}} \cup_{j \in 1 \ldots i+1} H''_{A,j}$,
- $M \cup M''_1 \cup \ldots \cup M''_n$ is one-to-one, and
\[ \text{Case: } \hat{\nu} = \text{list } \hat{\nu}_1 \ldots \hat{\nu}_n \]

The function Mark does not add markers to abstract lists, so Mark(\(\hat{\nu}, H_A\)) = (\(\hat{\nu}, H_A\)). Let \(\hat{\nu}' = \hat{\nu}\), \(H' = \emptyset\), and \(M' = \emptyset\). By the Extra Markers lemma, we have \(|\hat{\nu}'| \subseteq A, M \equiv \hat{\nu}'\). We also have \(H_C \cap \text{dom}(M \oplus M') = \emptyset\), so \(H_C \equiv H_C' \subseteq M \oplus M' \cap H_A\). Finally, it is trivially the case that \(M \oplus M'\) is one-to-one and \(\text{dom}(M') \subseteq H_C'\).

\[ \text{Case: } \hat{\nu} = n \]

By the definition of Mark, \(\hat{\nu}' = n\) and \(H_C' = H_C\). Furthermore, by the definitions of \(|\cdot|\) and \(\subseteq\), \(\hat{\nu} = \text{abs-nat}\). Let \(\hat{\nu}' = \text{abs-nat}, H_A' = H_A\), and \(M' = M\) to complete the proof.

\[ \text{Case: } \hat{\nu} = \text{str} \]

By the definition of Mark, \(\hat{\nu}' = \text{str}\) and \(H_C' = H_C\). Furthermore, by the definitions of \(|\cdot|\) and \(\subseteq\), \(\hat{\nu} = \text{abs-string}\). Let \(\hat{\nu}' = \text{abs-string}, H_A' = H_A\), and \(M' = M\) to complete the proof. \(\square\)

### I.10 New Behavior Lemma

Note that in this lemma, the subscript G stands for the “greater” term (i.e., the approximating one), and the subscript L stands for the “lesser” term (i.e., the approximated one).

**Lemma** (New Behavior). For all \(\hat{\beta}_L, \hat{\beta}_G, \hat{\alpha}_L, \hat{\alpha}_G, \hat{b}_L, \hat{b}_G, A, \) and \(M\), if

- either \(\hat{\alpha}_G \notin \text{dom}(\hat{\beta}_G), \hat{\alpha}_L \in \text{dom}(\hat{\beta}_L),\) or \(\hat{\alpha}_G\) is collective,
- \(\hat{\beta}_L \subseteq A, M \equiv \hat{\beta}_G,\)
- \(\hat{\alpha}_L \subseteq \hat{\alpha}_G,\)
- \(\hat{b}_L \subseteq A, M \equiv \hat{b}_G,\) and
- no behavior in \(\hat{\beta}_C(\hat{\alpha}_G)\) is handling an event,
then \( \hat{\beta}_L \oplus [\hat{a}_L \rightarrow \hat{b}_L] \subseteq_M \hat{\beta}_G \oplus [\hat{a}_G \rightarrow \hat{b}_G] \).

Proof. Let \( \hat{\beta}'_L = \hat{\beta}_L \oplus [\hat{a}_L \rightarrow \hat{b}_L] \) and \( \hat{\beta}'_G = \hat{\beta}_G \oplus [\hat{a}_G \rightarrow \hat{b}_G] \). To show that \( \hat{\beta}'_L \subseteq_M \hat{\beta}'_G \), first let \( \hat{a}'_L \) be a member of \( \text{dom}(\hat{\beta}'_L) \). We must show that there exists \( \hat{a}'_G \) such that \( \hat{a}'_L \subseteq \hat{a}'_G \) and \( \hat{\beta}'_L(\hat{a}'_L) \subseteq_M \hat{\beta}'_G(\hat{a}'_G) \).

If \( \hat{a}'_L \in \text{dom}(\hat{\beta}_L) \), then because \( \hat{\beta}_L \subseteq_M \hat{\beta}_G \), there exists \( \hat{a}'_G \) such that \( \hat{\beta}_L(\hat{a}'_L) \subseteq_M \hat{\beta}_G(\hat{a}'_G) \). To show that \( \hat{\beta}'_L(\hat{a}'_L) \subseteq_M \hat{\beta}'_G(\hat{a}'_G) \), let \( \hat{b}'_L \) be a member of \( \hat{\beta}'_L(\hat{a}'_L) \). If \( \hat{b}'_L = \hat{b}_L \), then it must be the case that \( \hat{a}_L = \hat{a}'_L \) and \( \hat{a}_G = \hat{a}'_G \), and we know that \( \hat{b}_G \in \hat{\beta}_G(\hat{a}_G) \) and \( \hat{b}_L \subseteq \hat{\beta}_G(\hat{a}_G) \). Otherwise, \( \hat{b}'_L \in \hat{\beta}_G(\hat{a}'_G) \), so there exists some \( \hat{b}_G' \in \hat{\beta}_G(\hat{a}_G) \) such that \( \hat{b}'_L \subseteq_M \hat{b}_G' \). We have \( \hat{\beta}_G(\hat{a}_G) \subseteq \hat{\beta}_G(\hat{a}'_G) \), so we also have \( \hat{b}_G' \in \hat{\beta}_G(\hat{a}_G) \). Finally, no behavior in \( \hat{\beta}_G(aaddrg) \) is handling an event and at most one behavior in \( \hat{\beta}_G(aaddrg') \) is handling an event for all other addresses \( \hat{a}'_G \), so at most one behavior in \( \hat{\beta}_G(aaddrg') \) is handling an event. Therefore \( \hat{\beta}'_L(\hat{a}'_L) \subseteq_M \hat{\beta}_G(\hat{a}'_G) \).

Otherwise, \( \hat{a}'_L \notin \text{dom}(\hat{\beta}_L) \), and therefore \( \hat{a}'_L = \hat{a}_L \). In that case, we know that \( \hat{a}_L \subseteq \hat{a}_G \) and \( \hat{\beta}_L(\hat{a}_L) = \{\hat{b}_L\} \). We also know that \( \hat{b}_L \subseteq_M \hat{b}_G \) and \( \hat{b}_G \in \hat{\beta}_G(\hat{a}_G) \).

Once again no behavior in \( \hat{\beta}_G(aaddrg) \) is handling an event, so at most one behavior in \( \hat{\beta}_G(aaddrg') \) is handling an event, and therefore \( \hat{\beta}'_L(\hat{a}_L) \subseteq_M \hat{\beta}_G(\hat{a}_G) \).

Second, let \( \hat{a}_G \) be an atomic address in \( \text{dom}(\hat{\beta}_G) \). We must show there exists a unique \( \hat{a}'_G \in \text{dom}(\hat{\beta}'_L) \) such that \( \hat{a}'_L \subseteq \hat{a}_G \). If \( \hat{a}'_G \in \text{dom}(\hat{\beta}_G) \), then because \( \hat{\beta}_L \subseteq_M \hat{\beta}_G \), we know there exists a unique \( \hat{a}'_L \in \text{dom}(\hat{\beta}'_L) \) such that \( \hat{a}'_L \subseteq \hat{a}_G \). If \( \hat{a}'_G = \hat{a}_G \), then by the precondition to this lemma, \( \hat{a}_L \in \text{dom}(\hat{\beta}_L) \). Therefore \( \text{dom}(\hat{\beta}_L) = \text{dom}(\hat{\beta}'_L) \), so \( \hat{a}_L \) is the unique such address in \( \text{dom}(\hat{\beta}'_L) \). Otherwise \( \hat{a}'_G \notin \hat{a}_G \), so \( \hat{a}_L \subseteq \hat{a}_G \) and therefore \( \hat{a}_L \subseteq \hat{a}'_G \), and therefore \( \hat{a}_L \subseteq \hat{a}'_G \).

Otherwise, \( \hat{a}'_G \notin \text{dom}(\hat{\beta}_L) \). Because \( \hat{\beta}_L \subseteq_M \hat{\beta}_G \), for every member of \( \text{dom}(\hat{\beta}_L) \) there is a member of \( \text{dom}(\hat{\beta}_G) \) that approximates it. For a given \( A \), any address can be approximated by at most one address, so because \( \hat{a}'_G \notin \text{dom}(\hat{\beta}_L) \) there is no \( \hat{a}'_G \in \text{dom}(\hat{\beta}_L) \) such that \( \hat{a}'_L \subseteq \hat{a}_G \). Because \( \hat{a}'_G \notin \text{dom}(\hat{\beta}_G) \), it must be the case that \( \hat{a}'_G = \hat{a}_G \), and therefore \( \hat{a}_L \) is the unique address in \( \text{dom}(\hat{\beta}'_L) \) approximated by \( \hat{a}'_G \).

Third, let there be \( \hat{a}_L \) and \( \hat{b}_G \) such that \( \hat{b}'_G \in \hat{\beta}_G(\hat{a}_G) \) and \( \hat{b}'_G \) is handling an event. We must show there exist \( \hat{a}'_L \) and \( \hat{b}'_L \in \hat{\beta}_L(\hat{a}'_L) \) such that \( \hat{a}'_L \subseteq \hat{a}_G \) and \( \hat{b}'_L \subseteq_M \hat{b}_G \). If \( \hat{a}'_G \notin \hat{a}_G \), then \( \hat{b}_G \notin \hat{\beta}_G(\hat{a}_G) \). Then because \( \hat{\beta}_L \subseteq_M \hat{\beta}_G \), there exists some \( \hat{a}'_L \) and \( \hat{b}'_L \) such that \( \hat{b}'_L \in \hat{\beta}_L(\hat{a}'_L) \), \( \hat{b}'_L \) is handling an event, \( \hat{a}'_L \subseteq \hat{a}_G \), and \( \hat{b}'_L \subseteq_M \hat{b}_G \). Because \( \hat{a}'_G \notin \hat{a}_G \) and at most one address can approximate \( \hat{a}_L \), \( \hat{a}'_G \notin \hat{a}_L \). Therefore \( \hat{b}'_L \notin \hat{\beta}(\hat{a}'_L) \). Otherwise, \( \hat{a}'_L = \hat{a}_L \), and therefore \( \hat{b}'_L = \hat{b}_G \). We know that \( \hat{b}_L \in \hat{\beta}(\hat{a}_L) \), \( \hat{a}_L \subseteq \hat{a}_G \), and \( \hat{b}_L \subseteq_M \hat{b}_G \). Furthermore, by the definition of \( c \), \( \hat{b}_L \) must be handling a message.

We have shown that each of the three conditions of the rule for actor-behavior-map approximation hold, so \( \hat{\beta}'_L \subseteq_M \hat{\beta}_G \).

**Corollary I.10.1.** For all \( \beta, \hat{\beta}, a, \hat{a}, b, \hat{b}, A, \) and \( M \), if
APPENDIX I. PROOFS FOR ABSTRACT CONFORMANCE

- either $\hat{a} \notin \text{dom}(\hat{\beta})$, $a \in \text{dom}(\hat{\beta})$, or $\hat{a}$ is collective,
- $|\hat{\beta}| \subseteq_{A,M} \hat{\beta}$,
- $|a| \subseteq_{A} \hat{\beta}$,
- $|\hat{b}| \subseteq_{A,M} \hat{b}$, and
- no behavior in $\hat{\beta}(\hat{a})$ is handling an event,
then $|\hat{\beta}[a \rightarrow b]| \subseteq_{A,M} \hat{\beta} \uplus |a \rightarrow b|$.

Proof. By the definition of $|\cdot|$, if $a \in \text{dom}(\hat{\beta})$, then $|a| \in \text{dom}(|\hat{\beta}|)$. Also by the definition of $|\cdot|$, $|\hat{\beta}[a \rightarrow b]| = |\hat{\beta}| \uplus |a| \rightarrow |b|$. Therefore, by the New Behavior lemma, $|\hat{\beta}[a \rightarrow b]| \subseteq_{A,M} \hat{\beta} \uplus |a \rightarrow b|$.

I.11 Replaced Behavior Lemma

Note that as in the previous lemma, the subscript $G$ in this lemma stands for the “greater” term (i.e., the approximating one), and the subscript $L$ stands for the “lesser” term (i.e., the approximated one).

Lemma (Replaced Behavior). For all $\hat{\beta}_L$, $\hat{\beta}_G$, $a$, $\hat{a}$, $\hat{b}$, $\hat{b}'$, $\hat{B}$, $\hat{A}$, and $\hat{M}$, if

- $\hat{\beta}_L \subseteq_{A,M} \hat{\beta}_G$,
- $\hat{a}_L \subseteq A \hat{\beta}_G$,
- $\hat{\beta}_L(\hat{a}_L) = \hat{\beta}_L \cup \{\hat{b}_L\}$,
- $\hat{\beta}_G(\hat{a}_G) = \hat{\beta}_G \cup \{\hat{b}_G\}$,
- no behavior in $\hat{B}_G$ is handling an event,
- there is no $\hat{a}'_L \neq \hat{a}_L$ such that $\hat{a}'_L \subseteq_{A} \hat{\alpha}_G$ and some behavior in $\hat{\beta}_L(\hat{a}'_L)$ is handling an event,
- $\hat{b}'_L \subseteq_{A,M} \hat{b}_G$,

then $\hat{\beta}_L[\hat{a}_L \rightarrow \hat{\beta}_L \cup \{\hat{b}'_L\}] \subseteq_{A,M} \hat{\beta}_G[\hat{a}_G \rightarrow \hat{B}_G \cup \{\hat{b}'_G\}]$.

Proof. Let $\hat{\beta}'_L = \hat{\beta}_L[\hat{a}_L \rightarrow \hat{\beta}_L \cup \{\hat{b}'_L\}]$ and $\hat{\beta}'_G = \hat{\beta}_G[\hat{a}_G \rightarrow \hat{B}_G \cup \{\hat{b}'_G\}]$. To show that $\hat{\beta}'_L \subseteq_{A,M} \hat{\beta}'_G$, first let $\hat{a}'_L$ be a member of $\text{dom}(\hat{\beta}'_L)$. We must show that there exists $\hat{a}'_G$ such that $\hat{a}'_L \subseteq_{A} \hat{\beta}_G$ and $\hat{\beta}'_L(\hat{a}'_L) \subseteq_{A,M} \hat{\beta}'_G(\hat{a}'_G)$. Because $\hat{\beta}_L \subseteq_{A,M} \hat{\beta}_G$, we know there exists $\hat{a}'_G$ such that $\hat{a}'_L \subseteq_{A} \hat{\beta}_G$ and $\hat{\beta}_L(\hat{a}'_L) \subseteq_{A,M} \hat{\beta}_G(\hat{a}'_G)$. If $\hat{a}'_G \neq \hat{a}_G$, then $\hat{a}'_L \neq \hat{a}_L$, because an address can be approximated by at most one other address for any given $A$. In that case, $\hat{\beta}'_L(\hat{a}'_L) = \hat{\beta}_L(\hat{a}'_L)$ and $\hat{\beta}'_G(\hat{a}'_G) = \hat{\beta}_G(\hat{a}'_G)$, so $\hat{\beta}'_L(\hat{a}'_L) \subseteq_{A,M} \hat{\beta}'_G(\hat{a}'_G)$.
I.11. REPLACED BEHAVIOR LEMMA

Otherwise, \( \hat{a}_G = \hat{a}_G \) and it remains to show \( \hat{\beta}_L'(\hat{a}_L') \subseteq_{A,M} \hat{\beta}_G'\hat{a}_G \). For the first requirement for that rule, let \( \hat{b}_L'' \) be a member of \( \hat{\beta}_L'(\hat{a}_L') \). If \( \hat{b}_L'' = \hat{b}_L' \), then we know that \( \hat{b}_G' \subseteq \hat{\beta}_G'(\hat{a}_G) \) and \( \hat{b}_L'' \subseteq_{A,M} \hat{b}_G' \). Otherwise, we know there exists some \( \hat{b}_G'' \subseteq \hat{\beta}_G(\hat{a}_G) \) such that \( \hat{b}_L'' \subseteq_{A,M} \hat{b}_G'' \). By the preconditions on this lemma, \( \hat{b}_L' \) is not handling an event, so by the definition of \( \cap \) neither is \( \hat{b}_G'' \). Therefore \( \hat{b}_G'' \neq \hat{b}_G' \), and therefore \( \hat{b}_G'' \subseteq \hat{\beta}_G'(\hat{a}_G') \).

For the second requirement to show \( \hat{\beta}_L'(\hat{a}_L') \subseteq_{A,M} \hat{\beta}_G'(\hat{a}_G) \), we have that either \( \hat{\beta}_G'(\hat{a}_G') = \hat{\beta}_G(\hat{a}_G') \) or \( \hat{\beta}_G'(\hat{a}_G') = \hat{B}_G \cup \{ \hat{b}_G' \} \), so at most one behavior in \( \hat{\beta}_G'(\hat{a}_G') \) is handling an event.

Second, let \( \hat{a}_G' \) be an atomic address in \( dom(\hat{\beta}_G') \). We must show there exists a unique \( \hat{a}_L' \in dom(\hat{\beta}_L') \) such that \( \hat{a}_L' \subseteq \hat{a}_G' \). We have that \( dom(\hat{\beta}_G') = dom(\hat{\beta}_G) \), so there exists such a \( \hat{a}_L' \) in \( dom(\hat{\beta}_L) \). We also have that \( dom(\hat{\beta}_L') = dom(\hat{\beta}_L) \), so \( \hat{a}_L' \in dom(\hat{\beta}_L) \), and we're done.

Third, let there be \( \hat{a}_G' \) and \( \hat{b}_G' \) such that \( \hat{b}_G'' \subseteq \hat{\beta}_G'(\hat{a}_G') \) and \( \hat{b}_G'' \) is handling an event. We must show there exist \( \hat{a}_L' \) and \( \hat{b}_L'' \subseteq \hat{\beta}_L'(\hat{a}_L') \) such that \( \hat{a}_L'' \subseteq \hat{a}_G' \) and \( \hat{b}_L'' \subseteq_{A,M} \hat{b}_G'' \). If \( \hat{a}_G' = \hat{a}_G' \), then it must be the case that \( \hat{b}_G'' = \hat{b}_G' \), so let \( \hat{a}_L' = \hat{a}_L' \) and \( \hat{b}_L'' = \hat{b}_L' \) to complete the proof. Otherwise, \( \hat{b}_G'' \subseteq \hat{\beta}_G'(\hat{a}_G') \), and so there exist \( \hat{a}_L' \) and \( \hat{b}_G'' \subseteq \hat{\beta}_L'(\hat{a}_L') \) such that \( \hat{a}_L'' \subseteq \hat{a}_G' \) and \( \hat{b}_L'' \subseteq_{A,M} \hat{b}_G'' \). Once again an address can be approximated by at most one other address for any given \( A \), so \( \hat{a}_L' \neq \hat{a}_L' \), and therefore \( \hat{\beta}_L'(\hat{a}_L') = \hat{\beta}_L'(\hat{a}_L') \).

We have shown that each of the three conditions of the rule for actor-behavior-map approximation hold, so \( \hat{\beta}_L' \subseteq_{A,M} \hat{\beta}_G' \).

**Corollary I.11.1.** For all \( \hat{b}, \hat{\beta}, a, \hat{a}, \hat{b}, \hat{b}', \hat{B}, A, \) and \( M \), if

- \( |\hat{b}| \subseteq_{A,M} \hat{\beta} \),
- \( |a| \subseteq_A \hat{a} \),
- \( a \in dom(\hat{\beta}) \),
- \( \hat{\beta}(\hat{a}) = \hat{B} \cup \{ \hat{b} \} \),
- \( \hat{b} \) is handling an event,
- there is no \( a' \neq a \) such that \( a' \subseteq_A \hat{a} \) and \( \hat{\beta}(a') \) is handling an event, and
- \( |\hat{b}| \subseteq_{A,M} \hat{b}' \),

then \( |\hat{\beta}|(a \rightarrow \hat{b}) \subseteq_{A,M} \hat{\beta}|(a \rightarrow \hat{b} \cup \{ \hat{b} \}) | \).

**Proof.** By the above preconditions and the definition of \( |\hat{\beta}| \), we have that

- \( |\hat{b}|(a \rightarrow b') \subseteq_{A,M} \hat{\beta}|(a \rightarrow \{ \hat{b} \}) | \) for some \( b' \),
- \( |\hat{b}|(a \rightarrow \hat{b}) | = |\hat{\beta}|(a \rightarrow \{ \hat{b} \}) | \), and
- there is no \( \hat{a}' \neq |a| \) such that \( \hat{a}' \subseteq_A \hat{a} \) and \( \hat{\beta}|(\hat{a}') \) is handling an event.

Then by the Replaced Behavior lemma, \( |\hat{\beta}|(a \rightarrow \hat{b}) \subseteq_{A,M} \hat{\beta}|(a \rightarrow \hat{b} \cup \{ \hat{b} \}) | \).
I.12 Maximal Value Lemma

Lemma (Maximal Value). For all \( \hat{v}, \tau, \) and \( A \) such that

- \( \hat{v} \in \text{ExtRepMsgs}(\tau) \) and
- \( A(\hat{a}) = \hat{a} \) for all external addresses \( \hat{a} \),

there exists \( \hat{v} \) such that

- \( \hat{v} \in \text{MaxVals}(\tau) \) and
- \( |\hat{v}| \subseteq_{A,M} \hat{v} \) for all \( M \).

Proof. The proof is by structural induction on \( \hat{v} \). The cases for lists and addresses are given below. The case for dictionaries is similar to the one for lists, and the others are straightforward.

Case: \( \hat{v} = (\text{list } \hat{v}_1 \ldots \hat{v}_n) \)

By the definition of \( \text{ExtRepMsgs} \), \( \tau = (\text{List } \tau') \) for some \( \tau' \), and \( \hat{v}'_i \in \text{ExtRepMsgs}(\tau') \) for all \( i \in 1 \ldots n \). Then \( \text{MaxVals}(\tau) = \{ (\text{list } \hat{V}) \} \), where \( \hat{V} = \text{MaxVals}(\tau') \).

By the definition of \( |\cdot| \), \( |\hat{v}| = (\text{list } \bigcup_{i \in 1 \ldots n} \{ |\hat{v}'_i| \} ) \). By the induction hypothesis, for all \( i \in 1 \ldots n \), there exists some \( \hat{v}'_i \in \hat{V} \) such that \( |\hat{v}'_i| \subseteq_{A,M} \hat{v}'_i \) for all \( M \). Also because all of the elements of \( \hat{V} \) are (closed) values, we know they contain no free variables. Let \( \hat{v}' = (\text{list } \hat{V}) \) and we have that \( \hat{v}' \in \text{MaxVals}(\tau) \), and \( |\hat{v}'| \subseteq_{A,M} \hat{v}' \) for all \( M \).

Case: \( \hat{v} = (\text{addr } \ell n)@H \)

By the definition of \( \text{ExtRepMsgs} \), there exists \( \tau' \) such that \( \tau = (\text{Addr } \tau') \), \( \text{ExtRepAddr}(\tau') = (\text{addr } \ell n) \), \( \text{addr } \ell n \) is external, and \( H = \emptyset \). Then we have that \( \text{MaxVals}(\tau) = \{ (\text{collective-addr } \ell) @ \emptyset \} \).

Let \( \hat{v} = (\text{collective-addr } \ell) @ \emptyset \). By the definition of \( |\cdot| \), because \( \text{addr } \ell n \) is external, \( |\hat{v}| = (\text{collective-addr } \ell) @ \emptyset \). Then because \( A((\text{collective-addr } \ell)) = (\text{collective-addr } \ell) \subseteq_{A} (\text{collective-addr } \ell) \), we have that \( (\text{collective-addr } \ell) \subseteq_{A} (\text{collective-addr } \ell) \).

Furthermore, we have \( \emptyset \subseteq_{M} \emptyset \) for all \( M \). Therefore, we have \( |\hat{v}| \subseteq_{A,M} \hat{v} \) for all \( M \), which completes the proof.

\[ \square \]

I.13 Well-Typed Maximal Value Lemma

Lemma (Well-Typed Maximal Value). For all \( \tau \) and \( \hat{v} \) such that \( \hat{v} \in \text{MaxVals}(\tau) \), \( \emptyset, \emptyset \vdash \hat{v} : \tau \).

Proof. By a straightforward structural induction on \( \hat{v} \).

\[ \square \]
I.14 Internal Address Types Lemma

**Lemma** (Internal Address Types). For all \( \hat{v}, \hat{v}', \hat{\rho}, \hat{\rho}, A, \) and \( M \), if

- \( |\hat{v}| \subseteq_{A,M} \hat{v}, \)
- \( \text{IntAddrTypes}(\hat{v}, \tau) = \hat{\rho}, \) and
- \( \text{IntAddrTypes}(\hat{v}', \tau) = \hat{\rho}, \)

then \( |\hat{\rho}| \subseteq_{A,M} \hat{\rho}. \)

*Proof.* Straightforward structural induction on \( \hat{v}. \)

I.15 Functional-Step Soundness Lemma

**Lemma.** For all \( \hat{e}, \hat{e}', \hat{\rho}, A, \) and \( M \), if \( |\hat{e}| \subseteq_{A,M} \hat{e} \) and \( \hat{e} \rightarrow \hat{e}' \), then there exists \( \hat{e}' \) such that \( \hat{e} \rightarrow \hat{e}' \) and \( |\hat{e}'| \subseteq_{A,M} \hat{e}'. \)

*Proof.* Straightforward case analysis based on the rule enabling the step \( \hat{e} \rightarrow \hat{e}'. \)

The function \( \text{EvalAbsPrimop} \) that evaluates primitive operations on abstract values is defined such that for all \( o, \hat{v}_1, \ldots, \hat{v}_n, \hat{v}', \hat{v}_1, \ldots, \hat{v}_n, A, \) and \( M \), if \( |\hat{v}_i| \subseteq_{A,M} \hat{v}_i \) for all \( i \in 1 \ldots n \) and \( \text{EvalPrimop}(o, \hat{v}_1, \ldots, \hat{v}_n) = \hat{v}' \), there exists \( \hat{v}' \in \text{EvalAbsPrimop}(o, \hat{v}_1, \ldots, \hat{v}_n) \) such that \( |\hat{v}'| \subseteq_{A,M} \hat{v}'. \)

I.16 Merge Unreachability Lemma

**Lemma** (Merge Unreachability). For all \( \hat{v}, \hat{v}', \hat{v}'' \), if \( \text{Merge}(\hat{v}, \hat{v}') \) is not defined, then \( \hat{v} \neq \text{Merge}(\hat{v}', \hat{v}'') \)

*Proof.* Straightforward structural induction on \( \hat{v}. \)

I.17 Mergeability Preservation Lemma

**Lemma** (Mergeability Preservation). For all \( \hat{v}, \hat{v}', \) and \( \hat{v}'' \), \( \text{Merge}(\hat{v}, \hat{v}') \) and \( \text{Merge}(\hat{v}', \hat{v}'') \) are defined if and only if \( \text{Merge}(\text{Merge}(\hat{v}, \hat{v}''), \hat{v}') \) is defined.

*Proof.* Showing that \( \text{Merge}(\hat{v}, \hat{v}') \) and \( \text{Merge}(\hat{v}', \hat{v}'') \) are defined implies a definition for \( \text{Merge}(\text{Merge}(\hat{v}, \hat{v}''), \hat{v}') \) is a straightforward structural induction on \( \hat{v}. \) The other direction is a consequence of the associativity and commutativity of \( \text{Merge}. \)
I.18 Quasi-Commutativity Theorem

**Theorem (Message-Addition Quasi-Commutativity).** For all $\hat{\mu}, \hat{\alpha}_1, \hat{\alpha}_2, H_1, H_2, \hat{v}_1$, and $\hat{v}_2$, $\hat{\mu} \oplus (\hat{\alpha}_1 \oplus H_1, \hat{v}_1) \oplus (\hat{\alpha}_2 \oplus H_2, \hat{v}_2) = \hat{\mu} \oplus (\hat{\alpha}_2 \oplus H_2, \hat{v}_2) \oplus (\hat{\alpha}_1 \oplus H_1, \hat{v}_1))$.

**Proof.** By the definition of $\oplus$, there exist $\mu', \mu'', \hat{v}_1', \ldots, \hat{v}_n'$, and $m$ such that

- $\mu' = \mu'' \oplus \mu''$,
- $\forall \langle \hat{\alpha}' \oplus H', \mu' \rangle \in \text{dom}(\mu')$. $\hat{\alpha}_1 \oplus H_1 = \hat{\alpha}' \oplus H'$ and $\text{Merge}(\hat{v}_1, \mu')$ is defined,
- $\forall \langle \hat{\alpha}' \oplus H', \mu' \rangle \in \text{dom}(\mu')$. $\hat{\alpha}_1 \oplus H_1 \neq \hat{\alpha}' \oplus H'$ or $\text{Merge}(\hat{v}_1, \mu')$ is not defined,
- $\forall \langle \hat{\alpha}' \oplus H', \mu' \rangle \in \text{dom}(\mu')$. $\hat{\alpha}_1 \oplus H_1 = \hat{\alpha}' \oplus H'$ and $\text{Merge}(\hat{v}_1, \mu')$ is defined, and $\mu'' \oplus (\hat{\alpha}_1 \oplus H_1, \hat{v}_1) = \mu'' \oplus \langle \hat{\alpha}_1 \oplus H_1, \hat{v}_1, \hat{v}_1', \ldots, \hat{v}_n' \rangle \rightarrow m$.

Furthermore, for all $\langle \hat{\alpha}' \oplus H', \mu' \rangle \in \text{dom}(\mu')$, either $\hat{\alpha}_1 \oplus H_1 \neq \hat{\alpha}' \oplus H'$, or $\text{Merge}(\hat{v}_1, \mu')$ is not defined and therefore by the Merge Unreachability lemma $\hat{v} \neq \text{Merge}(\hat{v}_1, \mu')$. Therefore, $\langle \hat{\alpha}_1 \oplus H_1, \text{Merge}(\hat{v}_1, \mu') \rangle \notin \text{dom}(\mu'')$ and $\mu'' \oplus (\hat{\alpha}_1 \oplus H_1, \hat{v}_1) = \mu'' \oplus \langle \hat{\alpha}_1 \oplus H_1, \text{Merge}(\hat{v}_1, \mu') \rangle \rightarrow m$. The remainder of the proof breaks down into two cases.

**Case: $\hat{\alpha}_1 \oplus H_1 = \hat{\alpha}_2 \oplus H_2$ and $\text{Merge}(\hat{v}_1, \hat{v}_2)$ is defined**

By the Mergeability Preservation lemma, $\text{Merge}(\hat{v}_2, \text{Merge}(\hat{v}_1, \hat{v}_1', \ldots, \hat{v}_n'))$ is defined, and for all $\langle \hat{\alpha}' \oplus H', \mu' \rangle \in \text{dom}(\mu'')$, either $\hat{\alpha}_2 \oplus H_2 \neq \hat{\alpha}' \oplus H'$ or $\text{Merge}(\hat{v}_2, \mu')$ is not defined. Therefore, $\mu'' \oplus (\hat{\alpha}_1 \oplus H_1, \hat{v}_1) \oplus (\hat{\alpha}_2 \oplus H_2, \hat{v}_2) = \mu'' \oplus \langle \hat{\alpha}_1 \oplus H_1, \text{Merge}(\hat{v}_2, \mu') \rangle \rightarrow m$. The quantity must be many because we are merging at least two messages together.

By a similar argument, we can show that $\mu'' \oplus (\hat{\alpha}_2 \oplus H_2, \hat{v}_2) \oplus (\hat{\alpha}_1 \oplus H_1, \hat{v}_1) = \mu'' \oplus \langle \hat{\alpha}_1 \oplus H_1, \text{Merge}(\hat{v}_2, \mu') \rangle \rightarrow m$. Then because $\text{Merge}$ is commutative and associative, we have $\mu'' \oplus (\hat{\alpha}_1 \oplus H_1, \hat{v}_1) \oplus (\hat{\alpha}_2 \oplus H_2, \hat{v}_2) = \mu'' \oplus (\hat{\alpha}_1 \oplus H_1, \hat{v}_1) \oplus (\hat{\alpha}_2 \oplus H_2, \hat{v}_2)$.

**Case: $\hat{\alpha}_1 \oplus H_1 \neq \hat{\alpha}_2 \oplus H_2$ or $\text{Merge}(\hat{v}_1, \hat{v}_2)$ is not defined**

Let $\mu_{R1} = \langle \hat{\alpha}_1 \oplus H_1, \text{Merge}(\hat{v}_1, \hat{v}_1', \ldots, \hat{v}_n') \rangle \rightarrow m$. If $\text{Merge}(\hat{v}_1, \hat{v}_2)$ is not defined, then by the Mergeability Preservation lemma, $\text{Merge}(\hat{v}_2, \text{Merge}(\hat{v}_1, \hat{v}_1', \ldots, \hat{v}_n'))$ is not defined. Therefore, there exist $\mu'''$, $\mu''''$, $\hat{v}_1'$, $\ldots$, $\hat{v}_m'$, and $m'$ such that

- $\mu'' = \mu''' \oplus \mu''''$,
- $\forall \langle \hat{\alpha}' \oplus H', \mu'' \rangle \in \text{dom}(\mu'''$. $\hat{\alpha}_2 \oplus H_2 = \hat{\alpha}' \oplus H'$ and $\text{Merge}(\hat{v}_2, \mu'' \rangle$ is defined,
- $\forall \langle \hat{\alpha}' \oplus H', \mu'' \rangle \in \text{dom}(\mu''')$. $\hat{\alpha}_2 \oplus H_2 \neq \hat{\alpha}' \oplus H'$ or $\text{Merge}(\hat{v}_2, \mu'' \rangle$ is not defined,
- $\forall \langle \hat{\alpha}' \oplus H', \mu'' \rangle \in \text{dom}(\mu''').$ $\mu'' \oplus (\hat{\alpha}_2 \oplus H_2, \hat{v}_2) \rightarrow m$. The quantity must be many.

The remainder of the proof breaks down into two cases.
I.19. APPROXIMATION MERGEABILITY LEMMA

- $\hat{\mu} \oplus (\hat{a}_1 @ H_1, \hat{v}_1) \oplus (\hat{a}_2 @ H_2, \hat{v}_2) = \left[ \left( \hat{\mu}'''' \uplus \hat{\mu}_{R_2} \right) \left( \hat{a}_2 @ H_2, \text{Merge}(\hat{v}_2, v''', \ldots, v''_p) \right) \rightarrow m' \right].$

For all $\langle \hat{a}' @ H', \hat{v}' \rangle \in \text{dom}(\hat{\mu}'')$, either $\hat{a}_1 @ H_1 \neq \hat{a}' @ H'$, or $\text{Merge}(\hat{v}_1, \hat{v}')$ is not defined and therefore by the Merge Unreachability lemma $\hat{v}' \neq \text{Merge}(\hat{v}_2, v''', \ldots, v''_p)$. Therefore, $\langle \hat{a}_2 @ H_2, \text{Merge}(\hat{v}_2, v''', \ldots, v''_p) \rangle \notin \text{dom}(\hat{\mu}''') \cup \{ \langle \hat{a}_1 @ H_1, \text{Merge}(\hat{v}_1, v'\ldots, v'_n) \rangle \}$. Let $\hat{\mu}_{R_2} = \left[ \langle \hat{a}_2 @ H_2, \text{Merge}(\hat{v}_2, v''', \ldots, v''_p) \rangle \rightarrow m' \right]$; then $\hat{\mu} \oplus (\hat{a}_1 @ H_1, \hat{v}_1) \oplus (\hat{a}_2 @ H_2, \hat{v}_2) = \hat{\mu}'''' \uplus \hat{\mu}_{R_1} \uplus \hat{\mu}_{R_2}$.

Now consider the addition of messages in the other direction, i.e., $\hat{\mu} \oplus (\hat{a}_2 @ H_2, \hat{v}_2) \oplus (\hat{a}_1 @ H_1, \hat{v}_1)$. By the above definitions, we have that $\hat{\mu} = \hat{\mu}' \uplus \hat{\mu}'''' \uplus \hat{\mu}''''.

We know that for all $\langle \hat{a}' @ H', \hat{v}' \rangle \in \text{dom}(\hat{\mu}'')$, either $\hat{a}_2 @ H_2 \neq \hat{a}' @ H'$ or $\text{Merge}(\hat{v}_2, \hat{v}')$ is not defined. We also know that for all $\langle \hat{a}' @ H', \hat{v}' \rangle \in \text{dom}(\hat{\mu})$, either $\hat{a}_2 @ H_2 \neq \hat{a}' @ H'$ or, by the Mergeability Preservation lemma, $\text{Merge}(\hat{v}_2, \hat{v}')$ is not defined. Therefore, $\hat{\mu} \oplus (\hat{a}_2 @ H_2, \hat{v}_2) = (\hat{\mu} \uplus \hat{\mu}'') \left[ \langle \hat{a}_2 @ H_2, \text{Merge}(\hat{v}_2, v''', \ldots, v''_p) \rangle \rightarrow m' \right]$. Again by the Merge Unreachability lemma, we also have $\hat{\mu} \oplus (\hat{a}_2 @ H_2, \hat{v}_2) = \hat{\mu}' \uplus \hat{\mu}''' \uplus \left[ \langle \hat{a}_2 @ H_2, \text{Merge}(\hat{v}_2, v''', \ldots, v''_p) \rangle \rightarrow m' \right]$, or written another way, $\hat{\mu} \oplus (\hat{a}_2 @ H_2, \hat{v}_2) = \hat{\mu}' \uplus \hat{\mu}''' \uplus \hat{\mu}_{R_2}$.

Again, by the Mergeability Preservation lemma, we have that for all $\langle \hat{a}' @ H', \hat{v}' \rangle \in \text{dom}(\hat{\mu}_{R_2})$, either $\hat{a}_1 @ H_1 \neq \hat{a}' @ H'$ or $\text{Merge}(\hat{v}_1, \hat{v}')$ is not defined. Also, $\text{dom}(\hat{\mu}''') \subseteq \text{dom}(\hat{\mu}'')$, so we have that for all $\langle \hat{a}' @ H', \hat{v}' \rangle \in \text{dom}(\hat{\mu}'')$, either $\hat{a}_1 @ H_1 \neq \hat{a}' @ H'$ or $\text{Merge}(\hat{v}_1, \hat{v}')$ is not defined. Therefore, $\hat{\mu} \oplus (\hat{a}_2 @ H_2, \hat{v}_2) \oplus \langle \hat{a}_1 @ H_1, \hat{v}_1 \rangle = \left[ \left( \hat{\mu}''' \uplus \hat{\mu}_{R_2} \right) \left( \langle \hat{a}_1 @ H_1, \text{Merge}(\hat{v}_1, v'\ldots, v'_n) \rangle \rightarrow m \right) \right]$. Again by the Merge Unreachability lemma, $\langle \hat{a}_1 @ H_1, \text{Merge}(\hat{v}_1, v'\ldots, v'_n) \rangle \notin \text{dom}(\hat{\mu}''' \uplus \hat{\mu}_{R_2})$. Therefore, $\hat{\mu} \oplus (\hat{a}_2 @ H_2, \hat{v}_2) \oplus (\hat{a}_1 @ H_1, \hat{v}_1) = \hat{\mu}''' \uplus \hat{\mu}_{R_2} \uplus \hat{\mu}_{R_1}$. We showed above that $\hat{\mu} \oplus (\hat{a}_1 @ H_1, \hat{v}_1) \oplus (\hat{a}_2 @ H_2, \hat{v}_2) = \hat{\mu}''' \uplus \hat{\mu}_{R_1} \uplus \hat{\mu}_{R_2}$, so we have $\hat{\mu} \oplus (\hat{a}_2 @ H_2, \hat{v}_2) \oplus (\hat{a}_1 @ H_1, \hat{v}_1) \oplus (\hat{a}_2 @ H_2, \hat{v}_2) = \hat{\mu} \oplus (\hat{a}_1 @ H_1, \hat{v}_1) \oplus (\hat{a}_2 @ H_2, \hat{v}_2).$

\[ \square \]

### I.19 Approximation Mergeability Lemma

**Lemma** (Approximation Mergeability). For all $\hat{v}_1, \hat{v}_1', \hat{v}_2, \hat{v}_2', A, \text{ and } M$, if $\hat{v}_1 \sqsubseteq_{A,M} \hat{v}_2, \hat{v}_1' \sqsubseteq_{A,M} \hat{v}_2'$, and $\text{Merge}(\hat{v}_1, \hat{v}_1')$ is defined, then $\text{Merge}(\hat{v}_2, \hat{v}_2')$ is defined.

**Proof:** The proof is by structural induction on $\hat{v}_1$. The case for addresses is given below; others are straightforward.

**Case:** $\hat{v}_1 = \hat{a}_1 @ H_1$

By the definition of $\text{Merge}(\hat{v}_1, \hat{v}_1'), \hat{v}_1' = \hat{v}_1 = \hat{a}_1 @ H_1$. By the definition of $\sqsubseteq$ for marked addresses, there can be at most one abstract marked address that approximates another abstract marked address for a given $A$ and $M$. Therefore, there exists $\hat{a}_2$ and $H_2$ such that $\hat{v}_2 = \hat{v}_2' = \hat{a}_2 @ H_2$. Therefore, $\text{Merge}(\hat{v}_2, \hat{v}_2')$ is defined.

\[ \square \]
Corollary I.19.1. For all \( \hat{v}_1, \hat{v}_2, \hat{v}_2', A, \) and \( M, \) if \( \hat{v}_1 \sqsubseteq_{A,M} \hat{v}_2 \) and \( \hat{v}_1 \sqsubseteq_{A,M} \hat{v}_2', \) then \( \text{Merge}(\hat{v}_2, \hat{v}_2') \) is defined.

Proof. Let \( \hat{v}_1' = \hat{v}_1. \) All abstract values are mergeable with themselves (i.e., \( \text{Merge}(\hat{v}, \hat{v}) \) is defined for all \( \hat{v} \)), so \( \text{Merge}(\hat{v}_1, \hat{v}_1') \) is defined. This matches the preconditions necessary for the lemma. \( \square \)

I.20 Merge Argument Lemma

Lemma. For all \( \hat{v}, \hat{v}', \hat{v}'', A, \) and \( M, \) if \( \text{Merge}(\hat{v}, \hat{v}') = \hat{v}'' \) and \( \hat{v}' \sqsubseteq_{A,M} \hat{v}'' \), then \( \hat{v} \sqsubseteq_{A,M} \hat{v}'' \) and \( \hat{v}' \sqsubseteq_{A,M} \hat{v}'' \).

Proof. The proof is by structural induction on \( \hat{v} \). The case for lists is shown below. The case for dictionaries is similar to the one for lists, and the others are straightforward.

Case: \( \hat{v} = \text{(list } \{\hat{v}_1, \ldots, \hat{v}_n\} \) \)

By the definition of \( \text{Merge} \), there exist \( \hat{v}_1', \ldots, \hat{v}_m' \) such that \( \hat{v}' = \text{(list } \{\hat{v}_1', \ldots, \hat{v}_m'\} \) \) and \( \hat{v}'' = \text{(list } \{\hat{v}_1, \ldots, \hat{v}_n\}) \cup \{\hat{v}_1', \ldots, \hat{v}_m'\} \) ). Then by the definition of \( \sqsubseteq \), there exist \( \hat{v}_1'', \ldots, \hat{v}_p'' \) such that

- \( \hat{v}'' = \text{(list } \{\hat{v}_1'', \ldots, \hat{v}_p''\} \)
- for all \( i \in 1 \ldots n \), there exists \( j \in 1 \ldots p \) such that \( \hat{v}_i \sqsubseteq_{A,M} \hat{v}_j'' \), and
- for all \( i \in 1 \ldots m \), there exists \( j \in 1 \ldots p \) such that \( \hat{v}_i' \sqsubseteq_{A,M} \hat{v}_j'' \).

Also, because all of the \( \hat{v}_1'', \ldots, \hat{v}_p'' \) are (closed) values, they have no free variables. Thus, the above results are sufficient to show that \( \hat{v} \sqsubseteq_{A,M} \hat{v}'' \) and \( \hat{v}' \sqsubseteq_{A,M} \hat{v}'' \). \( \square \)

I.21 Merge Result Lemma

Lemma (Merge Result). For all \( \hat{v}, \hat{v}', \hat{v}'', A, \) and \( M, \) if \( \hat{v} \sqsubseteq_{A,M} \hat{v}' \) and \( \text{Merge}(\hat{v}', \hat{v}'') = \hat{v}'' \), then \( \hat{v} \sqsubseteq_{A,M} \hat{v}'' \).

Proof. The proof is by structural induction on \( \hat{v} \). The case for lists is shown below. The case for dictionaries is similar to the one for lists, and the others are straightforward.

Case: \( \hat{v} = \text{(list } \{\hat{v}_1, \ldots, \hat{v}_n\} \) \)

By the definition of \( \sqsubseteq \), there exist \( \hat{v}_1', \ldots, \hat{v}_m' \) such that

- \( \hat{v}' = \text{(list } \{\hat{v}_1', \ldots, \hat{v}_m'\} \),

...
I.22. MULTIPLE MERGE RESULT LEMMA

Lemma. For all \( \tilde{v}_1, \ldots, \tilde{v}_n, \tilde{v}'_1, \ldots, \tilde{v}'_m, \tilde{v}'', \tilde{v}''', A, \) and \( M \), if

- for all \( i \in 1 \ldots n \), there exists \( j \in 1 \ldots m \) such that \( \tilde{v}_i \subseteq_{A,M} \tilde{v}'_j \),
- every free variable occurring in \( \{ \tilde{v}'_1, \ldots, \tilde{v}'_m \} \) occurs in \( \{ \tilde{v}_1, \ldots, \tilde{v}_n \} \).

By the definition of \( \text{Merge} \), there exist \( \tilde{v}''_1, \ldots, \tilde{v}''_p \) such that \( \tilde{v}'' = (\text{list } \{ \tilde{v}''_1, \ldots, \tilde{v}''_p \}) \) and \( \tilde{v}''' = (\text{list } \{ \tilde{v}''_1, \ldots, \tilde{v}''_p \} \cup \{ \tilde{v}'_1, \ldots, \tilde{v}'_m \}) \). Because all of the \( \tilde{v}'_1, \ldots, \tilde{v}'_m \) appear in \( \tilde{v}''' \), and because all of the \( \tilde{v}''_1, \ldots, \tilde{v}''_p \) and \( \tilde{v}'_1, \ldots, \tilde{v}'_m \) are (closed) values, they have no free variables. Therefore, \( \tilde{v} \subseteq_{A,M} \tilde{v}''' \).

\( \square \)

I.22 Multiple Merge Result Lemma

**Lemma.** For all \( \tilde{v}_1, \ldots, \tilde{v}_n, \tilde{v}'_1, \ldots, \tilde{v}'_m, \tilde{v}'', \tilde{v}''', A, \) and \( M \), if

- for all \( i \in 1 \ldots n \) there exists \( j \in 1 \ldots m \) such that \( \tilde{v}_i \subseteq_{A,M} \tilde{v}'_j \),
- \( \text{Merge}(\tilde{v}_1, \ldots, \tilde{v}_n) = \tilde{v}'' \), and
- \( \text{Merge}(\tilde{v}'_1, \ldots, \tilde{v}'_m) = \tilde{v}''' \),

then \( \tilde{v}'' \subseteq_{A,M} \tilde{v}''' \).

**Proof.** The proof is by structural induction on \( \tilde{v}_1 \). The case for lists is shown below. The case for dictionaries is similar, and all others are straightforward.

**Case:** \( \tilde{v}_1 = (\text{list } \tilde{V}_1) \)

By the definition of \( \text{Merge} \), there exist \( \tilde{V}_2, \ldots, \tilde{V}_n \) such that for all \( i \in 2 \ldots n \), \( \tilde{v}_i = (\text{list } \tilde{V}_i) \). Then by the definition of \( \subseteq \), for all \( i \in 1 \ldots n \) there exists some \( j \in 1 \ldots m \) and some \( \tilde{V}'_j \) such that \( \tilde{v}'_j = (\text{list } \tilde{V}'_j) \) and for all \( \tilde{v}''_j \in \tilde{v}_i \) there exists some \( \tilde{v}'''_j \in \tilde{V}'_j \) such that \( \tilde{v}''_j \subseteq_{A,M} \tilde{v}'''_j \).

Then by the definition of \( \text{Merge} \), there exist \( \tilde{V}'_1, \ldots, \tilde{V}'_m \) such that for all \( j \in 1 \ldots m \), \( \tilde{v}'_j = (\text{list } \tilde{V}'_j) \). Therefore, \( \tilde{v}'' = (\text{list } \bigcup_{i \in 1 \ldots n} \tilde{V}_i) \) and \( \tilde{v}''' = (\text{list } \bigcup_{j \in 1 \ldots m} \tilde{V}'_j) \). Then for all \( \tilde{v}'''' \in \bigcup_{i \in 1 \ldots n} \tilde{V}_i \), there exists some \( \tilde{v}''''' \in \bigcup_{j \in 1 \ldots m} \tilde{V}'_j \) such that \( \tilde{v}''''' \subseteq_{A,M} \tilde{v}'''''' \). Furthermore, because all of the elements of \( \tilde{v}''''' \) are (closed) values, there are no free variables in \( \tilde{v}''''' \). Therefore, \( \tilde{v}'' \subseteq_{A,M} \tilde{v}''''' \).

\( \square \)

I.23 Fully Merged Preservation Lemma

**Lemma** (Fully Merged Preservation). For all \( \tilde{\mu}, \tilde{a}, H, \) and \( \tilde{v} \), if \( \tilde{\mu} \) is fully merged, then \( \tilde{\mu} \oplus (\tilde{a} \oplus H, \tilde{v}) \) is fully merged.

**Proof.** Let \( \tilde{\mu} \) be a fully merged abstract message map, and let there be \( \tilde{a}, H, \) and \( \tilde{v} \). By the definition of \( \oplus \) there exist \( \tilde{\mu}', \tilde{\mu}'', \tilde{v}'_1, \ldots, \tilde{v}'_n \), and \( m \) such that

- \( \tilde{\mu} = \tilde{\mu}' \cup \tilde{\mu}'' \),

- \( \tilde{a} = \tilde{a}' \cup \tilde{a}'' \),

- \( \tilde{v} = \tilde{v}' \cup \tilde{v}'' \),

- \( H = H' \cup H'' \),

then \( \tilde{\mu} \oplus (\tilde{a} \oplus H, \tilde{v}) = \tilde{\mu}' \oplus (\tilde{a}' \oplus H', \tilde{v}') \cup \tilde{\mu}'' \oplus (\tilde{a}'' \oplus H'', \tilde{v}'') \) is fully merged.

\( \square \)
Lemma (Message-Addition Soundness). For all $\bar{\mu}_1$, $\bar{\mu}_2$, $\bar{a}_1$, $\bar{a}_2$, $H_1$, $H_2$, $\bar{v}_1$, $\bar{v}_2$, $A$, and $M$, if

- $\bar{\mu}_1 \subseteq_{A,M} \bar{\mu}_2$.
- $\bar{\mu}_2$ is fully merged.
- $\bar{a}_1@H_1 \subseteq_{A,M} \bar{a}_2@H_2$, and
- $\bar{v}_1 \subseteq_{A,M} \bar{v}_2$,

then $\bar{\mu}_1 \oplus (\bar{a}_1@H_1, \bar{v}_1) \subseteq_{A,M} \bar{\mu}_2 \oplus (\bar{a}_2@H_2, \bar{v}_2)$.

Proof. By the definition of $\oplus$, for all $i \in \{1, 2\}$, there exist $\bar{\mu}'_i$, $\bar{\mu}''_i$, $\bar{v}'_{i,1}, \ldots, \bar{v}'_{i,n_i}$, and $m_i$ such that

- $\bar{\mu}_i = \bar{\mu}'_i \cup \bar{\mu}''_i$,
- $\forall \langle \bar{a}'@H', \bar{v}' \rangle \in \text{dom}(\bar{\mu})$, $\bar{a}'@H' = \bar{a}'@H$ and $\text{Merge}(\bar{v}, \bar{v}')$ is defined,
- $\forall \langle \bar{a}'@H', \bar{v}' \rangle \in \text{dom}(\bar{\mu}'_i)$, $\bar{a}'@H' \neq \bar{a}'@H$ or $\text{Merge}(\bar{v}, \bar{v}')$ is not defined,
- $\bar{\mu}'_i \oplus (\bar{a}_i@H_1, \bar{v}_{i,1}) \cup \bar{\mu}'_i \cup \bar{\mu}''_i = \{ \bar{v}' \}$, $\bar{a}_i@H_1 = \bar{a}'@H'$ and $\text{Merge}(\bar{v}, \bar{v}')$ is defined,
- $\bar{\mu}'_i \oplus (\bar{a}_i@H_1, \bar{v}_{i,1}) \cup \bar{\mu}'_i \cup \bar{\mu}''_i = \{ \bar{v}' \}$, $\bar{a}_i@H_1 \neq \bar{a}'@H'$ or $\text{Merge}(\bar{v}, \bar{v}')$ is not defined,
- $m_i = \text{single}$ if $n_i = 0$, $\text{many}$ otherwise, and
I.24. MESSAGE-ADDITION SOUNDNESS LEMMA

I.24.1 Property 1

\[ \mu_i \oplus (\hat{a}_i @ H, \hat{v}_i) = \mu''_i \left( \langle \hat{a}_i @ H, \text{Merge}(\hat{v}_i, \hat{v}'_1, \ldots, \hat{v}'_{n_i}) \rangle \rightarrow m_i \right) . \]

Furthermore, for all \( i \in \{1, 2\} \) and all \( \langle \hat{a}' @ H', \hat{v}' \rangle \in \text{dom}(\mu''_i) \), either

\[ \hat{a}_i @ H \neq \hat{a}' @ H' \], or \( \text{Merge}(\hat{v}_i, \hat{v}') \) is not defined and therefore by the Merge Unreachability lemma \( \hat{v}' \neq \text{Merge}(\hat{v}_i, \hat{v}'_1, \ldots, \hat{v}'_{n_i}) \). Therefore, for all \( i \in \{1, 2\} \),

\[ \langle \hat{a}_i @ H, \text{Merge}(\hat{v}_i, \hat{v}'_1, \ldots, \hat{v}'_{n_i}) \rangle \notin \text{dom}(\mu''_i) \text{ and } \mu_i \oplus (\hat{a}_i @ H, \hat{v}_i) = \mu''_i \omega \left( \langle \hat{a}_i @ H, \text{Merge}(\hat{v}_i, \hat{v}'_1, \ldots, \hat{v}'_{n_i}) \rangle \rightarrow m_i \right) . \]

By the definition of \( \subset \) for message maps \( \mu \), to show that \( \mu_1 \oplus (\hat{a}_1 @ H, \hat{v}_1) \subset_{A, M} \mu_2 \oplus (\hat{a}_2 @ H, \hat{v}_2) \), there are three properties to prove:

1. every message in \( \mu_1 \oplus (\hat{a}_1 @ H, \hat{v}_1) \) is approximated by a unique message (with an approximating quantity) in \( \mu_2 \oplus (\hat{a}_2 @ H, \hat{v}_2) \),

2. every single message in \( \mu_2 \oplus (\hat{a}_2 @ H, \hat{v}_2) \) approximates a unique single message in \( \mu_1 \oplus (\hat{a}_1 @ H, \hat{v}_1) \), and

3. \( \mu_2 \oplus (\hat{a}_2 @ H, \hat{v}_2) \) does not contain two distinct messages that can be merged.

We prove each of the three properties separately below.

I.24.1 Property 1

Let \( \langle \hat{a}'_1 @ H', \hat{v}'_1 \rangle \) be a member of \( \text{dom}(\mu'_1 \oplus (\hat{a}_1 @ H, \hat{v}_1)) \). By the definition of \( \text{dom}(\mu'_1 \oplus (\hat{a}_1 @ H, \hat{v}_1)) \) above, there are two cases: either \( \langle \hat{a}'_1 @ H', \hat{v}'_1 \rangle \in \text{dom}(\mu''_1) \), or \( \langle \hat{a}'_1 @ H', \hat{v}'_1 \rangle = \langle \hat{a}_1 @ H, \text{Merge}(\hat{v}_1, \hat{v}'_1, \ldots, \hat{v}'_{n_1}) \rangle \).

**Case 1:** \( \langle \hat{a}'_1 @ H', \hat{v}'_1 \rangle \in \text{dom}(\mu''_1) \)

We know that \( \text{dom}(\mu''_1) \subset \text{dom}(\mu_1) \), so by the definition of \( \mu_1 \subset_{A, M} \mu_2 \), there exists a unique \( \langle \hat{a}'_2 @ H', \hat{v}'_2 \rangle \in \text{dom}(\mu_2) \) such that \( \hat{a}'_1 @ H' \subset_{A, M} \hat{a}'_2 @ H', \hat{v}'_1 \subset_{A, M} \hat{v}'_2 \), and \( \mu_1 \langle \hat{a}'_1 @ H', \hat{v}'_1 \rangle \subset_{A, M} \mu_2 \langle \hat{a}'_2 @ H', \hat{v}'_2 \rangle \). Then by the definition of \( \text{dom}(\mu_2 \oplus (\hat{a}_2 @ H, \hat{v}_2)) \), there are two cases: either \( \langle \hat{a}'_2 @ H', \hat{v}'_2 \rangle \in \text{dom}(\mu''_2) \), or \( \exists i \in 1 \ldots n \) such that \( \hat{a}_2 @ H = \hat{a}_2 @ H \) and \( \hat{v}'_2 \subset \hat{v}'_{2,i} \).

**Subcase 1** For the first subcase, assume \( \langle \hat{a}'_2 @ H', \hat{v}'_2 \rangle \in \text{dom}(\mu''_2) \). Then we know that \( \langle \hat{a}'_2 @ H', \hat{v}'_2 \rangle \in \text{dom}(\mu_2 \oplus (\hat{a}_2 @ H, \hat{v}_2)) \). Furthermore, we know that \( \mu_1 \langle \hat{a}'_1 @ H', \hat{v}'_1 \rangle \subset_{A, M} \mu_2 \langle \hat{a}'_2 @ H', \hat{v}'_2 \rangle \), and because \( \omega \) does not change the mappings for messages in \( \mu''_1 \) or \( \mu_2 \), we have \( \mu''_1 \langle \hat{a}'_1 @ H', \hat{v}'_1 \rangle \subset_{A, M} \mu''_2 \langle \hat{a}'_2 @ H', \hat{v}'_2 \rangle \).

It remains only to show that this message is the only message in \( \text{dom}(\mu_2 \oplus (\hat{a}_2 @ H, \hat{v}_2)) \) that approximates \( \langle \hat{a}'_1 @ H', \hat{v}'_1 \rangle \). We already know that \( \langle \hat{a}'_2 @ H', \hat{v}'_2 \rangle \) is a unique such message in \( \text{dom}(\mu_2) \), so we only have to show that

\[ \langle \hat{a}_2 @ H, \text{Merge}(\hat{v}_2, \hat{v}'_{2,1}, \ldots, \hat{v}'_{2,n_2}) \rangle \text{ is not such an approximating message.} \]

Because \( \langle \hat{a}'_2 @ H', \hat{v}'_2 \rangle \in \text{dom}(\mu''_2) \), we know that either \( \hat{a}'_2 @ H' \neq \hat{a}_2 @ H \), or that \( \text{Merge}(\hat{v}'_{2,1}, \ldots, \hat{v}'_{2,n_2}) \) is not defined. By the definition of \( \subset \), there is exactly one marked address \( \hat{a}_2 @ H \) for the given \( A \) and \( M \) such that \( \hat{a}'_1 @ H' \subset_{A, M} \hat{a}_2 @ H \), so if \( \hat{a}_2 @ H \neq \hat{a}_2 @ H \), then \( \hat{a}'_1 @ H' \not\subset_{A, M} \hat{a}_2 @ H \).
APPENDIX I. PROOFS FOR ABSTRACT CONFORMANCE

Otherwise, Merge(\hat{v}_2', \hat{v}_2) is not defined. Let \hat{v}_2'' = Merge(\hat{v}_2, \hat{v}_{2,1}', \ldots, \hat{v}_{2,n_2}'). By the Mergeability Preservation theorem, Merge(\hat{v}_2', \hat{v}_2'') is also undefined. Additionally, We have that \hat{v}_2'' \subseteq_{A,M} \hat{v}_2'. By the corollary to the Approximation Mergeability lemma, if we had \hat{v}_2'' \subseteq_{A,M} \hat{v}_2', then Merge(\hat{v}_2', \hat{v}_2'') would be defined. We already know that Merge(\hat{v}_2', \hat{v}_2'') is not defined, though, so \hat{v}_2'' \not\subseteq_{A,M} \hat{v}_2'.

Subcase 2 For the second subcase, assume \exists i \in 1...n_2 such that \hat{a}_i'@H' = \hat{a}_2@H_2 and \hat{v}_i'' = \hat{v}_{2,i}'. In this case, We have that $\hat{a}_i'@H_1' \subseteq_{A,M} \hat{a}_2@H_2$. By the Merge Result lemma, \hat{v}_i'' \subseteq_{A,M} Merge(\hat{v}_2, \hat{v}_{2,1}', \ldots, \hat{v}_{2,n_2}')'. Finally, because \hat{n}_2 > 0, m_2 = many, so \hat{\mu}_2 \models (\hat{a}_2@H_2, \hat{v}_2) (\langle \hat{v}_2', \hat{v}_2'' \rangle) \exists \hat{\mu}_2 \models (\hat{a}_2@H_2, \hat{v}_2) (\langle \hat{v}_2', \hat{v}_2'' \rangle).

It remains to show that there is no message in dom(\hat{\mu}_2) that approximates \langle \hat{a}_1@H_1', \hat{v}_1'' \rangle. We have already shown that \langle \hat{a}_2@H_2', \hat{v}_2'' \rangle is the only message in dom(\hat{\mu}_2) that approximates \langle \hat{a}_1@H_1', \hat{v}_1'' \rangle, and we know that \hat{\mu}_2 = \hat{\mu}_2' \cup \hat{\mu}_2'' and \langle \hat{a}_2@H_2', \hat{v}_2'' \rangle \in dom(\hat{\mu}_2), so the property hold.

Case 2: \langle \hat{a}_1@H_1', \hat{v}_1'' \rangle = \langle \hat{a}_1@H_1, Merge(\hat{v}_1, \hat{v}_{1,1}', \ldots, \hat{v}_{1,n_1}) \rangle.

We will first show that the message \langle \hat{a}_2@H_2, Merge(\hat{v}_2, \hat{v}_{2,1}', \ldots, \hat{v}_{2,n_2}) \rangle approximates \langle \hat{a}_1@H_1', \hat{v}_1'' \rangle, then show that no message in dom(\hat{\mu}_2) approximates \langle \hat{a}_1@H_1', \hat{v}_1'' \rangle.

Let \hat{i} be a number in 1...n_1. By the definition of \hat{\mu}_1 \models (\hat{a}_1@H_1, \hat{v}_1) (\langle \hat{a}_1@H_1', \hat{v}_1'' \rangle), (\hat{a}_1@H_1, \hat{v}_1') \in dom(\hat{\mu}_1). Then because \hat{\mu}_1 \subseteq_{A,M} \hat{\mu}_2, there exists some (\langle \hat{a}''@H'', \hat{v}'' \rangle) \in dom(\hat{\mu}_2) such that \hat{a}_1'@H_1' \subseteq_{A,M} \hat{a}_1''@H'' and \hat{v}_1' \subseteq_{A,M} \hat{v}'' by the definition of \subseteq, the marked address \hat{a}_1@H_1 can be approximated by at most one marked address, so because \hat{a}_1@H_1 \subseteq_{A,M} \hat{a}_2@H_2, we have \hat{a}_1@H'' = \hat{a}_2@H_2. We also have that Merge(\hat{v}_1', \hat{v}_1) is defined and \hat{v}_1 \subseteq_{A,M} \hat{v}_1'. Then by the Approximation Mergeability lemma, Merge(\hat{v}_1', \hat{v}_1) is defined. Therefore, there exists some \hat{j} \in 1...n_2 such that \hat{v}'' = \hat{v}_j'.

Thus, for all \hat{i} \in 1...n_1, there exists \hat{j} \in 1...n_2 such that \hat{v}_i' \subseteq_{A,M} \hat{v}_j'. We also know that \hat{v}_i \subseteq_{A,M} \hat{v}_j, so by the Multiple Merge Result lemma, Merge(\hat{v}_1, \hat{v}_{1,1}', \ldots, \hat{v}_{1,n_1}) \subseteq_{A,M} Merge(\hat{v}_2, \hat{v}_{2,1}', \ldots, \hat{v}_{2,n_2}). By the above arguments, it also must be the case that if \hat{n}_1 > 0, then \hat{n}_2 > 0. Therefore, \hat{m}_1 = many implies that \hat{m}_2 = many, so we have \hat{m}_1 \subseteq \hat{m}_2.

It remains to show that no message in dom(\hat{\mu}_2) approximates \langle \hat{a}_1@H_1', \hat{v}_1'' \rangle. Let there be (\langle \hat{a}''@H'', \hat{v}'' \rangle) \in dom(\hat{\mu}_2). By the definition of \hat{\mu}_2, either \hat{a}_1''@H'' \neq \hat{a}_2@H_2, or Merge(\hat{v}'', \hat{v}_2) is not defined. In the former case, we have already shown above that \hat{a}_2@H_2 is the only marked address that approximates \hat{a}_1@H_1, so \hat{a}_1@H_1 \not\subseteq_{A,M} \hat{a}_2@H_2.

In the latter case, Merge(\hat{v}'', \hat{v}_2) is not defined. By the Mergeability Preservation theorem, Merge(\hat{v}'', Merge(\hat{v}_2, \hat{v}_{2,1}', \ldots, \hat{v}_{2,n_2}')) is also undefined. We know that Merge(\hat{v}_1, \hat{v}_{1,1}', \ldots, \hat{v}_{1,n_1}) \subseteq_{A,M} Merge(\hat{v}_2, \hat{v}_{2,1}', \ldots, \hat{v}_{2,n_2}), so by the corollary to the Approximation Mergeability theorem, Merge(\hat{v}_1, \hat{v}_{1,1}', \ldots, \hat{v}_{1,n_1}) \not\subseteq_{A,M} \hat{v}''.


Here, we must show that for all $\langle \bar{a}_2 @ H_2, \bar{v}_2 \rangle$ such that $\hat{\mu}_2 \oplus \langle \bar{a}_2 @ H_2, \bar{v}_2 \rangle (\langle \bar{a}_2 @ H_2', \bar{v}_2' \rangle)$ is \textit{single}, there exists a unique $\langle \bar{a}_1' @ H_1', \bar{v}_1' \rangle$ such that $\bar{a}_1' @ H_1' \subseteq_{A,M} \bar{a}_2 @ H_2', \bar{v}_1' \subseteq_{A,M} \bar{v}_2'$, and $\hat{\mu}_1 \oplus \langle \bar{a}_1 @ H_1, \bar{v}_1 \rangle (\langle \bar{a}_1' @ H_1', \bar{v}_1' \rangle)$ is \textit{single}.

As in the proof for property 1 above, there are two cases: either $\langle \bar{a}_2 @ H_2, \bar{v}_2 \rangle \in dom(\hat{\mu}_2)$, or $\langle \bar{a}_2 @ H_2, \bar{v}_2 \rangle = \langle \bar{a}_2 @ H_2, \text{\texttt{Merge}}(\bar{v}_2, \bar{v}_2', \ldots, \bar{v}_{n_2}) \rangle$.

Case 1: $\langle \bar{a}_2' @ H_2', \bar{v}_2' \rangle \in dom(\hat{\mu}_2)$

First, we focus on finding an approximating message for $\langle \bar{a}_2' @ H_2', \bar{v}_2' \rangle$. We know that $\hat{\mu}_2 = \hat{\mu}_2' \oplus \hat{\mu}_2''$, so because $\hat{\mu}_1 \subseteq_{A,M} \hat{\mu}_2$, there exists a unique $\langle \bar{a}_1' @ H_1', \bar{v}_1' \rangle$ such that $\bar{a}_1' @ H_1' \subseteq_{A,M} \bar{a}_2' @ H_2', \bar{v}_1' \subseteq_{A,M} \bar{v}_2'$, and $\hat{\mu}_1 \langle \bar{a}_1' @ H_1', \bar{v}_1' \rangle)$ is \textit{single}. We will show that $\langle \bar{a}_1' @ H_1', \bar{v}_1' \rangle \in dom(\hat{\mu}_1)$.

By the definition of $\hat{\mu}_2''$, either $\bar{a}_2 @ H_2' \neq \bar{a}_2 @ H_2$, or $\text{\texttt{Merge}}(\bar{v}_2, \bar{v}_2')$ is not defined. In the former case, we have already shown above that $\bar{a}_2 @ H_2$ is the only marked address that approximates $\bar{a}_1 @ H_1$, so because $\bar{a}_1' @ H_1' \subseteq_{A,M} \bar{a}_2' @ H_2'$ and $\bar{a}_2' @ H_2'$ is not defined, we have $\hat{\mu}_1 \langle \bar{a}_1' @ H_1', \bar{v}_1' \rangle \in dom(\hat{\mu}_1)$.

In the latter case, $\text{\texttt{Merge}}(\bar{v}_2, \bar{v}_2')$ is not defined. We know that $\bar{v}_1 \subseteq_{A,M} \bar{v}_2$ and $\bar{v}_1' \subseteq_{A,M} \bar{v}_2'$. By the Approximation Mergeability lemma, $\text{\texttt{Merge}}(\bar{v}_1, \bar{v}_1')$ is similarly undefined, and therefore $\langle \bar{a}_1' @ H_1', \bar{v}_1' \rangle \in dom(\hat{\mu}_1)$.

It remains to show that there is no other message in $\text{\texttt{dom}}(\hat{\mu}_1 \langle \bar{a}_1 @ H_1, \bar{v}_1 \rangle)$ that approximates $\langle \bar{a}_2 @ H_2', \bar{v}_2' \rangle$. We have already shown that $\langle \bar{a}_1' @ H_1', \bar{v}_1' \rangle \in dom(\hat{\mu}_1)$, so the problem reduces to showing that $\langle \bar{a}_1 @ H_1, \text{\texttt{Merge}}(\bar{v}_1, \bar{v}_1', \ldots, \bar{v}_{n_2}) \rangle$ is not approximated by $\langle \bar{a}_2 @ H_2, \bar{v}_2' \rangle$.

Again, we have two cases: either $\bar{a}_2' @ H_2' \neq \bar{a}_2 @ H_2$, or $\text{\texttt{Merge}}(\bar{v}_2, \bar{v}_2')$ is not defined. In the former case, by the definition of $\subset$ there is at most marked address that can approximate $\bar{a}_1 @ H_1$. Then because $\bar{a}_1 @ H_1 \subseteq_{A,M} \bar{a}_2 @ H_2$ and $\bar{a}_2 @ H_2$ is not defined, we have $\hat{\mu}_1 \langle \bar{a}_1 @ H_1, \bar{v}_1 \rangle \subseteq_{A,M} \hat{\mu}_2 @ H_2'$.

In the latter case, $\text{\texttt{Merge}}(\bar{v}_2, \bar{v}_2')$ is not defined. By the Mergeability Preservation lemma, $\text{\texttt{Merge}}(\bar{v}_2, \bar{v}_2')$ is also undefined. We also know that $\text{\texttt{Merge}}(\bar{v}_1, \bar{v}_1', \ldots, \bar{v}_{n_1}) \subseteq_{A,M} \text{\texttt{Merge}}(\bar{v}_2, \bar{v}_2', \ldots, \bar{v}_{n_2})$, so by the corollary to the Approximation Mergeability lemma, $\text{\texttt{Merge}}(\bar{v}_1, \bar{v}_1', \ldots, \bar{v}_{n_1}) = \bar{v}_2$.

Case 2: $\langle \bar{a}_2 @ H_2, \bar{v}_2 \rangle = \langle \bar{a}_2 @ H_2, \text{\texttt{Merge}}(\bar{v}_2, \bar{v}_2', \ldots, \bar{v}_{n_2}) \rangle$

Because $\hat{\mu}_2 @ \langle \bar{a}_2 @ H_2, \bar{v}_2 \rangle (\langle \bar{a}_2 @ H_2', \bar{v}_2' \rangle) = \text{\texttt{single}}$, $\text{\texttt{Merge}}(\bar{v}_2, \bar{v}_2', \ldots, \bar{v}_{n_2}) = \bar{v}_2$ by the definition of $\oplus$. We will show that $\text{\texttt{Merge}}(\bar{v}_1, \bar{v}_1', \ldots, \bar{v}_{n_1}) = \bar{v}_1$.

Let $\langle \bar{a}_1' @ H_1', \bar{v}_1' \rangle$ be a member of $\text{\texttt{dom}}(\hat{\mu}_1)$. Because $\hat{\mu}_1 \subseteq_{A,M} \hat{\mu}_2$, and in this case $\hat{\mu}_2 = \hat{\mu}_2''$, there exists some $\langle \bar{a}_2 @ H_2', \bar{v}_2' \rangle$ such that $\bar{a}_1' @ H_1' \subseteq_{A,M} \bar{a}_2 @ H_2'$ and $\bar{v}_1' \subseteq_{A,M} \bar{v}_2'$. By the definition of $\hat{\mu}_2''$, either $\bar{a}_2 @ H_2' \neq \bar{a}_2 @ H_2$, or $\text{\texttt{Merge}}(\bar{v}_2', \bar{v}_2)$ is not defined.

In the former case, by the definition of $\subset$ there is at most one marked address that can approximate $\bar{a}_1' @ H_1'$. Then because $\bar{a}_1' @ H_1' \subseteq_{A,M} \bar{a}_2 @ H_2'$ and
\[ a''_1 @ H''_2 \neq a@H_2, \quad a''_1 @ H''_1 \not\subset_{AM} a@H_2, \quad a''_1 @ H''_1 \neq a@H_1, \quad \text{so} \quad (\hat{a''}_1 @ H''_1, \hat{v''}) \in \text{dom}(\hat{\mu}_1) \].

In the latter case, \( \text{Merge}(\hat{v''}, \hat{v}_2) \) is not defined. We have that \( \hat{v}_1 \subset_{AM} \hat{v}_2 \) and \( \hat{v''} \subset_{AM} \hat{v''}_1 \), so by the Approximation Mergeability lemma, \( \text{Merge}(\hat{v''}, \hat{v}_1) \) is not defined. Therefore, \( (\hat{a''}_1 @ H''_1, \hat{v''}_1) \in \text{dom}(\text{ammap}_1) \).

The above arguments show that \( \text{dom}(\hat{\mu}_1) \subset \text{dom}(\hat{\mu}'_1) \). Then because \( \hat{\mu}_1 = \hat{\mu}'_1 \oplus \hat{\mu}_1, \hat{\mu}'_1 = \emptyset \). Therefore, \( \text{Merge}(\hat{v}'_1, \hat{v}'_1; \ldots; \hat{v}'_1; n) = \hat{v}_1 \) and \( m_1 = \text{single} \). As a result, we have that \( \hat{a}_1 @ H'_1 \subset_{AM} \hat{a}_2 @ H'_2 \), \( \hat{v}_1 \subset_{AM} \hat{v}_2 \), and \( \hat{\mu}_1 \oplus \hat{a}_1 @ H_1, \hat{v}_1) \). Therefore, there is at most one marked address that can approximate \( \hat{a''}_1 @ H''_1 \). Then because \( \hat{a''}_1 @ H''_1 \subset_{AM} \hat{a}_2 @ H'_2 \) and \( \hat{a''}_1 @ H''_1 \neq \hat{a}_2 @ H'_2, \hat{a}_1 @ H'_1 \not\subset_{AM} \hat{a}_2 @ H'_2 \). In the latter case, \( \text{Merge}(\hat{v''}, \hat{v}_2) \) is not defined. We have that \( v'' \subset_{AM} v_2, \) so by the Approximation Mergeability lemma, \( v'' \not\subset_{AM} v_2 \). Therefore, there exists no \( (\hat{a''}_1 @ H''_1, \hat{v}'') \in \text{dom}(\hat{\mu}'_1) \) such that \( \hat{a''}_1 @ H''_1 \subset_{AM} \hat{a}_2 @ H'_2 \), \( \hat{v''} \subset_{AM} \hat{v}_2 \), and \( \hat{\mu}'_1 \oplus (\hat{a''}_1 @ H''_1, \hat{v}'') \) = \text{single}.

I.24.3 Property 3

By the Fully Merged Preservation lemma.

I.25 Soundness of Abstract CSA Lemma

This lemma shows that the (restricted) abstract semantics for CSA is sound with respect to the (restricted) concrete semantics. The idea is to show that the abstract program can simulate any restricted step of the concrete program.

Lemma (Soundness of Transitions for Abstract CSA). For all \( \bar{K} = \langle \langle \hat{\hat{\beta}} \mid \hat{\mu} \mid H \rangle \rangle^{\hat{\hat{\beta}}}, \)
\[ \bar{K}' = \langle \langle \hat{\hat{\beta}}' \mid \hat{\mu}' \mid H' \rangle \rangle^{\hat{\hat{\beta}}'}, \hat{I}, \bar{K}, A, \) and \( M, \) if

- \( \bar{K} \) is a well-formed, externals-only, single-handler configuration,
- \( |\bar{K}| \subset_{AM} \bar{K} \),
- \( \bar{K} \rightarrow_{R} \bar{K}' \),
- \( \text{dom}(A) = \text{dom}(\hat{\beta}) \cup \text{ExtAddr} \),
- \( \text{dom}(M) \subseteq H, \) and
- for all \( \hat{a} \in \text{ExtAddr}, A(\hat{a}) = \hat{a} \),
then there exist \( \tilde{l}, \tilde{K}', A', \) and \( M' \) such that

- \( \tilde{K} \xrightarrow{\tilde{l}}_{\text{RA}} \tilde{K}' \),
- \( |\tilde{l}| \subseteq A_{\tilde{l}} \cup A', M \subseteq M' \),
- \( |K'| \subseteq A_{\tilde{l}} \cup A', M \subseteq M' \),
- \( \text{dom}(A') = \text{dom}(\tilde{\beta}') - \text{dom}(\tilde{\beta}) \), and
- \( \text{dom}(M') \subseteq H' - H \).

Proof: The step \( \tilde{K} \xrightarrow{l} \tilde{K}' \) is possible only if \( \tilde{K} \xrightarrow{l} \tilde{K}' \), so the proof proceeds by case analysis on the transition-relation rule that enables \( \tilde{K} \xrightarrow{l} \tilde{K}' \). Each of the below cases is divided into sections that show

1. the concrete transition can be matched by an abstract transition of the \( \xrightarrow{} \) relation with a similar rule (e.g., an A-\text{RECEIVE\EXTERNAL} transition for a concrete M-\text{RECEIVE\EXTERNAL} transition),
2. there exist \( A' \) and \( M' \) that have the required properties and relate the transition labels \( \tilde{l} \) and \( \tilde{\beta} \) and the reached configurations \( \tilde{K}' \) and \( \tilde{K} \), and
3. the abstract transition from the first step is also a valid transition of the restricted abstract-transition relation \( \xrightarrow{\text{RA}} \).

**Case: M-\text{RECEIVE\EXTERNAL}**

By the definition of this rule, there exist \( a, H_C, H'_C, H''_C, \tilde{v}, \tilde{\beta}', \tau, \tilde{Q}_1, \ldots, \tilde{Q}_n, \bar{x}, \bar{\tilde{c}} \) such that

- \( \tilde{l} = a : \text{rcv-ext}(H_C, \tilde{v}', \tau) \),
- \( \langle \tilde{Q}_1 \ldots \tilde{Q}_n, (\text{receive } x \bar{x} \bar{\tilde{c}}) \rangle \) is the behavior of the actor at \( a \) in \( K \),
- \( \text{Used}(\tilde{K}) = H'_C \),
- \( (a@H_C, \tau) \) is a receptionist on \( \tilde{K} \),
- \( \emptyset, \emptyset \vdash \tilde{v} : \tau \),
- \( (\tilde{v}', H''_C) \in \text{Markings}(\tilde{v}, H'_C) \),
- for every marked external address \( a'@H'' \) in \( \tilde{v}, H'' = \emptyset \), and
- \( \tilde{K}' \) is identical to \( \tilde{K} \) except the behavior for \( a \) is \( \langle \tilde{Q}, \bar{e}[x \leftarrow \bar{v}] \rangle \), and \( \text{Used}(\tilde{K}') = H''_C \).

Additionally, the restrictions from the transition relation \( \xrightarrow{R} \) enforce that

- every address in \( \tilde{\beta} \) is external,
• $\bar{v} \in \text{ExtRepMsgs}(\tau)$,
• no actor in $\bar{\tilde{K}}$ is handling an event, and
• $\langle \bar{v}', H''_C \rangle = \text{Mark}(\bar{v}, H''_A)$.

**A-ReceiveExternal Transition**

We focus first on showing that $\bar{\tilde{K}}$ can take a similar step with the rule A-ReceiveExternal. In the following, let $\bar{\tilde{K}} = \langle \langle \tilde{\mu} \mid H_A' \rangle \rangle^{\bar{v}}$. First, by the definitions of $|\cdot|$ and $\subseteq$, there exist $\tilde{a}, H_A, \tilde{Q}_1, \ldots, \tilde{Q}_n, \bar{v},$ and $\bar{\tilde{c}}$ such that

- $|a| \subseteq \tilde{a}, H_C \sqsubseteq H_A, H'_C \sqsubseteq H'_A, |\tilde{Q}_i| \subseteq A,M \tilde{Q}_i$ for all $i \in 1 \ldots n$, $|\bar{v}| \subseteq A,M \bar{v},$
- $\langle \tilde{Q}_1 \ldots \tilde{Q}_n, (\text{receive } x \bar{v} \bar{\tilde{c}}) \rangle \in \tilde{\mu}(\tilde{a})$, and
- $\langle \tilde{a}@H_A, \tau \rangle \in \tilde{\mu}$.

By the preconditions to this lemma, we know that $A(\tilde{a}') = \tilde{a}'$ for all abstract external addresses $\tilde{a}'$. Then by the Maximal Value lemma, there exists $\bar{v}$ such that $\bar{v} \in \text{MaxVals}(\tau)$ and $|\bar{v}| \subseteq A,M \bar{v}$. By the definition of MaxVals, $\bar{v}$ contains no internal addresses, so the IntAddrTypes side-condition for A-ReceiveExternal is vacuously true, and $H''' = \emptyset$ for every marked external address $\tilde{a}@H'''$ in $\bar{v}$. By the Well-Typed Maximal Value lemma, $\emptyset \vdash \bar{v} : \tau$.

No markers appear in $\bar{v}$ by the definition of ExtRepMsgs. We already know that $H'_C \sqsubseteq H'_A$ and $|\bar{v}| \subseteq A,M \bar{v}$, and $M$ is one-to-one by the definition of $|\bar{K}| \subseteq A,M \bar{K}$. Then by the Marker Soundness lemma there exist $\bar{v}', H''_A$, and $M'$ such that

- $\text{Mark}(\bar{v}, H'_A) = \langle \bar{v}', H''_A \sqcup H''_A \rangle$,
- $|\bar{v}'| \subseteq A,M \sqcap M' \bar{v}'$,
- $H'_C \sqsubseteq H''_C \sqsubseteq M \sqcap M'H'_A \sqcup H''_A$,
- $M \sqcap M'$ is one-to-one, and
- $\text{dom}(M') \subseteq H''_C$.

By the Deterministic Marking lemma, we also have that $\langle \bar{v}', H''_C \rangle \in \text{Markings}(\bar{v}, H''_A)$. As a result, there exist $\bar{\tilde{K}}'$ and $\tilde{\bar{\tilde{c}}} = \tilde{a} : \text{rcv-ext}(H_A', \bar{v}', \tau)$ such that $\bar{\tilde{K}} \xrightarrow{\tilde{\bar{\tilde{c}}}} \bar{\tilde{K}}'$ and $\bar{\tilde{K}}' = \langle \langle \tilde{\mu} \leftarrow \langle \tilde{Q}_1 \ldots \tilde{Q}_n, \bar{v}[x \leftarrow \bar{v}'] \rangle \rangle \mid \tilde{\mu}[H'_A \sqcup H''_A] \rangle \rangle^{\bar{v}}$.
Having shown that \( |\hat{\beta} [a \rightarrow \langle Q_1 \ldots Q_n, e[x \rightarrow \hat{v}] \rangle] | \subseteq A_{M \oplus M'} \hat{\beta} \equiv \hat{a} \rightarrow \langle Q_1 \ldots Q_n, e[x \rightarrow \hat{v}] \rangle \), where \( \hat{\beta} \) is the actor-behavior map from \( \hat{K} \).

We know from \( |\hat{K}| \subseteq_A M \hat{K} \) that \( |\hat{\beta}| \subseteq_A M \hat{\beta} \). The configuration \( \hat{K} \) is well-formed, so every marker in \( \hat{K} \) appears in \( H_C' \), and therefore \( |\hat{\beta}| \subseteq_A M \oplus M' \hat{\beta} \).

We also know that \( |\hat{\epsilon}| \subseteq A_{M \oplus M'} \hat{\epsilon} \) and \( |\hat{\nu}'| \subseteq A_{M \oplus M'} \hat{\nu}' \), so by the Abstract Substitution lemma, we have \( |\hat{\epsilon}[x \rightarrow \hat{v}]| \subseteq_A M \oplus M' \hat{\epsilon}[x \rightarrow \hat{v}] \). Therefore, \( |\langle \hat{Q}, \hat{\epsilon}[x \rightarrow \hat{v}] \rangle| \subseteq A_{M \oplus M'} \langle \hat{Q}_1 \ldots \hat{Q}_n, \hat{\epsilon}[x \rightarrow \hat{v}] \rangle \), and we already know that \( |a| \subseteq A \hat{\alpha} \).

We also know that \( a \in \text{dom}(\hat{\beta}) \), and because \( |\hat{\beta}| \subseteq_A M \oplus M' \hat{\beta}, \) no other behavior in \( \hat{\beta}(\hat{\alpha}) \) is handling an event. Then by corollary I.10.1 to the New Behavior lemma, we have \( |\hat{\beta} [a \rightarrow \langle \hat{Q}, \hat{\epsilon}[x \rightarrow \hat{v}] \rangle] | \subseteq A_{M \oplus M'} \hat{\beta} \equiv \hat{a} \rightarrow \langle \hat{Q}_1 \ldots \hat{Q}_n, \hat{\epsilon}[x \rightarrow \hat{v}] \rangle \).

Let \( A' = \emptyset \); then we have \( |I| \subseteq A_{A' \oplus M} \hat{I} \) and \( |\hat{K}'| \subseteq A_{A' \oplus M} \hat{K}' \). Also, this step creates no new internal addresses, so \( \text{dom}(A') = \text{dom}(\hat{\beta}') = \text{dom}(\hat{\beta}) \), where \( \hat{\beta}' \) is the actor-behavior map for \( \hat{K}' \).

**Restricted Transition**

Having shown that \( \hat{K} \xrightarrow{\hat{I}} \hat{K}' \), we must also show that \( \hat{K} \xrightarrow{\text{RA}} \hat{K}' \). That is, we must show that the restrictions imposed by the relation \( \xrightarrow{\text{RA}} \) do not prevent the transition. We argue for each of the conditions of \( \xrightarrow{\text{RA}} \) below.

1. We have already shown that \( \hat{K} \xrightarrow{\hat{I}} \hat{K}' \).
2. \( \hat{I} \) is not a handler-continuation label.
3. By the definitions of \( |\cdot| \) and \( \subseteq \), if an actor in \( \hat{K} \) is handling an event, then there must be an actor in \( \hat{K} \) also handling an event. But by the definition of \( \xrightarrow{\text{RA}} \), there is no such actor in \( \hat{K} \). Therefore, there is no actor handling an event in \( \hat{K} \).
4. \( \hat{I} \) is not a \textit{spawn} label.
5. We have already shown that \( \text{Mark}(\hat{\nu}, H_A') = \langle \hat{\nu}', H_A'' \rangle \) and \( \hat{\nu} \in \text{MaxVals}(\tau) \).
6. \( \hat{I} \) is not a \textit{send-ext} label.

**Case: M-RECEIVEINTERNAL**

This case is nearly identical to the previous one. The proof here focuses on the differences.

By the definition of this rule, there exist \( \hat{\beta}, \hat{\mu}, H_C, \hat{\beta}, a, Q_1, \ldots, Q_n, x, \hat{e}, \hat{\nu}, \hat{\nu}', \hat{H}_C, H''_C \) such that

- \( \hat{K} = \langle [\hat{\beta} [a \rightarrow \langle Q_1 \ldots Q_n, \text{receive } x \hat{e} \hat{\nu} \rangle] \mid \hat{\mu} \cup \{a@H'_C, \hat{\nu}\} \mid H_C] \rangle \hat{\beta} \),
- \( \langle \hat{\nu}', H''_C \rangle \in \text{Markings}(\hat{\nu}, H'_C) \) if \( H' \neq \emptyset \), else \( \langle \hat{\nu}', H''_C \rangle = \langle \hat{\nu}, H_C \rangle \).
By the Merge Argument lemma, we also have that $\hat{m}'_{\text{proof}}$ is divided into cases depending on the value of $\text{dom}$. The transition $\hat{K}'$ is of the form $\langle \hat{\beta} \mid a \rightarrow \langle \hat{Q}_1 \ldots \hat{Q}_n, \hat{e}[x \leftarrow \hat{v}'] \rangle \mid \hat{\mu} \mid H_C' \rangle^{\hat{\rho}}$.

Because $\hat{K}$ is an externals-only configuration, we also know that $H_C' = \emptyset$, so $\hat{v}' = \hat{v}$ and $H_C' = H_C$.

A-RECEIVEINTERNAL Transition

There must exist $\hat{\beta}$, $\hat{\mu}_A$, $H_A$, and $\hat{\rho}$ such that $\hat{K} = \langle \langle \hat{\beta} \mid \hat{\mu}_A \mid H_A \rangle^{\hat{\rho}} \rangle^{\hat{\rho}}$. Similar to the previous case, by the definitions of $\mid$ and $\sqsubseteq$, there exist $\hat{a}$, $\hat{Q}_1, \ldots, \hat{Q}_n$, $\hat{e}$, and $\hat{t}$ such that

- $|a| \sqsubseteq_A \hat{a}$, $H_C \sqsubseteq_M H_A$, $|\hat{Q}_i| \sqsubseteq_{A,M} \hat{Q}_i$ for all $i \in 1 \ldots n$, $|\hat{e}| \sqsubseteq_{A,M} \hat{e}$, and
- $\langle \hat{Q}_1 \ldots \hat{Q}_n, (\text{receive} x \leftarrow \hat{t}) \rangle \in \hat{\beta} (\hat{a})$.

To find a similar abstract message to receive, let $\hat{\mu}_C = |\hat{\mu} \uplus \langle a@\emptyset, \hat{v} \rangle|$. By the definition of $\mid$, there exist $\hat{v}'_1, \ldots, \hat{v}'_p$ and $\hat{\mu}'$ such that $\hat{\mu} = \hat{\mu}' \uplus \{ \langle a@\emptyset, \hat{v}'_1 \rangle, \ldots, \langle a@\emptyset, \hat{v}'_p \rangle \}$ and $\langle \langle \hat{\beta} \mid \hat{\mu}' \mid M_{\text{receive}}(\hat{v}'_1|\hat{v}'_2|\ldots|\hat{v}'_p) \rangle^{\hat{\rho}} \rangle^{\hat{\rho}} \in \text{dom}(\hat{\mu}_C)$. Then by $\hat{\mu}_C \sqsubseteq_{A,M} \hat{\mu}_A$, there exists a unique $\hat{v}$ such that

- $\langle a@\emptyset, \hat{v} \rangle \in \text{dom}(\hat{\mu}_A)$,
- $\text{Merge}(\hat{v}|\hat{v}'_1|\ldots|\hat{v}'_p) \sqsubseteq_{A,M} \hat{v}$, and
- $\hat{\mu}_C(\langle a@\emptyset \mid \text{Merge}(\hat{v}|\hat{v}'_1|\ldots|\hat{v}'_p) \rangle) \sqsubseteq \hat{\mu}_A(\langle a@\emptyset, \hat{v} \rangle)$.

By the Merge Argument lemma, we also have that $|\hat{e}| \sqsubseteq_{A,M} \hat{e}$.

Therefore, there exist $\hat{t}$ and $\hat{K}'$ such that

- $\hat{K} \xrightarrow{\hat{t}} \hat{K}'$,
- $\hat{t} = \hat{a} : \text{rcv-int}(\emptyset, \hat{v})$, and
- $\hat{K}' = \langle \langle \hat{\beta} \oplus \hat{a} \rightarrow \langle \hat{Q}_1 \ldots \hat{Q}_n, \hat{e}[x \leftarrow \hat{v}'] \rangle \mid \hat{\mu} \oplus \langle a@\emptyset, \hat{v} \rangle \mid H_A \rangle^{\hat{\rho}} \rangle^{\hat{\rho}}$.

Approximating Configuration

Let $A' = \emptyset$ and $M' = \emptyset$. The transition $\hat{K} \xrightarrow{\hat{t}} \hat{K}'$ adds no new actors or markers, so $\text{dom}(A')$ contains the addresses of all new actors, and $\text{dom}(M')$ contains all new markers.

It remains to show that $|\hat{t}| \sqsubseteq_{A,M} \hat{t}$ and $|\hat{K}'| \sqsubseteq_{A,M} \hat{K}'$. The argument is largely similar to the argument in the previous case, using the Abstract Substitution lemma and corollary I.10.1 to the New Behavior lemma to show a correspondence between the transitioned actors. It remains to show that $|\hat{t}| \sqsubseteq_{A,M} \hat{t}$.

We know that there exists some $\hat{\mu}_C$, $\hat{\mu}_A$, $m_C$, and $m_A$ such that $\hat{\mu}_C = \hat{\mu}_C \uplus \{ \langle a@\emptyset, \hat{v}'_1 \rangle, \ldots, \langle a@\emptyset, \hat{v}'_p \rangle \} \rightarrow m_C$ and $\hat{\mu}_A = \hat{\mu}_A \uplus \{ \langle a@\emptyset, \hat{v} \rangle \rightarrow m_A \}$. The proof is divided into cases depending on the value of $m_A$. 


I.25. SOUNDNESS OF ABSTRACT CSA LEMMA

Case 1  In the first case, \(m_A = \text{single}\). Then by the definition of \(\emptyset\), \(\hat{\mu}_A \bowtie (\hat{a} @ \emptyset, \hat{v}) = \hat{\mu}'_A\), so we must show that \(|\hat{\mu}| \subseteq M_{\hat{\mu}}\).

Because \(\hat{\mu}_C(\hat{a} @ \emptyset), \text{Merge}(|\hat{v}|, |\hat{v}'_1|, \ldots, |\hat{v}'_p|)\) \(\subseteq \hat{\mu}_A(\hat{a} @ \emptyset, \hat{v})\), it must be the case that \(\hat{\mu}_C(\hat{a} @ \emptyset), \text{Merge}(|\hat{v}|, |\hat{v}'_1|, \ldots, |\hat{v}'_m|) = \text{single}\). Then by the definition of \(|\emptyset|\) and \(\emptyset, p = 0\). Therefore, by the definitions of \(|\emptyset|\) and \(\emptyset, |\hat{\mu}| = \hat{\mu}'_C\). So we must show that \(\hat{\mu}'_C \subseteq M_{\hat{\mu}}\).

To show the first property for \(\hat{\mu}'_C \subseteq M_{\hat{\mu}}\), let \(\langle \hat{a}'_A @ \emptyset, \hat{v}'_A \rangle\) be a member of \(\text{dom}(\hat{\mu}'_A)\); we must show there is a unique \(\langle \hat{a}'_A @ \emptyset, \hat{v}'_A \rangle \in \text{dom}(\hat{\mu}'_A)\) such that

- \(\hat{a}'_A @ \emptyset \subseteq M_{\hat{\mu}}, \hat{a}'_A @ \emptyset,\)
- \(\hat{v}'_A \subseteq M_{\hat{\mu}}, \hat{v}'_A,\)
- \(\hat{\mu}'_C(\hat{a}'_A @ \emptyset, \hat{v}'_A) \subseteq \hat{\mu}_A(\hat{a}'_A @ \emptyset, \hat{v}'_A).\)

Because \(\hat{\mu}_C \subseteq M_{\hat{\mu}}\), we know that there exists such a \(\langle \hat{a}'_A @ \emptyset, \hat{v}'_A \rangle\) in \(\text{dom}(\hat{\mu}_A)\).

Also by the definition of \(\hat{\mu}_C \subseteq M_{\hat{\mu}}\), because \(\hat{\mu}_A(\hat{a} @ \emptyset, \hat{v}) = \text{single}\), we know that the only concrete message that the received message \((\hat{a} @ \emptyset, \hat{v})\) approximates is \(|\langle a @ \emptyset, |v| \rangle|\). Therefore, \(\hat{a}'_A @ \emptyset \neq a @ \emptyset\) and \(\hat{v}'_A \neq \hat{v}\). Therefore, the required message \(\langle \hat{a}'_A @ \emptyset, \hat{v}'_A \rangle\) is in \(\text{dom}(\hat{\mu}_A)\), and because \(\hat{\mu}_A = \hat{\mu}'_A \cup |\langle \hat{a} @ \emptyset, \hat{v} \rangle \rightarrow \text{single} |\), \(\hat{\mu}'_C(\hat{a}'_A @ \emptyset, \hat{v}'_A) \subseteq \hat{\mu}_A(\hat{a}'_A @ \emptyset, \hat{v}'_A).\)

Second, let \(\langle \hat{a}'_A @ \emptyset, \hat{v}'_A \rangle\) be a member of \(\text{dom}(\hat{\mu}_C)\) such that \(\langle \hat{a}'_A @ \emptyset, \hat{v}'_A \rangle = \text{single}\). We must show there is a unique \(\langle \hat{a}'_A @ \emptyset, \hat{v}'_A \rangle \in \text{dom}(\hat{\mu}_C)\) such that

- \(\hat{a}'_A @ \emptyset \subseteq M_{\hat{\mu}}, \hat{a}'_A @ \emptyset,\)
- \(\hat{v}'_A \subseteq M_{\hat{\mu}}, \hat{v}'_A,\)
- \(\hat{\mu}'_C(\hat{a}'_A @ \emptyset, \hat{v}'_A) = \text{single}.\)

Because \(\hat{\mu}_C \subseteq M_{\hat{\mu}}\), we know that there exists such a \(\langle \hat{a}'_A @ \emptyset, \hat{v}'_A \rangle\) in \(\text{dom}(\hat{\mu}_C)\).

Also by the definition of \(\hat{\mu}_C \subseteq M_{\hat{\mu}}\), we know \(|\langle a @ \emptyset, |v| \rangle|\) is approximated only by \((\hat{a} @ \emptyset, \hat{v})\). Therefore, \(\hat{a}'_A @ \emptyset \neq a @ \emptyset\) and \(\hat{v}'_A \neq |v|\). Therefore, \(\langle \hat{a}'_A @ \emptyset, \hat{v}'_A \rangle\) \(\in \text{dom}(\hat{\mu}_C)\), and because \(\hat{\mu}_C = \hat{\mu}'_C \cup |\langle \hat{a} @ \emptyset, \hat{v} \rangle \rightarrow \text{single} |\), \(\hat{\mu}'_C(\hat{a}'_A @ \emptyset, \hat{v}'_A) = \text{single}.\)

Finally, we must show that \(\hat{\mu}'_C\) is fully merged. It suffices to know that \(\hat{\mu}_A\) is fully merged because \(\hat{\mu}_C \subseteq M_{\hat{\mu}}\), and that \(\hat{\mu}_A = \hat{\mu}'_A \cup |\langle \hat{a} @ \emptyset, \hat{v} \rangle \rightarrow m_A |\).

Case 2  In the second case, \(m_A = \text{many}\). Then by the definition of \(\emptyset\), \(\hat{\mu}_A \bowtie \langle \hat{a} @ \emptyset, \hat{v} \rangle = \hat{\mu}_A\), so we must show that \(|\hat{\mu}| \subseteq M_{\hat{\mu}}\).

Recall from above that \(\hat{\mu} = \hat{\mu}' \cup \langle \langle a @ \emptyset, \hat{v} \rangle, \langle a @ \emptyset, \hat{v}'_1 \rangle, \ldots, \langle a @ \emptyset, \hat{v}'_p \rangle \rangle \) and \(\hat{\mu}_C = \hat{\mu}'_C \cup \langle \langle a @ \emptyset, \text{Merge}(|\hat{v}|, |\hat{v}'_1|, \ldots, |\hat{v}'_p|) \rangle \rightarrow m_C |\). If \(p = 0\), then \(|\hat{\mu}| = \hat{\mu}_C\). Otherwise, there exists some \(m'_{\hat{\mu}}\) such that \(|\hat{\mu}| = \hat{\mu}_C \cup \langle |\langle a @ \emptyset, \text{Merge}(|\hat{v}|, |\hat{v}'_1|, \ldots, |\hat{v}'_p|) |\rangle \rightarrow m'_{\hat{\mu}} |\).

To show the first property for \(|\hat{\mu}| \subseteq M_{\hat{\mu}}\), let \(\langle \hat{a}'_A @ \emptyset, \hat{v}'_A \rangle\) be a member of \(\text{dom}(\hat{\mu}_C)\). We must show there is a unique \(\langle \hat{a}'_A @ \emptyset, \hat{v}'_A \rangle \in \text{dom}(\hat{\mu}_C)\) such that

- \(\hat{a}'_A @ \emptyset \subseteq M_{\hat{\mu}}, \hat{a}'_A @ \emptyset,\)
• $\delta_C^\prime \subseteq_{A,M} \delta_C'^n$, and

• $|\mu|(\delta_C^\prime \otimes \delta_C'^n) \subseteq \mu_A(\delta_A^\prime \otimes \delta_A'^n)$.

If $\langle \delta_C^\prime \otimes \delta_C'^n \rangle \in \text{dom}(\mu_C)$, then there exists such a $\langle \delta_A^\prime \otimes \delta_A'^n \rangle$ because $\mu_C \subseteq_{A,M} \mu_A$. Otherwise, $\langle \delta_C^\prime \otimes \delta_C'^n \rangle = \langle [a \otimes \varnothing, \text{Merge}(\delta_C'^1, \ldots, \delta_C'^p)] \rangle$ and $p > 0$. By the construction of the step $\hat{K} \rightarrow \hat{K}'$ above, we know that $\langle \hat{a} @ \varnothing, \hat{v} \rangle$ is a unique message in $\text{dom}(\mu_A)$ such that

• $[a @ \varnothing] \subseteq_{A,M} \hat{a} @ \varnothing$,

• $\text{Merge}(\delta_C'^1, \ldots, \delta_C'^p) \subseteq_{A,M} \hat{v}$, and

• $\mu_C([a @ \varnothing, \text{Merge}(\delta_C'^1, \ldots, \delta_C'^p)]) \subseteq \mu_A(\hat{a} @ \varnothing, \hat{v})$.

By the Merge Argument lemma, $\text{Merge}(\delta_C'^1, \ldots, \delta_C'^p) \subseteq_{A,M} \hat{v}$. Because $p > 0$, by the definition of $\oplus$, $\mu_C([a @ \varnothing, \text{Merge}(\delta_C'^1, \ldots, \delta_C'^p)]) = \text{many}$. Therefore, $\mu_A(\hat{a} @ \varnothing, \hat{v}) = \text{many}$ and $|\mu|(\langle [a @ \varnothing, \text{Merge}(\delta_C'^1, \ldots, \delta_C'^p)] \rangle) \subseteq \mu_A(\hat{a} @ \varnothing, \hat{v})$.

To show that no other message in $\text{dom}(\mu_A)$ can approximate $\langle [a @ \varnothing, \text{Merge}(\delta_C'^1, \ldots, \delta_C'^p)] \rangle$, let there be $\langle \delta_A^\prime \otimes \delta_A'^n \rangle \in \text{dom}(\mu_A)$ such that $\langle \delta_A^\prime @ \varnothing, \delta_A'^n \rangle \neq \langle \hat{a} @ \varnothing, \hat{v} \rangle$. Because $\mu_A$ is fully merged, either $\delta_A^\prime @ \varnothing \neq \hat{a} @ \varnothing$ or $\text{Merge}(\delta_A'^1, \hat{v})$ is not defined. In the former case, there can be at most one marked address that approximates $[a @ \varnothing]$, so $[a @ \varnothing] \not\subseteq_{A,M} \delta_A^\prime @ \varnothing$. In the latter case, we also have that $\text{Merge}(\delta_C'^1, \ldots, \delta_C'^p) \subseteq_{A,M} \hat{v}$, so by the Approximation Mergeability lemma, $\text{Merge}(\delta_C'^1, \ldots, \delta_C'^p) \not\subseteq_{A,M} \delta_A'^n$.

To show the second property for $|\mu| \subseteq_{A,M} \mu_A$, let $\langle \delta_A^\prime @ \varnothing, \delta_A'^n \rangle$ be a member of $\text{dom}(\mu_A)$ such that $\mu_A(\delta_A^\prime @ \varnothing, \delta_A'^n) = \text{single}$. We must show there is a unique $\langle \delta_C^\prime @ \varnothing, \delta_C'^n \rangle \in \text{dom}(\hat{\mu})$ such that

• $\delta_C^\prime \subseteq_{A,M} \delta_A^\prime @ \varnothing$,

• $\delta_C'^n \subseteq_{A,M} \delta_A'^n$, and

• $|\hat{\mu}|(\delta_C^\prime @ \varnothing, \delta_C'^n) = \text{single}$.

Because $\mu_C \subseteq_{A,M} \mu_A$, we know that there exists such a $\langle \delta_C^\prime @ \varnothing, \delta_C'^n \rangle$ in $\text{dom}(\mu_C)$. We must show that $\langle \delta_C^\prime @ \varnothing, \delta_C'^n \rangle \in \text{dom}(\mu_C)$, which reduces to showing that $\langle \delta_C^\prime @ \varnothing, \delta_C'^n \rangle \neq \langle [a @ \varnothing, \text{Merge}(\delta_C'^1, \ldots, \delta_C'^p)] \rangle$. Because $\mu_A(\hat{a} @ \varnothing, \hat{v}) = \text{many}$, we know that $\langle \hat{a} @ \varnothing, \hat{v} \rangle \not\subseteq_{A,M} \delta_C^\prime @ \varnothing$. Because $\mu_C \subseteq_{A,M} \mu_A$, no message in $\mu_A$ other than $\langle \hat{a} @ \varnothing, \hat{v} \rangle$ approximates $\langle [a @ \varnothing, \text{Merge}(\delta_C'^1, \ldots, \delta_C'^p)] \rangle$. Therefore, $\langle \delta_C^\prime @ \varnothing, \delta_C'^n \rangle \neq \langle [a @ \varnothing, \text{Merge}(\delta_C'^1, \ldots, \delta_C'^p)] \rangle$.

It remains to show that $\langle \delta_C^\prime @ \varnothing, \delta_C'^n \rangle$ is a unique such message in $|\hat{\mu}|$. We know that $\mu_C = \mu_C \oplus \langle [a @ \varnothing, \text{Merge}(\delta_C'^1, \ldots, \delta_C'^p)] \rangle \rightarrow m_C$ and that either
As in the previous case, we must also show that 

\[ |\bar{\mu}| = \tilde{\mu}_C \lor |\bar{\mu}| = \hat{\mu}_C \subseteq |\langle a@\emptyset, \text{Merge}(v'_1, \ldots, v'_p) \rangle \rightarrow m'_C | \text{ for some } m'_C. \]

Because \( \hat{\mu}_C \subseteq A_M \bar{\mu}_A \), we know that there is no other such corresponding message in \( \hat{\mu}_C' \). Therefore, it remains to show that \( |\langle a@\emptyset, \text{Merge}(v'_1, \ldots, v'_p) \rangle \text{ is not a corresponding message for } |\tilde{\mu}_A \subseteq \hat{\mu}_A \rangle \).

Because \( \hat{\mu}_A \) is fully merged and \( \langle \tilde{\mu}_A \subseteq \hat{\mu}_A \rangle \neq \langle \tilde{a}_A' \subseteq \hat{a}_A' \rangle \), either \( \tilde{\mu}_A \subseteq \tilde{a}_A' \neq \hat{a}_A' \subseteq \emptyset \) or \( \text{Merge}(\tilde{v}, \hat{v}') \) is undefined. In the former case, there can be at most one marked address that approximates \( |a@\emptyset| \), so \( |a@\emptyset| \subseteq A_M \hat{a}_A' \subseteq \emptyset \). In the latter case, we have that \( \text{Merge}(\tilde{v}, \hat{v}') = \emptyset \), and by the Merge Argument lemma,

\[ \text{Merge}(\tilde{v}', \hat{v}') \subseteq A_M \hat{v}. \]

Then by the Approximation Mergeability lemma,

\[ \text{Merge}(\tilde{v}', \hat{v}') \subseteq A_M \hat{v}' \cdot \]

The third property for \( |\tilde{\mu}| \subseteq A_M \hat{\mu}_A \) requires that \( \hat{\mu}_A \) be fully merged. This is guaranteed by \( \hat{\mu}_C \subseteq \hat{\mu}_A \).

**Restricted Transition**

As in the previous case, we must also show that \( \tilde{K} \rightarrow_{\text{RA}} \tilde{K}' \). We argue for each of the conditions of \( \rightarrow_{\text{RA}} \) below.

1. We have already shown that \( \tilde{K} \rightarrow_{\text{RA}} \tilde{K}' \).
2. \( \tilde{\iota} \) is not a handler-continuation label.
3. By the definitions of \( |\cdot| \) and \( \subseteq \), if an actor in \( \tilde{K} \) is handling an event, then there must be an actor in \( \tilde{K} \) also handling an event. But by the definition of \( \rightarrow_{\text{RA}} \), there is no such actor in \( \tilde{K} \). Therefore, there is no actor handling an event in \( \tilde{K} \).
4. \( \tilde{\iota} \) is not a \text{spawn} label.
5. \( \tilde{\iota} \) is not a \text{rcv-ext} label.
6. \( \tilde{\iota} \) is not a \text{send-ext} label.

**Case: M-TIMEOUT**

This case is nearly identical to the one for M-RECEIVEEXTERNAL. The proof here focuses on the differences.

By the definition of this rule, there exist \( \tilde{\beta}, \tilde{\mu}, H_C, \tilde{\rho}, a, \tilde{Q}_1, \ldots, \tilde{Q}_n, x, \tilde{\bar{e}}, n, \tilde{\bar{e}}', H'_C, \tilde{v}, \tilde{v}', \) and \( H''_C \) such that

\[
K = \langle \tilde{\beta} | a \rightarrow (\tilde{Q}_1 \ldots \tilde{Q}_n, \{ \text{receive } x \ \tilde{\bar{e}} \ [ (\text{timeout } n, \tilde{\bar{e}}') ] \}) | \tilde{\mu} | H_C \rangle^{\tilde{\beta}},
\]

\( \tilde{\iota} = a : \text{timeout} \), and

\[
K' = \langle \tilde{\beta} | a \rightarrow (\tilde{Q}_1 \ldots \tilde{Q}_n, \tilde{v}') | \tilde{\mu} | H_C \rangle^{\tilde{\beta}}.
\]
A-TIMEOUT Transition

First, we show that a similar transition from $\hat{K}$ with A-TIMEOUT is possible.

There must exist $\hat{\beta}$, $\hat{\mu}$, $H_A$, and $\hat{\rho}$ such that $\hat{K} = \langle \hat{\beta} \mid H_A \rangle^{\hat{\rho}}$. Similar to the previous case, by the definitions of $|\cdot|$ and $\sqsubset$, there exist $\hat{a}$, $\hat{Q}_1, \ldots, \hat{Q}_n$, $\hat{e}$, and $\hat{e}'$ such that

- $|a| \sqsubseteq A \hat{a}$, $H_C \sqsubseteq M H_A$, $|\hat{Q}_i| \sqsubseteq A_M \hat{Q}_i$ for all $i \in 1 \ldots n$, $|\hat{e}| \sqsubseteq A_M \hat{e}$, $|\hat{e}'| \sqsubseteq A_M \hat{e}'$, and
- $\langle \hat{Q}_1 \ldots \hat{Q}_n, \text{(receive } x \hat{e} \left[ \text{(timeout abs-nat) } \hat{e}' \right] \rangle \in \hat{\rho}(\hat{a})$.

Then by the rule A-TIMEOUT, there exist $\hat{\ell}$ and $\hat{K}'$ such that

- $\hat{K} \overset{\hat{\ell}}{\rightarrow} \hat{K}'$,
- $\hat{\ell} = \hat{a} : \text{timeout}$, and
- $\hat{K}' = \langle \hat{\beta} \oplus [\hat{a} \rightarrow \langle \hat{Q}_1 \ldots \hat{Q}_n, \hat{e}' \rangle] \mid \hat{\mu} \mid H_A \rangle^{\hat{\rho}}$.

Approximating Configuration

Let $A' = \emptyset$ and $M' = \emptyset$. The transition $\hat{K} \overset{\hat{\ell}}{\rightarrow} \hat{K}'$ adds no new actors or markers, so $\text{dom}(A')$ contains the addresses of all new actors, and $\text{dom}(M')$ contains all new markers.

The argument that $|\hat{\ell}| \sqsubseteq A_{\hat{\ell}}A', M_{\hat{\ell}}M'$ $\hat{\ell}$ and $|\hat{K}'| \sqsubseteq A_{\hat{\ell}}A', M_{\hat{\ell}}M'$ $\hat{K}'$ is similar to the argument in the M-RECEIVEINTERNAL case, using the Abstract Substitution lemma and corollary I.10.1 to the New Behavior lemma to show a correspondence between the transitioned actors; the only difference is that the message maps $\hat{\mu}$ and $\hat{\rho}$ do not change.

Restricted Transition

We must also show that $\hat{K} \overset{\hat{\ell}}{\rightarrow}_{\text{RA}} \hat{K}'$. We argue for each of the conditions of $\overset{\hat{\ell}}{\rightarrow}_{\text{RA}}$ below.

1. We have already shown that $\hat{K} \overset{\hat{\ell}}{\rightarrow} \hat{K}'$.
2. $\hat{\ell}$ is not a handler-continuation label.
3. By the definitions of $|\cdot|$ and $\sqsubseteq$, if an actor in $\hat{K}$ is handling an event, then there must be an actor in $\hat{K}$ also handling an event. But by the definition of $\overset{\hat{\ell}}{\rightarrow}_{\text{R}}$, there is no such actor in $\hat{K}$. Therefore, there is no actor handling an event in $\hat{K}$.
4. $\hat{\ell}$ is not a spawn label.
5. $\hat{\ell}$ is not a rcv-ext label.
6. $\hat{\ell}$ is not a send-ext label.
**Case: M-SendExternal**

By the definition of this rule, there exist ˘\(\hat{\beta}\), ˘\(\hat{\mu}\), ˘\(\hat{\rho}\), \(\hat{a}\), ˘\(\hat{Q}_1\), ..., ˘\(\hat{Q}_n\), ˘\(\hat{E}\), \(\hat{a}'\), ˘\(\hat{v}'\), ˘\(\hat{H}'_C\), \(\hat{v}\), and ˘\(\hat{\rho}'\) such that

- \(\hat{K} = \langle \langle ˘\hat{\beta} \mid ˘\hat{\mu} \mid ˘H_C \rangle \rangle^{˘\hat{\rho}}\),
- \(˘\hat{\beta}(a) = \langle \hat{Q}_1 \ldots \hat{Q}_n.˘\hat{E}[(\text{send } \hat{a}'@˘H'_C \hat{v})] \rangle\),
- \(\hat{a}'\) is external,
- \(\text{ActorType}(˘\hat{a}') = \tau\),
- \(\langle \hat{v}', ˘\hat{H}'_C \rangle \in \text{Markings}(\hat{v}, ˘H_C)\),
- \(\text{IntAddrTypes}(˘\hat{v}', \tau) = ˘\hat{\rho}'\),
- \(I = a : \text{send-ext}(\hat{a}'@˘H'_C, ˘\hat{v}')\), and
- \(\hat{K}' = \langle \langle ˘\hat{\beta}[a \rightarrow \langle \hat{Q}_1 \ldots \hat{Q}_n.˘\hat{E}[(\text{variant Unit})] \rangle] \mid ˘\hat{\mu} \mid ˘H'_C \rangle \rangle^{˘\hat{\rho} \cup ˘\hat{\rho}'}\).

Also, by the restrictions imposed by the transition relation \(\rightarrow_{RA}\), we have that \(\text{Mark}(\hat{v}, ˘H_C) = \langle \hat{v}', ˘H'_C \rangle\) (i.e., the message was marked deterministically).

**A-SendExternal Transition**

First, we show that a similar transition from ˘\(\hat{K}\) with A-SendExternal is possible. By the definitions of \(\mid \mid\) and \(\sqsubseteq\), there exist ˘\(\hat{\beta}\), ˘\(\hat{\mu}\), ˘\(\hat{\rho}\), \(\hat{a}\), ˘\(\hat{B}\), ˘\(\hat{Q}_1\), ..., ˘\(\hat{Q}_n\), ˘\(\hat{C}\), and ˘\(\hat{\epsilon}\) such that

- \(\hat{K} = \langle \langle ˘\hat{\beta} \mid ˘\hat{\mu} \mid ˘H_A \rangle \rangle^{˘\hat{\rho}}\),
- \(˘\hat{\beta}(\hat{a}) = ˘B \cup \{\langle \hat{Q}_1 \ldots \hat{Q}_n, \hat{C}[˘\hat{\epsilon}] \rangle\}\), and
- \(\mid \mid \hat{\beta} \mid \sqsubseteq_{A,M} ˘\hat{\beta}\), \(\mid \mid a \mid \sqsubseteq_{A} ˘\hat{a}\), \(\mid \mid \hat{Q}_i \mid \sqsubseteq_{A,M} ˘\hat{Q}_i\) for all \(i \in 1 \ldots n\),
- \(˘\hat{E}[(\text{send } \hat{a}'@˘H'_C \hat{v})] \mid \sqsubseteq_{A,M} ˘\hat{E}\), \(\mid \mid \hat{\mu} \mid \sqsubseteq_{A,M} ˘\hat{\mu}\), \(˘H_C \sqsubseteq_{M} H_A\), and \(\mid \mid \hat{\rho} \mid \sqsubseteq_{A,M} ˘\hat{\rho}\).

Then by corollary I.4.1 to the Abstract Context lemma and the definitions of \(\mid \mid\) and \(\sqsubseteq\), there exist ˘\(\hat{E}\), ˘\(\hat{\alpha}'\), ˘\(H'_A\), and ˘\(\hat{\epsilon}\) such that

- \(˘\hat{\epsilon} = ˘\hat{E}[(\text{send } \hat{\alpha}'@˘H'_A \hat{\epsilon})]\),
- \(\mid \mid \hat{E} \mid \sqsubseteq_{A,M} ˘\hat{E}\),
- \(\hat{a}'@˘H'_C \sqsubseteq_{A,M} ˘\hat{\alpha}'@˘H'_A\),
- \(\mid \mid \hat{\epsilon} \mid \sqsubseteq_{A,M} ˘\hat{\epsilon}\),
- \(\hat{\alpha}'\) is external, and
- \(\text{ActorType}(˘\hat{\alpha}') = \tau\).
APPENDIX I. PROOFS FOR ABSTRACT CONFORMANCE

If \( C = \emptyset \) and \( \hat{E} \neq \hat{E}' \cdot (\text{for/fold} \ [x \ \hat{E}'' \ [x' \ \hat{v}' \ldots \hat{v}'']) \) for some \( \hat{E}'', \hat{E}''', \hat{v}'', \) and \( \hat{v}''' \), then by the Marker Soundness lemma, there exist \( \hat{v}'', H''_A, M' \) such that

1. Mark\((\hat{v}, H_A) = (\hat{v}'', H''_A)\),
2. \( |\hat{v}'| \subseteq A_{A \cup M'} \hat{v}' \),
3. \( H''_C \subseteq M \cup M' \hat{v}'' \),
4. \( M \not\subseteq M' \) is one-to-one, and
5. \( \text{dom}(M') \subseteq H''_C - H_C \).

By the Deterministic Marking lemma, we also have \( (\hat{v}', H''_A) \in \text{Markings}(\hat{v}, H_A) \)

Otherwise, if \( \hat{E} = \hat{E}' \cdot (\text{for/fold} \ [x \ \hat{E}'' \ [x' \ \hat{v}' \ldots \hat{v}'']) \) or \( C \neq \emptyset \), let \( \hat{v}' = \hat{v}, H''_A = H_A \) and \( M' = \emptyset \). By the Extra Markers lemma, we have \( |\hat{v}'| \subseteq A_{A \cup M} \hat{v}' \). It is also trivially the case that \( \text{dom}(M') \subseteq H''_C - H_C \).

Furthermore, let \( \hat{v}' = \text{IntAddrTypes}(\hat{v}', r) \). Then there exist \( \hat{v} \) and \( \hat{K}' \) such that

1. \( \hat{K} \xrightarrow{\hat{v}} \hat{K}' \),
2. \( \hat{v} = \hat{a} : \text{send-ext}(\hat{a}'@H_A', \hat{v}) \), and
3. \( \hat{K}' = \langle \hat{v} \cdots \hat{v}' \rangle \cdot \langle \hat{Q}_1 \ldots \hat{Q}_n, C \cdot \hat{E} \cdot (\text{variant Unit}) \rangle \)

Approximating Configuration

Because \( \hat{K} \) is well-formed, \( H_C \) includes every marker appearing in \( \hat{K} \), so we have

1. \( |\hat{\beta}| \subseteq A_{A \cup M'} \hat{\beta} \),
2. \( |a| \subseteq A \hat{a} \),
3. \( |\hat{Q}_i| \subseteq A_{A \cup M'} \hat{Q}_i \) for all \( i \in 1 \ldots n \),
4. \( |\hat{E}| \subseteq A_{A \cup M'} \hat{E} \),
5. \( |\hat{\beta}| \subseteq A_{A \cup M'} \hat{\beta} \),
6. \( |\hat{\beta}| \subseteq A_{A \cup M'} \hat{\beta} \).

It is trivially the case that \(|(\text{variant Unit})| \subseteq_{A \cup M'} (\text{variant Unit})\), so by corollary I.3.1 to the Abstract Substitution lemma, we have that \(|\hat{E} \cdot (\text{variant Unit})| \subseteq_{A \cup M'} \hat{E} \cdot (\text{variant Unit})\). Let \( \hat{b} = \langle \hat{Q}_1 \ldots \hat{Q}_n, \hat{E} \cdot (\text{variant Unit}) \rangle \) and \( \hat{b} = \langle \hat{Q}_1 \ldots \hat{Q}_n, C \cdot \hat{E} \cdot (\text{variant Unit}) \rangle \); then we have \(|\hat{b}| \subseteq_{A \cup M'} \hat{b} \). We also know that \( a \in \text{dom}(\hat{\beta}) \), and because \(|\hat{\beta}| \subseteq_{A \cup M'} \hat{\beta} \), no behavior in \( \hat{B} \) is handling an event. Also, because \( \hat{K} \) is a single-handler configuration, there is no actor at some \( a' \neq a \) approximated by \( \hat{a} \) that is handling an event. Therefore by corollary I.11.1 to the Replaced Behavior lemma, we also have \(|\hat{\beta}| a \rightarrow \hat{b} | \subseteq_{A \cup M'} \hat{\beta} \cdot a \rightarrow \hat{B} \cup \{\hat{b}\} |\).
It remains to show that $|\hat{\rho} \cup \hat{\rho}'| \sqsubseteq_{A,M \cup M'} \hat{\rho} \cup \hat{\rho}'$. We already know that $|\hat{\rho}| \sqsubseteq_{A,M \cup M'} \hat{\rho}$, and by the Internal Address Types lemma we have $|\hat{\rho}'| \sqsubseteq_{A,M \cup M'} \hat{\rho}'$, which suffices to show that $|\hat{\rho} \cup \hat{\rho}'| \sqsubseteq_{A,M \cup M'} \hat{\rho} \cup \hat{\rho}'$.

We have already shown that $dom(M') \subseteq H'_{C} - H_{C}$. Let $A' = \emptyset$. Then by the above arguments, $|\hat{K}'| \sqsubseteq_{A \sqcup A', M \cup M'} \hat{K}'$ and $|\hat{l}| \sqsubseteq_{A \sqcup A', M \cup M'} \hat{l}$. The transition $\hat{K} \xrightarrow{\hat{l}} \hat{K}'$ adds no new actors, so $dom(A')$ contains the addresses of all new actors.

**Restricted Transition**

We must also show that $\hat{K} \xrightarrow{\hat{l}}_{RA} \hat{K}'$. We argue for each of the conditions of $\xrightarrow{RA}$ below.

1. We have already shown that $\hat{K} \xrightarrow{\hat{l}} \hat{K}'$.

2. By contradiction, assume there is some $\hat{a}'$ such that $\hat{a} < \hat{a}'$ and the actor at $\hat{a}'$ in $\hat{K}$ is handling an event. Because $|\hat{K}| \sqsubseteq_{A,M} \hat{K}$, there exists some $a'$ such that the actor at $a'$ in $\hat{K}$ is handling an event. Furthermore, because $|\cdot|$ preserves the locations on addresses, it must be the case that $a < a'$. This leads to a contradiction, however, because by the definition of $\hat{K} \xrightarrow{\hat{l}}_{R} \hat{K}'$, there can be no such $a'$. Therefore, there is no $\hat{a}'$ such that $\hat{a} < \hat{a}'$ and the actor at $\hat{a}'$ in $\hat{K}$ is handling an event.

3. $\hat{l}$ is not a handler-start label.

4. $\hat{l}$ is not a spawn label.

5. $\hat{l}$ is not a rcv-ext label.

6. We have already shown that either $H''_{A} = H_{A}$ or $\text{Mark}(\hat{v}, H_{A}) = \langle \hat{v}', H''_{A} \rangle$.

**Case: M-Send**

This case is similar to the previous one. The proof here focuses on the differences.

By the definition of this rule, there exist $\hat{\beta}, \hat{\mu}, H_{C}, \hat{\rho}, a, Q_{1}, \ldots, Q_{n}, \hat{E}, a', H'_{C}, \hat{v}, \hat{v}'$, and $H''_{C}$ such that

- $\hat{K} = \langle \langle \hat{\beta} \mid \hat{\mu} \mid H_{C} \rangle \rangle^{\hat{\rho}}$,
- $\hat{\beta}(a) = \langle Q_{1} \ldots Q_{n}, \hat{E} \mid (\text{send } a' @ H'_{C} \hat{v}) \rangle$,
- $a'$ is internal,
- if $H'_{C} = \emptyset$, then $\hat{v}' = \hat{v}$ and $H''_{C} = H_{C}$,
- $\hat{l} = a : \text{send-int}(a' @ H'_{C}, \hat{v}')$, and
We already know that $H_C$ is an externals-only configuration, so we also know that $H'_C = \emptyset$, so $\vartheta' = \hat{\vartheta}$ and $H''_C = H_C$.

**A-SendInternal Transition**

First, we show that a similar transition from $\hat{K}$ with A-SendInternal is possible.

By the definitions of $|\cdot|$ and $\sqsubseteq$, there exist $\hat{\beta}, \hat{\mu}, H_A, \hat{\rho}, \hat{\alpha}, \hat{B}, \hat{Q}_1, \ldots, \hat{Q}_n, C$, and $\hat{v}$ such that

- $\hat{K} = \langle \hat{\beta} | \hat{\mu} | H_A \rangle^\hat{\beta}$,
- $\hat{\beta} \hat{\alpha} = \hat{B} \sqcup \{\langle \hat{Q}_1 \ldots \hat{Q}_n, C[\hat{\vartheta}] \rangle\}$, and
- $|\hat{\beta}| \sqsubseteq_{A,M} \hat{\beta}$, $|a| \sqsubseteq A \hat{\alpha}$, $|\hat{Q}_i| \sqsubseteq_{A,M} \hat{Q}_i$ for all $i \in 1 \ldots n$,
- $|\hat{E}| (send a'@H'_C \hat{\vartheta}) | \sqsubseteq A,M \hat{\vartheta}$, $|\hat{\mu}| \sqsubseteq_{A,M} \hat{\mu}$, $H_C \sqsubseteq M H_A$, and $|\hat{\rho}| \sqsubseteq_{A,M} \hat{\rho}$.

Similar to the previous case, corollary I.4.1 to the Abstract Context lemma gives us that there exist $\hat{E}, \hat{\alpha}'$, and $\hat{v}$ such that

- $\hat{v} = \hat{E}[send \hat{\alpha}'@\emptyset \hat{v}]$,
- $|\hat{E}| \sqsubseteq_{A,M} \hat{E}$,
- $a'@\emptyset \sqsubseteq_{A,M} \hat{\alpha}'@\emptyset$,
- $|\hat{\vartheta}| \sqsubseteq_{A,M} \hat{\vartheta}$, and
- $\hat{\alpha}'$ is internal.

Let $\hat{v} = \hat{v}$ and $H'_A = H_A$. Then there exist $\hat{E}, \hat{\alpha}'$, and $\hat{v}$ such that

- $\hat{K} \xrightarrow{\hat{\alpha}} \hat{K}'$,
- $\hat{\alpha} = \hat{\alpha} : send-int(\hat{\alpha}'@\emptyset, \hat{\vartheta})$, and
- $\hat{K}' = \langle \hat{\beta} | \hat{\alpha} \rightarrow \hat{E} \cup \{\hat{b}\} | \hat{\mu} \oplus (\hat{\alpha}'@\emptyset, \hat{\vartheta}') | H''_A \rangle^\hat{\beta}$, where $\hat{b} = \langle \hat{Q}_1 \ldots \hat{Q}_n, C[\hat{E}[send (\hat{\vartheta}')]] \rangle$.

**Approximating Configuration**

We already know that $H_C \sqsubseteq M H_A$, and $|\hat{\rho}| \sqsubseteq_{A,M} \hat{\rho}$. Furthermore, we know that $|a| \sqsubseteq A \hat{\alpha}$, $a'@\emptyset \sqsubseteq_{A,M} \hat{\alpha}'@\emptyset$, and $|\hat{\vartheta}'| \sqsubseteq_{A,M} \hat{\vartheta}'$, so we have that $|\hat{\alpha}| \sqsubseteq_{A,M} \hat{\alpha}$.

The proof that $\hat{\beta} | a \rightarrow \hat{B} \sqcup \{\langle \hat{Q}_1 \ldots \hat{Q}_n, C[\hat{E}[send (\hat{\vartheta}')]] \rangle\}$ approximates $|\hat{\beta} | a \rightarrow \langle \hat{Q}_1 \ldots \hat{Q}_n, \hat{E}[send (\hat{\vartheta}')]] \rangle$ is identical to the one for the previous case, using corollaries I.3.1 and I.11.1.

It remains to show that $|\hat{\mu} \oplus (\hat{\alpha}'@\emptyset, \hat{\vartheta}')| \sqsubseteq_{A,M} \hat{\mu} \oplus (\hat{\alpha}'@\emptyset, \hat{\vartheta}')$. We know that...
• \(|\hat{\mu}| \subseteq_{A,M} \hat{\mu}\),
• \(|a'@\emptyset| \subseteq_{A,M} a'@\emptyset\), and
• \(|\hat{\nu}| \subseteq_{A,M} \hat{\nu}\).

By the definition of \(|\cdot|\), \(|\hat{\mu} \cup (a'@\emptyset, \hat{\nu}')| = |\hat{\mu} \oplus (a'@\emptyset, \hat{\nu}')|\). Furthermore, because \(|\hat{\mu}| \subseteq_{A,M} \hat{\mu}\), \(\hat{\mu}\) is fully merged. Then by the Message-Addition Soundness lemma, we have \(|\hat{\mu} \cup (a'@\emptyset, \hat{\nu}')| \subseteq_{A,M} \hat{\mu} \oplus (\hat{a'}@\emptyset, \hat{\nu}')\).

Let \(A' = \emptyset\) and \(M' = \emptyset\). The transition \(\hat{\mu} \xrightarrow{i} \hat{\nu}\) adds no new actors or markers, so \(\text{dom}(A')\) contains the addresses of all new actors, and \(\text{dom}(M')\) contains all new markers.

**Restricted Transition**

We must also show that \(\hat{\mu} \xrightarrow{i} \hat{\nu}\). We argue for each of the conditions of \(\xrightarrow{RA}\) below.

1. We have already shown that \(\hat{\mu} \xrightarrow{i} \hat{\nu}\).
2. Identical to the argument in the M-SENDEXTERNAL case above.
3. \(i\) is not a handler-start label.
4. \(i\) is not a spawn label.
5. \(i\) is not a rcv-ext label.
6. \(i\) is not a send-ext label.

**Case: M-SPAWN**

By the definition of this rule, there exist \(\hat{\beta}, \hat{\rho}, H_C, \rho, a, Q_1, \ldots, Q_n, E, \ell, \tau, \hat{e}, Q'_1, \ldots, Q'_m, a', n, \) and \(\hat{b}\) such that

- \(\hat{\mu} = \langle \hat{\beta} \mid \hat{\rho} \mid H_C \rangle^\rho\),
- \(\hat{\mu}(a) = \langle Q_1 \ldots, Q_n, E \mid (\text{spawn}^\ell \tau \hat{e} Q'_1 \ldots Q'_m) \rangle\),
- \(a' = \langle \text{addr} \ell n \rangle\),
- \(a' \notin (a) \cup \text{dom}(\hat{\beta})\),
- \(\hat{b} = \langle \hat{Q}'_1[\text{self} \leftarrow a'@\emptyset] \ldots \hat{Q}'_m[\text{self} \leftarrow a'@\emptyset], \hat{e}[\text{self} \leftarrow a'@\emptyset] \rangle\),
- \(\hat{\nu} = a: \text{spawn}(a')\), and
- \(\hat{\nu}' = \langle \hat{\beta} \mid a \leftarrow \langle \hat{Q}_1 \ldots, Q_n, E \mid a'@\emptyset \rangle \mid a' \leftarrow \hat{b} \rangle \mid \hat{\rho} \mid H_C \rangle^\hat{\nu}\).
APPENDIX I. PROOFS FOR ABSTRACT CONFORMANCE

A-SPAWN Transition

First, we show that a similar transition from $\tilde{K}$ with A-SPAWN is possible.

By the definitions of $|$ and $\subseteq$, there exist $\tilde{\beta}, \tilde{\mu}, H_A, \tilde{\rho}, \tilde{a}, B, Q_1, \ldots, Q_n, C$, and $\tilde{e}$ such that

- $\tilde{K} = \langle \tilde{\beta} \mid \tilde{\mu} \mid H_A \rangle^{\tilde{\rho}}$
- $\tilde{\beta}(\tilde{a}) = \tilde{B} \cup \{Q_1, \ldots, Q_n, C[\tilde{e}]\}$, and
- $\tilde{\beta} | \subseteq A_M \tilde{\beta}$, $|a| \subseteq A \tilde{a}$, $|Q_i| \subseteq A_M \tilde{Q}_i$ for all $i \in 1 \ldots n$, $|E| \subseteq A_M \tilde{e}$, $|\tilde{\beta}| \subseteq A_M \tilde{\mu}$, $H_C \subseteq_M H_A$, and $|\tilde{e}| \subseteq A_M \tilde{\rho}$.

By corollary I.4.1 to the Abstract Context lemma and the definitions of $|$ and $\subseteq$, there exist $\bar{E}, \bar{e},$ and $\tilde{Q}_1, \ldots, \tilde{Q}_m$ such that such that

- $\bar{e} = \bar{E}(|\text{spawn}^\ell \tau \bar{e}' \tilde{Q}_1' \ldots \tilde{Q}_m'|)$,
- $|\bar{E}| \subseteq A_M \bar{E}$,
- $|\bar{e}| \subseteq A_M \bar{e}'$, and
- $|\tilde{Q}_i'| \subseteq A_M \tilde{Q}_i'$ for all $i \in 1 \ldots m$.

If $\bar{E} = \bar{E}'(|\text{for/fold} \ [x \ \bar{E}''] \ [x' \ \bar{v}] \ \bar{e}'')|$ for some $\bar{E}', x, \bar{E}'', x', \bar{v}$, and $\bar{e}'$, then let $\tilde{a} = (\text{collective-addr } \ell)$. Otherwise, let $n$ be the smallest natural number such that $(\text{addr } \ell n) \notin \text{dom}(\tilde{\beta}) \cup \{\tilde{a}\}$, and let $\tilde{a} = (\text{addr } \ell n)$. Also, if $\bar{E} = \bar{E}'(|\text{for/fold} \ [x \ \bar{E}''] \ [x' \ \bar{v}] \ \bar{e}'') | C = (\text{in-loop } []))$, let $C' = (\text{in-loop } [])$; otherwise, let $C' = [].$ Finally, let $\bar{b} = (\tilde{Q}_1'[\text{self} \rightarrow \bar{a}' @ \phi] \ldots \tilde{Q}_m'[\text{self} \rightarrow \bar{a}' @ \phi], C'[\bar{e}' \rightarrow \bar{a}' @ \phi])$. Then there exist $\bar{\ell}$ and $\tilde{K}'$ such that

- $\bar{\ell} = \bar{a} : \text{spawn}(\bar{a}')$,
- $\tilde{K}' = \langle \tilde{\beta} | \bar{a} \rightarrow B \cup \{Q_1 \ldots Q_n, C[\bar{E}[\bar{a}' @ \phi]]\} \oplus \bar{e}' \rightarrow \bar{b} | \tilde{\mu} \mid H_A \rangle^{\tilde{\rho}}$

Approximating Configuration

Let $A' = |a' \rightarrow \bar{a}'|$. By the definition of $|\tilde{K}| \subseteq A_M \tilde{K}$, $\text{dom}(\tilde{\beta}) \subseteq \text{dom}(A)$. Then because $\tilde{K}$ is well-formed, every internal address in $\tilde{\mu}, H_C$, and $\tilde{\rho}$ is in $\text{dom}(\tilde{\beta})$.

Therefore, we have $|\tilde{\beta}| \subseteq A_{A',M} \tilde{\mu}, H_C \subseteq_M H_A$, and $|\tilde{\rho}| \subseteq A_M \tilde{\rho}$.

It remains to show that $\tilde{\beta}|a' \rightarrow B \cup \{Q_1 \ldots Q_n, E[\bar{a}' @ \phi]\}) \oplus \bar{e}' \rightarrow \bar{b}$ approximates $|\tilde{\beta}|a' \rightarrow (Q_1 \ldots Q_n, E[a' @ \phi]), a' \rightarrow \bar{b}]$. Let $\tilde{\beta}' = \tilde{\beta}|a \rightarrow (Q_1 \ldots Q_n, E[\bar{a}' @ \phi])$, and let $\tilde{\rho}' = \tilde{\beta}|a \rightarrow B \cup \{Q_1 \ldots Q_n, C[\bar{E}[\bar{a}' @ \phi]]\}$; that is, each one is the original actor-behavior map with the spawning actor updated. By the definition of $A'$, we have that $|a' @ \phi| \subseteq A_{A',M} \bar{a}' @ \phi$. Then similarly to the previous case, we can use
I.25. SOUNDNESS OF ABSTRACT CSA LEMMA

343

corollary I.3.1 to the Abstract Substitution lemma and corollary I.11.1 to the
Replaced Behavior lemma to show that \( |\beta'| \sqsubseteq_{A\oplus A',M} \beta' \).

It remains to show that \( |\beta'[a' \rightarrow b]| \sqsubseteq_{A\oplus A',M} \beta' \oplus [a' \rightarrow b] \).
The behaviors \( \hat{b} \) and \( \tilde{b} \) were defined such that \( \hat{b} = \langle Q_1'[\text{self} \rightarrow a'@\phi] \ldots Q_n'[\text{self} \rightarrow a'@\phi], e[\text{self} \rightarrow a'@\phi] \rangle \), and \( \tilde{b} = \langle Q_1'[\text{self} \rightarrow a'@\phi] \ldots Q_m'[\text{self} \rightarrow a'@\phi], C[\text{self} \rightarrow a'@\phi] \rangle \). From the above arguments, we know that \( |\check{e}| \sqsubseteq_{A,M} \check{e}' \) and \( |Q_i'| \sqsubseteq_{A,M} Q_i' \) for all \( i \in 1 \ldots m \). Furthermore, because the new concrete address \( a' \) cannot appear in \( \check{e} \) or \( Q_1' \ldots Q_m' \), we also have \( |\check{e}| \sqsubseteq_{A\oplus A',M} \check{e}' \) and \( |Q_i'| \sqsubseteq_{A\oplus A',M} Q_i' \) for all \( i \in 1 \ldots m \). Then by the Abstract Substitution lemma, we have that \( e[\text{self} \rightarrow a'@\phi] \sqsubseteq_{A\oplus A',M} \check{e}'[\text{self} \rightarrow a'@\phi] \) and \( Q_i'[\text{self} \rightarrow a'@\phi] \sqsubseteq_{A\oplus A',M} Q_i'[\text{self} \rightarrow a'@\phi] \) for all \( i \in 1 \ldots n \).

Let \( M' = \emptyset \). The transition \( \hat{K} \xrightarrow{\ell} R \hat{K}' \) adds no new markers, so \( \text{dom}(M') \) contains all new markers. We also have \( \text{dom}(A') = \{a\} \), so it contains the addresses of all new actors, and by the above arguments, \( |\hat{l}| \sqsubseteq_{A\oplus A',M \cup M'} \hat{\ell} \) and \( |\hat{K}'| \sqsubseteq_{A\oplus A',M \cup M'} \hat{K}' \).

Restricted Transition

We must also show that \( \hat{K} \xrightarrow{\ell} R \hat{K}' \). We argue for each of the conditions of \( \rightarrow_{RA} \) below.

1. We have already shown that \( \hat{K} \xrightarrow{\ell} \hat{K}' \).
2. Identical to the argument in the M-SENDEXTERNAL case above.
3. \( \hat{\ell} \) is not a handler-start label.
4. The address \( \check{a}' \) was defined with the minimum possible identifier \( n \).
5. \( \hat{\ell} \) is not a rcv-ext label.
6. \( \hat{\ell} \) is not a send-ext label.

Case: M-GOTO

By the definition of this rule, there exist \( \hat{\beta}, \hat{\mu}, H_C, \hat{\rho}, a, Q_1, \ldots, Q_n, \hat{E}, q, \check{v}_1, \ldots, \check{v}_m, x_1, \ldots, x_m, t_1, \ldots, t_m, x', \hat{\epsilon}, \hat{\iota}, \check{c} \) and \( \check{\epsilon}' \) such that

\[
\hat{K} = \langle \langle \hat{\beta} \mid \check{\mu} \mid H_C \rangle \rangle^{\hat{\rho}}.
\]
APPENDIX I. PROOFS FOR ABSTRACT CONFORMANCE

First, we show that a similar transition from $\preceq A-GOTO$ such that

- $1 \leq k \leq n$,
- $\bar{Q}_k = \langle \text{define-state} \ (q \ [x_1 \tau_1] \ldots \ [x_m \tau_m]) \ x' \ e \ \bar{c} \rangle$,
- $\bar{l} = a : \text{goto}$,
- $\hat{e}' = \langle \text{receive} \ x' \ e \ \bar{c} \rangle [x_1 \leftarrow \bar{v}_1] \ldots [x_m \leftarrow \bar{v}_m]$, and
- $\hat{K}' = \langle \hat{\beta} [a \rightarrow (\bar{Q}_1 \ldots \bar{Q}_n, \hat{e}')] \mid \hat{\mu} \mid H_C \rangle^\beta$.

A-GOTO Transition

First, we show that a similar transition from $\hat{K}$ with A-GOTO is possible. By the definitions of $\lvert \cdot \rvert$ and $\sqsubseteq$, there exist $\hat{\beta}, \hat{\mu}, H_A, \hat{\rho}, \hat{\alpha}, \bar{B}, \bar{Q}_1, \ldots, \bar{Q}_n, C$, and $\hat{e}$ such that

- $\hat{K} = \langle \hat{\beta} \mid \hat{\mu} \mid H_A \rangle^\beta$
- $\hat{\beta}(a) = \bar{B} \cup \{\langle \bar{Q}_1 \ldots \bar{Q}_n, C[e] \rangle\}$,
- $\lvert \hat{e} \rvert \sqsubseteq_{A,M} \hat{\beta}$, $\lvert a \rvert \sqsubseteq_A \hat{\alpha}$, $\lvert \bar{Q}_i \rvert \sqsubseteq_{A,M} \hat{\bar{Q}}_i$ for all $i \in 1 \ldots n$,
- $\lvert \bar{E}((\text{goto} q \bar{v}_1 \ldots \bar{v}_m)) \rvert \sqsubseteq_{A,M} \hat{\bar{e}}$, $\lvert \hat{\bar{e}} \rvert \sqsubseteq_{A,M} \hat{\bar{\mu}}$, $H_C \sqsubseteq_{A,M} H_A$, and $\lvert \hat{\bar{e}} \rvert \sqsubseteq_{A,M} \hat{\bar{\beta}}$.

By corollary I.4.1 to the Abstract Context lemma and the definitions of $\lvert \cdot \rvert$ and $\sqsubseteq$, there exist $\bar{E}$ and $\bar{v}_1, \ldots, \bar{v}_m$ such that such that

- $\hat{e} = \bar{E}((\text{goto} q \bar{v}_1 \ldots \bar{v}_m))$,
- $\lvert \bar{E} \rvert \sqsubseteq_{A,M} \bar{E}$, and
- $\lvert \bar{v}_j \rvert \sqsubseteq_{A,M} \bar{v}_j$ for all $j \in 1 \ldots m$.

Because $\lvert \bar{Q}_k \rvert \sqsubseteq_{A,M} \hat{\bar{Q}}_k$, there also exist $\hat{e}'$ and $\hat{\bar{c}}$ such that

- $\hat{Q}_k = \langle \text{define-state} \ (q \ [x_1 \tau_1] \ldots \ [x_m \tau_m]) \ x' \ e' \ \bar{c} \rangle$,
- $\lvert \hat{e}' \rvert \sqsubseteq_{A,M} \hat{e}'$, and
- $\lvert \hat{\bar{c}} \rvert \sqsubseteq_{A,M} \hat{\bar{c}}$.

Then by the rule A-GOTO, there exist $\hat{e}''$, $\hat{l}$ and $\hat{K}'$ such that

- $\hat{K} \xrightarrow{\hat{l}} \hat{K}'$
- $\hat{e}'' = \langle \text{receive} \ x' \ e' \ \bar{c} \rangle [x_1 \leftarrow \bar{v}_1] \ldots [x_m \leftarrow \bar{v}_m]$,
- $\hat{l} = \hat{\alpha} : \text{goto}$,
- $\hat{K}' = \langle \hat{\beta} [\hat{\alpha} \rightarrow \hat{\bar{B}} \cup \{\langle \bar{Q}_1 \ldots \bar{Q}_n, \hat{e}'' \rangle \}] \mid \hat{\mu} \mid H_A \rangle^\beta$.
Approximating Configuration

Most of the components of the configurations did not change, so it remains to show only that \( \hat{\beta}[a \to \hat{B} \cup \{(Q_1 \ldots Q_n, \hat{E}[\hat{e}'])\}] \) approximates \( |\hat{\beta}[a \to (Q_1 \ldots Q_n, \hat{e})]| \).

By repeated uses of the Abstract Substitution lemma (one for each of the \( m \) arguments to the state), we have \( |\hat{e}'| \preceq A, M \hat{\beta} \). That gives us that \( |(Q_1 \ldots Q_n, \hat{e})| \preceq A, M \hat{\beta} \). We also know that \( a \in \text{dom}(\hat{\beta}) \), and because \( \hat{\beta} \) is a single-handler configuration, there is no actor at some \( a' \neq a \) approximated by \( \hat{a} \) that is handling an event. Therefore by corollary I.11.1 to the Replaced Behavior lemma, we have \( |\hat{\beta}[a \to (Q_1 \ldots Q_n, \hat{e})]| \preceq A, M \hat{\beta}[a \to \hat{B} \cup \{(Q_1 \ldots Q_n, \hat{e}''\}]\).

Let \( A' = \emptyset \) and \( M' = \emptyset \). The transition \( \hat{K} \xrightarrow{i} \hat{K}' \) adds no new actors or markers, so \( \text{dom}(A') \) contains the addresses of all new actors, and \( \text{dom}(M') \) contains all new markers. Then by the above arguments, \( |\hat{l}| \preceq A \cup A', M \cup M' \hat{l} \) and \( |\hat{K}'| \preceq A \cup A', M \cup M' \hat{K}' \).

Restricted Transition

We must also show that \( \hat{K} \xrightarrow{\hat{l}}_{RA} \hat{K}' \). We argue for each of the conditions of \( \xrightarrow{\hat{l}}_{RA} \) below.

1. We have already shown that \( \hat{K} \xrightarrow{\hat{l}} \hat{K}' \).
2. Identical to the argument in the M-SENDEXTERNAL case above.
3. \( \hat{l} \) is not a handler-start label.
4. \( \hat{l} \) is not a spawn label.
5. \( \hat{l} \) is not a rcv-ext label.
6. \( \hat{l} \) is not a send-ext label.

Case: M-FUNC

By the definition of this rule, there exist \( \hat{\beta}, \hat{\mu}, H_C, \hat{\rho}, a, Q_1, \ldots, Q_n, \hat{E}, \hat{e}, \) and \( \hat{e}' \) such that

- \( \hat{K} = \langle \hat{\beta} \mid \hat{\mu} \mid H_C \rangle^\hat{\rho} \),
- \( \hat{\beta}(a) = \langle Q_1 \ldots Q_n, \hat{E}[\hat{e}] \rangle \),
- \( \hat{e} \to \hat{e}' \),
- \( \hat{I} = a : \text{func} \),
- \( \hat{K}' = \langle \hat{\beta}[a \to (Q_1 \ldots Q_n, \hat{E}[\hat{e}'])] \mid \hat{\mu} \mid H_C \rangle^\hat{\rho} \).
APPENDIX I. PROOFS FOR ABSTRACT CONFORMANCE

A-FUNC Transition

First, we show that a similar transition from \( \hat{K} \) with A-FUNC is possible.

By the definitions of \(|·|\) and \(\sqsubseteq\), there exist \(\hat{\beta}, \hat{\mu}, H_A, \hat{\rho}, \hat{\alpha}, B, \hat{Q}_1, \ldots, \hat{Q}_n, C, \) and \(\hat{e} \) such that

- \( \hat{K} = \left< \left< \hat{\beta} \mid \hat{\rho} \mid H_A \right> \right>^\hat{\rho} \),
- \( \hat{\beta}(\hat{\alpha}) = \hat{B} \cup \{ \langle \hat{Q}_1, \ldots, \hat{Q}_n, C[\hat{e}] \rangle \} \), and
- \( |\hat{\beta}| \sqsubseteq_{A,M} \hat{\beta}, |a| \sqsubseteq_{A} \hat{\alpha}, |\hat{Q}_i| \sqsubseteq_{A,M} \hat{Q}_i \) for all \( i \in 1 \ldots n \), \( |\hat{E}[\hat{e}]| \sqsubseteq_{A,M} \hat{e}, |\hat{\mu}| \sqsubseteq_{A,M} \hat{\mu}, H_C \sqsubseteq_{M} H_A \), and \( |\hat{\rho}| \sqsubseteq_{A,M} \hat{\rho} \).

By corollary I.4.1 to the Abstract Context lemma, there exist \(\hat{E}\) and \(\hat{e}'\) such that

- \( |\hat{e}| = \hat{E}[\hat{e}'] \),
- \( |\hat{E}| \sqsubseteq_{A,M} \hat{E}, \) and
- \( |\hat{e}| \sqsubseteq_{A,M} \hat{e}' \).

By the Functional-Step Soundness lemma, there exists \(\hat{e}''\) such that \(\hat{e}' \rightarrow \hat{e}''\) and \( |\hat{e}'| \sqsubseteq_{A,M} \hat{e}'' \). Therefore, by the rule A-FUNC, there exist \(\hat{I}\) and \(\hat{K}'\) such that

- \( \hat{K} \xrightarrow{\hat{I}} \hat{K}' \),
- \( \hat{I} = \hat{\alpha}: \text{func} \),
- \( \hat{K}' = \left< \left< \hat{\beta}[\hat{\alpha} \rightarrow B \cup \{ \langle \hat{Q}_1, \ldots, \hat{Q}_n, C[\hat{E}[\hat{e}'']] \rangle \} \right| \hat{\rho} \mid H_A \right> \right>^\hat{\rho} \).

Approximating Configuration

As in the previous case, most of the components did not change, so it remains only to show an approximation relationship between the two actors. The result from the Functional-Step Soundness lemma gives us that \( |\hat{e}| \sqsubseteq_{A,M} \hat{e}' \), and corollary I.3.1 to the Abstract Substitution lemma then gives us that \( |\hat{E}[\hat{e}']| \sqsubseteq_{A,M} \hat{E}[\hat{e}''] \). The rest of the proof is identical to the one for the M-SENDEXTERNAL case, using corollary I.11.1 to the Replaced Behavior lemma to show the relationship.

Restricted Transition

We must also show that \( \hat{K} \xrightarrow{\hat{I}}_{RA} \hat{K}' \). We argue for each of the conditions of \( \rightarrow_{RA} \) below.

1. We have already shown that \( \hat{K} \xrightarrow{\hat{I}} \hat{K}' \).
2. Identical to the argument in the M-SENDEXTERNAL case above.
I.26. SOUNDNESS OF EVENT STEPS LEMMA

3. \( \hat{l} \) is not a handler-start label.
4. \( \hat{l} \) is not a spawn label.
5. \( \hat{l} \) is not a rcv-ext label.
6. \( \hat{l} \) is not a send-ext label.

\[\]

I.26 Soundness of Event Steps Lemma

Lemma (Soundness of Event Steps). For all \( \hat{\mathcal{K}} = \langle \beta \mid H \rangle^\rho, \hat{l}, \hat{\mathcal{K}}' = \langle \beta' \mid H' \rangle^\rho' \), \( \mathcal{K}, \mathcal{A}, \) and \( \mathcal{M} \), if

- \( \hat{\mathcal{K}} \) is a well-formed, externals-only, single-handler configuration,
- \( |\hat{\mathcal{K}}| \subseteq \mathcal{A}, \mathcal{M} \hat{\mathcal{K}} \)
- \( \hat{\mathcal{K}} \xrightarrow{\hat{l}_1, \ldots, \hat{l}_n} \hat{\mathcal{K}}' \)
- \( \mathcal{A}(\hat{\mathcal{K}}) = \mathcal{A}(\hat{\mathcal{K}}') \)
- \( \mathcal{M}(\hat{\mathcal{K}}') \subseteq \mathcal{H} \), and
- for all \( \hat{a} \in \text{ExtAddr} \), \( \mathcal{A}(\hat{a}) = \hat{a} \),
then there exist \( \hat{l}_1, \ldots, \hat{l}_n, \hat{\mathcal{K}}', \mathcal{A}', \) and \( \mathcal{M}' \) such that

- \( \hat{\mathcal{K}} \xrightarrow{\hat{l}_1, \ldots, \hat{l}_n} \hat{\mathcal{K}}' \)
- \( |\hat{l}_i| \subseteq \mathcal{A}, \mathcal{M} \hat{\mathcal{K}}' \) for all \( i \in 1 \ldots n \)
- \( |\hat{\mathcal{K}}'| \subseteq \mathcal{A}, \mathcal{M} \hat{\mathcal{K}}' \)
- \( \mathcal{A}'(\hat{\mathcal{K}}') = \mathcal{A}(\hat{\mathcal{K}}') - \mathcal{A}(\hat{\beta}), \) and
- \( \mathcal{M}'(\hat{\mathcal{K}}') \subseteq \mathcal{H}' - \mathcal{H} \).

Proof. By the definition of the concrete event-step relation, there exist \( \hat{\mathcal{K}}_1, \ldots, \hat{\mathcal{K}}_{n+1} \) such that

- \( \hat{\mathcal{K}}_1 = \hat{\mathcal{K}} \)
- \( \hat{\mathcal{K}}_{n+1} = \hat{\mathcal{K}}' \)
- \( \hat{\mathcal{K}}_1 \xrightarrow{l_1} \cdots \xrightarrow{l_n} \hat{\mathcal{K}}_{n+1} \)
- \( n > 0 \),
• some actor in $\hat{K}_i$ is handling an event for all $i \in 2 \ldots n$, and
• no actor in $\hat{K}_n + 1$ is handling an event.

The proof is by induction on $n$. In the base case, $n = 1$. Then by the Soundness of Abstract CSA lemma, we have that there exist $\hat{l}_1$, $\hat{K}'$, $A'$, and $M'$ such that

- $\hat{K} \xrightarrow{\hat{l}_1} RA \hat{K}'$,
- $|\hat{l}_1| \sqsubseteq_{A \cup A', M \cup M'} \hat{l}_1$,
- $|\hat{K}'| \sqsubseteq_{A \cup A', M \cup M'} \hat{K}'$,
- $dom(A') = dom(\hat{\beta}') - dom(\hat{\beta})$, and
- $dom(M') \subseteq H' - H$.

Furthermore, because $|\hat{K}'| \sqsubseteq_{A \cup A', M \cup M'} \hat{K}'$ and no actor in $\hat{K}'$ is handling an event, we know that no actor in $\hat{K}'$ is handling an event. Therefore, we have $\hat{K} \xrightarrow{\hat{l}_1} \hat{K}'$.

In the inductive case, $n > 0$. Let $\hat{K}_2 = \langle \langle \hat{\beta}'', \{H''\} \rangle \rangle \hat{K}''$. Again by the Soundness of Abstract CSA lemma, there exist $\hat{l}_2, \ldots, \hat{l}_n$, $\hat{K}'', A''$, and $M''$ such that

- $\hat{K} \xrightarrow{\hat{l}_1} RA \hat{K}_2$,
- $|\hat{l}_1| \sqsubseteq_{A \cup A'', M \cup M''} \hat{l}_1$,
- $|\hat{K}_2| \sqsubseteq_{A \cup A'', M \cup M''} \hat{K}_2$,
- $dom(A') = dom(\hat{\beta}'') - dom(\hat{\beta})$, and
- $dom(M') \subseteq H'' - H$.

By the definition of the $\xrightarrow{R}$ relation, $dom(\hat{\beta}') \subseteq dom(\hat{\beta}'')$ and $H' \subseteq H''$. As a result, $dom(A' \cup A'') = dom(\hat{\beta}'') \cup ExtAddr$ and $dom(M \cup M'') \subseteq H''$.

By the Well-Formed Preservation, Externals-Only Preservation, and Single-Handler Preservation lemmas, $\hat{K}_2$ is a well-formed, externals-only, single-handler configuration. Then by the induction hypothesis, there exist $\hat{l}_2, \ldots, \hat{l}_n$, $\hat{K}'$, $A''$, and $M''$ such that

- $\hat{K}_2 \xrightarrow{\hat{l}_2, \ldots, \hat{l}_n} \hat{K}'$,
- $|\hat{l}_i| \sqsubseteq_{A \cup A'' \cup A'' \cup M \cup M''} \hat{l}_i \text{ for all } i \in 2 \ldots n$,
- $|\hat{K}'| \sqsubseteq_{A \cup A'' \cup A'' \cup M \cup M''} \hat{K}'$,
- $dom(A'') = dom(\hat{\beta}'') - dom(\hat{\beta}'')$, and
- $dom(M'') \subseteq H' - H''$. 


I.27. Event-Step Execution Soundness Lemma

By the definition of $\rightarrow_R$ (and specifically the rules defining its intermediate $\rightarrow_R$ transitions), $\text{dom}(\beta_2) \subseteq \text{dom}(\beta')$ and $H_2 \subseteq H'$. As a result, we also have $\text{dom}(\hat{A}'' \cup \hat{A}''' \cup \hat{A}''') = \text{dom}(\beta') \cup \text{ExtAddr}$ and $\text{dom}(M' \cup M'' \cup M''') \subseteq H'$.

By the definition of the abstract event-step relation, there exist $\hat{K}_{n+1} = \hat{K}'$ and $\hat{K}_1 \rightarrow_{RA} \hat{K}_{n+1} \rightarrow_{RA} \hat{K}_{n+1}$. We know that there is an actor handling an event in $\hat{K}_2$, so because $|\hat{K}_2| \subseteq M' \hat{K}_2$, there is also an actor handling an event in $\hat{K}_2$. Therefore, we have $\hat{K} \xrightarrow{I_1, \ldots, I_n} \hat{K}'$.

By the definition of the event-step relation (specifically, the rules for its intermediate $\rightarrow_R$ transitions), every internal address appearing in some $I_1$ in the execution must also be in $\text{dom}(\beta_2)$, and every marker appearing $I_1$ must also be in $H_2$. We already have that every abstract external address $\hat{a}$ is in $\text{dom}(A)$. As a result, $\hat{A}' = \hat{A}'' \cup \hat{A}'''$ and $M' \cup M'' \cup M'''$ give the exact same results as $A \cup A''$ and $M \cup M'' \cup M'''$, respectively, for all addresses and markers appearing in $|I_1|$. Therefore, $|\hat{I}_1| \subseteq A \cup A'' \cup A''' \cup M \cup M'' \cup M''' \hat{I}_1$.

Let $A' = A'' \cup A'''$ and $M' = M'' \cup M'''$. Then we have

- $|\hat{I}_1| \subseteq A \cup A' \cup M' \hat{I}_1$ for all $i \in 1 \ldots n$,
- $|\hat{K}'| \subseteq A \cup A' \cup M' \hat{K}'$,
- $\text{dom}(A') = \text{dom}(\beta') - \text{dom}(\beta)$, and
- $\text{dom}(M') \subseteq H' - H$,

which completes the proof. $\square$

I.27 Event-Step Execution Soundness Lemma

**Lemma** (Event-Step Execution Soundness). For all concrete event-step executions $\hat{K}_1 \xrightarrow{I_{1,1}, \ldots, I_{1,n}} \hat{K}_2 \xrightarrow{I_{2,1}, \ldots, I_{2,n}} \ldots$, where $\hat{K}_1 = \langle \hat{A}, \hat{M} \rangle$, and all $\hat{K}_1$, $A$, and $M$ such that

- all of the $\hat{K}_i$ are well-formed, externals-only, single-handler configurations,
- $|\hat{K}_1| \subseteq A, M \hat{K}_1$,
- $\text{dom}(A) = \text{dom}(\beta) \cup \text{ExtAddr}$,
- $\text{dom}(M) \subseteq H'$, and
- for all $\hat{a} \in \text{ExtAddr}$, $A(\hat{a}) = \hat{a}$,

there exist $A'$, $M'$, and an abstract event-step execution $\hat{K}_1 \xrightarrow{I_{1,1}, \ldots, I_{1,n}} \hat{K}_2 \xrightarrow{I_{2,1}, \ldots, I_{2,n}} \ldots$ such that

- $|\hat{K}_i| \subseteq A \cup A', M \cup M' \hat{K}_i$ for all corresponding configurations in the two executions and
APPENDIX I. PROOFS FOR ABSTRACT CONFORMANCE

350

• \( |\tilde{I}_{i,j}| \subseteq A \cup A', M \cup M' \tilde{I}_{i,j} \) for all corresponding labels in the two executions.

Proof. Let \( L = n + 1 \) if \( \text{len} (\tilde{K}_1 \xrightarrow{I_{1,1}, \ldots, I_{1,n}} \cdots) = n \), or \( \omega \) if \( \text{len} (\tilde{K}_1 \xrightarrow{I_{1,1}, \ldots, I_{1,n}} \cdots) = \omega \). Let \( \tilde{K}_i = \left( \left\langle \tilde{\mu}_i \mid \nu_i \right\rangle H_i \right) \) for all \( i < L \). We first prove there exist an abstract event-step execution \( \tilde{K}_1 \xrightarrow{I_{1,1}, \ldots, I_{1,n}} \tilde{K}_2 \xrightarrow{I_{2,1}, \ldots, I_{2,n}} \cdots \) and correspondence functions \( A'_1, \ldots, \) and \( M'_1, \ldots \) such that for all \( i < L \), there exist \( A'' = \cup_{j<i} A'_j \) and \( M'' = \cup_{j<i} M'_j \) such that

• \( |\tilde{K}_i| \subseteq A \cup A', M \cup M' \tilde{K}_i \),

• if \( i > 1 \), then \( |\tilde{I}_{i-1,j}| \subseteq A \cup A', M \cup M' \tilde{I}_{i-1,j} \) for all \( j \in 1 \ldots n \),

• \( \text{dom}(A \cup A'') = \text{dom}(\tilde{\mu}_i) \cup \text{ExtAddr} \), and

• \( \text{dom}(M \cup M'') \subseteq H_i \).

The execution and correspondence functions are defined inductively. In the base case, when \( i = 1 \), let \( A_1 = \emptyset \) and \( M_1 = \emptyset \). We already have a definition for \( \tilde{K}_1 \), and we have that \( |\tilde{K}_1| \subseteq A_1 \cup A', M_1 \cup M' \tilde{K}_1 \). We also know that \( \text{dom}(A \cup A'_1) = \text{dom}(\tilde{\mu}_1) \cup \text{ExtAddr} \) and \( \text{dom}(M \cup M'_1) \subseteq H_1 \).

In the inductive case, we have some step \( \tilde{K}_{i-1} \xrightarrow{I_{i-1,1}, \ldots, I_{i-1,n}} \tilde{K}_i \) in the execution. By induction, there exist some \( \tilde{K}_{i-1}, A'' = \cup_{j<i} A'_j \), and \( M'' = \cup_{j<i} M'_j \) such that

• \( |\tilde{K}_{i-1}| \subseteq A \cup A'', M \cup M' \tilde{K}_{i-1} \),

• \( \text{dom}(A \cup A'') = \text{dom}(\tilde{\mu}_{i-1}) \cup \text{ExtAddr} \), and

• \( \text{dom}(M \cup M'') \subseteq H_{i-1} \).

Then by the Soundness of Abstract Event Steps lemma, there exist \( \tilde{I}_{i-1,1}, \ldots, \tilde{I}_{i-1,n}, \tilde{K}_i, A'_i, \) and \( M'_i \) such that

• \( \tilde{K}_{i-1} \xrightarrow{I_{i-1,1}, \ldots, I_{i-1,n}} \tilde{K}_i \),

• \( |\tilde{I}_{i-1,j}| \subseteq A \cup A'' \cup A'_i \cup M'' \cup M'_i \tilde{I}_{i-1,j} \) for all \( j \in 1 \ldots n \),

• \( |\tilde{K}_i| \subseteq A \cup A'' \cup A'_i \cup M'' \cup M'_i \tilde{K}_i \),

• \( \text{dom}(A'_i) = \text{dom}(\tilde{\mu}_i) - \text{dom}(\tilde{\mu}_{i-1}) \), and

• \( \text{dom}(M'_i) \subseteq H_i - H_{i-1} \).

By the definition of \( \xrightarrow{R} \) (and specifically the rules defining its intermediate \( \xrightarrow{R} \) transitions), \( \text{dom}(\tilde{\mu}_{i-1}) \subseteq \text{dom}(\tilde{\mu}_i) \) and \( H_{i-1} \subseteq H_i \). As a result, we also have \( \text{dom}(A \cup A'' \cup A'_i) = \text{dom}(\tilde{\mu}_i) \cup \text{ExtAddr} \) and \( \text{dom}(M \cup M'' \cup M'_i) \subseteq H_i \), which satisfies the necessary conditions.
I.28. SOUNDNESS OF FAIR EXECUTIONS LEMMA

Having defined the execution and the series of correspondence functions, let $A' = \bigcup_{i < L} A_i'$ and let $M' = \bigcup_{i < L} M_i'$. We will show that $A \uplus A'$ and $M \uplus M'$ relate all of the configurations and labels in the two executions.

Let there be $i < L$, and let $A'' = \bigcup_{j \not= i} A_j'$, and $M'' = \bigcup_{j \not= i} M_j'$. Because $\tilde{K}_i$ is well-formed, the only addresses appearing in $[\tilde{K}_i]$ are in $\text{dom}(\tilde{\beta}_i) \cup \text{ExtAddr}$. Similarly, the only markers appearing in $[\tilde{K}_i]$ must be in $H_i$. Therefore, $A \uplus A'$ and $M \uplus M'$ give the exact same results as $A \uplus A''$ and $M \uplus M''$, respectively, for all addresses and markers appearing in $[\tilde{K}_i]$. Therefore, $[\tilde{K}_i] \sqsubseteq_{A \uplus A', M \uplus M'} \tilde{K}_i$.

Finally, let there be $j$ such that $l_{i-1,j}$ is a label in the concrete execution. By the definition of the event-step relation $\longrightarrow$ (specifically, the rules for its intermediate $\rightarrow$ transitions), every internal address appearing in some $\tilde{l}_{i-1,j}$ in the execution must also be in $\text{dom}(\tilde{\beta}_i)$, and every marker appearing $\tilde{l}_{i-1,j}$ must also be in $H_i$. We already have that every abstract external address $\tilde{a}$ is in $\text{dom}(A)$. As a result, $A \uplus A'$ and $M \uplus M'$ give the exact same results as $A \uplus A''$ and $M \uplus M''$, respectively, for all addresses and markers appearing in $[\tilde{l}_{i-1,j}]$. Therefore, $[\tilde{l}_{i-1,j}] \sqsubseteq_{A \uplus A', M \uplus M'} \tilde{l}_{i-1,j}$, which completes the proof.

I.28 Soundness of Fair Executions Lemma

**Lemma** (Soundness of Fair Executions). For all concrete event-step executions $\tilde{K}_1 l_{1,1} \ldots l_{1,n_1} \tilde{K}_2 l_{2,1} \ldots l_{2,n_2} \ldots$ all abstract event-step executions $K_1 l_{1,1} \ldots l_{1,n_1} K_2 l_{2,1} \ldots l_{2,n_2} \ldots$, of the same length, and all $A$ and $M$, if

- the concrete execution is fair,
- all of the $K_i$ in the concrete execution are externals-only configurations,
- $[\tilde{K}_i] \sqsubseteq_{A,M} \tilde{K}_i$ for all corresponding configurations in the two executions, and
- $[l_{i,j}] \sqsubseteq_{A,M} \tilde{l}_{i,j}$ for all corresponding labels in the two executions,

then the abstract execution is also fair.

**Proof.** Let $\tilde{K}_1 l_{1,1} \ldots l_{1,n} \tilde{K}_2 l_{2,1} \ldots l_{2,n} \ldots$ be a concrete, fair event-step execution such that all of the $K_i$ are externals-only configurations, and let $\tilde{K}_1 l_{1,1} \ldots l_{1,n} \tilde{K}_2 l_{2,1} \ldots l_{2,n} \ldots$ be an abstract event step execution. Finally, let there be $A$ and $M$ such that $[\tilde{K}_i] \sqsubseteq_{A,M} \tilde{K}_i$ for all corresponding configurations in the two executions and $[l_{i,j}] \sqsubseteq_{A,M} \tilde{l}_{i,j}$ for all corresponding labels in the two executions. We must show that the abstract execution is fair.

In the following, let $\langle \tilde{\beta}_i | \tilde{\mu}_i | H_i \rangle^{\tilde{K}_i} = \tilde{K}_i$ and $\langle \tilde{\beta}_i | \tilde{\mu}_i | H_i' \rangle^{\tilde{K}_i} = \tilde{K}_i$ for all $\tilde{K}_i$ and $\tilde{K}_i$ in the two executions. We must show that for every configuration in the abstract execution, every necessarily-enabled actor eventually runs or is no longer necessarily enabled, and every message with a quantity of single is either eventually received or no longer has a quantity of single.
First, let $\hat{K}_i$ be a configuration in the execution, and let $\hat{a}$ be the address of a necessarily-enabled actor in $\hat{K}_i$. By the definition of necessarily-enabled, there exists some $\hat{l}$ such that

- an abstract step labeled with $\hat{l}$ is enabled in $\hat{K}_i$,
- $\hat{l}$ is not a \texttt{rcv-ext} label,
- if $\hat{a}$ is collective, then $\hat{l}$ is not a \texttt{rcv-int} or \texttt{timeout} label, and
- if $\hat{l} = \hat{a}$: \texttt{rcv-int}(H', $\hat{v}$) for some $\hat{v}$ and $H'$, then $\hat{\mu}_i(\hat{a} @ H', \hat{v}) = \texttt{single}$.

If $\hat{l}$ is not a \texttt{rcv-ext}, \texttt{rcv-int}, or \texttt{timeout} label, then the actor at $\hat{a}$ in $\hat{K}_i$ must be handling an event. Then by the definition of $\mid\cdot\mid$ and $\sqsubseteq$, there exists some $a$ such that $\mid a \mid \sqsubseteq_A \hat{a}$, and the actor at $a$ in $\hat{K}_i$ is handling an event. Otherwise, $\hat{l}$ must be a \texttt{rcv-int} or \texttt{timeout} label, so $\hat{a}$ must be atomic. Then by the definition of $\mid\cdot\mid$ and $\sqsubseteq$ there exists some $a$ such that $\mid a \mid \sqsubseteq_A \hat{a}$. Then by the definition of $\sqsubseteq$, there exists some $l$ such that $a$ identifies the active actor for $\hat{l}$ and $\hat{l}$ is enabled in $\hat{K}_i$. Therefore in either case, because the given concrete execution is fair, there must be some step $\hat{K}_{i+1} \xrightarrow{\hat{l}_{i+1}, \ldots, \hat{l}_{i+j}} \hat{K}_{i+j+1}$ in the execution such that $a$ identifies the active actor for $\hat{l}_{i+1}$. Then because $\mid \hat{l}_{i+1} \mid \sqsubseteq_A \hat{l}_{i+1}, \hat{a}$ must identify the active actor for $\hat{l}_{i+1}$.

Second, let $\hat{K}_i$ be a configuration in the constructed execution, and let there be $\hat{a}$, $H''$, and $\hat{v}$ such that $\hat{\mu}_i(\hat{a} @ H'', \hat{v}) = \texttt{single}$. Then by the definition of $\mid\cdot\mid$ and $\sqsubseteq$, there exists exactly one $a$, $H'''$, and $\hat{v}$ such that $\langle a @ H''', \hat{v} \rangle \in \hat{\mu}_i$, $\mid a @ H''' \mid \sqsubseteq_A \hat{a} @ H''$, and $\mid \hat{v} \mid \sqsubseteq_A \hat{a} @ H''$. Because the given concrete execution is fair, there exists some $j > 0$ such that $\hat{K}_{i+j} \xrightarrow{\hat{l}_{i+j}, \ldots, \hat{l}_{i+j+n}} \hat{K}_{i+j+n}$ and $\hat{l}_{i+j} = a: \texttt{rcv-int}(H''', \hat{v})$. Because $\mid \hat{l}_{i+j} \mid \sqsubseteq_A \hat{l}_{i+j}, \hat{l}_{i+j} = \hat{a}: \texttt{rcv-int}(H''', \hat{v})$ for some $\hat{v}'$ such that $\mid \hat{v} \mid \sqsubseteq_A \hat{a} @ H''$. If $\hat{v} = \hat{v}'$, then we're done. Otherwise, the transition with the label $\hat{l}_{i+j}$ must be a use of the M-RECEIVEINTERNAL rule, and the transition with the label $\hat{l}_{i+j, 1}$ must be a use of the A-RECEIVEINTERNAL rule. Because $\hat{K}_i$ is an externals-only configuration, $H''' = \emptyset$, and then by the definitions of $\mid\cdot\mid$ and $\sqsubseteq$, $H'' = \emptyset$. Then by the definitions of those transition rules, $\langle a @ H''', \hat{v} \rangle \in \hat{\mu}_{i+j}$ and $\langle \hat{a} @ H'', \hat{v}' \rangle \in \text{dom}(\hat{\mu}_{i+j})$. Then because $\mid \hat{K}_{i+j} \mid \sqsubseteq_A \hat{K}_{i+j}, \langle \hat{a} @ H'', \hat{v}' \rangle$ must be the only element of $\text{dom}(\hat{\mu}_{i+j})$ such that $\mid a @ H''' \mid \sqsubseteq_A \hat{a} @ H''$ and $\mid \hat{v} \mid \sqsubseteq_A \hat{a} @ H''$. Therefore, $\langle \hat{a} @ H'', \hat{v} \rangle \notin \text{dom}(\hat{\mu})$, so we're done.

\section*{I.29 Monitored Correspondents Lemma}

This lemma is used in the completeness results for PSM transitions below to show that whenever two PSMs $s$ and $s'$ are related by $M$ (i.e., $M(s) = s'$) and they both transition with labels $\mid \hat{l} \mid$ and $\mid \hat{l} \mid$ related by $\mu$ (i.e., $\mid \hat{l} \mid \sqsubseteq \hat{M} \mid \hat{l} \mid$), then the input-monitored or output-monitored markers $H'_N$ in $\hat{l}$ are exactly the $M$-correspondents of the input-monitored or output-monitored markers $H_N$ in $\hat{l}$, respectively.
Lemma (Monitored Correspondents Lemma). For all $H_L, H'_L, H_N, H'_N,$ and $M,$ if $H_L \sqsubseteq_M H'_L,$ $M(H_N) = H'_N,$ and $M$ is one-to-one, then $M(H_L \cap H_N) = H'_L \cap H'_N.$

By the definition of $\sqsubseteq,$ there exist $H_{L1}$ and $H_{L2}$ such that $H_L = H_{L1} \sqcup H_{L2},$ $M(H_{L1}) = H'_L,$ and $H_{L2} \cap \text{dom}(M) = \emptyset.$ We know that $M(H_N)$ is defined only when $H_N \subseteq \text{dom}(M),$ so $H_{L2} \cap H_N = \emptyset.$ Therefore $H_L \cap H_N = H_{L1} \cap H_N,$ and it remains to show that $M(H_{L1} \cap H_N) = H'_L \cap H'_N.$

To show that equality, we show a subset relationship in both directions. First, to show $M(H_{L1} \cap H_N) \subseteq H'_L \cap H'_N,$ let $\eta$ be an element of $M(H_{L1} \cap H_N).$ Then there exists some $\eta' \in H_{L1} \cap H_N$ such that $M(\eta') = \eta.$ Therefore $\eta \in M(H_{L1})$ and $\eta' \in H'_N.$ By the above equalities, $\eta \in H'_L$ and $\eta' \in H'_N,$ so $\eta \in H'_L \cap H'_N.$

To show that $H'_L \cap H'_N \subseteq M(H_{L1} \cap H_N),$ let $\eta$ be a member of $H'_L \cap H'_N.$ By the above equalities, $\eta \in M(H_{L1})$ and $\eta \in M(H_N).$ Because $M$ is one-to-one, there exists some $\eta'$ such that $M(\eta') = \eta,$ $\eta' \in H_{L1},$ and $\eta' \in H_N.$ Then $\eta' \in H_{L1} \cap H_N,$ so $\eta \in M(H_{L1} \cap H_N).$

I.30 Input Pattern Lemma

Lemma (Input Pattern). For all $\bar{v}, \bar{v}, M, A, pi, x_1, \ldots, x_n,$ and $\eta_1, \ldots, \eta_n,$ if

- $|\bar{v}| \subseteq_{A,M} \bar{v},$
- $\bar{v} \sim pi \triangleright [x_1 \leftarrow \eta_1, \ldots, x_n \leftarrow \eta_n],$ and
- $|H| = 1$ for all $a@H$ in $\bar{v},$

then there exist $\eta'_1, \ldots, \eta'_n$ such that $\bar{v} \sim pi \triangleright [x_1 \leftarrow \eta'_1, \ldots, x_n \leftarrow \eta'_n]$ and $M(\eta'_i) = \eta_i$ for all $i \in 1 \ldots n.$

Proof. The proof is by structural induction on $pi.$ The case for $pi = x$ is shown below; the others are straightforward.

Case: $pi = x$

By the definition of this rule, there exist $\bar{a}, H,$ and $\eta$ such that $\bar{v} = \bar{a}@H$ and $\eta = \text{max}(H).$ Then by the definitions of $|\bar{v}|$ and $\subseteq,$ there exist $a$ and $H'$ such that $\bar{v} = a@H'.$ By the precondition for this lemma, $|H'| = 1.$ Therefore there exists $\eta'$ such that $H' = \{\eta'\}$ and $a@H' \sim x \triangleright [x \leftarrow \eta'].$

It remains to show that $M(\eta') = \eta.$ By the definitions of $|\bar{v}|$ and $\subseteq,$ $|H| \leq |H'|,$ but because $\text{max}(H)$ is defined, $|H| > 0.$ Therefore, $|H| = 1$ (i.e., $H = \{\eta\},$ so $M(H') = H,$ and therefore $M(\eta') = \eta.$

I.31 Output Pattern Lemma

Lemma (Output Pattern). For all $\bar{v}, \bar{v}, M, po, H,$ and $S,$ if

- $|\bar{v}| \subseteq_{A,M} \bar{v},$
• $\hat{v} \sim pi \triangleright H, S$, and
• $|H| = 1$ for all $a@H$ in $\hat{v}$ where $a$ is internal,
there exist $H'$ and $S'$ such that $\hat{v} \sim pi \triangleright H', S', M(H') = H$, and $M(S') = S$.

**Proof.** The proof is by induction on $po$. The cases for **self-addr** and **delayed-fork-addr** patterns are given below. The others are straightforward.

**Case: $pi = \text{self-addr}$**

By the definition of this rule, there exist $\hat{a}, H''$, and $\eta$ such that

- $\hat{v} = \hat{a}@H''$,
- $\hat{a}$ is internal,
- $\max(H'') = \eta$,
- $H = \{\eta\}$, and
- $S = \varnothing$.

Then by the definitions of $|\cdot|$ and $\sqsubseteq$, there exist $a$ and $H'''$ such that $\hat{v} = a@H'''$ and $a$ is internal. By the precondition for this lemma, $|H'''| = 1$. Therefore, there exists $\eta'$ such that $H''' = \{\eta'\}$ and $a@H''' \sim \text{self-addr} \triangleright \{H\}, \varnothing$.

It remains to show that $M(\eta') = \eta$; the proof is similar to the argument in the proof for the Input Pattern lemma. By the definitions of $|\cdot|$ and $\sqsubseteq$, $|H'''| \leq |H''|$, but because $\max(H'')$ is defined, $|H''| > 0$. Therefore, $|H'''| = 1$ (i.e., $H'' = \{\eta\}$), so $M(H'''') = H''$, and therefore $M(\eta') = \eta$.

**Case: $po = (\text{delayed-fork-addr} \ (\text{goto} \ \varphi \ ) \ \overline{\Phi})$**

The proof for this case is similar to the previous one. By the definition of this rule, there exist $\hat{a}, H''$, and $\eta$ such that

- $\hat{v} = \hat{a}@H''$,
- $\hat{a}$ is internal,
- $\max(H'') = \eta$,
- $H = \varnothing$, and
- $S = \{\langle \{\eta\}, \varphi : e, \overline{\Phi}, \varnothing \rangle\}$.

Then by the definitions of $|\cdot|$ and $\sqsubseteq$, there exist $a$ and $H'''$ such that $\hat{v} = a@H'''$ and $a$ is internal. By the precondition for this lemma, $|H'''| = 1$. Therefore, there exists $\eta'$ such that $H''' = \{\eta'\}$ and $a@H''' \sim (\text{delayed-fork-addr} \ (\text{goto} \ \varphi \ ) \ \overline{\Phi}) \triangleright \varnothing, \{\langle \{\eta'\}, \varphi : e, \overline{\Phi}, \varnothing \rangle\}$.

It remains to show that $M(\eta') = \eta$; the proof is identical to the one in the previous case. By the definitions of $|\cdot|$ and $\sqsubseteq$, $|H'''| \leq |H''''|$, but because $\max(H'')$ is defined, $|H''| > 0$. Therefore, $|H'''| = 1$ (i.e., $H'' = \{\eta\}$), so $M(H'''') = H'''$, and therefore $M(\eta') = \eta$. 

\[\square\]
I.32 Label Erasure Lemma

Lemma (Label Erasure). For all \( \tilde{I}, \hat{I}, A, \) and \( M \), if

- \( |\tilde{I}| \subseteq_{A,M} \hat{I} \), and
- there exist \( \bar{K} \) and \( \bar{K}' \) such that \( \bar{K} \) is a well-formed, externals-only configuration and \( \bar{K} \stackrel{\tilde{I}}{\rightarrow}_R \bar{K}' \),

then either \( |\tilde{I}| = |\hat{I}| = \bullet \), or there exist \( a, \bar{a}, H, H', \bar{v}, \) and \( \bar{v} \) such that
  - \( |a@H| \subseteq_{A,M} \bar{a}@H' \),
  - \( |\bar{v}| \subseteq_{A,M} \bar{v} \), and
  - either
    - \( |\tilde{I}| = a@H?\bar{v} \) and \( |\hat{I}| = \bar{a}@H'？\bar{v} \), or
    - \( |\tilde{I}| = a@H!\bar{v} \) and \( |\hat{I}| = \bar{a}@H'！\bar{v} \).

Proof. The proof is by cases on the shape of \( |\tilde{I}| \).

Case: \( |\tilde{I}| = \bullet \)

By the definition of \( |\_\_| \), \( \tilde{I} \) is either a \texttt{rcv-int}, \texttt{send-int}, \texttt{timeout}, \texttt{func}, \texttt{goto}, or \texttt{spawn} label. In the latter four cases, \( \tilde{I} \) is a similar kind of label by the definitions of \( |\_| \) and \( \subseteq \), so by the definition of \( |\_\_| \), \( |\tilde{I}| = \bullet \).

If \( \tilde{I} \) is a \texttt{rcv-int} or \texttt{send-int} label, then by the definition of \( |\_| \) and \( \subseteq \), there exist \( a, a', H, \) and \( \bar{v} \) such that \( \tilde{I} = a: \texttt{rcv-int}(H, \bar{v}) \) or \( \tilde{I} = a': \texttt{send-int}(a@H, \bar{v}) \). In either case, the transition \( \bar{K} \stackrel{\tilde{I}}{\rightarrow}_R \bar{K}' \) requires that \( a \) is an internal address. Then because \( \bar{K} \) is an externals-only configuration, \( H = \emptyset \), and therefore \( |\tilde{I}| = \bullet \).

Case: \( |\tilde{I}| = \bar{a}@H?\bar{v} \)

By the definition of \( |\_\_| \), there exists \( \tau \) such that \( \tilde{I} = \bar{a}: \texttt{rcv-ext}(H, \bar{v}, \tau) \). Then by the definitions of \( |\_| \) and \( \subseteq \), there exist \( a, H', \) and \( \bar{v} \) such that \( \tilde{I} = a: \texttt{rcv-ext}(H', \bar{v}, \tau) \), \( |a@H'| \subseteq_{A,M} \bar{a}@H \), and \( |\bar{v}| \subseteq_{A,M} \bar{v} \). Because \( \bar{K} \) is well-formed and the transition \( \bar{K} \stackrel{\tilde{I}}{\rightarrow}_R \bar{K}' \) requires that there be some receptionist \( \langle a@H', \tau \rangle \) on \( \bar{K} \), \( |H'| \neq \emptyset \). Therefore, \( |\hat{I}| = a@H'?\bar{v} \).

Case: \( |\tilde{I}| = \bar{a}@H!\bar{v} \)

By the definition of \( |\_\_| \), there exists \( \bar{a}' \) such that \( \tilde{I} = \bar{a}': \texttt{send-ext}(\bar{a}@H, \bar{v}) \). Then by the definitions of \( |\_| \) and \( \subseteq \), there exist \( a, a', H', \) and \( \bar{v} \) such that \( \tilde{I} = a': \texttt{send-ext}(a@H', \bar{v}) \), \( |a@H'| \subseteq_{A,M} \bar{a}@H \), and \( |\bar{v}| \subseteq_{A,M} \bar{v} \). Because \( \bar{K} \) is well-formed and the transition \( \bar{K} \stackrel{\tilde{I}}{\rightarrow}_R \bar{K}' \) requires that \( a \) is external, \( H' \neq \emptyset \). Therefore, \( |\hat{I}| = a@H'!\bar{v} \).

\[\Box\]
I.33 PSM Completeness Lemma

Lemma (PSM Completeness). For all \( s, s', s'', M, \lambda, \lambda', S', \) and \( O' \), if

1. \( M \) is one-to-one
2. \( M(s) = s' \),
3. either \( \lambda = \lambda' = \circ \), or there exist \( a, \hat{a}, H, H', \hat{v}, \hat{v}, \) and \( A \) such that
   - \( |a@H| \subseteq_{A,M} \hat{a}@H' \),
   - \( |\hat{v}| \subseteq_{A,M} \hat{v}, \)
   - \(|H| = 1 \) for all \( a@H \) in \( \hat{v} \) where either \( \lambda = a@H?\hat{v} \) or \( a \) is internal, and
   - either \( \lambda = a@H?\hat{v} \) and \( \lambda = \hat{a}@H'?\hat{v} \), or \( \lambda = a@H!\hat{v} \) and \( \lambda = \hat{a}@H'!\hat{v} \), and
4. \( s' \xrightarrow{\lambda, O', S'} s'' \),

then there exists \( s''' \), \( O \), and \( S \) such that such that

1. \( s \xrightarrow{\lambda, O, S} s''' \),
2. \( M(O) = O' \),
3. \( M(S) = S' \), and
4. \( M(s''') = s'' \).

Proof. The proof is by cases on the rule enabling the transition \( s' \xrightarrow{\lambda, O', S'} s'' \).

Case: P-UNMONITOREDRECEIVE

By the definition of this rule, there exist \( \hat{a}, H', \) and \( \hat{v} \) such that \( \hat{\lambda} = \hat{a}@H'?\hat{v} \). Then by the preconditions to this lemma, there exist \( a, H, \) and \( \hat{v} \) such that \( \lambda = a@H?\hat{v} \).

Let \( H'' \) be the set of input-monitored markers in \( s \), and let \( H''' \) be the set of input-monitored markers in \( s' \). By the definition of P-UNMONITOREDRECEIVE, \( H' \cap H''' = \emptyset \). Then by the Monitored Correspondents lemma, \( H \cap H'' = M(H' \cap H''') = \emptyset \), so \( s \xrightarrow{\lambda, \emptyset, \emptyset} s \). Let \( s''' = s, O = \emptyset \), and \( S = \emptyset \) to complete the proof.

Case: P-MONITOREDRECEIVE

By the definition of this rule, there exist \( \hat{a}, H', \) and \( \hat{v} \) such that \( \hat{\lambda} = \hat{a}@H'?\hat{v} \). Then by the preconditions to this lemma, there exist \( a, H, \) and \( \hat{v} \) such that \( \lambda = a@H?\hat{v}, H \subseteq_{M} H, \) and \( |\hat{v}| \subseteq_{A,M} \hat{v}. \)

To show that \( s \) can also take a step with this rule, let \( H'' \) be the set of input-monitored markers in \( s \), and let \( H''' \) be the set of input-monitored markers in \( s' \). By the definition of P-UNMONITOREDRECEIVE, \( H' \cap H''' \neq \emptyset \). Then by the Monitored Correspondents lemma, \( H \cap H'' = M(H' \cap H''') \neq \emptyset \).
By the definition of this rule, there exists some same state definition \((\text{define-state } (\varphi x_1 \ldots x_n) \delta [pi \rightarrow \overrightarrow{f} (\text{goto } \varphi' x'_1 \ldots x'_m)]) \delta')\) in \(s\) such that \(\hat{\nu} \cong pi \triangleright [x_1 \rightarrow \eta_1, \ldots, x_n \rightarrow \eta_n]\). The state definitions \(\varPhi\) are identical in both \(s\) and \(s'\), so \(s'\) has the same state definition.

By another precondition on this lemma, every marked address \(a@H''''\) in \(\hat{\nu}\) is marked with exactly one marker (i.e., \(|H''''| = 1\)). Then by the Input Pattern lemma, there exist \(\eta'_1, \ldots, \eta'_n\) such that \(\hat{\nu} \cong pi \triangleright [x_1 \rightarrow \eta'_1, \ldots, x_n \rightarrow \eta'_n]\) and \(M(\eta'_i) = \eta_i\) for all \(i \in 1 \ldots n\).

At this point, we have that \(M(s) = s'\) and \(M(\eta'_i) = \eta_i\) for all \(i \in 1 \ldots n\). Thus, because \(M\) is one-to-one, \(M\) forms a bijection between all terms used in the rest of the rule. All of the remaining functions in the rule \((\text{PerformAll}, \text{OutMon}, \text{and Dist})\) treat each marker as an opaque distinct token but otherwise ignore its value, so there exist \(s''''\) and \(S\) such that such that

1. \(s \xrightarrow{\lambda, \varphi, S} s''''\),
2. \(M(S) = S'\), and
3. \(M(s''') = s''\).

Let \(O = O' = \emptyset\) to complete the proof.

**Case: P-FREE TRANSITION**

By the definition of this rule, \(\hat{\lambda} = \bullet\), so by the precondition to this lemma, \(\lambda = \bullet\). Similar to the previous rule after the pattern-match, \(M\) forms a bijection between \(s\) and \(s'\). Again, \(\text{PerformAll}\) and \(\text{Dist}\) treat each marker as an opaque distinct token but otherwise ignore its value, so there must exist \(s''''\) and \(S\) such that such that

1. \(s \xrightarrow{\lambda, \varphi, S} s''''\),
2. \(M(S) = S'\), and
3. \(M(s''') = s''\).

Let \(O = O' = \emptyset\) to complete the proof.

**Case: P-SEND**

By the definition of this rule, there exist \(\hat{\alpha}, H_A, \) and \(\hat{\nu}\) such that \(\hat{\lambda} = \hat{\alpha}@H_A\hat{\nu}\). Then by the preconditions on this lemma, there exist \(a, H_C, \hat{\nu}, \) and \(A\) such that \(\lambda = a@H_C\hat{\nu}\) and \(H_C \sqsubseteq_M H_A\) and \(|\hat{\nu}| \sqsubseteq_{A,M} \hat{\nu}\).

Let \(H'_C\) be the set of output-monitored markers in \(s\), and let \(H'_A\) be the set of output-monitored markers in \(s'\). By the definition of P-\text{SEND}, there exist \(\eta'_1, \ldots, \eta'_n\) such that \(H_A \cap H'_A = \{\eta'_1, \ldots, \eta'_n\}\). Then by the Monitored Correspondents lemma, we have that \(M(H_C \cap H'_C) = \{\eta'_1, \ldots, \eta'_n\}\). Because \(M\) is one-to-one,
there also exist $\eta_1, \ldots, \eta_n$ such that $H_C \cap H'_C = \{\eta_1, \ldots, \eta_n\}$ and $M(\eta_i) = \eta'_i$ for all $i \in 1 \ldots n$.

By the definition of this rule, $s'$ has a series of obligations $O' = \{(\eta'_1, p_{01}), \ldots, (\eta'_n, p_{0n})\}$. Then because $M(s) = s'$, $s'$ has a similar series of obligations with the same patterns, $O = \{(\eta_1, p_{01}), \ldots, (\eta_n, p_{0n})\}$, so $M(O) = O'$.

Also by the definition of this rule, there exist $H''_{A,1}, \ldots, H''_{A,n}$ and $S''_1, \ldots, S''_n$ such that $\bar{v} \sim p_{0i} > H''_{A,i}, S''_i$ for all $i \in 1 \ldots n$. A precondition to this lemma enforces that every marked internal address in $\bar{\eta}$ such that $\bar{\eta} \sim \bar{v}$ is marked with exactly one marker, so by the Output Pattern lemma, there exist $H''_{C,1}, \ldots, H''_{C,n}$ and $S''_1, \ldots, S''_n$ such that $\bar{v} \sim \bar{v} > H''_{C,i}, S''_i$, $M(H''_{C,i}) = H''_{A,i}$, and $M(S''_i) = S''_i$ for all $i \in 1 \ldots n$. By the definition of this rule, $S' = \bigcup_{i \in 1 \ldots n} S''_i$. Let $S = \bigcup_{i \in 1 \ldots n} S''_i$ and we have $M(S) = S'$.

Let $H''_C$ be the input-monitored markers in $s$, and let $H''_{A}$ be the input-monitored markers in $s'$; we have that $MH''_C = H''_{A}$. By the definition of this rule, we know that $|H''_C \cup H''_{A,1} \cup \ldots \cup H''_{A,n}| \leq 1$. Then because $M$ is one-to-one and each individual set is related by $M$, we have that $|H''_C \cup H''_{A,1} \cup \ldots \cup H''_{A,n}| \leq 1$.

Finally, by the definition of this rule, no marker from the output-monitored set $H'_C$ appears on an address in $\bar{v}$. By the definitions of $|\cdot|$ and $\subset$, every marker $\eta$ appearing in $\bar{v}$ is either not in $\text{dom}(M)$ (and therefore not in the output-monitored set $H'_C$, because $H'_C \subseteq \text{dom}(M)$), or has a $M$-correspondent in $\bar{v}$. In the latter case, because $M(H'_C) = H'_A$, and because $M$ is one-to-one, $\eta \in H'_C$.

Let $O''$ be the unused obligations from $s$, and let $O''$ be the unused obligations from $s'$. Then because $M(s) = s'$ and $M(O) = O'$, we have $M(O'') = O''$. Thus, there exists $s''$ such that $s \xrightarrow{a \in O_S} s''$, where $s''$ is identical to $s$ except that its input-monitored markers are $\{H''_C \cup H''_{A,1} \cup \ldots \cup H''_{A,n}\}$ and its obligations are $O''$. Then $M(s'') = s''$, which completes the proof.  

I.34 Configuration Completeness Lemma

**Lemma** (Configuration Completeness). For all $S_C, S_A, S'_A, M, \lambda, \hat{\lambda}$ and $O_A$, if

- $M$ is one-to-one
- $M(S_C) = S_A$.
- either $\lambda = \hat{\lambda} = \bullet$, or there exist $a, \hat{a}, H, H', \bar{v}, \bar{\eta},$ and $A$ such that
  - $[a@H] \subseteq_{A,M} \hat{a}@H'$,
  - $[\bar{v}] \subseteq_{A,M} \bar{\eta}$,
  - $[H] = 1$ for all $a@H$ in $\bar{v}$ where either $\lambda = a@H?\bar{v}$ or $a$ is internal, and
  - either $\lambda = a@H?\bar{v}$ and $\hat{\lambda} = \hat{a}@H'?\bar{v}$, or $\lambda = a@H!\bar{v}$ and $\hat{\lambda} = \hat{a}@H!'\bar{v}$, and
- $S_A \xrightarrow{\lambda, O_A} S'_A$.  


then there exists $S'_C$ and $O_C$ such that

- $S_C \xrightarrow{\lambda, O_C} S'_C$,
- $M(O_C) = O_A$, and
- $M(S'_C) = S'_A$.

Proof. The proof is by cases on the rule enabling the transition $S_A \xrightarrow{\lambda} S'_A$.

Case: S-SENDORRECEIVE

Let $S_A = \{s_{A,1}, \ldots, s_{A,n}\}$. By the definition of this rule, there exist $S''_{A,1}, \ldots, S''_{A,n}$, $O'_{A,1}, \ldots, O'_{A,n}$, and $s'_{A,1}, \ldots, s'_{A,n}$ such that

- $S_A \xrightarrow{\lambda} S'_A$,
- $O_A = \biguplus_{i=1}^{n} O'_{A,i}$,
- $S_A' = \{s'_{A,1}, \ldots, s'_{A,n}\} \cup S''_{A,1} \cup \ldots \cup S''_{A,n}$.

Also by the definition of this rule, $\lambda \neq \bullet$. Then by a precondition to this lemma, $\lambda \neq \bullet$.

Because $M$ is one-to-one and $M(S_C) = S_A$, there exist $s_{C,1}, \ldots, s_{C,n}$ such that $S_C = \{s_{C,1}, \ldots, s_{C,n}\}$ and $M(s_{C,i}) = s_{A,i}$ for all $i \in 1 \ldots n$. Then by the PSM Completeness lemma, there exist $S''_{C,1}, \ldots, S''_{C,n}$, $O'_{C,1}, \ldots, O'_{C,n}$, and $s'_{C,1}, \ldots, s'_{C,n}$ such that for all $i \in 1 \ldots n$,

- $S_C \xrightarrow{\lambda, O_C} S'_C$,
- $M(O'_{C,i}) = O'_{A,i}$,
- $M(S''_{C,i}) = S''_{A,i}$; and
- $M(s'_{C,i}) = s'_{A,i}$.

Let $O_C = \biguplus_{i=1}^{n} O'_{C,i}$ and $S'_C = \{s'_{C,1}, \ldots, s'_{C,n}\} \cup S''_{C,1} \cup \ldots \cup S''_{C,n}$. Then we have that $S_C \xrightarrow{\lambda, O_C} S'_C$. Also, as a result of the above arguments, we have that $M(O_C) = O_A$ and $M(S'_C) = S'_A$, which completes the proof.

Case: S-FREETRANSITION

By the definition of this rule, there exist some $s_A, s'_A, S''_A$, and $S''_A$ such that

- $\lambda = \bullet$. 

APPENDIX I. PROOFS FOR ABSTRACT CONFORMANCE

- \( S_A = \{ s_A \} \cup S''_A \),
- \( s_A \xrightarrow{\varphi, S''_A} s'_A \),
- \( O_A = \emptyset \),
- \( S'_A = \{ s'_A \} \cup S''_A \cup S'''_A \).

By a precondition to this lemma, \( \lambda = \ast \). For the other premise for this rule, \( MS = S_A \) and \( M \) is one-to-one, so there exist \( s_C \) and \( S''_C \) such that

- \( S_C = \{ s_C \} \cup S''_C \),
- \( M(s_C) = s_A \),
- \( M(S''_C) = S''_A \).

Then by the PSM Completeness lemma, there exist \( S'''_C \) and \( s'_C \) such that

- \( s_C \xrightarrow{\varphi, S'''_C} s'_C \),
- \( M(S'''_C) = S'''_A \),
- \( M(s'_C) = s'_A \).

Let \( S'_C = \{ s'_C \} \cup S''_C \cup S'''_C \) and let \( O_C = \emptyset \). Then we have \( M(O_C) = O_A \) and \( M(S'_C) = S'_A \), which completes the proof.

**Corollary I.34.1.** For all \( S_C, S_A, S'_A \), and \( M \), if

- \( M \) is one-to-one
- \( M(S_C) = S_A \), and
- \( S_A \xrightarrow{\varphi} S'_A \),

then there exists \( S'_C \), such that

- \( S_C \xrightarrow{\varphi} S'_C \) and
- \( M(S'_C) = S'_A \).

**Proof.** By the definition of \( \xrightarrow{\varphi} \), there exist \( S''_{A,1}, \ldots, S''_{A,n} \) such that

- \( S''_{A,1} = S_A \),
- \( S''_{A,n} = S'_A \), and
- \( S''_{A,1} \xrightarrow{\varphi} \ldots \xrightarrow{\varphi} S''_{A,n} \).

By a straightforward induction on \( n \), it is easy to see that there exist \( S''_{C,1}, \ldots, S''_{C,n} \) such that
I.34. CONFIGURATION COMPLETENESS LEMMA

• \( S_{C,1}'' = S_C \),
• \( M(S''_{C,i}) = S''_{A,i} \) for all \( i \in 1 \ldots n \), and
• \( S''_{C,1} \overset{\varphi}{\ldots} \overset{\varphi}{\ldots} \overset{\varphi}{\ldots} S''_{C,n} \).

Let \( S'_C = S''_{C,n} \). Then we have \( S_C \overset{\varphi}{\ldots} \overset{\varphi}{\ldots} \overset{\varphi}{\ldots} S'_C \) and \( M(S'_C) = S'_A \).

\[ \square \]

Corollary I.34.2. For all \( S_C, S_A, S'_A, M, \hat{l}, \hat{\hat{l}} \) and \( O_A \), if

• \( M \) is one-to-one
• \( M(S_C) = S_A \),
• \( [\hat{l}] \sqsubseteq_M \hat{\hat{l}} \),
• there exist \( \hat{K} \) and \( \hat{K}' \) such that \( \hat{K} \) is a well-formed, externals-only configuration and \( \hat{K} \overset{l}{\longrightarrow}_R \hat{K}' \), and
• \( S_A \overset{[\hat{l}], O_A}{\longrightarrow} S'_A \).

then there exists \( S'_C \) and \( O_C \) such that

• \( S_C \overset{[\hat{l}], O_C}{\longrightarrow} S'_C \),
• \( M(O_C) = O_A \), and
• \( M(S'_C) = S'_A \).

Proof. If \( [\hat{l}] = \bullet \), then by the definition of \( [\cdot] \) and \( \sqsubseteq \), \( [\hat{l}] = \bullet \), and by the definition of the specification transition rules, \( O_A = \emptyset \). Let \( O_C = \emptyset \); then by corollary I.34.1, then there exists \( S'_C \) such that \( S_C \overset{[\hat{l}], O_C}{\longrightarrow} S'_C \), \( M(O_C) = O_A \), and \( M(S'_C) = S'_A \).

Otherwise, by the definition of \( \overset{\rightarrow}{\rightarrow} \), there exist \( S''_A \) and \( S''_A \) such that \( S_A \overset{\varphi}{\rightarrow} S''_A \), \( S''_A \overset{\varphi}{\rightarrow} S_A \). By corollary I.34.1, there exists \( S''_C \) such that \( S_C \overset{[\hat{l}], O_C}{\rightarrow} S''_C \) and \( M(S''_C) = S''_A \).

Because \( [\hat{l}] \sqsubseteq_M \hat{\hat{l}} \), there exists some \( A \) such that \( [\hat{l}] \sqsubseteq_{A, M} \hat{\hat{l}} \). Then by the Label Erasure lemma, there exist \( a, \hat{a}, H, H', \hat{\hat{a}}, \) and \( \hat{\hat{a}} \) such that

• \( [a@H] \sqsubseteq_{A, M} \hat{a}@H' \),
• \( [\hat{\hat{a}}] \sqsubseteq_{A, M} \hat{\hat{a}} \), and
• either
  - \( [\hat{l}] = a@H?\hat{a} \) and \( [\hat{\hat{l}}] = \hat{a}@H'?\hat{\hat{a}} \), or
  - \( [\hat{l}] = a@H!\hat{\hat{a}} \) and \( [\hat{\hat{l}}] = \hat{a}@H!\hat{\hat{a}} \).
The relation $\rightarrow_R$ enforces that for all $a@H$ in $\bar{v}$ where either $\lambda = a@H?\bar{v}$ or $a$ is internal, $|H| = 1$. Then by the Configuration Completeness lemma, there exist $S''_C$ and $O_C$ such that $S_C \xrightarrow{[\bar{i}],O_C} S''_C$, $M(O_C) = O_A$, and $M(S''_C) = S''_A$.

Finally, by corollary I.34.1 again, there exists $S'_C$ such that $S''_C \xrightarrow{\bar{i}} S'_C$ and $M(S'_C) = S'_A$. Therefore, we have $S_C \xrightarrow{[\bar{i}],O_C} S'_C$, which completes the proof.

\[\square\]

Corollary I.34.3. For all $S_C, S_A, S'_A, M, \bar{l}_1, \ldots, \bar{l}_n, \bar{\hat{l}}_1, \ldots, \bar{\hat{l}}_n$, and $O_{A,1}, \ldots, O_{A,n}$, if
- $M$ is one-to-one
- $M(S_C) = S_A$, $\bar{l}_i \subseteq M \bar{l}_i$ for all $i \in 1 \ldots n$,
- there exist $\bar{K}$ and $\bar{K}'$ such that $\bar{K}$ is a well-formed, externals-only configuration and $\bar{K} \xrightarrow{\bar{l}_1, \ldots, \bar{l}_n} \bar{K}'$, and
- $S_A \xrightarrow{\langle \bar{l}_1, O_{A,1} \rangle, \ldots, \langle \bar{l}_n, O_{A,n} \rangle} S'_A$.

then there exists $S'_C$ and $O_{C,1}, \ldots, O_{C,n}$ such that
- $S_C \xrightarrow{\langle \bar{l}_1, O_{C,1} \rangle, \ldots, \langle \bar{l}_n, O_{C,n} \rangle} S'_C$,
- $M(O_{C,i}) = O_{A,i}$ for all $i \in 1 \ldots n$, and
- $M(S'_C) = S'_A$.

Proof. By a simple induction on $n$ and corollary I.34.2 to the Configuration Completeness lemma.

I.35 Expression Reflexivity Lemma

**Lemma** (Expression Reflexivity). For all $\hat{\bar{e}}, A, M$, if
- $A = \text{id}|_D$ for some $D$ that contains every address appearing in $\hat{\bar{e}}$, and
- $M = \text{id}|_H$ for some $H$ that contains every marker appearing in $\hat{\bar{e}}$,
then $\bar{\hat{\mu}} \subseteq_{A,M} \bar{\mu}$.

Proof. Straightforward structural induction on $\hat{\bar{e}}$. In the case where $\hat{\bar{e}}$ is a marked address $\hat{\bar{a}}@H'$, the preconditions to the lemma give that $\hat{\bar{a}} \subseteq_{A} \hat{\bar{a}}$ and $H' \subseteq_M H'$.

\[\square\]
I.36 Message-Map Reflexivity Lemma

**Lemma** (Message-Map Reflexivity). For all $\hat{\mu}$, $A$, and $M$, if

- $\hat{\mu}$ is fully merged,
- $A = id|_D$ for some $D$ that contains every address appearing in $\hat{\mu}$, and
- $M = id|_{H'}$ for some $H'$ that contains every marker appearing in $\hat{\mu}$,

then $\hat{\mu} \sqsubseteq_{A,M} \hat{\mu}$.

*Proof.* Let $\langle \hat{a}@H''', \hat{v} \rangle$ be a member of $\text{dom}(\hat{\mu})$. By the Expression Reflexivity lemma, $\hat{a}@H'' \sqsubseteq_{A,M} \hat{a}@H''$ and $\hat{v} \sqsubseteq_{A,M} \hat{v}$. By the approximation rules for abstract quantities, we have $\hat{\mu}(\hat{a}@H''', \hat{v}) \sqsubseteq \hat{\mu}(\hat{a}@H'', \hat{v})$.

To show that this is the only message in $\hat{\mu}$ that can approximate $\langle \hat{a}@H''', \hat{v} \rangle$, let there be $\langle \hat{a}@H''', \hat{v}' \rangle \in \text{dom}(\hat{\mu})$ such that $\langle \hat{a}@H''', \hat{v}' \rangle \neq \langle \hat{a}@H''', \hat{v} \rangle$. Because $\hat{\mu}$ is fully merged, it must be the case that either $\hat{a}@H'' \neq \hat{a}@H'''$ or $\text{Merge}(\hat{v}, \hat{v}')$ is not defined. In the former case, by the definition of $\sqsubseteq$ there is at most one marked address that can approximate $\hat{a}@H''$, so because $\hat{a}@H'' \sqsubseteq_{A,M} \hat{a}@H''$, $\hat{a}@H'' \sqsubseteq_{A,M} \hat{a}@H'''$. In the latter case, we have $\hat{v} \sqsubseteq_{A,M} \hat{v}$, so by corollary I.19.1 to the Approximation Mergeability lemma, $\hat{v} \not\sqsubseteq_{A,M} \hat{v}'$. Therefore, there is no message $\langle \hat{a}@H''', \hat{v}' \rangle \neq \langle \hat{a}@H''', \hat{v} \rangle$ in $\text{dom}(\hat{\mu})$ such that

- $\hat{a}@H'' \sqsubseteq_{A,M} \hat{a}@H'''$,
- $\hat{v} \sqsubseteq_{A,M} \hat{v}'$, and
- $\hat{\mu}(\hat{a}@H''', \hat{v}) \sqsubseteq \hat{\mu}(\hat{a}@H''', \hat{v}')$.

\[\blacksquare\]

I.37 Approximation Reflexivity Lemma

**Lemma** (Approximation Reflexivity). For all $\hat{K} = \langle \langle \hat{\mu} \rangle \rangle$ $H$, $A$, and $M$, if

- $\hat{K}$ is well-formed,
- $A = id|_D$ for some $D$ such that $\text{dom}(\hat{\mu}) \cup \text{ExtAddr} \subseteq D$, and
- $M = id|_{H'}$ for some $H'$ such that $H \subseteq H'$,

then $\hat{K} \sqsubseteq_{A,M} \hat{K}$.

*Proof.* The proof is by a straightforward inspection of the rules defining $\sqsubseteq_{A,M}$. Some notes on the less straightforward cases are given below.

- The marker-correspondence function $M$ is obviously one-to-one.
- Because $\hat{K}$ is well-formed, every address appearing in it is in $\text{dom}(\hat{\mu}) \cup \text{ExtAddr}$, which implies that it is in $\text{dom}(A)$.
Also because $\tilde{K}$ is well-formed, every marker appearing in $\tilde{K}$ is in $H$, which implies that it is in $\text{dom}(M)$.

The Expression Reflexivity lemma proves reflexivity for abstract expressions.

The Message-Map Reflexivity lemma gives the proof that $\tilde{\mu} \subseteq_{A,M} \tilde{\mu}$ (\(\tilde{\mu}\) is fully merged by the definition of a well-formed abstract program configuration).

\[\square\]

### I.38 Expression Transitivity Lemma

**Lemma (Expression Transitivity).** For all $\hat{\varepsilon}$, $\hat{\varepsilon}'$, $\hat{\varepsilon}''$, $A$, $M$, $A'$, and $M'$, if $M$ and $M'$ are one-to-one, $\hat{\varepsilon} \subseteq A,M \hat{\varepsilon}'$ and $\hat{\varepsilon}' \subseteq A',M' \hat{\varepsilon}''$, then $\hat{\varepsilon} \subseteq A=M \hat{\varepsilon}''$.

**Proof.** Straightforward structural induction on $\hat{\varepsilon}$. \[\square\]

### I.39 Message-Map Transitivity Lemma

**Lemma (Message-Map Transitivity).** For all $\tilde{\mu}$, $\tilde{\mu}'$, $\tilde{\mu}''$, $A$, $M$, $A'$, and $M'$, if $M$ and $M'$ are one-to-one, $\tilde{\mu} \subseteq_{A,M} \tilde{\mu}'$ and $\tilde{\mu}' \subseteq_{A',M'} \tilde{\mu}''$, then $\tilde{\mu} \subseteq_{A=M} \tilde{\mu}''$.

**Proof.** First, let $(\hat{a}@H, \hat{b})$ be a message in $\tilde{\mu}$. By the rules for $\subseteq$, there exist some unique $\langle \hat{a}'@H', \hat{b}' \rangle$ in $\tilde{\mu}'$ and some unique $\langle \hat{a}''@H'', \hat{b}'' \rangle$ in $\tilde{\mu}''$ such that $\hat{a}@H \subseteq_{A,M} \hat{a}'@H' \subseteq_{A,M'} \hat{a}''@H''$, $\hat{b} \subseteq_{A,M} \hat{b}' \subseteq_{A,M'} \hat{b}''$, $\hat{a}@H, \hat{b} \subset \hat{a}'@H', \hat{b}'$, and $\hat{a}@H, \hat{b} \subset \hat{a}''@H'', \hat{b}''$. Then we have $\hat{\mu}(\hat{a}@H, \hat{b}) \subseteq \hat{\mu}(\hat{a}'@H', \hat{b}')$ and by the Expression Transitivity lemma, $\hat{\mu}(\hat{a}@H, \hat{b}) \subseteq \hat{\mu}(\hat{a}''@H'', \hat{b}'')$, and by the Approximation Mergeability lemma, any other message in $\tilde{\mu}''$ that approximates $(\hat{a}@H, \hat{b})$ would be mergeable with $(\hat{a}''@H'', \hat{b}'')$. We already know that $\tilde{\mu}''$ is fully merged, however, so there is no other such message and therefore $\langle \hat{a}''@H'', \hat{b}'' \rangle$ is the only message in $\tilde{\mu}''$ that approximates $(\hat{a}@H, \hat{b})$.

Next, let there be some $\langle \hat{a}''@H'', \hat{b}'' \rangle$ such that $\tilde{\mu}''(\hat{a}''@H'', \hat{b}'') = \text{single}$. Again by the Expression Transitivity lemma, we have $\hat{a}@H \subseteq_{A,A,M} \hat{a}''@H''$ and $\hat{b} \subseteq_{A,M} \hat{b}''$. To show that that message is unique, suppose there is some other $\langle \hat{a}'''@H'''', \hat{b}''' \rangle$ in $\tilde{\mu}$ approximated by $\langle \hat{a}''@H'', \hat{b}'' \rangle$. By the rules for $\subseteq$, there exists some unique $\langle \hat{a}''@H'', \hat{b}'' \rangle$ in $\tilde{\mu}'$ that approximates $\hat{a}@H, \hat{b}$, and some unique $\langle \hat{a}'''@H'''', \hat{b}''' \rangle$ in $\tilde{\mu}''$ that approximates $\hat{a}@H, \hat{b}$. By the Expression Transitivity lemma, $\langle \hat{a}'''@H'''', \hat{b}''' \rangle$ approximates $\langle \hat{a}''@H'', \hat{b}'' \rangle$. Then by the Approximation Mergeability lemma, $\langle \hat{a}'''@H'''', \hat{b}''' \rangle = \langle \hat{a}''@H'', \hat{b}'' \rangle$. We know there is at most one message in $\tilde{\mu}$ approximated by $(\hat{a}@H, \hat{b})$, so $\langle \hat{a}''@H'', \hat{b}'' \rangle = \langle \hat{a}@H, \hat{b} \rangle$. Again we know there is at most one message in $\tilde{\mu}$ approximated by $(\hat{a}@H', \hat{b}')$, so $\langle \hat{a}''@H'', \hat{b}'' \rangle = \langle \hat{a}@H', \hat{b}' \rangle$.

Finally, we know that $\tilde{\mu}''$ is fully merged by the rules for $\subseteq_{A,M} \tilde{\mu}''$, so we have $\tilde{\mu} \subseteq_{A=M} \tilde{\mu}''$. \[\square\]
I.40 Approximation Transitivity Lemma

Lemma (Approximation Transitivity). For all \( \bar{K}, \bar{K}', \bar{K}'', A, M, A', and M' \), if \( \bar{K} \sqsubseteq_{A,M} \bar{K}' \) and \( \bar{K}' \sqsubseteq_{A,M'} \bar{K}'' \), then \( \bar{K} \sqsubseteq_{A',\bar{A},M''} \bar{K}'' \).

Proof. Let \( \langle \beta, H \rangle \bar{K} = \bar{K}', \langle \beta', H' \rangle \bar{K}' = \bar{K}'' \). We will show approximation for each of the components in turn.

First, let \( \bar{a} \) be a member of \( \text{dom}(\beta) \). By the rules for \( \sqsubseteq \), there exists \( \bar{a}' \in \text{dom}(\beta') \) and \( \bar{a}'' \in \text{dom}(\beta'') \) such that \( \bar{a} \sqsubseteq_A \bar{a}' \sqsubseteq_{A'} \bar{a}'' \), \( \beta(\bar{a}) \sqsubseteq_{A,M} \beta'(\bar{a}') \), and \( \beta'(\bar{a}') \sqsubseteq_{A,M'} \beta''(\bar{a}'') \). By the Expression Transitivity lemma, \( \bar{a} \sqsubseteq_{A',\bar{A},M''} \bar{a}'' \). To show that \( \beta(\bar{a}) \sqsubseteq_{A',\bar{A},M''} \beta'(\bar{a}') \), let \( \bar{b} \) be a member of \( \beta(\bar{a}) \). By the rules for \( \sqsubseteq \), there exists some \( \bar{b}' \in \beta'(\bar{a}') \) and some \( \bar{b}'' \in \beta''(\bar{a}'') \) such that \( \bar{b} \sqsubseteq_{A,M} \bar{b}' \) and \( \bar{b}' \sqsubseteq_{A,M'} \bar{b}'' \). It is easy to show by the Expression Transitivity lemma and an inspection of the rules for \( \sqsubseteq \) that this implies that \( \bar{b} \sqsubseteq_{A',\bar{A},M''} \beta'(\bar{a}') \). The judgment \( \bar{b}' \sqsubseteq_{A,M'} \bar{b}'' \) also implies that at most one behavior in \( \beta''(\bar{a}'') \) is handling an event, so we have \( \beta(\bar{a}) \sqsubseteq_{A',\bar{A},M''} \beta'(\bar{a}') \).

We have \( \bar{H} \sqsubseteq_{A',\bar{A},M''} \bar{H}'' \) by the Message-Map Transitivity lemma.

By the rules for \( \sqsubseteq \), we have that \( M(\bar{H}) = \bar{H}' \) and \( M'(\bar{H}') = \bar{H}'' \). Therefore, \( M' \circ M(\bar{H}) = \bar{H}' \), and therefore \( \bar{H} \sqsubseteq_{M,M'} \bar{H}'' \).

Finally, let \( \langle \bar{a} @ \bar{H}', \tau \rangle \) be a member of \( \bar{H} \). By the rules for \( \sqsubseteq \), there exist \( \bar{a}' \), \( \bar{a}'' \), \( \bar{H}''' \), and \( \bar{H}'''' \) such that \( \langle \bar{a}' @ \bar{H}'', \tau \rangle \in \beta', \langle \bar{a}'' @ \bar{H}'''', \tau \rangle \in \beta'' \). By the Expression Transitivity lemma, \( \bar{a} @ \bar{H}'''' \sqsubseteq_{A',\bar{A},M''} \bar{a}'' @ \bar{H}''''' \). So \( \bar{H} \sqsubseteq_{A',\bar{A},M''} \bar{H}'''' \).

\( \Box \)

I.41 Externals-Only Preservation Lemma

Lemma (Externals-Only Preservation). For all \( \bar{K}, l_1, \ldots, l_n, \) and \( \bar{K}' \), if \( \bar{K} \) is an externals-only configuration and \( \bar{K} \xrightarrow{l_1,\ldots,l_n} \bar{K}' \), then \( \bar{K}' \) is an externals-only configuration.

Proof. We prove first that the externals-only property is preserved for a single restricted transition \( \bar{K} \xrightarrow{\hat{i}} \bar{K}' \), then extend that result to event-step transitions.

I.41.1 Restricted Transition

Let there be \( \bar{K}, l, \) and \( \bar{K}' \) such that \( \bar{K} \) is an externals-only configuration and \( \bar{K} \xrightarrow{\hat{i}} \bar{K}' \). The transition \( \xrightarrow{\hat{i}} \) is defined such that \( \bar{K} \xrightarrow{\hat{i}} \bar{K}' \) only if \( \bar{K} \xrightarrow{\hat{i}} \bar{K} \) (see the transitions defined in appendices B–E). Therefore the proof proceeds by cases on the rule enabling the transition \( \bar{K} \xrightarrow{\hat{i}} \bar{K}' \).

M-Receive Internal Case This case involves receiving an in-flight message in \( \bar{K} \) of the form \( \langle a @ \bar{H}, \bar{v} \rangle \). The rule requires that \( a \) identify an actor in \( \bar{K} \), so \( a \) must be an internal address. Then because \( \bar{K} \) is an externals-only configuration,
it must be the case that \( H = \emptyset \). Therefore this step does not change the markings on any addresses, nor does it introduce any new addresses, so \( \tilde{K}' \) is an externals-only configuration.

**M-ReceiveExternal Case** For this rule, there exist some \( \tilde{v}, \tilde{v}', H, \) and \( H' \) such that \( \langle \tilde{v}', H' \rangle \in \text{Markings}(\tilde{v}, H) \). The transition \( \tilde{K} \xrightarrow{R} \tilde{K}' \) only if \( \tilde{K} \xrightarrow{EO} \tilde{K}' \) (see appendix B). That implies that \( \tilde{v} \) contains no internal addresses, and the M-ReceiveExternal rule requires that for all \( a@H'' \) appearing in \( \tilde{v} \), \( H'' = \emptyset \). A marking from Markings adds one marker to each contained address in \( \tilde{v} \), so no internal address appears in \( \tilde{v}' \), and for all \( a@H'' \) appearing in \( \tilde{v}' \), \( |H''| = 1 \). This accounts for all new marked addresses \( a@H'' \) introduced by this rule, so \( \tilde{K}' \) is an externals-only configuration.

**M-SendInternal Case** This case adds new markers only if the target \( a@H \) of the send is marked with at least one marker (i.e., \( H \neq \emptyset \)). However, the rule requires that \( a \) is an internal address, and \( \tilde{K} \) is an externals-only configuration, so \( H = \emptyset \). Therefore this transition introduces no new marked addresses, so \( \tilde{K}' \) is an externals-only configuration.

**M-SendExternal Case** This rule does not add any new marked addresses to the actor map \( \tilde{\beta} \) or the message multiset \( \tilde{\mu} \). If there are new receptionists \( \tilde{\rho}' \), then by the definition of this rule there exist \( \tilde{v}, \tilde{v}', H, \) and \( H' \) such that \( \langle \tilde{v}', H' \rangle \in \text{Markings}(\tilde{v}, H) \) and \( \text{IntAddrTypes}(\tilde{v}', \tau) = \tilde{\rho}' \). Because \( \tilde{K} \) is an externals-only configuration, no internal address in \( \tilde{v} \) has any markers on it. A marking from Markings adds one marker to each contained address in \( \tilde{v} \), and because \( \text{IntAddrTypes} \) only returns the internal marked addresses from \( \tilde{v}' \), it is the case that for all \( \langle a@H', \tau \rangle \in \tilde{\rho}' \), \( |H'| = 1 \). This accounts for all new marked addresses \( a@H'' \) introduced by this rule, so \( \tilde{K}' \) is an externals-only configuration.

**M-Spawn Case** The only new marked address \( a@H \) introduced in this step is the address of the created actor. That address \( a \) is internal by definition, and the rule marks it with no addresses (i.e., \( H = \emptyset \)), so \( \tilde{K}' \) is an externals-only configuration.

**Other Cases** None of the other cases introduce new marked addresses, so \( \tilde{K}' \) is an externals-only configuration in each case.

### I.41.2 Event-Step Transition

It remains to show that the event-step transition \( \xrightarrow{E} \) preserves the externals-only property. Let there be \( \tilde{K}_1, \tilde{l}_1, \ldots, \tilde{l}_n, \) and \( \tilde{K}_{n+1} \) such that \( \tilde{K}_1 \) is an externals-only configuration and \( \tilde{K}_1 \xrightarrow{\tilde{l}_1 \cdots \tilde{l}_n} \tilde{K}_{n+1} \). By the definition of the event-step transition, there must exist \( \tilde{K}_2, \ldots, \tilde{K}_n \) such that \( \tilde{K}_1 \xrightarrow{R} \tilde{K}_2 \xrightarrow{R} \ldots \xrightarrow{R} \tilde{K}_{n+1} \). The above
section proved that each transition \( \hat{K}_i \xrightarrow{\hat{l}_i} \hat{K}_{i+1} \) preserves the externals-only property, so by induction, \( \hat{K}_{n+1} \) is an externals-only configuration.

\[ \square \]

I.42 Abstract Externals-Only Preservation Lemma

**Lemma** (Abstract Externals-Only Preservation). For all \( \hat{K}, \hat{l}, \) and \( \hat{K}' \), if \( \hat{K} \) is an externals-only configuration and \( \hat{K} \xrightarrow{\hat{l}} \hat{K}' \), then \( \hat{K}' \) is an externals-only configuration.

**Proof.** The proof is similar to the argument made in the Restricted Transition part of the proof for the Externals-Only Preservation lemma above. The only differences are that external addresses and receptionists are marked with at most one marker instead of exactly one, and the A-RECEIVE\text{-}INTERNAL and A-SEND\text{-}INTERNAL rules never add new markers, regardless of whether there are markers on the destination address or not.

\[ \square \]

**Corollary I.42.1.** For all \( \hat{K}, \hat{l}_1, \ldots, \hat{l}_n, \) and \( \hat{K}' \), if \( \hat{K} \) is an externals-only configuration and \( \hat{K} \xrightarrow{\hat{l}_1 \ldots \hat{l}_n} \hat{K}' \), then \( \hat{K}' \) is an externals-only configuration.

**Proof.** By the definition of the event-step relation, there exist \( \hat{K}_1'', \ldots, \hat{K}_{n+1}'' \) such that
\[
\hat{K}_1'' \xrightarrow{\hat{l}_1} \ldots \xrightarrow{\hat{l}_n} \hat{K}_{n+1}'' = \hat{K}, \quad \text{and} \quad \hat{K}_{n+1}'' = \hat{K}'.
\]
Then by induction on \( i \) and the Abstract Externals-Only Preservation lemma, \( \hat{K}_i'' \) is an externals-only configuration for all \( i \in 1 \ldots n \). Then because \( \hat{K}_{n+1}'' = \hat{K}', \hat{K}' \) is an externals-only execution.

\[ \square \]

**Corollary I.42.2.** For all \( \hat{K}, L, \) and \( \hat{K}' \), if \( \hat{K} \) is an externals-only configuration and \( \hat{K} \xrightarrow{L} \hat{K}' \), then \( \hat{K}' \) is an externals-only configuration.

**Proof.** By the definition of a summary transition, there exist some \( \hat{l}_1, \ldots, \hat{l}_n \) such that \( \hat{K} \xrightarrow{\hat{l}_1 \ldots \hat{l}_n} \hat{K}' \). Then by corollary I.42.1, \( \hat{K}' \) is an externals-only configuration.

\[ \square \]

I.43 Single-Handler Preservation Lemma

**Lemma** (Single-Handler Preservation). For all \( \hat{K}, \hat{l}, \) and \( \hat{K}' \), if \( \hat{K} \) is a single-handler configuration and \( \hat{K} \xrightarrow{\hat{l}} \hat{K}' \), then \( \hat{K}' \) is a single-handler configuration.

**Proof.** We prove first that the single-handler property is preserved for a single restricted transition \( \hat{K} \xrightarrow{\hat{l}} \hat{K}' \), then extend that result to event-step transitions.
I.43.1 Restricted Transition

Let there be $\bar{K}, \bar{l},$ and $\bar{K}'$ such that $\bar{K}$ is a single-handler configuration and $\bar{K} \xrightarrow{l} \bar{K}'$. The transition $\xrightarrow{l}$ is defined such that $\bar{K} \xrightarrow{l} \bar{K}'$ only if $\bar{K} \xrightarrow{l} \bar{K}'$ (see the transitions defined in appendices B–E). Therefore the proof proceeds by cases on the rule enabling the transition $\bar{K} \xrightarrow{l} \bar{K}'$. Each case relies on the fact that the transition $\xrightarrow{l}$ is defined such that $\bar{K} \xrightarrow{l} \bar{K}'$ only if $\bar{K} \xrightarrow{l} \bar{K}'$ (see appendix E).

**M-ReceiveInternal Case** For this rule, $\bar{l}$ must be a handler-start label (see section 2.5.4) of the form $a : \text{rcv-int}(H, \bar{v})$. The transition relation $\xrightarrow{SH}$ allows a transition with a handler-start label only if no actor in $\bar{K}$ is handling an event, and an M-ReceiveInternal transition results in just one additional actor handling an event. Therefore, only one actor in $\bar{K}'$ is handling an event, so $\bar{K}'$ is a single-handler configuration.

**M-ReceiveExternal Case** Same as the M-ReceiveInternal Case, with a handler-start label of the form $a : \text{rcv-ext}(H, \bar{v}, \tau)$.

**M-Timeout Case** Same as the M-ReceiveInternal Case, with a handler-start label of the form $a : \text{timeout}$.

**M-Spawn Case** For this case, $\bar{l}$ must be a handler-continuation label (again, see section 2.5.4) of the form $a : \text{spawn}(a')$, where $a = (\text{addr } \ell' \ n)$ and $a' = (\text{addr } \ell' \ n')$. This rule creates a new actor at $a'$, so we must show that there is no other actor spawned from $\ell'$ currently handling an event in $\bar{K}$. We will prove this by contradiction.

Assume that such an actor does exist, at address $a'' = (\text{addr } \ell'' \ n'')$. Because the actor at $a$ is spawning an actor at $a'$, it must be the case that the spawn expression at location $\ell'$ must be lexically nested inside the spawn expression at location $\ell$. Therefore, $\ell < \ell'$ (see appendix D).

By the definition of $<$ for addresses, $a < a''$. The transition relation $\xrightarrow{SH}$ allows a transition with a handler-continuation label only if there is no $a''$ such that $a < a''$ and the actor at $a''$ in $\bar{K}$ is handling an event. The transition $\bar{K} \xrightarrow{l} \bar{K}'$ is possible though, so this is a contradiction. Therefore there is no actor spawned from $\ell'$ currently handling an event in $\bar{K}$.

A M-Spawn transition does not otherwise change which actors are handling events, so $\bar{K}'$ is a single-handler configuration.

**Other Cases** None of the other transition rules change the number of actors handling an event, so $\bar{K}'$ is a single-handler configuration in each case.
I.43.2 Event-Step Transition

It remains to show that the event-step transition preserves the single-handler property. Let there be $\hat{K}_1, \hat{I}_1, \ldots, \hat{I}_n$, and $\hat{K}_{n+1}$ such that $\hat{K}_1$ is a single-handler configuration and $\hat{K}_1 \xrightarrow{\hat{I}_1} \hat{K}_2 \xrightarrow{\hat{I}_2} \cdots \xrightarrow{\hat{I}_n} \hat{K}_{n+1}$. By the definition of the event-step transition, there must exist $\hat{K}_2, \ldots, \hat{K}_n$ such that $\hat{K}_1 \xrightarrow{\hat{I}_1} \hat{K}_2 \cdots \xrightarrow{\hat{I}_n} \hat{K}_{n+1}$. The above section proved that each transition $\hat{K}_i \xrightarrow{\hat{I}_i} \hat{K}_{i+1}$ preserves the single-handler property, so by induction, $\hat{K}_{n+1}$ is a single-handler configuration.

I.44 Abstract Conformance Theorem

**Theorem** (Abstract Conformance). For all $P$ and $\Sigma$, if $P \models_A \Sigma$, then $P \models_{PSM} \Sigma$.

**Proof.** Let there be $P$ and $\Sigma$ such that $P \models_A \Sigma$. By the definition of $\models_A$, there must exist some maximal instantiation $\langle \hat{K}_{init}, \{s_{init}\} \rangle$ of $P$ and $\Sigma$ such that $|\hat{K}_{init}| \models_A s_{init}$. We will show that $\hat{K}_{init} \models_{PSM} s_{init}$, which implies $P \models_{PSM} \Sigma$.

By the definition of $\models_A$, there must exist some abstract-conformance-dense relation $R_A$ such that $\langle |\hat{K}_{init}|, s_{init} \rangle \in R_A$. Let $R_C$ be the set of all $\langle \hat{K}, s \rangle$, with $\hat{K} = \left\langle \hat{\beta} \mid H \right\rangle^\delta$, such that there exist some $\hat{K}, s', A, \text{and} M$ such that

1. $\hat{K}$ is a well-formed, externals-only, single-handler configuration,
2. $\hat{K}$ is non-stuck,
3. $|\hat{K}| \subseteq_{A,M} \hat{K}$,
4. $M(s) = s'$,
5. $\text{dom}(A) = \text{dom}(\hat{\beta}) \cup \text{ExtAddr}$,
6. $\text{dom}(M) \subseteq H$,
7. for all $\hat{a} \in \text{ExtAddr}$, $A(\hat{a}) = \hat{a}$, and
8. $\langle \hat{K}, s' \rangle \in R_C$.

We will show that $\langle \hat{K}_{init}, s_{init} \rangle \in R_C$ and that $R_C$ is PSM-conformance-dense, thus proving that $\hat{K}_{init} \models_{PSM} s_{init}$. To show that $\langle \hat{K}_{init}, s_{init} \rangle \in R_C$, let $\hat{K}_{init} = \left\langle \hat{\beta} \mid H \right\rangle^\delta$, let $\hat{K} = |\hat{K}_{init}|$, and let $s' = s_{init}$. Furthermore, let $A = \text{id}|_{\text{dom}(\hat{\beta}) \cup \text{ExtAddr}}$ and let $M = \text{id}|_H$. The proofs for each of the conditions in the definition of $R_C$ follow.

1. By the definition of instantiation for a program configuration (section 3.5.2),
   - $H_C$ contains all markers in the configuration,
• every external address is marked with one marker,
• every receptionist is marked with a unique marker,
• every internal address in $\mathcal{K}_{init}$ has no markers,
• every external address in $\mathcal{K}_{init}$ has exactly one marker, and
• every actor in $\mathcal{K}_{init}$ is awaiting an event.

2. By the definition of instantiation, every actor in $\mathcal{K}_{init}$ is awaiting an event. Then by the definition of $|\cdot|$, every actor in $\mathcal{K}$ is awaiting an event. An actor awaiting an event is not stuck, so $\mathcal{K}$ is non-stuck.

3. From the definition of $|\cdot|$ and because $\mathcal{K}_{init}$ is well-formed, it is easy to see that $|\mathcal{K}_{init}|$ is well-formed (the message-map component $\hat{\mu}$ in $|\mathcal{K}_{init}|$ must be fully merged by the definition of $|\cdot|$). Then by the Approximation Reflexivity lemma, $|\mathcal{K}_{init}| \subseteq_{A,M} |\mathcal{K}_{init}|$.

4. By the definition of an instantiation of a program and specification (sections 3.5.2 and 3.6.2), every marker appearing in $s$ is in $H_C$, and therefore is in $\text{dom}(M)$. Therefore because $M$ is a restricted version of the identity function, $M(s) = s = s'$.

5. By the definition of $A$.

6. By the definition of $M$.

7. By the definition of $A$.

8. By the definition of $R_A$.

It remains to show that $R_C$ is PSM-conformance-dense. Let $\langle \mathcal{K}_1, s \rangle$ be a member of $R_C$, with $\mathcal{K}_1 = \langle \hat{\beta} \hat{\mu} | H \rangle_{\hat{\beta}}$, and let $\mathcal{K}_1 \xrightarrow{\ell_1, \ldots, \ell_n} \ldots$ be a fair, non-stuck execution. By the Well-Formed Preservation, Externals-Only Preservation, and Single-Handler Preservation lemmas, all of the $\mathcal{K}_i$ in the execution are well-formed, externals-only, single-handler configurations. By the definition of $R_C$, there exist some $\hat{\mathcal{K}}_1$, $s'$, $A$, and $M$ such that

• $\hat{\mathcal{K}}_1$ is non-stuck,
• $|\hat{\mathcal{K}}_1| \subseteq_{A,M} \mathcal{K}_1$,
• $M(s) = s'$,
• $\text{dom}(A) = \text{dom}(\hat{\beta}) \cup \text{ExtAddr}$,
• $\text{dom}(M) \subseteq H$,
• for all $\hat{a} \in \text{ExtAddr}$, $A(\hat{a}) = \hat{a}$, and
• $\langle \hat{\mathcal{K}}_1, s' \rangle \in R_A$. 
I.44. ABSTRACT CONFORMANCE THEOREM

By the Event-Step Execution Soundness lemma, there also exist $A'$, $M'$, and an abstract event-step execution $\hat{K}_1 \xrightarrow{\bar{l}_{i,1},...\bar{l}_{i,n}} ...$ of the same length as $\bar{K}_1 \xrightarrow{\bar{l}_{1,1},...\bar{l}_{1,n}} ...$ such that

- $|\hat{K}_i| \subseteq A \cup A', M \cup M' \xrightarrow{\bar{l}_{i},\bar{l}_{i,n}} \hat{K}_i$ for all corresponding configurations in the two executions and

- $|\bar{l}_{i,j}| \subseteq A \cup A', M \cup M' \xrightarrow{\bar{l}_{i,j}}$ for all corresponding labels in the two executions.

Because an event-step execution can be stuck only when its first configuration is stuck, that execution is non-stuck. Furthermore, by the Soundness of Fair Executions lemma, that execution is fair.

By the definition of an abstract-conformance-dense relation, there must be some fair specification execution $S'_1 \langle \bar{l}_{1,1}, O_{1,1} \rangle, ... \langle \bar{l}_{1,m}, O_{1,m} \rangle ...$ with the same length as $\hat{K}_1 \xrightarrow{\bar{l}_{1,1},...\bar{l}_{1,n}} ...$ such that $S'_1 = \{s'\}$ and for all $\hat{K}_i$ and $S'_i$ in the respective executions and all $s'' \in S'_i$, $\langle \hat{K}_i, s'' \rangle \in R_A$.

Because every marker in $s$ appears in $dom(M)$, we have that $(M \cup M'(s) = s'$. Furthermore, because $|\hat{K}_i| \subseteq A \cup A', M \cup M'$ is one-to-one. Then let $S_1 = \{s\}$, so that $M'(S_1) = S'_1$. By repeated uses of corollary I.34.3 to the Configuration Completeness lemma, there exists a specification execution $S_1 \langle \bar{l}_{1,1}, O_{1,1} \rangle, ... \langle \bar{l}_{1,m}, O_{1,m} \rangle ...$ with the same length as $S'_1 \langle \bar{l}_{1,1}, O_{1,1} \rangle, ... \langle \bar{l}_{1,m}, O_{1,m} \rangle ...$ such that $M'(S_i) = S'_i$ for all $S_i$ and $S'_i$ in the respective executions, and $M'(O'_i \langle \bar{l}_{i,1}, O_{i,1} \rangle, ... \langle \bar{l}_{i,m}, O_{i,m} \rangle ...$ is fair, and because that execution is related to the $S_1 \langle \bar{l}_{1,1}, O_{1,1} \rangle, ... \langle \bar{l}_{1,m}, O_{1,m} \rangle ...$ execution by a one-to-one marker-correspondence function $M'$, the execution $S_1 \langle \bar{l}_{1,1}, O_{1,1} \rangle, ... \langle \bar{l}_{1,m}, O_{1,m} \rangle ...$ is also fair.

It remains to show that for all $\hat{K}_i$ and $S_i$ in the respective concrete executions and all $s'' \in S_i$, $\langle \hat{K}_i, s'' \rangle \in R_C$. For each such pair, let $\hat{K}_i$ be the corresponding abstract configuration required for the definition of $R_C$, and let the PSM $s''$ in $S_i$ such that $M'(s'') = s'''$ be the corresponding PSMa (such a PSM must exist by the definition of $M(S_i) = S'_i$). Furthermore, let $\hat{K}_i = \langle \bar{\beta}' | \bar{\beta}' | H' \rangle$, let $A'' = (A \cup A')|_{dom(\bar{\beta}')}$.ExtAddr, and let $M'' = (M \cup M')|_H$. The proofs for each of the conditions in the definition of $R_C$ follow.


2. The abstract execution $\hat{K}_1 \xrightarrow{\bar{l}_{1,1},...\bar{l}_{1,n}} ...$ is non-stuck, so by the definition of a stuck abstract execution, $\hat{K}_i$ is not stuck.
3. From above, we know that $|\tilde{K}_i| \subseteq A \circ M \circ M' \tilde{K}_i$. Because $\tilde{K}_i$ is well-formed, the only addresses appearing in $|\tilde{K}_i|$ are in $\text{dom}(\beta') \cup \text{ExtAddr}$. Similarly, the only markers appearing in $|\tilde{K}_i|$ must be in $H'$. Therefore, $A^n$ and $M^n$ give the exact same results as $A \cup A'$ and $M \cup M'$ for all addresses and markers appearing in $|\tilde{K}_i|$, so $|\tilde{K}_i| \subseteq A^n \circ M^n \tilde{K}_i$.

4. By the definition of $s''$.

5. By the definition of $A^n$.

6. By the definition of $M^n$.

7. By the definition of $A^n$.

8. By the definition of the execution $S_i' \overrightarrow{(i_{1,1},O_{i,1}) \ldots (i_{1,n},O_{i,n})} \ldots$ from which $s''$ came.

Thus, for all $\tilde{K}_i$ and $S_i$ in the respective concrete executions and all $s'' \in S_i$, $\langle \tilde{K}_i, s'' \rangle \in R_C$, and $R_C$ is PSM-conformance-dense.

\section*{I.45 Summary Conformance Theorem}

\textbf{Theorem (Summary Conformance).} For all $P$ and $\Sigma$, if $P \vdash_\Sigma \Sigma$, then $P \vdash_A \Sigma$.

\begin{proof}
Let there be $P$ and $\Sigma$ such that $P \vdash_\Sigma \Sigma$. By the definition of $\vdash_\Sigma$, there exists some maximal instantiation $\langle K_{\text{init}}, s_{\text{init}} \rangle$ of $P$ and $\Sigma$ such that $|K_{\text{init}}| \vdash_\Sigma s_{\text{init}}$. That in turn implies there must exist some summary-conformance-dense relation $R$ such that $\langle |K_{\text{init}}|, s_{\text{init}} \rangle \in R$. We will show that $R$ is also abstract-conformance-dense, which implies that $|K_{\text{init}}| \vdash_A s_{\text{init}}$ and therefore $P \vdash_A \Sigma$.

Let $\langle K_1, s \rangle$ be a member of $R$, and let $\tilde{K}_1 \overrightarrow{i_{1,1} \ldots i_{1,n}} \ldots$ be a fair, non-stuck execution. We must show that there exists a fair specification execution $S_1 \overrightarrow{[i_{1,1} \ldots i_{1,n}]} \ldots$ with the same length such that $S_1 = \{s\}$ and for all $\tilde{K}_i$ and $S_i$ in the respective executions and all $s' \in S_i$, $\langle \tilde{K}_i, s' \rangle \in R$.

Because $R$ is summary-conformance-dense, there exists a fair summary execution $\tilde{K}_1 \overrightarrow{L_1} \ldots$ that summarizes $\tilde{K}_1 \overrightarrow{i_{1,1} \ldots i_{1,n}} \ldots$, and a fair specification summary execution $S_1 \overrightarrow{(i_2,O_2)} \ldots S_2 \overrightarrow{(i_3,O_3)} \ldots$ with the same length such that $S_1 = \{s\}$ and $\langle \tilde{K}_i, s' \rangle \in R$ for all $\tilde{K}_i$ and $S_i$ in the respective executions and all $s' \in S_i$. Then by the definition of a specification summary execution, there exists a specification execution $S_1 \overrightarrow{[i_{1,1},O_{i,1}^1 \ldots (i_{1,n},O_{i,1}^n)] \ldots}$ such that for all $O_i$ in the specification summary execution, $O_i \subseteq O_{i,1}^1 \cup \ldots \cup O_{i,n}^n$.

It remains to show that the execution $S_1 \overrightarrow{[i_{1,1},O_{i,1}^1 \ldots (i_{1,n},O_{i,1}^n)] \ldots}$ is fair, meaning that every obligation in every configuration $S_i$ is eventually fulfilled. Let $S_i$ be a configuration in that execution, and let there be $\langle \eta, po \rangle \in \text{Obls}(S_i)$. Because the summary execution is fair, there exists some $j$ such that $\langle \eta, po \rangle \in$
\( O_{i+j} \). Then because \( O_{i+j} \subseteq O_{i+j,1} \cup \ldots \cup O_{i+j,n_{i+j}} \), there exists some \( k \leq n_{i+j} \) such that \( \langle \eta, po \rangle \in O_{i+j,k} \). Therefore, all obligations are fulfilled, and the execution 
\[
S_1 \langle \hat{t}_{1,1}, O_{1,1} \rangle, \ldots, \langle \hat{t}_{1,n}, O_{1,n} \rangle, \ldots
\] is fair, which completes the proof. \( \square \)
Appendix J

Type Preservation Proofs

This appendix provides proofs that program instantiation, abstraction, and transitions on concrete and abstract program configurations all preserve well-typedness. These properties are necessary to show that the ModelCheck algorithm terminates (appendix N), as well as to show that externals-only conformance implies standard conformance (appendix B).

J.1 Substitution Lemma

Lemma (Substitution). For all \( \Gamma, x, \tau, \Theta, \bar{e}, \bar{Q}, \bar{t}c, \text{ and } \bar{e}', \text{ if } \Gamma, \emptyset \vdash \bar{e}' : \tau', \text{ then} \)

- \( \text{if } \Gamma [x \rightarrow \tau'], \Theta \vdash \bar{e} : \tau, \text{ then } \Gamma, \emptyset \vdash \bar{e}[x \rightarrow \bar{e}'] : \tau, \)
- \( \text{if } \Gamma [x \rightarrow \tau'], \Theta \vdash \text{state } \bar{Q}, \text{ then } \Gamma, \emptyset \vdash \text{state } \bar{Q}[x \rightarrow \bar{e}'], \text{ and} \)
- \( \text{if } \Gamma [x \rightarrow \tau'], \Theta \vdash \bar{t}c, \text{ then } \Gamma, \emptyset \vdash \bar{t}c[x \rightarrow \bar{e}']. \)

Proof. Straightforward induction on the derivation of \( \Gamma [x \rightarrow \tau'], \Theta \vdash \bar{e} : \tau, \Gamma [x \rightarrow \tau'], \Theta \vdash \text{state } \bar{Q}, \text{ or } \Gamma [x \rightarrow \tau'], \Theta \vdash \bar{t}c \bar{t}c \) as appropriate. Note that because the substituted expression \( \bar{e} \) must type-check in an environment without any state definitions, it cannot be a \texttt{goto} expression, so crossing into the body of a \texttt{spawn} expression with a different state environment from that of the current term does not affect the typing. \( \square \)

Corollary J.1.1. For all \( \Gamma, x_1, \ldots, x_n, \tau, \tau'_1, \ldots, \tau'_n, \Theta, \bar{e}, \bar{Q}, \bar{t}c, \text{ and } \bar{e}'_1, \ldots, \bar{e}'_n, \text{ if } \)

- \( \text{if } \emptyset \vdash \bar{e}_i : \tau'_i \text{ for all } i \in 1 \ldots n, \text{ then} \)
  - \( \text{if } \Gamma = \bar{e} \text{ and } \Gamma [x_1 \rightarrow \tau'_1, \ldots, x_n \rightarrow \tau'_n], \Theta \vdash \bar{e} : \tau \text{ then } \)
    \( \Gamma, \emptyset \vdash \bar{e}[x_1 \rightarrow \bar{e}'_1] \ldots [x_n \rightarrow \bar{e}'_n] : \tau, \)
  - \( \text{if } \Gamma = \bar{Q} \text{ and } \Gamma [x_1 \rightarrow \tau'_1, \ldots, x_n \rightarrow \tau'_n], \text{state } \bar{Q}, \text{ then } \)
    \( \Gamma, \emptyset \vdash \text{state } \bar{Q}[x_1 \rightarrow \bar{e}'_1] \ldots [x_n \rightarrow \bar{e}'_n], \text{ and} \)
  - \( \text{if } \Gamma = \bar{t}c \text{ and } \Gamma [x_1 \rightarrow \tau'_1, \ldots, x_n \rightarrow \tau'_n], \text{tc } \bar{t}c, \text{ then } \)
    \( \Gamma, \emptyset \vdash \bar{t}c[x_1 \rightarrow \bar{e}'_1] \ldots [x_n \rightarrow \bar{e}'_n]. \)

375
Proof. Straightforward induction on \( n \).

### J.2 Instantiation Type Preservation Lemma

**Lemma** (Instantiation Type Preservation). If

1. \( \text{Inst}(P, (\text{addr } \ell_1 \ n_1) \ldots (\text{addr } \ell_m \ n_m')) = \bar{K} \),
2. \( \vdash_{\text{prog}} P \),
3. the provided external addresses \( (\text{addr } \ell_1 \ n_1) \ldots (\text{addr } \ell_m \ n_m') \) are distinct from each other,
4. for all \( i \in 1 \ldots m \), \( \ell_i = (\text{env } n') \) for some \( n' \),
5. the declared externals in \( P \) are \( [x_1 \tau_1] \ldots [x_m \tau_m] \),
6. and \( \tau_i = \tau_j \) whenever \( \ell_i = \ell_j \) for all \( i, j \in 1 \ldots m \),

then \( \vdash_{\text{cfg}} \bar{K} \).

**Proof.** In the following, let \( \langle \beta \mid \mu \mid H \rangle^{\bar{\rho}} = \bar{K} \). The proofs for each requirement of a well-typed configuration are below (the English descriptions correspond to the premises of the type rule for configurations).

**Every receptionist corresponds to an internal actor, and the address type-checks at the receptionist’s type.** Let \( \langle a@H, \tau \rangle \) be a member of \( \bar{\rho} \). By the definition of instantiation, there exist \( x, x'_1, \ldots, x'_p, \tau'_1, \ldots, \tau'_p \), and \( a'_1, \ldots, a'_p \) such that \( [x \tau] \) is a declared receptionist in \( P \), \( a = x[x'_1 \leftarrow a'_1] \ldots [x'_p \leftarrow a'_p] \), and for all \( i \in 1 \ldots p \), \( \tau'_i \) is the type for the spawn expression for the internal actor declared with name \( x'_i \), and \( a'_i \) is an internal actor such that \( \text{ActorType}(a'_i) = \tau'_i \). By that definition of \( a \), \( a \) must be internal. Because \( P \) is well-typed, there exists some \( \Gamma = [x'_1 \leftarrow (\text{Addr } \tau'_1), \ldots, x'_p \leftarrow (\text{Addr } \tau'_p)] \) such that \( \Gamma, \emptyset \vdash x : (\text{Addr } \tau) \). Then by corollary J.1.1 to the Substitution lemma, \( \Gamma, \emptyset \vdash a@H : (\text{Addr } \tau) \).

**All addresses of actors in the configuration are internal.** By the definition of \( \text{Inst} \) (each address uses a location \( \ell \) from one of the spawn expressions in \( P \), which is internal by definition).

**Every internal actor’s behavior type-checks** Because \( P \) is well-typed, the initial expression and state definitions for each initial spawn expression must be well-typed when given the proper types for the externals and previous actors in scope. For each actor behavior in \( \bar{K} \), the state definitions are merely the definitions from one of those spawn expressions with the in-scope addresses substituted in. The other component of each actor’s behavior is a special receive term in the initial state given by the spawn expression, which was reached by substituting the in-scope addresses into the goto arguments and substituting...
those arguments into the message handler and timeout clause. Instantiation ensures that the type of each external in $\chi$ matches its declared type in $P$, and that the types of all internal addresses match their declared spawn expressions in $P$. Furthermore, because $P$ is well-typed, the initial goto arguments for each actor match the types for the corresponding state’s formal parameters. Therefore, by corollary J.1.1 of the Substitution lemma, each actor’s behavior in $K$ type-checks.

The type of every in-transit message matches its destination address’s type. The instantiated configuration has no in-transit messages, so this is trivially true.

### J.3 Markings Type Preservation Lemma

**Lemma** (Markings Type Preservation). For all $\vec{v}, \vec{v}', H, H', \Gamma, \Theta$, and $\tau$, if $\Gamma, \Theta \vdash \vec{v} : \tau$ and $\langle \vec{v}', H' \rangle \in \text{Markings}(\vec{v}, H)$, then $\Gamma, \Theta \vdash \vec{v}' : \tau$.

**Proof:** Straightforward structural induction on $\vec{v}'$, following the definition of Markings.

### J.4 Type Inversion Lemma

**Lemma** (Type Inversion). For all $\vec{v}$ and $\tau$ such that $\phi, \phi \vdash \vec{v} : \tau$, the following conditions hold.

1. If $\vec{v} = (\text{variant } t \ \vec{v}_1 \ldots \vec{v}_m)$, then $\tau = (\text{Variant } \{t_1 \ \tau_{1,1} \ldots \tau_{1,m}\} \ldots \{t_n \ \tau_{n,1} \ldots \tau_{n,m}\})$, and there exists $i \in 1 \ldots n$ such that $t = t_i$ and $\phi, \phi \vdash \vec{v}_i' : \tau_{i,j}$ for all $j \in 1 \ldots m$.

2. If $\vec{v} = (\text{record } [r_1 \ \vec{v}_1'] \ldots [r_n \ \vec{v}_n'])$, then $\tau = (\text{Record } [r_1 \ \tau_1'] \ldots [r_n \ \tau_n'])$ and $\phi, \phi \vdash \vec{v}_i' : \tau_i$ for all $j \in 1 \ldots n$.

3. If $\vec{v} = a$, then $\tau = (\text{Addr } \tau')$ and $\tau' <: \text{ActorType}(a)$.

4. If $\vec{v} = (\text{fold } (\text{rec } X \ \tau') \ \vec{v}')$, then $\phi, \phi \vdash \vec{v}' : \tau'[X \leftarrow (\text{rec } X \ \tau')]$ and $(\text{rec } X \ \tau') <: \tau$.

5. If $\vec{v} = (\text{list } \vec{v}_1' \ldots \vec{v}_n')$, then $\tau = (\text{List } \tau')$ and $\phi, \phi \vdash \vec{v}_i' : \tau'$ for all $i \in 1 \ldots n$.

6. If $\vec{v} = (\text{dict } [\vec{v}_1' \ldots \vec{v}_m'] \ [\vec{v}_1'' \ldots \vec{v}_n''])$, then $\tau = (\text{Dict } \tau' \ \tau'')$ and $\phi, \phi \vdash \vec{v}_i' : \tau'$ for all $i \in 1 \ldots n$ and $\phi, \phi \vdash \vec{v}_j'' : \tau''$ for all $j \in 1 \ldots m$.

**Proof:** By induction on the derivation of $\phi, \phi \vdash \vec{v} : \tau$. The proof is similar to the proof of the Abstract Type Inversion lemma below.
J.5 Context Lemma

Lemma (Context). For all \( \Theta, \bar{E}, \bar{e}, \) and \( \tau \), if \( \phi, \Theta \vdash \bar{E}[\bar{e}] : \tau \), then there exist \( \tau' \) and \( x \) such that \( \phi, \Theta \vdash x : \tau' \), \( x \) is not free in \( \bar{E} \) or \( \bar{e} \), and \( [x \leftarrow \tau'], \Theta \vdash \bar{E}[x] : \tau \).

Proof. Straightforward induction on the derivation of \( \phi, \Theta \vdash \bar{E}[\bar{e}] : \tau \).

\( \square \)

J.6 IntAddrTypes Correctness Lemma

Lemma (IntAddrTypes Correctness). For all \( \bar{v} \) and \( \tau \), if \( \phi, \Theta \vdash \bar{v} : \tau \), then for all \( (a@H, \tau') \in \text{IntAddrTypes}(\bar{v}, \tau) \), \( \phi, \Theta \vdash a@H : (\text{Addr} \tau') \).

Proof. Straightforward induction on the derivation of \( \phi, \Theta \vdash \bar{v} : \tau \), following the definition of IntAddrTypes.

\( \square \)

J.7 Functional Step Type Preservation Lemma

Lemma (Functional Step Type Preservation). For all \( \Theta, \bar{e}, \bar{e}', \) and \( \tau \), if \( \phi, \Theta \vdash \bar{e} : \tau \) and \( \bar{e} \rightarrow \bar{e}' \), then \( \phi, \Theta \vdash \bar{e} : \tau \).

Proof. The proof is by case analysis on the rule enabling the transition \( \bar{e} \rightarrow \bar{e}' \).

Case 1

In this case, \( \bar{e} = (\text{begin} \ \bar{v}) \) and \( \bar{e}' = \bar{v} \). By a straightforward induction on the derivation of \( \phi, \Theta \vdash \bar{e} : \tau \), we have \( \phi, \bar{v} \vdash \tau \), which completes the proof.

Case 2

In this case, \( \bar{e} = (\text{begin} \ \bar{v} \ \bar{e}_1'' \ldots \bar{e}_n'') \) and \( \bar{e}' = (\text{begin} \ \bar{e}_1''' \ldots \bar{e}_n''') \). By a straightforward induction on the derivation of \( \phi, \Theta \vdash \bar{e} : \tau \), there exist \( \tau_1', \ldots, \tau_n' \) such that \( \phi, \Theta \vdash \bar{e}_i'' : \tau_i' \) for all \( i \in 1 \ldots n \) and \( \tau_1' = \tau \). Therefore, by the type-checking rule for \text{begin}, \phi, \Theta \vdash \bar{e} : \tau \).

Case 3

In this case, \( \bar{e} = (: (\text{record} [r' \bar{v}'] [r \bar{v}] [r'' \bar{v}'']) r) \) and \( \bar{e}' = \bar{v} \). By a straightforward induction on the derivation of \( \phi, \Theta \vdash \bar{e} : \tau \), we have \( \phi, \bar{v} \vdash \tau \), which completes the proof.

Case 4

In this case, \( \bar{e} = (\text{case} (\text{variant} t \ \bar{v}_1 \ldots \bar{v}_n) \ldots [(t \ x_1 \ldots x_n) \ \bar{e}'']) \) and \( \bar{e}' = \bar{e}''[x_1 \leftarrow \bar{v}_1] \ldots [x_n \leftarrow \bar{v}_n] \). By a straightforward induction on the derivation of \( \phi, \Theta \vdash \bar{e} : \tau \), there exist \( \tau_1', \ldots, \tau_n' \) such that \( \phi, \Theta \vdash \bar{v}_i : \tau_i' \) for all \( i \in 1 \ldots n \) and \( [x_1 \leftarrow \tau_1', \ldots, x_n \leftarrow \tau_n'], \Theta \vdash \bar{e}''' : \tau \). By corollary J.1.1 to the Substitution lemma, \( \phi, \Theta \vdash \bar{e}''[x_1 \leftarrow \bar{v}_1] \ldots [x_n \leftarrow \bar{v}_n] : \tau \), which completes the proof.
Case 5

In this case, \( \hat{e} = (\text{unfold } \tau') (\text{fold } \tau'' \hat{v}) \) and \( \hat{e}' = \hat{v} \). By a straightforward induction on the derivation of \( \phi, \Theta \vdash \hat{e} : \tau, \phi, \Theta \vdash \hat{v} : \tau \), which completes the proof.

Case 6

In this case, \( \hat{e} = (o \hat{v}_1 \ldots \hat{v}_n) \) and \( \hat{e}' = \text{EvalPrimop}(a, \hat{v}_1 \ldots \hat{v}_n) \). By the definition of \( \text{EvalPrimop} \), \( \phi, \Theta \vdash \hat{e}' : \tau \).

Case 7

In this case, \( \hat{e} = (\text{for/fold } [x \hat{v}] [x' (\text{list } \hat{v}_1' \hat{v}_2' \ldots \hat{v}_n')] \hat{e}'') \) and \( \hat{e}' = \hat{e}'' \). By a straightforward induction on the derivation of \( \phi, \Theta \vdash \hat{e} : \tau \), there exists \( \tau' \) such that \( \phi, \Theta \vdash \hat{v} : \tau \), \( \phi, \Theta \vdash \hat{v}_i : \tau' \) for all \( i \in 1 \ldots n \), \( \phi, \Theta \vdash (\text{list } \hat{v}_1' \hat{v}_2' \ldots \hat{v}_n') : (\text{List } \tau') \), and \( [x \rightarrow \tau, x' \rightarrow \tau'], \Theta \vdash \hat{e}'' : \tau \).

By corollary 1.1.1 to the Substitution lemma, \( \phi, \Theta \vdash \hat{e}'' : \tau \). By the type-checking rule for lists, we have \( \phi, \Theta \vdash (\text{list } \hat{v}_1' \hat{v}_2' \ldots \hat{v}_n') : (\text{List } \tau') \). Then by the type-checking rule for \( \text{for/fold} \), we have \( \phi, \Theta \vdash (\text{for/fold } [x \hat{v}'] [x' (\text{list } \hat{v}_1' \hat{v}_2' \ldots \hat{v}_n')] \hat{e}) : \tau \), which completes the proof.

square

J.8 Type Preservation Lemma

Lemma (Type Preservation). For all \( \hat{K}, \hat{K}' \), and \( \hat{l} \), if \( \vdash_{\text{cfg}} \hat{K} \) and \( \hat{K} \xrightarrow{\hat{l}} \hat{K}' \), then \( \vdash_{\text{cfg}} \hat{K}' \).

Proof: In the following, let \( \langle \hat{\beta} \mid \hat{\mu} \mid H \rangle^{\hat{\beta}} = \hat{K} \) and \( \langle \hat{\beta}' \mid \hat{\mu}' \mid H' \rangle^{\hat{\beta}'} = \hat{K}' \). The proof is by case analysis on the rule enabling the transition \( \hat{K} \xrightarrow{\hat{l}} \hat{K}' \).

M-GOTO

By the definition of this rule, there exist \( a, \hat{b}, \hat{b}', \hat{Q}, Q, q, \) and \( \hat{v}_1, \ldots, \hat{v}_n \) such that

- \( \hat{\beta}(a) = \hat{b} = \langle \hat{Q}, \hat{E} \mid (\text{goto } q \hat{v}_1 \ldots \hat{v}_n) \rangle \),
- \( (\text{define-state} (q \ [x_1 \tau_1] \ldots [x_n \tau_n]) x' \hat{e} \hat{c} \) is in \( \hat{Q} \), and
- \( \hat{\beta}'(a) = \hat{b}' = \langle \hat{Q}, (\text{receive} x' \hat{e} \hat{c}) [x_1 \rightarrow \hat{v}_1] \ldots [x_n \rightarrow \hat{v}_n] \rangle \).
Because $\vdash_{\text{cfg}} \tilde{K}$, it must be the case that there exists some $\tau$ such that $\tau, \varnothing \vdash_{\text{beh}} \tilde{b}$ and therefore there exists some $\Theta$ such that $\varnothing, \Theta \vdash_{\text{state}} \tilde{Q}$ for each $\tilde{Q} \in \overline{Q}$ and $\varnothing, \Theta \vdash \tilde{E}[(\text{goto } q \overset{\upsilon_1}{\ldots} \overset{\upsilon_n}{\ldots})] : \bot$. By the Context lemma, there exist $\tau'$ and $x$ such that $\varnothing, \Theta \vdash (\text{goto } q \overset{\upsilon_1}{\ldots} \overset{\upsilon_n}{\ldots}) : \tau'$. By a straightforward induction on the derivation of that judgment, it is easy to see that $\tau' \vdash \bot$ and for all $i \in 1\ldots n$, $\varnothing, \Theta \vdash \tilde{v}_i : \tau_i$. Because values contain neither free variables nor spawn expressions, the environments are irrelevant, so we also have $\varnothing, \varnothing \vdash \tilde{v}_i : \tau_i$ for all $i \in 1\ldots n$. Then by corollary J.1.1 to the Substitution lemma, $\varnothing, \varnothing \vdash (\text{receive } \xi \rightarrow \xi \mathcal{H})[\xi_1 = \xi_1 \ldots [\xi_n = \xi_n] : \bot$. Therefore, $\tau, \varnothing \vdash_{\text{beh}} \tilde{b}'$, and so $\vdash_{\text{cfg}} \tilde{K}$.

**M-ReceiveInternal**

By the definition of this rule, there exist $a, \tilde{b}, \tilde{b}', \overline{Q}, x, \xi, \mathcal{H}', \overline{v},$ and $\overline{\upsilon}'$ such that

- $\overline{b}(a) = \tilde{b} = \langle \overline{Q}, (\text{receive } x \rightarrow \xi \mathcal{H}) \rangle$.
- $\langle a @ \mathcal{H}', \overline{v} \rangle \in \overline{\mu}$,
- $\langle \overline{\upsilon}', \mathcal{H}' \rangle \in \text{Markings}(\overline{v}, \mathcal{H})$ if $\mathcal{H}' \neq \varnothing$, else $\langle \overline{\upsilon}', \mathcal{H}' \rangle = \langle \overline{v}, \mathcal{H} \rangle$, and
- $\overline{b}'(a) = \tilde{b}' = \langle \overline{Q}, \xi[x \rightarrow \overline{\upsilon}'] \rangle$.

Let $\tau = \text{ActorType}(a)$. Because $\vdash_{\text{cfg}} \tilde{K}$, it must be the case that $\tau, \varnothing \vdash_{\text{beh}} \tilde{b}$ and therefore there exists some $\Theta$ such that $\varnothing, \Theta \vdash_{\text{state}} \tilde{Q}$ for each $\tilde{Q} \in \overline{Q}$ and $[\overline{x} \rightarrow \tau], \Theta \vdash \xi : \bot$. Also by the program configuration type-checking rules, $\varnothing, \Theta \vdash \overline{v} : \tau$, so by the Markings Type Preservation lemma, $\varnothing, \Theta \vdash \overline{v}' : \tau$. Then by the Substitution lemma, $\varnothing, \Theta \vdash \xi[x \rightarrow \overline{v}'] : \bot$, so $\tau, \varnothing \vdash_{\text{beh}} \tilde{b}'$.

The rest of the configuration did not change, so $\vdash_{\text{cfg}} \tilde{K}$.

**M-ReceiveExternal**

The proof is similar to the proof for M-ReceiveInternal. The only difference is that by the definition of this rule, the message $\overline{v}$ is received by a receptionist with a type $\tau'$. By the definition of this rule, $\varnothing, \varnothing \vdash \overline{v} : \tau'$. Because $\vdash_{\text{cfg}} \tilde{K}$, we also have that $\varnothing, \varnothing \vdash a : (\text{Addr } \tau')$. By the Type Inversion lemma, $\tau' \vdash \bot$, where $\text{ActorType}(a) = \tau$. Then by the subsumption rule, $\varnothing, \varnothing \vdash \overline{v} : \tau$. The rest of the proof is analogous to the one for M-ReceiveInternal.

**M-Timeout**

Similar to the proof for M-ReceiveInternal, except there is no message that gets substituted into the behavior expression.
M-SENDINTERNAL

By the definition of this rule, there exist \( a, \dot{b}, \dot{b}', \overline{Q}, \overline{E}, a', H'', \dot{v}, \) and \( \dot{v}' \) such that

- \( \dot{b}(a) = \dot{b} = \langle \overline{Q}, \overline{E}[\text{send } a'@H''] \rangle \),
- \( \langle \dot{v}', H' \rangle \in \text{Markings}(\dot{v}, H) \) if \( H'' \neq \emptyset \), else \( \langle \dot{v}', H' \rangle = \langle \dot{v}, H \rangle \),
- \( \dot{\mu}' = \dot{\mu} \cup \{ \langle a'@H'', \dot{v}' \rangle \} \), and
- \( \dot{b}'(a) = \dot{b}' = \langle \overline{Q}, \overline{E}[\text{variant Unit}] \rangle \).

Because \( \vdash_{\text{cfg}} \overline{K} \), it must be the case that \( r, \emptyset \vdash_{\text{beh}} \dot{b} \) and therefore there exists some \( \Theta \) such that \( \emptyset, \Theta \vdash_{\text{state}} \overline{Q} \) for each \( \overline{Q} \in \overline{Q} \) and \( \emptyset, \Theta \vdash \overline{E}[\text{send } a'@H''] : \bot \). By the Context lemma, there exist \( \tau' \) and \( x \) such that \( \emptyset, Q \vdash (\text{send } a'@H'') \vdash \tau' \), \( x \) is not free in \( \overline{E} \) or \( \dot{e} \), and \( \Gamma [x \rightarrow \tau'], \Theta \vdash \overline{E} [x] : \tau \). By a straightforward induction on the derivation of \( \emptyset, Q \vdash (\text{send } a'@H'') \vdash \tau' \), it is easy to see that there exists some \( \tau'' \) such that \( \tau' = (\text{Variant Unit}), \emptyset, Q \vdash a'@H'': (\text{Addr } \tau'') \), and \( \emptyset, Q \vdash \dot{v} : \tau'' \).

The variant typing rule immediately gives us that \( \emptyset, Q \vdash (\text{variant Unit}) : (\text{Variant Unit}) \), so by the Substitution lemma, \( \emptyset, \Theta \vdash \overline{E}[\text{variant Unit}] : \bot \), and therefore \( r, \emptyset \vdash_{\text{beh}} \dot{b}' \).

By the Markings Type Preservation lemma, \( \emptyset, Q \vdash \dot{v}' : \tau'' \). Because \( \dot{v}' \) contains no free variables or goto expressions, the environments are irrelevant, so we also have \( \emptyset, \emptyset \vdash \dot{v}' : \tau'' \). By the Type Inversion lemma, \( \tau'' <: \tau''' \), where \( \text{ActorType}(a) = \tau''' \). Then by the subsumption rule, \( \emptyset, \emptyset \vdash \dot{v}' : \tau''' \), which satisfies the rule for the message map. None of the rest of the configuration changed, so \( \vdash_{\text{cfg}} \overline{K}' \).

M-SENDEXTERNAL

The proof is similar to the previous case, except that rather than creating a new in-transit message, the transition sends the message to the environment and therefore creates new receptionists. We must prove that those new receptionists type-check.

By an argument similar to the one in the previous case, there exist \( a', H'', \dot{v}'' \), and \( \tau'' \) such that \( \emptyset, \emptyset \vdash a'@H'' : (\text{Addr } \tau'') \), and \( \emptyset, \emptyset \vdash \dot{v} : \tau'' \). Let \( \tau''' = \text{ActorType}(a') \). By the Type Inversion lemma, \( \tau''' <: \tau'' \), and by the subsumption rule, \( \emptyset, \emptyset \vdash \dot{v}' : \tau''' \).

By the definition of this rule, there exists \( \dot{\rho}'' = \text{IntAddrTypes}(\dot{v}'', \tau''') \) such that \( \dot{\rho}' = \dot{\rho} \cup \dot{\rho}'' \). Then by the \text{IntAddrTypes} Correctness lemma, for all \( \langle a''@H'''', \tau''' \rangle \in \dot{\rho}'' \), \( \emptyset, \emptyset \vdash a''@H'''' : (\text{Addr } \tau'''') \).

M-SPAWN

By the definition of this rule, there exist \( a, \dot{b}, \dot{b}', \overline{Q}, \overline{E}, \ell, \dot{r}, \dot{e}, \overline{Q}', a', \) and \( \dot{b}'' \) such that
• \( \tilde{\beta}(a) = \tilde{b} = \langle Q, \bar{E} \{ (\text{spawn}' \, τ' \, \bar{e} \, \bar{Q'}) \} \rangle \),

• \( \text{dom}(\tilde{\beta}) = \text{dom}(\tilde{\beta}) \cup \{a'\} \),

• \( a' \) is internal,

• \( \text{ActorType}(a') = \tau' \),

• \( \tilde{\beta}(a) = \tilde{b}' = \langle Q, \bar{E}a@\phi \rangle \), and

• \( \tilde{\beta}(a') = \tilde{b}'' = \langle Q[\text{self} \rightarrow a@\phi], \bar{e}[\text{self} \rightarrow a@\phi] \rangle \).

Because \( \vdash_{\text{cfg}} \bar{K} \), it must be the case that \( \tau, \emptyset \vdash_{\text{beh}} \tilde{b} \) where \( \tau = \text{ActorType}(a) \), and therefore there exists some \( \Theta \) such that \( \emptyset, \Theta \vdash_{\text{state}} \bar{Q} \) for each \( \bar{Q} \in Q \) and \( \emptyset, \Theta \vdash \bar{E} \{ (\text{spawn}' \, τ' \, \bar{e} \, \bar{Q'}) \} : \bot \). By the Context lemma, there exist \( τ'' \) and \( x \) such that \( \emptyset, \emptyset \vdash (\text{spawn}' \, τ' \, \bar{e} \, \bar{Q'}) : τ'' \), \( x \) is not free in \( \bar{E} \) or \( \bar{e} \), and \( [x \rightarrow τ''] \), \( \emptyset \vdash \bar{E} [x] : \bot \). By a straightforward induction on the derivation of that judgment, it is easy to see that there exist some \( \Gamma \) and \( \Theta' \) such that \( τ'' = (\text{Addr} \, τ') \), \( \Gamma = \{ \text{self} \rightarrow (\text{Addr} \, τ') \} \), \( \Gamma, \Theta' \vdash_{\text{state}} \bar{Q}' \) for each \( \bar{Q}' \in \bar{Q} \), and \( \Gamma, \Theta' \vdash \bar{e} : \bot \). By the Substitution lemma, \( \emptyset, \Theta' \vdash \bar{Q}'[\text{self} \rightarrow a@\phi] \) for each \( \bar{Q}' \in \bar{Q} \), and \( \emptyset, \Theta' \vdash e[\text{self} \rightarrow a@\phi] : \bot \). Therefore, \( τ', \emptyset \vdash_{\text{beh}} \tilde{b}' \).

For \( \tilde{b}' \), we have immediately by the type rule for addresses that \( \emptyset, \emptyset \vdash a'@\phi : (\text{Addr} \, τ') \). Therefore, by the Substitution lemma, \( \emptyset, \emptyset \vdash \bar{E} \{ a'@\phi \} : \bot \), and therefore \( τ, \emptyset \vdash_{\text{beh}} \tilde{b}' \).

No other parts of the configuration changed, so we have \( \vdash_{\text{cfg}} \bar{K}' \).

\section*{M-FUNC}

By the definition of this rule, there exist \( a, \tilde{b}, \tilde{b}', \bar{Q}, \bar{E}, \bar{e}, \) and \( \bar{e}' \) such that

• \( \tilde{\beta}(a) = \tilde{b} = \langle Q, \bar{E} \{ \bar{e} \} \rangle \),

• \( \bar{e} \rightarrow \bar{e}' \), and

• \( \tilde{\beta}(a) = \tilde{b}' = \langle Q, \bar{E} \{ \bar{e}' \} \rangle \).

Because \( \vdash_{\text{cfg}} \bar{K} \), it must be the case that \( \tau, \emptyset \vdash_{\text{beh}} \tilde{b} \) where \( \tau = \text{ActorType}(a) \), and therefore there exists some \( \Theta \) such that \( \emptyset, \Theta \vdash_{\text{state}} \bar{Q} \) for each \( \bar{Q} \in Q \) and \( \emptyset, \Theta \vdash \bar{E} \{ \bar{e} \} : \bot \). By the Context lemma, there exist \( τ' \) and \( x \) such that \( \emptyset, \emptyset \vdash \bar{e} : τ' \), \( x \) is not free in \( \bar{E} \) or \( \bar{e} \), and \( [x \rightarrow τ''] \), \( \emptyset \vdash \bar{E} [x] : \bot \). By the Functional Step Type Preservation lemma, \( \emptyset, \emptyset \vdash \bar{E} \{ \bar{e}' \} : \bot \). By the Substitution lemma, \( \emptyset, \emptyset \vdash \bar{E} \{ \bar{e}' \} : \bot \), so \( τ, \emptyset \vdash_{\text{beh}} \tilde{b}' \). No other changes were made to the configuration, so \( \vdash_{\text{cfg}} \bar{K}' \).

\( \square \)
J.9 Abstraction Type Preservation

Lemma (Abstraction Type Preservation). For all $\vec{K}$, if $\vdash_{cfg} \vec{K}$, then there exists $d$ such that $\vdash_{cfg} [\vec{K}] : d$.

Proof. By a straightforward structural induction on $\vec{K}$. The depth $d$ at each point is selected to be the maximum depth inferred for any given component. $\square$

J.10 Subtype Inversion Lemma 1

Lemma (Subtype Inversion 1). For all $\tau$ and $\tau'$ such that $\tau <: \tau'$, the following conditions hold.

- If $\tau' = \text{Nat}$ then $\tau = \text{Nat}$.
- If $\tau' = \text{String}$ then $\tau = \text{String}$.
- If $\tau' = (\text{Variant} \ [t_1 \overline{\tau'_1}] \ldots [t_n \overline{\tau'_n}])$, then $\tau = (\text{Variant} \ [t_1 \overline{\tau_1}] \ldots [t_n \overline{\tau_n}])$, where
  - $\{t'_1, \ldots, t'_m\} \subseteq \{t_1, \ldots, t_n\}$ and
  - for all $j \in 1 \ldots m$, there exists $i \in 1 \ldots n$ such that
    - $t_i = t'_j$,
    - $\overline{\tau_i} = \overline{\tau'_i}$, $\ldots$, $\overline{\tau'_{i,p'}}$,
    - $\overline{\tau_j} = \overline{\tau_{j,1}}$, $\ldots$, $\overline{\tau_{j,p}}$, and
    - $\tau_{j,k} <: \tau'_{i,k}$ for all $k \in 1 \ldots p$.
- If $\tau' = (\text{Record} \ [r_1 \overline{\tau'_1}] \ldots [r_n \overline{\tau'_n}])$, then $\tau = (\text{Record} \ [r_1 \overline{\tau_1}] \ldots [r_n \overline{\tau_n}])$ and $\tau_i <: \tau'_i$ for all $i \in 1 \ldots n$.
- If $\tau' = (\text{Addr} \overline{\tau''})$, then $\tau = (\text{Addr} \overline{\tau''})$ and $\tau'' <: \overline{\tau''}$.
- If $\tau' = (\text{rec} X \overline{\tau''})$, then $\tau = (\text{rec} X' \overline{\tau''})$.
- If $\tau' = (\text{List} \overline{\tau''})$, then $\tau = (\text{List} \overline{\tau''})$ and $\overline{\tau''} <: \overline{\tau''}$.
- If $\tau' = (\text{Dict} \overline{\tau'_1 \overline{\tau'_2}})$, then $\tau = (\text{Dict} \overline{\tau_1 \overline{\tau_2}})$, $\tau_1 <: \tau'_1$, and $\tau_2 <: \tau'_2$.

Proof. Straightforward structural induction on the derivation of $\tau <: \tau'$.$\square$

J.11 Subtype Inversion Lemma 2

This lemma is the dual to the previous one, in that it requires we know the shape of the smaller type $\tau$, rather than the bigger one $\tau'$.

Lemma (Subtype Inversion 2). For all $\tau$ and $\tau'$ such that $\tau <: \tau'$, the following conditions hold.
If \( \tau = \text{Nat} \) then \( \tau' = \text{Nat} \).

If \( \tau = \text{String} \) then \( \tau' = \text{String} \).

If \( \tau = (\text{Variant} \ [t'_1 \overline{t}_1] \ldots [t'_m \overline{t}_m]) \),

then \( \tau' = (\text{Variant} \ [t_1 \overline{t}_1] \ldots [t_n \overline{t}_n]) \), where

- \( \{t'_1, \ldots, t'_m\} \subseteq \{t_1, \ldots, t_n\} \) and
- for all \( j \in 1 \ldots m \), there exists \( i \in 1 \ldots n \) such that
  - \( t_i = t'_j \),
  - \( \overline{t}_i = \tau'_{i,1}, \ldots, \tau'_{i,p} \),
  - \( \tau_{j,1}, \ldots, \tau_{j,p}, \) and
  - \( \tau_{j,k} <: \tau'_{i,k} \) for all \( k \in 1 \ldots p \).

If \( \tau = (\text{Record} \ [r_1 \tau_1] \ldots [r_n \tau_n]) \) then \( \tau' = (\text{Record} \ [r_1 \tau'_1] \ldots [r_n \tau'_n]) \),

and \( \tau_i <: \tau'_i \) for all \( i \in 1 \ldots n \).

If \( \tau = (\text{Addr} \ \tau'') \), then \( \tau' = (\text{Addr} \ \tau''') \) and \( \tau''' <: \tau'' \).

If \( \tau = (\text{rec} X \ \tau'''') \), then \( \tau' = (\text{rec} X \ \tau''') \).

If \( \tau = (\text{List} \ \tau''''') \), then \( \tau' = (\text{List} \ \tau''') \) and \( \tau'''' <: \tau''' \).

If \( \tau = (\text{Dict} \ \tau_1 \ \tau_2) \), then \( \tau' = (\text{Dict} \ \tau'_1 \ \tau'_2) \), \( \tau_1 <: \tau'_1 \), and \( \tau_2 <: \tau'_2 \).

Proof: Straightforward structural induction on the derivation of \( \tau <: \tau' \). \( \square \)

### J.12 Abstract Canonical Forms Lemma

**Lemma** (Abstract Canonical Forms). For all \( \bar{v} \) and \( \tau \) such that \( \phi, \phi \vdash \bar{v} : \tau \), the following conditions hold.

- If \( \tau = \text{Nat} \), then \( \bar{v} = \text{abs-nat} \).
- If \( \tau = \text{String} \), then \( \bar{v} = \text{abs-string} \).
- If \( \tau = (\text{Variant} \ [t_1 \tau_{1,1} \ldots \tau'_{1,m}] \ldots [t_n \tau_{n,1} \ldots \tau'_{n,m}]) \), then there exists \( i \in 1 \ldots n \) and \( \bar{v}'_1, \ldots, \bar{v}'_m \) such that \( \bar{v} = (\text{variant} \ t_i \bar{v}'_1 \ldots \bar{v}'_m) \).
- If \( \tau = (\text{Record} \ [r_1 \tau'_1] \ldots [r_n \tau'_n]) \), then there exist \( \bar{v}'_1, \ldots, \bar{v}'_n \) such that \( \bar{v} = (\text{record} \ [r_1 \bar{v}'_1] \ldots [r_n \bar{v}'_n]) \).
- If \( \tau = (\text{Addr} \ \tau') \), then there exist \( \bar{a} \) such that \( \bar{v} = \bar{a} \).
- If \( \tau = (\text{rec} X \ \tau') \), then there exist \( X', \ \tau'' \), and \( \bar{v}' \) such that \( \bar{v} = (\text{fold} \ (\text{rec} X' \ \tau'') \ \bar{v}') \).
- If \( \tau = (\text{List} \ \tau') \), then there exist \( \bar{v}'_1, \ldots, \bar{v}'_n \) such that \( \bar{v} = (\text{list} \ \{\bar{v}'_1, \ldots, \bar{v}'_n\}) \).
J.13. ABSTRACT TYPE INVERSION LEMMA

Lemma (Abstract Type Inversion). For all \( \hat{v} \) and \( \tau \) such that \( \phi, \emptyset \vdash \hat{v} : \tau \), the following conditions hold.

1. If \( \hat{v} = (\text{variant } t \ \hat{v}_1' \ldots \hat{v}_m') \), then \( \tau = (\text{Variant } \{t_1 \ \tau_{1,1} \ldots \tau_{1,m} \} \ldots \{t_n \ \tau_{n,1} \ldots \tau_{n,m} \}) \) and there exists \( i \in 1 \ldots n \) such that \( t = t_i \) and \( \phi, \emptyset \vdash \hat{v}_i' : \tau_{i,j} \) for all \( j \in 1 \ldots m \).

2. If \( \hat{v} = (\text{record } [r_1 \ \hat{v}_1'] \ldots [r_n \ \hat{v}_n']) \), then \( \tau = (\text{Record } [r_1 \ \tau_1'] \ldots [r_n \ \tau_n']) \) and \( \phi, \emptyset \vdash \hat{v}_i' : \tau_{i,j} \) for all \( j \in 1 \ldots n \).

3. If \( \hat{v} = \hat{a} \), then \( \tau = (\text{Addr } \tau') \) and \( \tau' <:\text{ActorType}(\hat{a}) \).

4. If \( \hat{v} = (\text{fold } (\text{rec } X \ \tau') \ \hat{v}') \), then \( \phi, \emptyset \vdash \hat{v}' : \tau'[X \rightarrow (\text{rec } X \ \tau')] \) and \( (\text{rec } X \ \tau') <:\tau \).

5. If \( \hat{v} = (\text{list } \{\hat{v}_1' \ldots \hat{v}_n'\}) \), then \( \tau = (\text{List } \tau') \) and \( \phi, \emptyset \vdash \hat{v}_i' : \tau' \) for all \( i \in 1 \ldots n \).

6. If \( \hat{v} = (\text{dict } \{\hat{v}_1' \ldots \hat{v}_n'\} \ \{\hat{v}_1'' \ldots \hat{v}_m''\}) \), then \( \tau = (\text{Dict } \tau' \ \tau'') \) and \( \phi, \emptyset \vdash \hat{v}_i' : \tau' \) for all \( i \in 1 \ldots n \) and \( \phi, \emptyset \vdash \hat{v}_j'' : \tau'' \) for all \( j \in 1 \ldots m \).

Proof. By induction on the derivation of \( \phi, \emptyset \vdash \hat{v} : \tau \). The proof for the list condition is given; others are similar.

---

- If \( \tau = (\text{Dict } \tau' \ \tau'') \), then there exist \( \hat{v}_1' \ldots \hat{v}_n' \) and \( \hat{v}_1'' \ldots \hat{v}_m'' \) such that \( \hat{v} = (\text{dict } \{\hat{v}_1', \ldots, \hat{v}_n'\} \ \{\hat{v}_1'', \ldots, \hat{v}_m''\}) \).

Proof. By induction on the derivation of \( \phi, \emptyset \vdash \hat{v} : \tau \). The case for List types is given; others are similar.

Case: \( \tau = (\text{List } \tau') \)

The last rule used in the derivation of \( \phi, \emptyset \vdash \hat{v} : \tau \) is either the List rule, the type-subsumption rule, or the depth subsumption rule.

- If it is the List rule, the proof is immediate by the definition of that rule.

- In the type-subsumption case there exists some \( \tau'' \) such that \( \tau'' <:\tau \) and \( \phi, \emptyset \vdash \hat{v} : \tau'' \). By the lemma Subtype Inversion 1, \( \tau'' = (\text{List } \tau'''') \) for some \( \tau'''' \). By the induction hypothesis, there exist \( \hat{v}_1' \ldots, \hat{v}_n' \) such that \( \hat{v} = (\text{list } \{\hat{v}_1', \ldots, \hat{v}_n'\}) \).

- In the depth-subsumption case, again by the induction hypothesis there exist \( \hat{v}_1' \ldots, \hat{v}_n' \) such that \( \hat{v} = (\text{list } \{\hat{v}_1', \ldots, \hat{v}_n'\}) \).

\( \square \)
By the Abstract Type Inversion lemma, for lists, 

By the Abstract Canonical Forms lemma, 

Case: \( \tau = (\text{List} \; \tau') \)

By the Abstract Canonical Forms lemma, \( \hat{\nu} \) is of the form \( (\text{list} \; \{\hat{\nu}_1', \ldots, \hat{\nu}_n'\}) \).

Case: \( \tau = (\text{rec} \; X \; (\text{Addr} \; \tau')) \)

By the syntactic restriction on \text{rec} \ types in CSA, \( \tau''' = (\text{Addr} \; \tau''') \) for some \( \tau''' \), and therefore \( \tau''|(X' \leftarrow \tau') = (\text{Addr} \; \tau'''|(X' \leftarrow \tau')) \). Then by the Abstract Canonical Forms lemma again, \( \hat{\nu}' = \hat{a} \) for some \( \hat{a} \). By the Abstract Type Inversion lemma, \( \tau'''|(X' \leftarrow \tau') \prec \text{ActorType}(\hat{a}) \).

Let \( \tau''' = \text{ActorType}(\hat{a}) \). By the type rule for addresses, \( \varnothing, \varnothing \vdash \hat{a} : (\text{Addr} \; \tau'''), 1 \). By the \text{Addr} rule for subtypes, \( (\text{Addr} \; \tau''') \prec (\text{Addr} \; \tau'') \), and therefore by the type-subsumption
J.15 Merge Type Preservation Lemma

Lemma (Merge Type Preservation). For all \( \tilde{\mu}, \tilde{\alpha}, H, \) and \( \tilde{v} \) if

\[
\text{there exists } \tau \text{ such that } \emptyset, \emptyset \vdash \tilde{\alpha}@H : (\text{Addr } \tau), d \text{ and } \emptyset, \emptyset \vdash \tilde{v} : \tau, d,
\]

- for all \( \langle \tilde{\alpha}@H', \tilde{v}' \rangle \in \text{dom}(\tilde{\mu}) \), there exists \( \tau' \) such that \( \emptyset, \emptyset \vdash \tilde{\alpha}'@H' : (\text{Addr } \tau'), d \) and \( \emptyset, \emptyset \vdash \tilde{v} : \tau', d \),

then for all \( \langle \tilde{\alpha}'@H'', \tilde{v}'' \rangle \in \text{dom}(\tilde{\mu} \oplus (\tilde{\alpha}@H, \tilde{v})) \), there exists \( \tau'' \) such that \( \emptyset, \emptyset \vdash \tilde{\alpha}''@H'' : (\text{Addr } \tau''), d \) and \( \emptyset, \emptyset \vdash \tilde{v} : \tau'', d \).

Proof. By the definition of \( \oplus \), there exist \( \tilde{\mu}', \tilde{\mu}'', \tilde{v}', \tilde{v}'', \ldots, \tilde{v}_n \), and \( m \) such that \( \tilde{\mu} = \tilde{\mu}' \uplus \tilde{\mu}'' \), \( \tilde{\mu} \oplus (\tilde{\alpha}@H, \tilde{v}) = \tilde{\mu}' \uplus \tilde{\mu}'' \uplus \langle \tilde{\alpha}@H, \text{Merge}(\tilde{v}', \tilde{v}'', \ldots, \tilde{v}_n) \rangle \rightarrow m \), and \( \langle \tilde{\alpha}@H, \tilde{v} \rangle \in \text{dom}(\tilde{\mu}'') \) for all \( i = 1 \ldots n \). For all \( \langle \tilde{\alpha}'@H', \tilde{v}' \rangle \in \text{dom}(\tilde{\mu}) \), we already know that there exists \( \tau' \) such that \( \emptyset, \emptyset \vdash \tilde{\alpha}'@H' : (\text{Addr } \tau'), d \) and \( \emptyset, \emptyset \vdash \tilde{v} : \tau', d \).

It remains to show there exists some \( \tau'' \) such that

\[
\emptyset, \emptyset \vdash \tilde{\alpha}''@H'' : (\text{Addr } \tau''), d \text{ and } \emptyset, \emptyset \vdash \tilde{v} : \tau'', d.
\]

We know that

- there exists \( \tau \) such that \( \emptyset, \emptyset \vdash \tilde{\alpha}@H : (\text{Addr } \tau), d \) and \( \emptyset, \emptyset \vdash \tilde{v} : \tau, d, \) and

- for all \( i = 1 \ldots n \), there exists \( \tau_i \) such that \( \emptyset, \emptyset \vdash \tilde{\alpha}@H : (\text{Addr } \tau_i), d \) and \( \emptyset, \emptyset \vdash \tilde{v} : \tau_i, d \).

Let \( \tau' = \text{ActorType}(\tilde{\alpha}) \). By the Abstract Type Inversion lemma, \( \tau <: \tau' \) and \( \tau_i <: \tau' \) for all \( i = 1 \ldots n \). Therefore by the type subsumption rule, \( \emptyset, \emptyset \vdash \tilde{v} : \tau', d \) and \( \emptyset, \emptyset \vdash \tilde{v}_i : \tau_i, d \) for all \( i = 1 \ldots n \). Then by induction on \( n \) and the Merge Type Preservation lemma, \( \emptyset, \emptyset \vdash \text{Merge}(\tilde{v}, \tilde{v}', \ldots, \tilde{v}_n) : \tau', d \). Because \( \tau' = \text{ActorType}(\tilde{\alpha}) \), we also have \( \emptyset, \emptyset \vdash \tilde{\alpha} : (\text{Addr } \tau'), d \), which completes the proof. \( \square \)
J.17 Abstract Typed Substitution Lemma

Lemma (Abstract Typed Substitution). For all $\Gamma$, $x$, $\tau$, $\Theta$, $\hat{\epsilon}$, $\hat{Q}$, $\hat{tc}$, $\hat{\epsilon}'$, $d$, and $d'$, if $\Gamma, \emptyset \vdash \hat{\epsilon} : \tau', d'$ and $\text{Depth}(\tau') = d'$, then

1. if $\Gamma[x \leadsto \tau'], \Theta \vdash \hat{\epsilon} : \tau, d$, then $\Gamma, \Theta \vdash \hat{\epsilon}[x \mapsto \hat{\epsilon}'] : \tau, d$.
2. if $\Gamma[x \leadsto \tau'], \Theta \vdash_{\text{state}} Q : d$, then $\Gamma, \Theta \vdash_{\text{state}} Q[x \mapsto \hat{\epsilon}'] : d, d'$.
3. if $\Gamma[x \leadsto \tau'], \Theta \vdash_{\text{tc}} \hat{tc} : d$, then $\Gamma, \Theta \vdash_{\text{tc}} \hat{tc}[x \mapsto \hat{\epsilon}'] : d$.

Proof. The proof is by induction on the derivation of $\Gamma[x \leadsto \tau'], \Theta \vdash \hat{\epsilon} : \tau, d$, $\Gamma[x \leadsto \tau'], \Theta \vdash_{\text{state}} Q : d$, or $\Gamma[x \leadsto \tau'], \Theta \vdash_{\text{tc}} \hat{tc} : d$ as appropriate. Most cases are straightforward.

In the case of the variable typing rule, $\hat{\epsilon} = x'$ for some $x'$. If $x \neq x'$, then $\hat{\epsilon}[x \mapsto \hat{\epsilon}'] = x'$, so we have $\Gamma, \Theta \vdash \hat{\epsilon}[x \mapsto \hat{\epsilon}'] : \tau, d$. Otherwise, $\hat{\epsilon}[x \mapsto \hat{\epsilon}'] = \hat{\epsilon}'$. Then by the definition of this rule, $\tau = \tau'$, and $d = \text{Depth}(\tau) = d'$. Then we know that $\Gamma, \emptyset \vdash \hat{\epsilon} : \tau', d'$, which completes the proof. \hfill \square

Corollary J.17.1. For all $\Gamma$, $x_1, \ldots, x_n$, $\tau$, $\tau_1', \ldots, \tau_n'$, $\Theta$, $\hat{\epsilon}$, $\hat{Q}$, $\hat{tc}$, $\hat{\epsilon}_1', \ldots, \hat{\epsilon}_n'$, and $d_1, \ldots, d_n$, if $\Gamma, \emptyset \vdash \hat{\epsilon}_i : \tau_i', d_i'$ and $\text{Depth}(\tau_i') = d_i'$, then

1. if $\Gamma[x_1 \leadsto \tau_1', \ldots, x_n \leadsto \tau_n'], \Theta \vdash \hat{\epsilon} : \tau, d$, then $\Gamma, \Theta \vdash \hat{\epsilon}[x_1 \mapsto \hat{\epsilon}_1'] \ldots [x_n \mapsto \hat{\epsilon}_n'] : \tau, d$.
2. if $\Gamma[x_1 \leadsto \tau_1', \ldots, x_n \leadsto \tau_n'], \Theta \vdash_{\text{state}} Q : d$, then $\Gamma, \Theta \vdash_{\text{state}} \hat{\epsilon}[x_1 \mapsto \hat{\epsilon}_1'] \ldots [x_n \mapsto \hat{\epsilon}_n'] : d, d'$.
3. if $\Gamma[x_1 \leadsto \tau_1', \ldots, x_n \leadsto \tau_n'], \Theta \vdash_{\text{tc}} \hat{tc} : d$, then $\Gamma, \Theta \vdash_{\text{tc}} \hat{tc}[x_1 \mapsto \hat{\epsilon}_1'] \ldots [x_n \mapsto \hat{\epsilon}_n'] : d$.

Proof. By a straightforward induction on $n$. \hfill \square

J.18 IntAddrTypes Depth Lemma

Lemma (IntAddrTypes Depth). For all $\tau$ and $\tau'$, if $\emptyset, \emptyset \vdash \hat{\nu} : \tau$, then for all $\langle \hat{a}@H, \tau' \rangle \in \text{IntAddrTypes}(\hat{\nu}, \tau)$, $\text{Depth}(\tau') \leq \text{Depth}(\tau) \emptyset, \emptyset \vdash \hat{a}@H : (\text{Addr } \tau'), d$.

Proof. By induction on the derivation of $\emptyset, \emptyset \vdash \hat{\nu} : \tau$, following the definition of IntAddrTypes. The interesting cases are the ones for a fold expression and for an address, given below.

$\hat{\nu} = (\text{fold } \tau \hat{\nu}')$, $\tau = (\text{rec } X \tau')$

In this case, $\text{IntAddrTypes}(\hat{\nu}, \tau) = \text{IntAddrTypes}(\hat{\nu}', \tau'[X \leadsto \tau])$. That corresponds to the typing rule for fold expressions, so by the induction hypothesis, for all $\langle \hat{a}@H, \tau' \rangle \in \text{IntAddrTypes}(\hat{\nu}', \tau'[X \leadsto \tau])$, $\text{Depth}(\tau') \leq \text{Depth}(\tau'[X \leadsto \tau])$. By the definition of Reachable, $\tau'[X \leadsto \tau] \in \text{Reachable}(\tau)$, so by the definition of Depth, $\text{Depth}(\tau'[X \leadsto \tau]) \leq \text{Depth}(\tau)$. Therefore, for all $\langle \hat{a}@H, \tau' \rangle \in \text{IntAddrTypes}(\hat{\nu}', \tau'[X \leadsto \tau])$, $\text{Depth}(\tau') \leq \text{Depth}(\tau)$. 

\hfill \square
\( \hat{\nu} = \hat{a} @ H, \tau = (kw\text{Addr } r') \)

In this case, \( \text{IntAddrTypes}(\hat{\nu}, \tau) = \{ (\hat{a} @ H, r') \} \). By the typing rule for addresses, \( \phi, \tau \vdash \hat{a} @ H : (\text{Addr } r'), d \). By the definition of \( \text{Depth} \), \( \text{Depth}(r') \leq \text{Depth}(\tau) \), so we’re done.

\[ \square \]

### J.19 Abstract Functional Step Type Preservation Lemma

**Lemma** (Functional Step Type Preservation). For all \( \Theta, \hat{e}, \hat{e}', \) and \( \tau \), if \( \phi, \Theta \vdash \hat{e} : \tau, d \) and \( \hat{e} \rightarrow \hat{e}' \), then \( \phi, \Theta \vdash \hat{e}' : \tau \).

**Proof.** The proof is by case analysis on the rule enabling the transition \( \hat{e} \rightarrow \hat{e}' \). The proof is largely similar to the one for the Functional Step Type Preservation lemma.

#### Case 1

In this case, \( \hat{e} = (\text{begin } \hat{\nu}) \) and \( \hat{e}' = \hat{\nu} \). By a straightforward induction on the derivation of \( \phi, \Theta \vdash \hat{\nu} : \tau, d \), we have \( \phi, \nu \vdash \tau : d - 1 \). Then by the depth-subsumption rule, \( \phi, \nu \vdash \tau : d \).

#### Case 2

In this case, \( \hat{e} = (\text{begin } \nu \hat{e}_1'' \ldots \hat{e}_n'' \text{ variant } t) \) and \( \hat{e}' = (\text{begin } \hat{e}_1'' \ldots \hat{e}_n'') \). By a straightforward induction on the derivation of \( \phi, \Theta \vdash \hat{e}_i : \tau_d \), there exist \( \tau_{d-1}, \ldots, \tau_1' \) such that \( \phi, \Theta \vdash \hat{e}_i : \tau_i', d - 1 \) for all \( i \in 1 \ldots n \) and \( \tau_n' = \tau \). Therefore, by the type-checking rule for \( \text{begin} \), \( \phi, \Theta \vdash \hat{e} : \tau, d \).

#### Case 3

In this case, \( \hat{e} = (\text{record } \{ r \hat{v} \}[r \hat{v}][r' \hat{v}']) \) and \( \hat{e}' = \hat{\nu} \). By a straightforward induction on the derivation of \( \phi, \Theta \vdash \hat{\nu} : \tau, d \), we have \( \phi, \Theta \vdash \hat{\nu} : \tau, d - 2 \). Then by the depth-subsumption rule, \( \phi, \Theta \vdash \hat{\nu} : \tau, d \).

#### Case 4

In this case, \( \hat{e} = (\text{case } \text{variant } t \hat{v}_1 \ldots \hat{v}_n)[ (t x_1 \ldots x_n) \hat{e}'' \text{ variant } t] \) and \( \hat{e}' = \hat{e}''|_{x_1 \rightarrow \hat{v}_1} | \ldots |_{x_n \rightarrow \hat{v}_n} \). By a straightforward induction on the derivation of \( \phi, \Theta \vdash \hat{e} : \tau, d \), there exist \( \tau_{d-1}, \ldots, \tau_1' \) such that \( \phi, \Theta \vdash \hat{v}_i : \tau_i' \) for all \( i \in 1 \ldots n \) and \( \phi, \Theta \vdash \hat{e}'' |_{x_1 \rightarrow \hat{v}_1} | \ldots |_{x_n \rightarrow \hat{v}_n} : \tau, d - 1 \). By the Typeled Value Depth lemma, there exist \( d_1', \ldots, d_n' \) such that \( \text{Depth}(\tau_i') = d_i' \) and \( \phi, \Theta \vdash \hat{v}_i : \tau_i', d_i' \) for all \( i \in 1 \ldots n \). By corollary J.17.1 to the Abstract Typed Substitution lemma, \( \phi, \Theta \vdash \hat{e}'' |_{x_1 \rightarrow \hat{v}_1} | \ldots |_{x_n \rightarrow \hat{v}_n} : \tau, d - 1 \). Then by the depth subsumption rule, \( \phi, \Theta \vdash \hat{e}'' |_{x_1 \rightarrow \hat{v}_1} | \ldots |_{x_n \rightarrow \hat{v}_n} : \tau, d \).
APPENDIX J. TYPE PRESERVATION PROOFS

Case 5

In this case, \( \hat{e} = (\text{unfold } \tau' (\text{fold } \tau'\hat{v})) \) and \( \hat{e}' = \hat{v} \). By a straightforward induction on the derivation of \( \varnothing, \Theta \vdash \hat{e} : \tau, d \), \( \varnothing, \Theta \vdash \hat{v} : \tau, d - 2 \), and by the depth-subsumption rule, \( \varnothing, \Theta \vdash \hat{v} : \tau, d \).

Case 6

In this case, \( \hat{e} = (o \hat{v}_1 \cdots \hat{v}_n) \) and \( \hat{e}' = \text{EvalAbsPrimop}(o, \hat{v}_1 \cdots \hat{v}_n) \). By the definition of \( \text{EvalAbsPrimop} \), \( \varnothing, \Theta \vdash \hat{e}' : \tau, d \).

Case 7

In this case, \( \hat{e} = (\text{for/fold } [x \hat{v}] [x' (\text{list } \{\hat{v}'_1, \ldots, \hat{v}'_n\}) \hat{e}']) \) and \( \hat{e}' = \hat{v} \). By a straightforward induction on the derivation of \( \varnothing, \Theta \vdash \hat{e} : \tau, d \), \( \varnothing, \Theta \vdash \hat{v} : \tau, d - 1 \), and by the depth-subsumption rule, \( \varnothing, \Theta \vdash \hat{v} : \tau, d \).

Case 8

In this case, \( \hat{e} = (\text{for/fold } [x \hat{v}] [x' (\text{list } \{\hat{v}'_1, \ldots, \hat{v}'_n\}) \hat{e}']) \) and there exists some \( k \in 1 \ldots n \) such that \( \hat{e}' = (\text{for/fold } [x \hat{v}'^m] [x' (\text{list } \{\hat{v}'_1, \ldots, \hat{v}'_n\}) \hat{e}'^m]) \) where \( \hat{e}'^m = \hat{e}[x \leftarrow \hat{v}]x' \leftarrow \hat{v}'_k \). By a straightforward induction on the derivation of \( \varnothing, \Theta \vdash \hat{e} : \tau, d \), there exists \( \tau' \) such that \( \varnothing, \Theta \vdash \hat{e} : \tau, d - 1 \), \( \varnothing, \Theta \vdash \hat{v}'_i : \tau', d - 2 \) for all \( i \in 1 \ldots n \), \( \varnothing, \Theta \vdash (\text{list } \{\hat{v}'_1, \hat{v}'_2, \ldots, \hat{v}'_n\}) : (\text{List } \tau'), d - 1 \), and \( [x \leftarrow \tau, x' \leftarrow \tau'], \Theta \vdash \hat{e}'^m : \tau, d - 1 \).

By the Typed Value Depth lemma, there exists \( \hat{d}' \) such that \( \text{Depth}(\hat{d}') = \hat{d}' \) and \( \varnothing, \Theta \vdash \hat{v}'_i : \tau', \hat{d}' \) for all \( i \in 1 \ldots n \). By corollary J.17.1 to the Substitution lemma, \( \varnothing, \Theta \vdash \hat{d}'^m : \tau, d - 1 \). Then by the type-checking rule for \( \text{for/fold} \), we have \( \varnothing, \Theta \vdash (\text{for/fold } [x \hat{v}'^m] [x' (\text{list } \{\hat{v}'_1, \ldots, \hat{v}'_n\}) \hat{e}'^m]) : \tau, d \), which completes the proof.

\square

J.20 Abstract Type Preservation Lemma

Lemma (Abstract Type Preservation). For all \( \hat{K}, \hat{K}', d, \) and \( \hat{l} \), if \( \vdash_{cfg} \hat{K} : d \) and \( \hat{K} \xrightarrow{\hat{l}}_{RA} \hat{K}' \), then \( \vdash_{cfg} \hat{K}' : d \).

Proof. In the following, let \( \langle \{ \beta \mid \hat{\mu} \mid H \} \rangle^{\hat{\sigma}} = \hat{K} \) and \( \langle \{ \beta' \mid \hat{\mu}' \mid H' \} \rangle^{\hat{\sigma}} = \hat{K}' \). The proof is similar to the proof of the Type Preservation lemma. The proof of the Message Addition Type Preservation lemma shows how well-typedness is preserved when messages are added to \( \hat{\mu} \). The rest of this proof focuses on showing how the depth part of the type judgment is preserved, proceeding by cases on the rule enabling the transition \( \hat{K} \xrightarrow{\hat{l}}_{RA} \hat{K}' \).
A-GOTO
In this case, \( \hat{b} = \langle \hat{Q}, E[\{\text{goto } q \hat{v}_1 \ldots \hat{v}_n\}] \rangle \) is a behavior for the actor at \( \hat{a} \) in \( \hat{K} \), and this transition replaces that behavior with \( \hat{b}' = \langle \hat{Q}, (\text{receive} x' \hat{e} \hat{tc})[x_1 \leftarrow \hat{v}_1] \ldots [x_n \leftarrow \hat{v}_n] \rangle \), where \((\text{define-state} (q [x_1 \tau_1] \ldots [x_n \tau_n]) x' \hat{e} \hat{tc})\) is in \( \hat{Q} \).

By the type-checking rule for goto expressions, there exists some \( \Theta \) such that \( \hat{v}_i : \tau_i \) for all \( i \in 1 \ldots n \). By the Typed Value Depth lemma, \( \hat{e} \text{ type-checks at type } \tau \). Therefore by the Abstract Typed Substitution lemma, the depth of the body expression \( \hat{e} \) used to create the new behavior is preserved, so \( \vdash_{\text{cfg}} K' : d \).

A-RECEIVEINTERNAL
Let \( \hat{a} \) be the address of the actor that receives a message. In this case, the relevant behavior of that actor is of the form \( \langle \hat{Q}, (\text{receive} x \hat{e} \hat{tc}) \rangle \). By the type-checking rules, we know that \( \hat{e} \) type-checks in an environment where \( x \) has type \( \tau \), where \( \tau = \text{ActorType}(\hat{a}) \). By the type-checking rules for messages, we also have that the message \( \hat{v} \) type-checks at type \( \tau \). Therefore by the Typed Value Depth lemma, that value type-checks with a depth bounded by \( \text{Depth}(\tau) \). Therefore by the Abstract Typed Substitution lemma, \( \vdash_{\text{cfg}} K' : d \).

A-RECEIVEEXTERNAL
As in the previous case, the behavior expression \( \hat{e} \) type-checks in an environment where \( x \) has type \( \tau \), where \( \tau = \text{ActorType}(\hat{a}) \). By the definition of this rule, the message \( \hat{v} \) received from the environment must have type \( \tau \). By the Typed Value Depth lemma, that value type-checks with a depth bounded by \( \text{Depth}(\tau) \). Therefore by the Abstract Typed Substitution lemma, \( \vdash_{\text{cfg}} K' : d \).

A-TIMEOUT
This rule only changes the transitioned actor's behavior from \( \langle \hat{Q}, (\text{receive} \hat{e} [\text{timeout } \hat{v} \hat{e}']) \rangle \) to \( \langle \hat{Q}, \hat{e}' \rangle \), so the depth is preserved, and \( \vdash_{\text{cfg}} K' : d \).

A-SENDINTERNAL
By an argument similar to the one for the M-SENDINTERNAL case in the proof of the Type Preservation lemma, there exists some \( \tau \) such that the marked address \( \hat{a}'@H'' \) has type \( \text{Addr } \tau' \) and the message \( \hat{v} \) to be sent has type \( \tau \). By the Abstract Type Inversion lemma, \( \tau < \tau' \), where \( \text{ActorType}(\hat{a}') = \tau' \). By the subsumption rule, \( \hat{v} \) has type \( \tau' \). By the type-checking rule for configurations, no
address appearing in $\hat{K}$ can have an $ActorType$ with depth greater than $d$, and so by the Typed Value Depth lemma, the depth of the message $\hat{v}$ is bounded by $d$.

The result of the $send$ expression is (variant $Unit$), which has depth 2. The $send$ expression must have a depth of at least 2, because the address must have a depth of at least 1, and the $send$ expression itself adds 1 to that, so the result also preserves the depth. Therefore, $\vdash_{cfg} \hat{K}': d$.

**A-SendExternal**

Rather than creating a new internal message, this transition creates new receptionists. The $IntAddrTypes$ Depth lemma tells us that the depths of the types for those receptionists can be no greater than the depth of the destination address for this $send$, so those receptionists have depth less than or equal to $d$. The argument for the result of the $send$ call is the same in this case as in the previous one.

**A-Spawn**

A $spawn$ expression reduces to an address of the type $\tau$ given in the $spawn$ expression. By the type rule for $spawn$ expressions, the $spawn$ expression type-checks at a depth of $d + 1$, where $\text{Depth}(\tau) = d$. Therefore the new address has a $Depth$ bounded by $d$. By the typing rule for addresses, the resulting address type-checks with depth 1, and we know that a $spawn$ expression has a depth of at least 1, so the depth is preserved there.

The new actor's behavior is made up of the state definitions $\hat{Q}'$ and behavior expression $\hat{e}$ from the $spawn$ expression, with the new address $\hat{a}'$ substituted in. Because those components are part of the original $spawn$ expression, they must type-check with some depth less than or equal to $d$. An address type-checks with a depth of 1, which is less than or equal to the depth of the self variable that it replaces. Therefore, the new behavior overall type-checks with a depth less than or equal to $d$. Therefore, $\vdash_{cfg} \hat{K}': d$.

**A-Func**

In this case, there exists some $\hat{a}$, $\hat{b}$, $\hat{b}'$, $\hat{Q}$, $\hat{E}$, $\hat{e}$, and $\hat{e}'$ such that

- the behavior of the actor at $\hat{a}$ in $\hat{K}$ is $\langle \hat{Q}, \hat{E}[\hat{e}] \rangle$,

- $\hat{e} \rightarrow \hat{e}'$, and

- the behavior of the actor at $\hat{a}$ in $\hat{K}$ is $\langle \hat{Q}, \hat{E}[\hat{e}'] \rangle$.

Because $\vdash_{cfg} \hat{K}: d$, it must be the case that there exist some $\Theta$, $\tau$, and $d'$ such that $\varphi, \Theta \vdash \hat{e} : \tau, d'$. By the Functional Step $Type$ Preservation lemma, $\varphi, \Theta \vdash \hat{e}' : \tau, d'$. The rest of the configuration remains the same, so we have $\vdash_{cfg} \hat{K}': d$. \qed
Corollary J.20.1. For all $\hat{K}$, $\hat{K}'$, $d$, and $L$, if $\vdash_{\text{cfg}} \hat{K} : d$ and $\hat{K} \xrightarrow{L} \hat{K}'$, then $\vdash_{\text{cfg}} \hat{K}' : d$.

Proof. By the definition of the summary transition relation, there exist $\hat{l}_1, \ldots, \hat{l}_n$ such that $\hat{K} \xrightarrow{\hat{l}_1, \ldots, \hat{l}_n}$. By the definition of the abstract event-step relation, there exist $\hat{K}'^1, \ldots, \hat{K}'^{n+1}$ such that $\hat{K}_1 \xrightarrow{\hat{l}_1} \hat{R}_1 \cdots \xrightarrow{\hat{l}_n} \hat{R}_n \xrightarrow{\text{RA}} \hat{K}'$, $\hat{K}_1 = \hat{K}$, and $\hat{K}'^{n+1} = \hat{K}'$. Then by induction on $n$ and the Abstract Type Preservation lemma, $\vdash_{\text{cfg}} \hat{K}'^n : d$. Therefore, because $\hat{K}'^{n+1} = \hat{K}'$, $\vdash_{\text{cfg}} \hat{K}' : d$. \qed
Appendix K

Proofs for Transformation Conformance

This appendix proves that transformation conformance implies summary conformance (section K.29), and it proves that each of the transformations defined throughout the dissertation is conformance-reflecting, as needed for transformation conformance. The formal definition of conformance reflection and other terms are below; the proofs follow.

K.1 Miscellaneous Definitions

Throughout this appendix, let abstract event step executions $\hat{K}_1 \hat{l}_1, \ldots, \hat{l}_m, \ldots$ be ranged over by the variable $\hat{aex}$.

For a PSM $s = \langle H, H', \varphi : \eta_1, \Phi, O \rangle$, section 3.6.5 defined $\text{OutMon}(s)$ as that PSM's set of output-monitored markers, i.e., $\text{OutMon}(s) = H'$. Similarly, $\text{InMon}(s)$ is defined that PSM's set of input-monitored markers, i.e., $\text{InMon}(s) = H$. The function $\text{Mon}(s)$ defines all of a PSM's monitored markers, i.e., $\text{Mon}(s) = \text{OutMon}(s) \cup \text{InMon}(s)$.

We say that all PSMs in a specification configuration $S$ monitor distinct markers if and only if for all $s \in S$ and $s' \in S$ such that $s \neq s'$, $\text{Mon}(s) \cap \text{Mon}(s') = \emptyset$.

It is sometimes useful to talk about the markers monitored by all of the PSMs of an entire specification configuration, so we also define

- $\text{OutMon}(S) = \bigcup_{s \in S} \text{OutMon}(s)$,
- $\text{InMon}(S) = \bigcup_{s \in S} \text{InMon}(s)$, and
- $\text{Mon}(S) = \bigcup_{s \in S} \text{Mon}(s)$.

A PSM $\langle H, H', \varphi : \eta_1, \ldots, \eta_n, \Phi, \langle \eta'_1, p_{o_1} \rangle, \ldots, \langle \eta'_m, p_{o_m} \rangle \rangle$ is well-formed if and only if $|H| \leq 1$ and $\{ \eta_1, \ldots, \eta_n, \eta'_1, \ldots, \eta'_m \} \subseteq H'$, i.e., it output-monitors all of its
state arguments and obligation destinations. A specification configuration \( S \) is well-formed if and only if all of its constituent PSMs are well-formed.

The function \( \text{Matchable} \) defines the markers from a label \( \lambda \) that can possibly be matched by some pattern. It is defined as follows.

\[
\text{Matchable}(\lambda) =
\begin{cases}
\{ \eta \mid \exists p, \tau, \eta', \hat{v} \sim p \tau \rightarrow [x \rightarrow \eta'] \text{ and } \eta \in \{ \eta' \} \} & \text{if } \lambda = \hat{a}'@H'\hat{v} \\
\{ \eta \mid \exists p, H, S, \hat{v} \sim p \rightarrow H, S \text{ and } \eta \in H \cup \text{Mon}(S) \} & \text{if } \lambda = \hat{a}'@H'!\hat{v} \\
\emptyset & \text{if } \lambda = \bullet
\end{cases}
\]

The relation \( \text{Simulates} \) defines when a summary specification execution simulates an event-step execution. Formally, \( \text{Simulates} \) is defined such that

\[
\text{Simulates} \left( S_1 \xrightarrow{(L_1, O_1)} \ldots, \hat{a} \hat{v} \right) \quad \text{if and only if} \quad \text{len}(\hat{a} \hat{v}) = \text{len} \left( S_1 \xrightarrow{(L_1, O_1)} \ldots \right) \quad \text{and} \quad \hat{K}_1 \xrightarrow{L_1} \ldots \text{summarizes} \hat{a} \hat{v}.
\]

Two abstract event-step executions \( \hat{K}_1 \xrightarrow{L_1} \ldots \text{ and } \hat{K}_1' \xrightarrow{L_1'} \ldots \) share a prefix of length \( i \) if and only if

- \( \text{len} \left( \hat{K}_1 \xrightarrow{L_1} \ldots \right) \geq i \),
- \( \text{len} \left( \hat{K}_1' \xrightarrow{L_1'} \ldots \right) \geq i \), and
- the first \( i + 1 \) configurations and the transition labels of the first \( i \) steps of \( \hat{K}_1 \xrightarrow{L_1} \ldots \text{ and } \hat{K}_1' \xrightarrow{L_1'} \ldots \) are identical.

Note that in the base case when \( i = 0 \), the initial configurations of each execution must still be identical.

Similarly, two specification summary executions \( S_1 \xrightarrow{(L_1, O_1)} \ldots \), and \( S_1' \xrightarrow{(L_1', O_1')} \ldots \) share a prefix of length \( i \) if and only if

- \( \text{len} \left( S_1 \xrightarrow{(L_1, O_1)} \ldots \right) \geq i \),
- \( \text{len} \left( S_1' \xrightarrow{(L_1', O_1')} \ldots \right) \geq i \),
- the first \( i + 1 \) configurations and the labels of the first \( i \) transitions of \( S_1 \xrightarrow{(L_1, O_1)} \ldots \), and \( S_1' \xrightarrow{(L_1', O_1')} \ldots \) are identical.

A strategy \( Z \) is deterministic if and only if for all \( \left( \hat{K} \xrightarrow{L} \hat{K}', s \right) \in \text{dom}(Z) \),

\[
\left| Z(\hat{K} \xrightarrow{L} \hat{K}', s) \right| \leq 1.
\]
K.2 Conformance Reflection

As mentioned in section 6.2.2, for a transformation to be useful for proving conformance, it must be conformance-reflecting. Intuitively, that means that if all pairs from a transformation $T(\hat{K}, s)$ “conform” to each other, then so does the pair $(\hat{K}, s)$. This section formally defines conformance reflection.

The definition of conformance reflection requires the existence of two associated functions for every transformation $T$: $\text{TransExec}_T$ and $\text{UntransExec}_T$. The idea is that given some execution $\hat{K}_1 \xrightarrow{\hat{I}_{1,1}, \ldots, \hat{I}_{1,n}} \ldots$ and a PSM $s_1$, $\text{TransExec}_T$ returns the set of executions that represent the “equivalent” executions starting from the program configurations in $T(\hat{K}_1, s_1)$. For example, in the case of a transformation such as $\text{Canonicalize}$ that renames addresses and history markers, $\text{TransExec}_{\text{Canonicalize}}$ simply applies the same renaming across the entire execution to get an equivalent execution starting from the renamed program configuration. The function $\text{UntransExec}_T$ then has a dual purpose: once simulating specification executions have been found for each of the transformed program executions from $\text{TransExec}_T$, $\text{UntransExec}_T$ combines those into a single specification execution $s_1 \xrightarrow{\Lambda_1} \ldots$ that simulates the original program execution $\hat{K}_1 \xrightarrow{\hat{I}_{1,1}, \ldots, \hat{I}_{1,n}} \ldots$ (summarized by $\hat{K}_1 \xrightarrow{L} \ldots$). For example, $\text{UntransExec}_{\text{Canonicalize}}$ reverses the renaming across the specification execution, thereby obtaining a simulating execution for the original program execution.

Unlike the transformation $T$ itself, the related functions $\text{TransExec}_T$ and $\text{UntransExec}_T$ do not need to be implemented as part of the model checker. They exist only as mathematical objects for the sake of proving conformance reflection for each transformation, and for proving the Transformation Conformance theorem.

K.2.1 Execution Maps

The input to $\text{UntransExec}_T$ is phrased in terms of an execution map that maps the abstract event-step executions from $\text{TransExec}_T$ to specification summary executions, formally defined as follows.

**Definition.** An execution map $X$ is a partial function from tuples of the form $\langle \hat{K}_1 \xrightarrow{\hat{I}_{1,1}, \ldots, \hat{I}_{1,n}} \ldots, s, A, M \rangle$ to specification summary executions $S_1 \xrightarrow{\Lambda_1} \ldots$.

The function $\text{UntransExec}_T$ is defined only when $X$ contains a simulating execution for all of the results from $\text{TransExec}_T$. This is formalized as follows.

**Definition.** An execution map $X$ contains a simulation for $\langle \bar{a} \bar{e} \bar{x}, s, A, M \rangle$ if and only if there exists a specification summary execution $S_1 \xrightarrow{(L_1, O_1)} \ldots$ such that

- $X(\bar{a} \bar{e} \bar{x}, s, A, M) = S_1 \xrightarrow{(d_1, O_1)} \ldots$,
- $S_1 = \{s\}$, and
- $\text{Simulates}(S_1 \xrightarrow{(L_1, O_1)} \ldots, \bar{a} \bar{e} \bar{x})$. 


K.2.2 Definition of Conformance Reflection

**Definition.** A transformation $T$ is conformance-reflecting if and only if there exist a total function $\text{TransExec}_T$ and a partial function $\text{UntransExec}_T$ such that all of the following properties hold.

**Properties for $T$**

For all $\hat{K}$ and $s$, if $\hat{K}$ and $s$ are well-formed, $\hat{K}$ is an externals-only configuration, $\text{Mon}(s) \subseteq \text{Used}(\hat{K})$, and no actor in $\hat{K}$ is handling an event, then $T(\hat{K}, s)$ is a set $\{⟨\hat{K}_1', s_1', A_1, M_1⟩, \ldots, ⟨\hat{K}_n', s_n', A_n, M_n⟩\}$ such that the following properties hold. In the following, let $\hat{\mu}$ be the message-map component of $\hat{K}$, and let $\hat{\mu}_1', \ldots, \hat{\mu}_n'$ be the message-map components of $\hat{K}_1', \ldots, \hat{K}_n'$, respectively.

**Well-Formed Preservation** For all $i \in 1 \ldots n$, $\hat{K}_i'$ and $s_i'$ are well-formed.

**Externals-Only Preservation** For all $i \in 1 \ldots n$, $\hat{K}_i'$ is an externals-only configuration.

**All-Awaiting Preservation** For all $i \in 1 \ldots n$, no actor in $\hat{K}_i'$ is handling an event.

**Used/Monitored Preservation** For all $i \in 1 \ldots n$, $\text{Mon}(s_i') \subseteq \text{Used}(\hat{K}_i')$.

**No New Enabled Actors** For all $i \in 1 \ldots n$ and all $\hat{a}'$ that identify a necessarily enabled actor in $\hat{K}_i'$, there exists $\hat{a}$ such that $A_i(\hat{a}) = \hat{a}'$ and $\hat{a}$ identifies a necessarily enabled actor in $\hat{K}$.

**Atomic Address Reflection** For all $i \in 1 \ldots n$ and all $\hat{a}$ and $\hat{a}'$, if $A_i(\hat{a}) = \hat{a}'$ and $\hat{a}'$ is atomic, then $\hat{a}$ is atomic.

**Unique Actor Correspondences** For all $i \in 1 \ldots n$ and for all $\hat{a}$, $\hat{a}'$, and $\hat{a}''$, if $A_i(\hat{a}) = \hat{a}'$, $A_i(\hat{a}) = \hat{a}''$, and $\hat{a}''$ is atomic, then $\hat{a} = \hat{a}'$.

**No New Single Messages** For all $i \in 1 \ldots n$ and all $\hat{a}'$, $H'$, and $\hat{v}'$ such that $\hat{\mu}_i'(\hat{a}'@H', \hat{v}') = \text{single}$, there exist $\hat{a}$, $H$, and $\hat{v}$ such that

- $A(\hat{a}) = \hat{a}'$,
- $M(H) = H'$,
- $\text{Remap}(\hat{v}, A, M) = \hat{v}'$, and
- $\hat{\mu}(\hat{a}@H, \hat{v}) = \text{single}$.

**Internal Address Reflection** For all $i \in 1 \ldots n$ and all $\hat{a}$ and $\hat{a}'$, if $\hat{a}'$ is internal and $A_i(\hat{a}) = \hat{a}'$, then $\hat{a}$ is internal.

**Unique Approximating Messages** For all $i \in 1 \ldots n$ and all $⟨\hat{a}@H, \hat{v}⟩ \in \text{dom}(\hat{\mu})$ such that $A_i(\hat{a})$ is internal, there exist $\hat{a}'$, $H'$, and $\hat{v}'$ such that

- $\hat{\mu}(\hat{a}@H, \hat{v}) \subseteq \hat{\mu}_i'(\hat{a}'@H', \hat{v}')$ and
K.2. CONFORMANCE REFLECTION

• \((\hat{a}@H',\hat{v}')\) is the only member of \(\text{dom}(\hat{\mu}')\) such that \(A_i(\hat{a}) = \hat{a}', M_i(H) = H',\) and \(\text{Remap}(\hat{v}, A_i, M_i) \sqsubseteq_{id,id} \hat{v}'\).

Unique Approximated Messages For all \(i \in 1...n\) and all \(\hat{a}', H',\) and \(\hat{v}'\) such that \(\hat{\mu}'(\hat{a}@H',\hat{v}') = \text{single}\), there exist \(\hat{a}, H,\) and \(\hat{v}\) such that

• \(\hat{\mu}(\hat{a}@H,\hat{v}) = \text{single}\) and
• \((\hat{a}@H,\hat{v})\) is the only member of \(\text{dom}(\hat{\mu})\) such that \(A_i(\hat{a}) = \hat{a}', M_i(H) = H',\) and \(\text{Remap}(\hat{v}, A_i, M_i) \sqsubseteq_{id,id} \hat{v}'\).

Properties for \(\text{TransExec}_T\)

For all abstract event-step executions \(\hat{K}_1 \xrightarrow{I_{1,1} \ldots I_{1,m}} \ldots\) and all PSMs \(s,\) if \(\hat{K}_1\) and \(s\) are well-formed, \(\hat{K}_1\) is an externals-only configuration, \(\text{Mon}(s) \sqsubseteq \text{Used}(\hat{K}_1),\) and no actor in \(\hat{K}_1\) is handling an event, then \(\text{TransExec}_T(\hat{K}_1 \xrightarrow{I_{1,1} \ldots I_{1,m}} \ldots, s)\) is a set \(\left\{ \left( \hat{K}'_{i,1} \xrightarrow{\gamma_{1,1} \ldots \gamma_{1,m}} \ldots, s'_i, A_i, M_i \right) \right\}_{i \in 1...n}\) such that the following properties hold.

Initial Pair Correctness For all \(i \in 1...n, \left( \hat{K}'_{i,1}, s'_i, A_i, M_i \right) \in T(\hat{K}_1, s)\).

Fairness Preservation 1 If \(\hat{K}_1 \xrightarrow{I_{1,1} \ldots I_{1,m}} \ldots\) is fair, then for all \(i \in 1...n, \hat{K}'_{i,1} \xrightarrow{\gamma_{1,1} \ldots \gamma_{1,m}} \ldots\) is fair.

Fairness Preservation 2 For all \(\hat{a}\) and \(j,\) if an actor at \(\hat{a}\) is necessarily enabled in \(\hat{K}_1\) and either

• the actor at \(\hat{a}\) is not necessarily enabled in \(\hat{K}_j\) or
• \(\hat{K}_j \xrightarrow{I_{j,1} \ldots I_{j,m}} \hat{K}_{j+1}\) is a step in the execution and \(\hat{a}\) identifies the active actor for \(I_{j,1},\)

then for all \(i \in 1...n\) there exists \(k \leq j\) such that either

• there is no necessarily enabled actor at \(A_i(\hat{a})\) in \(\hat{K}'_{i,k},\) or
• \(\hat{K}'_{i,k} \xrightarrow{\gamma_{i,k,1} \ldots \gamma_{i,k,m}} \hat{K}_{i,k+1}\) is a step in the execution and \(\hat{K}'_{i,1} \xrightarrow{\gamma_{i,1,1} \ldots \gamma_{i,1,m}} \ldots\) and \(A_i(\hat{a})\) identifies the active actor for \(I_{i,k,1}.'\)

Fairness Preservation 3 Let \(\left\langle \langle \hat{\beta} \mid \hat{\mu} \mid H \rangle \rangle_\hat{\nu} = \hat{K}_1\). For all \(\hat{a}, H', \hat{v},\) and \(j,\) if

• \(\hat{\mu}(\hat{a}@H',\hat{v}) = \text{single}\), and
APPENDIX K. PROOFS FOR TRANSFORMATION CONFORMANCE

- either $\vec{K}_j = \left\langle \vec{B} \left| \bar{\rho}' \right. \right\rangle^{H''}$ is a configuration in the execution such that $\langle \bar{a} @ H', \bar{v} \rangle \in \text{dom}(\bar{\rho}')$ or $\bar{\rho}'(\bar{a} @ H', \bar{v}) = \text{many}$, or $\vec{K}_j \xrightarrow{\text{rcv-int}(H', \bar{v})} \vec{K}_{j+1}$ is a step in the execution,

then for all $i \in 1 \ldots n$ there exist $\bar{a}' = A_1(\bar{a}), H'' = M_i(H'), \bar{v}' = A_1(M_i(\bar{v}))$, and $k \leq j$ such that $\vec{K}'_k = \left\langle \left\langle \bar{\rho}'' \left| \bar{\rho}'' \right. \right\rangle^{H''} \right\rangle$ is a configuration in the execution $\vec{K}'_k \xrightarrow{\bar{\rho}_{i,k}^{l_{1,k}^{\cdots} l_{1,m}^{\cdots}}} \vec{K}'_{k+1}$ is a step in that execution such that $\bar{\rho}_{i,k,1}^{l_{1,k}^{\cdots} l_{1,m}^{\cdots}} = \bar{a}'$:

\begin{itemize}
  \item $\langle \bar{a}' @ H'', \bar{v}' \rangle \notin \text{dom}(\bar{\rho}'')$,
  \item $\bar{\rho}''(\bar{a}' @ H'', \bar{v}') = \text{many}$, or
  \item $\vec{K}'_k \xrightarrow{\bar{\rho}_{i,k}^{l_{1,k}^{\cdots} l_{1,m}^{\cdots}}} \vec{K}'_{k+1}$ is a step in that execution such that $\bar{\rho}_{i,k,1}^{l_{1,k}^{\cdots} l_{1,m}^{\cdots}} = \bar{a}'$:
\end{itemize}

Properties for UntransExec\(_T\)

For all abstract event-step executions $\vec{K}_1 \xrightarrow{\bar{\rho}_{1,1}^{l_{1,1}^{\cdots} l_{1,m}^{\cdots}}} \ldots$, PSMs $s$, and execution maps $X$, if

- $\vec{K}_1$ and $s$ are well-formed,
- $\vec{K}_1$ is an externals-only configuration,
- $\text{Mon}(s) \subseteq \text{Used}(\vec{K}_1)$,
- no actor in $\vec{K}_1$ is handling an event, and
- $X$ contains a simulation for all members of $\text{TransExec}_T(\vec{K}_1 \xrightarrow{\bar{\rho}_{1,1}^{l_{1,1}^{\cdots} l_{1,m}^{\cdots}}} \ldots, s)$,

then $\text{UntransExec}_T(\vec{K}_1 \xrightarrow{\bar{\rho}_{1,1}^{l_{1,1}^{\cdots} l_{1,m}^{\cdots}}} \ldots, s, X)$ is a specification execution $S_1 \xrightarrow{\langle L_1, O_1 \rangle} \ldots$ such that the following properties hold.

Execution Simulation $S_1 = \{s\}$ and $\text{Simulates}(S_1 \xrightarrow{\langle L_1, O_1 \rangle} \ldots, \vec{K}_1 \xrightarrow{\bar{\rho}_{1,1}^{l_{1,1}^{\cdots} l_{1,m}^{\cdots}}} \ldots)$.

Prefix Consistency For all $\vec{K}_1 \xrightarrow{\bar{\rho}_{1,1}^{l_{1,1}^{\cdots} l_{1,m}^{\cdots}}} \ldots, X', S_1' \xrightarrow{\langle L_1', O_1' \rangle} \ldots$, and $i$, if

- $\vec{K}_1 \xrightarrow{\bar{\rho}_{1,1}^{l_{1,1}^{\cdots} l_{1,m}^{\cdots}}} \ldots$ and $\vec{K}'_1 \xrightarrow{\bar{\rho}_{1,1}^{l_{1,1}^{\cdots} l_{1,m}^{\cdots}}} \ldots$ share a prefix of length $i$,
- $\text{UntransExec}_T(\vec{K}_1' \xrightarrow{\bar{\rho}_{1,1}^{l_{1,1}^{\cdots} l_{1,m}^{\cdots}}} \ldots, s, X') = S_1' \xrightarrow{\langle L_1', O_1' \rangle} \ldots$,
- for all $\bar{a} \bar{e} \bar{x}, \bar{a} \bar{e} \bar{x}', s', A, M$, and $j$ such that
  - $\langle \bar{a} \bar{e} \bar{x}, s', A, M \rangle \in \text{TransExec}_T(\vec{K}_1 \xrightarrow{\bar{\rho}_{1,1}^{l_{1,1}^{\cdots} l_{1,m}^{\cdots}}} \ldots, s)$,
K.3. Definition of SimExecs

A proof of transformation conformance for a pair \( \langle K, s \rangle \) produces a strategy \( Z \). Then given an abstract event-step simulation starting from \( K \), one can use \( Z \) to find a simulating summary execution starting from \( s \), as required for abstract conformance. This section defines a function \( \text{SimExecs} \) that does just that. The proof of the Transformation Conformance theorem then uses this function to convert a proof of transformation conformance into a proof of abstract conformance.

The function definition requires a notion of “union” for specification summary executions. The union of two such executions is defined whenever they have the same number of steps and use the same program summary step labels at each step. Then the union itself just takes the union of the PSMs in each configuration at each point, as well as the multiset-sum of the fulfilled obligations in each step.

\[
\text{Definition. } \ \langle \text{SimExecs} \rangle = \bigcup_{i \leq i} \langle \text{SimExecs} \rangle' = (\text{SimExecs} \cup \text{SimExecs}') \text{ if and only if } \text{len}(\text{SimExecs}) = \text{len}(\text{SimExecs}')
\]
Given an abstract event-step execution \( \hat{K}_1 \xrightarrow{\hat{f}_{1,1}, \ldots, \hat{f}_{1,m}} \ldots \), a PSM \( s \), a transformation \( T \), and a strategy \( Z \), the idea of \( \text{SimExecs} \) is to return a set of summary executions starting from the specification configuration \( \{s\} \) such that each one simulates the given event-step execution. The definition assumes that the given transformation \( T \) has associated functions \( \text{TransExec}_T \) and \( \text{UntransExec}_T \) of the type described in section K.2.2. An explanation follows after the definition.

\[
\text{SimExecs}(\hat{K}_1 \xrightarrow{\hat{f}_{1,1}, \ldots, \hat{f}_{1,m}} \ldots, s, T) =
\begin{cases}
\text{If } \text{len} \left( \hat{K}_1 \xrightarrow{\hat{f}_{1,1}, \ldots, \hat{f}_{1,m}} \ldots \right) = 0: \\
\{\{s\}\}
\end{cases}
\]

Else:

\[
\begin{align*}
&\{s\} \xrightarrow{\langle L_1, O_1 \rangle} S_2 \xrightarrow{\langle L_2, O_2 \rangle} \ldots \\
&\exists X \text{ such that } \\
&\quad \bullet L_1 \text{ is the (lexicographically) least summary-transition label such that } \hat{K}_1 \xrightarrow{L_1} \hat{K}_2 \text{ summarizes } \hat{K}_1 \xrightarrow{\hat{f}_{1,1}, \ldots, \hat{f}_{1,m}} \hat{K}_2 \text{ and } Z(\hat{K}_1 \xrightarrow{L_1} \hat{K}_2, s) \neq \emptyset, \\
&\quad \bullet \langle O_1, S_2 \rangle \in Z(\hat{K}_1 \xrightarrow{L_1} \hat{K}_2, s), \\
&\quad \bullet \text{dom}(X) = \bigcup_{s' \in S_2} \text{TransExec}_T(\hat{K}_2 \xrightarrow{\hat{f}_{2,1}, \ldots, \hat{f}_{2,m}} \ldots, s'), \\
&\quad \bullet \forall \langle \hat{a} \hat{e}, s''', A, M \rangle \in \text{dom}(X). X(\hat{a} \hat{e}, s''', A, M) \in \text{SimExecs}(\hat{a} \hat{e}, s''', T, Z), \text{ and } \\
&\quad \bullet S_2 \xrightarrow{\langle L_2, O_2 \rangle} \ldots = \bigcup_{s' \in S_2} \text{UntransExec}_T(\hat{K}_2 \xrightarrow{\hat{f}_{2,1}, \ldots, \hat{f}_{2,m}} \ldots, s', X)
\end{align*}
\]

If the given execution has zero steps, then there are no steps to simulate, so the zero-step summary execution \( \{s\} \) is a valid simulation of the given execution. Otherwise, a simulating execution can be found by the following steps.

1. Find the (lexicographically) least summary step \( \hat{K}_1 \xrightarrow{L_1} \hat{K}_2 \) that summarizes the first step \( \hat{K}_1 \xrightarrow{\hat{f}_{1,1}, \ldots, \hat{f}_{1,m}} \hat{K}_2 \) of the given execution and has at least one matching step in the strategy \( Z \). (The least step is chosen so that \( \text{SimExecs} \) returns at most one result when given a deterministic strategy, as the \( \text{SimExecs} \) Determinism lemma below proves. The details of the lexicographic order on summary-transition labels are unimportant, as long as there exists some total order on those labels.)

2. Use the strategy \( Z \) to find a step \( \{s\} \xrightarrow{\langle L_1, O_1 \rangle} S_2 \) that simulates \( \hat{K}_1 \xrightarrow{L_1} \hat{K}_2 \).

3. For each PSM \( s' \) in the reached configuration \( S_2 \), use \( \text{TransExec}_T \) to transform the remainder of the event-step transition starting from \( \hat{K}_2 \) paired with each of the PSMs \( s' \in S_2 \) into a set of tuples \( \{\langle \hat{a} \hat{e}_1, s''_1, A_1, M_1 \rangle, \ldots, \langle \hat{a} \hat{e}_n, s''_n, A_n, M_n \rangle\} \).
4. For each \((\alpha \sigma \xi, s''_i, A_i, M_i)\), use a recursive call to \SimExecs to find a summary specification from \(s''_i\) that simulates \(\alpha \sigma \xi_i\). Record these simulating executions in the execution map \(X\).

5. Having found a simulating execution for each \(\alpha \sigma \xi_i\) and recorded them in \(X\), use \UntransExec\ to merge those executions together to create a simulating execution from each of the \(s' \in S_2\). Then take the union of those executions to produce an execution \(S_2 \xrightarrow{L_{2},O_{2}} \ldots \) that simulates the remainder \(K_2 \xrightarrow{I_{21},\ldots,I_{2m}} \ldots\) of the original event-step execution, which is what we want.

6. Add the step \(K_1 \xrightarrow{L_1} K_2\) found from the strategy to the front of the produced execution, then return the new execution as a result.

In this way, the possible simulating specification executions are defined coinductively. For each call to \SimExecs, we obtain the first step from the strategy, then obtain the remainder of the execution by using \TransExec\, a recursive call to \SimExecs, and \UntransExec. The function is defined in such a way that for each call \SimExecs\(\langle K_1 \xrightarrow{I_{11},\ldots,I_{1n}} \ldots, s, T, Z\rangle\), if \(\langle K_1, s\rangle\) is in a transformation-conformance-dense relation \(R\) for \(T\) with witness \(Z\), then for each recursive call \SimExecs\(\langle K_1', \xrightarrow{I_{11}',\ldots,I_{1m}'} \ldots, s'', T, Z\rangle\), \(\langle K_1', s''\rangle\) is also in \(R\). Thus, the strategy \(Z\) always provides a matching “next step”; the proof of the \SimExecs Non-Emptiness lemma in section K.25 formalizes this idea.

**K.4 Fair Suffix Lemma**

**Lemma** (Fair Suffix). For all abstract event-step executions \(K_1 \xrightarrow{I_{11},\ldots,I_{1n}} \ldots\) and all \(i \leq 1 + \text{len}(\hat{K}_1 \xrightarrow{I_{11},\ldots,I_{1n}} \ldots)\), if the suffix of the execution starting from \(\hat{K}_i\) is fair, then the entire execution is fair.

**Proof.** Let \(\hat{K}_j\) be a configuration in the execution, and let there be \(\alpha \) such that the actor at \(\alpha \) is necessarily enabled in \(\hat{K}_j\). If there is any \(k\) such that \(\hat{K}_{j+k}\) is a configuration in the execution and the actor at \(\alpha \) is not necessarily enabled in \(\hat{K}_{j+k}\), then we're done. Otherwise, there must be some \(k\) such that \(i+k > j\) and the actor at \(\alpha \) is enabled in \(\hat{K}_{i+k}\). Then by the fairness of the suffix execution, there exists some \(m\) such that \(\hat{K}_{i+k+m} \xrightarrow{I_{i+k+m,1},\ldots,I_{i+k+m,n}} \hat{K}_{i+k+m+1}\) is a step in the execution such that \(\alpha \) identifies the active actor for \(\hat{I}_{i+k+m,1}\). We have that \(i+k+m > j\), so we're done.

Second, let \(\hat{K}_j = \langle \hat{\beta} \mid \hat{\mu} \mid H \rangle\) be a configuration in the execution, and let there be \(\alpha, H',\) and \(\hat{\beta}\) such that \(\hat{\mu}(\alpha \hat{\circ} H', \hat{\beta}) = \text{single}\). If there is any \(k\) such that \(\hat{K}_{j+k} = \langle \hat{\beta} \mid \hat{\mu}' \mid H'' \rangle\) and either \(\langle \hat{\alpha} \hat{\circ} H', \hat{\beta}\rangle \in \text{dom}(\hat{\mu}')\) or \(\hat{\mu}'(\hat{\alpha} \hat{\circ} H', \hat{\beta}) = \text{many}\), then we're done. Otherwise, there must be some \(k\) such that \(i+k > j\), \(\hat{K}_{i+k} = \)
\begin{align*}
\langle \langle \beta^n, \mu^n, H'' \rangle \rangle^\rho'' \rangle \rangle, \quad \text{and} \quad \hat{\mu}(\hat{a} @ H', \hat{v}) = \text{single}. \quad \text{Then by the fairness of the suffix execution, there exists some} \quad m \quad \text{such that} \quad \hat{K}_{i+k+m+1} \xrightarrow{\hat{I}_{i+k+m,1}, \ldots, \hat{I}_{i+k+m,n}} \hat{K}_{i+k+m+1} \quad \text{is a step in the execution and} \quad \hat{I}_{i+k+m,1} = \hat{a} : \text{rcv-int}(H', \hat{v}). \quad \text{We have that} \quad i + k + m > j, \quad \text{so we're done.} \quad \square
\end{align*}

K.5 Specification Well-Formed Preservation

Lemma (Specification Well-Formed Preservation). For all \(s, s', \lambda, O, \) and \(S,\) if \(s\) is well-formed and \(s \xrightarrow{\lambda, O, S} s',\) then \(s'\) and \(S\) are well-formed.

Proof. The proof is by cases on the rule enabling the transition \(s \xrightarrow{\lambda, O, S} s'.\)

Case: P-UNMONITOREDRECEIVE
In this case, \(s' = s\) and \(S = \emptyset,\) so \(s'\) and \(S\) are well-formed.

Case: P-MONITOREDRECEIVE
The definition of this rule enforces that \(s'\) has all of its state arguments as output-monitored markers. The function \(\text{Dist}\) used to add new obligations to the PSMs enforces that for each PSM \(\langle H, H', \phi : \pi, O \rangle \in \{s'\} \cup S\) and every obligation \(\langle \eta', po \rangle \in O, \eta' \in H'.\) All of the PSMs in \(S\) come from the \text{PerformAll} function, which ensures that all forked PSMs have their state arguments as output-monitored markers. Finally, the PSMs created in \text{PerformAll} have no input-monitored markers and this rule does not change the input-monitored markers on the original PSM, so \(s'\) and \(S\) are well-formed.

Case: P-FREETRANSITION
Same as the previous case, except that in this case the state arguments for \(s'\) must be from the state arguments of \(s,\) so \(s'\) has all of its state arguments as output-monitored markers.

Case: P-SEND
In this case, none of the forked PSMs in \(S\) have state arguments or obligations. By the definition of the output-pattern-matching relation, each of the forked PSMs input-monitors exactly one marker, so are well-formed. The PSM \(s'\) does not change its state arguments or output-monitored markers, it does not gain any new obligations, and the rule explicitly enforces that \(s'\) input-monitors at most one marker. Therefore, because \(s\) is well-formed, \(s'\) is also well-formed. \(\square\)

Corollary K.5.1. For all \(S, S', \) and \(\lambda,\) if \(S\) is well-formed and \(S \xrightarrow{\lambda} S',\) then \(S'\) is well-formed.
Proof. The proof is by cases on the rule enabling the transition $S \xrightarrow{\lambda} S'$.

**Case: S-SENDORRECEIVE**

By the definition of this rule, there exist $s_1, \ldots, s_n$, $s'_1, \ldots, s'_n$, $S''_1, \ldots, S''_n$, and $O_1, \ldots, O_n$ such that

- $S = \{s_1, \ldots, s_n\}$,
- $S' = \{s'_1, \ldots, s'_n\} \cup S''_1 \cup \ldots \cup S''_n$, and
- \[ s_i \xrightarrow{\hat{a}@H''\in\{O_i\}, S''_i} s'_i \text{ for all } i \in 1 \ldots n. \]

By the definition of a well-formed specification configuration, $s_i$ is well-formed for all $i \in 1 \ldots n$. Then by the PSM Well-Formed Preservation lemma, for all $i \in 1 \ldots n$, $s'_i$ and $S''_i$ are well-formed. Therefore, $S'$ is well-formed.

**Case: S-FREETRANSITION**

By the definition of this rule, there exist $s, s', S''$, and $S'''$ such that

- $S = \{s\} \cup S''$,
- $s \xrightarrow{\sigma, S''} s'$,
- $S' = \{s'\} \cup S'' \cup S'''$.

By the definition of a well-formed specification configuration, $s$ and $S''$ are well-formed. Then by the PSM Well-Formed Preservation lemma, $s'$ and $S'''$ are well-formed. Therefore, $S'$ is well-formed.

**Corollary K.5.2.** For all $S$, $S'$, and $\hat{\lambda}$, if $S$ is well-formed and $S \xrightarrow{\hat{\lambda}} S'$, then $S'$ is well-formed.

**Proof.** If $\hat{\lambda} = \bullet$, then by the definition of the weak-step relation $\xrightarrow{\bullet}$, $S \xrightarrow{\bullet} \cdots \xrightarrow{\bullet} S'$. Then by induction and corollary K.5.1 of this lemma, $S'$ is well-formed.

If $\hat{\lambda} \neq \bullet$, then by the definition of the weak-step relation $\xrightarrow{\bullet}$, there exist $S''$ and $S'''$ such that $S \xrightarrow{\bullet} S'' \xrightarrow{\hat{\lambda}} S''' \xrightarrow{\bullet} S'$. By the argument for the previous case, $S''$ is well-formed. Then by corollary K.5.1 of this lemma, $S'''$ is well-formed. Finally, again by the argument for the previous case, $S'$ is well-formed.

**Corollary K.5.3.** For all $S$, $S'$, and $\hat{\lambda}_1, \ldots, \hat{\lambda}_n$, if $S$ is well-formed and $S \xrightarrow{\hat{\lambda}_1 \cdots \hat{\lambda}_n} S'$, then $S'$ is well-formed.

**Proof.** By the definition of the relation $\xrightarrow{\hat{\lambda}_1 \cdots \hat{\lambda}_n}$, $S \xrightarrow{\hat{\lambda}_1 \cdots \hat{\lambda}_n} S'$. By induction on the length of that execution and corollary K.5.2 of this lemma, $S'$ is well-formed.
Corollary K.5.4. For all \( S, S', L, \) and \( O \), if
- \( S \) is well-formed,
- \( S \xrightarrow{\Lambda} S' \), and
- there exist \( \hat{K} \) and \( \hat{K}' \) such that \( \hat{K} \xrightarrow{L} \hat{K}' \),
then \( S' \) is well-formed.

Proof. By the definition of the program summary-transition relation and the abstract event-step relation, there exist \( \hat{K}'_{1}, \ldots, \hat{K}'_{n+1} \) and \( \hat{l}_{1}, \ldots, \hat{l}_{n} \) such that
- \( \hat{K}'_{1} = \hat{K} \),
- \( \hat{K}'_{n+1} = \hat{K}' \), and
- \( \hat{K}'_{1} \xrightarrow{\Lambda} \cdots \xrightarrow{\Lambda} \hat{K}'_{n+1} \).

Then by the definition of the specification summary-transition relation, \( S \xrightarrow{\hat{l}_{1}, \ldots, \hat{l}_{n}} S' \). By corollary K.5.3 of this lemma, \( S' \) is well-formed.

K.6 Empty Matchable Output Lemma

Lemma (Empty Matchable Output). For all \( L = (\hat{l}', \hat{\mu}), \hat{l}_{1}, \ldots, \hat{l}_{n}, \hat{a}, H, \hat{v}, \hat{v}', \hat{K}, \) and \( \hat{K}' \), \( i \in 1 \ldots n \), if
- \( L \) summarizes \( \hat{l}_{1}, \ldots, \hat{l}_{n} \),
- \( \hat{l}_{i} = \hat{a} \hat{\mu} H \hat{v} \),
- \( \hat{\mu}(\hat{a} \hat{\mu} H, \hat{v}') = \text{many} \),
- \( \text{Merge}(\hat{v}, \hat{v}') \) is defined,
- \( \hat{K} \xrightarrow{L} \hat{K}' \), and
- \( \hat{K} \) is an externals-only configuration,
then Matchable(\( \hat{l}_{i} \)) = \( \emptyset \).

Proof. By the definition of \( \hat{K} \xrightarrow{L} \hat{K}' \), there exist an abstract event step \( \hat{K} \xrightarrow{\hat{l}_{j}', \hat{\mu}'} \hat{K}'' \), two distinct labels \( \hat{l}_{j} \) and \( \hat{l}_{h} \) from that step, \( \hat{v}'' \), and \( \hat{v}''' \) such that
- \( \hat{l}_{j}' = \hat{a} \hat{\mu} H \hat{v}'' \),
- \( \hat{l}_{h}' = \hat{a} \hat{\mu} H \hat{v}''' \), and
- \( \text{Merge}(\hat{v}', \hat{v}'') \) and \( \text{Merge}(\hat{v}', \hat{v}''') \) are defined.
K.7. EXTERNALS-ONLY LABEL LEMMA

By the definition of an event step, there also exist $\tilde{K}^{m_1}_1, \ldots, \tilde{K}^{m_{m+1}}_m$ such that $\tilde{K}^{m_1}_1 = \hat{K}$, $\tilde{K}^{m_{m+1}}_m = \hat{K}''$, and $\tilde{K}^{m_i}_i \overset{\mathcal{RA}}{\rightarrow} \tilde{K}^{m_{i+1}}_{i+1}$. By induction and repeated uses of the Abstract Externals-Only Preservation lemma, every configuration in that sequence is an externals-only configuration.

The labels $\tilde{l}^j_j$ and $\tilde{l}^k_k$ must correspond to two uses of the A-SendExternal rule. Thus, there must exist some $\tilde{v}'''$ and $\tilde{v}'''$ in $\tilde{K}^{m_j}_j$ and $\tilde{K}^{m_k}_k$, respectively, such that either

- $\tilde{v}''' = \tilde{v}''$ and $\tilde{v}''' = \tilde{v}'''$, or
- $\langle \tilde{v}'''', H'' \rangle = \text{Mark}(\tilde{v}'', H'')$ and $\langle \tilde{v}'''', H''' \rangle = \text{Mark}(\tilde{v}'', H'''')$ for some $H'$, $H''$, $H'''$, and $H''''$.  

Because both $\tilde{K}^{m_j}_j$ and $\tilde{K}^{m_k}_k$ are externals-only configurations, for all $\tilde{a}'@H'$ appearing in $\tilde{v}'''$ or $\tilde{v}'''$, if $\tilde{a}'$ is internal, then $H' = \phi$.

For a contradiction, assume that Matchable($\tilde{I}_j$) $\neq \phi$. Then by the definition of output-pattern-matching, there must be some $\tilde{a}'@H'$ appearing in $\tilde{v}$ outside of any list or dict expression such that $\tilde{a}'$ is internal and $H' \neq \phi$. Then by the definition of Merge, $\tilde{a}@H'$ similarly appears in $\tilde{v}'$, $\tilde{v}''$, and $\tilde{v}'''$ outside of any list or dict expression. Therefore, it cannot be the case that $\tilde{v}''' = \tilde{v}''$ and $\tilde{v}''' = \tilde{v}'''$.  

Thus, it must be the case that $\tilde{v}''$ and $\tilde{v}'''$ are both values marked during the step labeled with $\tilde{l}^j_j$ and $\tilde{l}^k_k$, respectively. However, the CSA semantics mark each sent value with fresh markers at each step, so it cannot be the case that $\tilde{a}'@H'$ appears in both $\tilde{v}''$ and $\tilde{v}'''$. Therefore we have a contradiction, so Matchable($\tilde{I}_j$) $= \phi$. \hfill \square

K.7 Externals-Only Label Lemma

Lemma (Externals-Only Label). For all $\tilde{K}$, $\tilde{K}'$, and $\tilde{I}$, if $\tilde{K}$ is an externals-only configuration and $\tilde{K} \overset{\mathcal{RA}}{\rightarrow} \tilde{K}'$, then

- Matchable($\tilde{I}$) $\cap$ Used($\tilde{K}$) $= \phi$ and
- if $\tilde{I} = \tilde{a}@H?\tilde{v}$ or $\tilde{I} = \tilde{a}@H!\tilde{v}$ for some $\tilde{a}$, $H$, and $\tilde{v}$, then
  - $|H| \leq 1$,
  - for all $\tilde{a}'@H'$ in $\tilde{v}$, if $\tilde{a}$ is internal or $\tilde{I} = \tilde{a}@H?\tilde{v}$, then $|H'| \leq 1$, and
  - no marker in Matchable($\tilde{I}$) appears more than once in $\tilde{v}$.

Proof: There are three cases, depending on the value of $\tilde{I}$. If $\tilde{I} = \bullet$, then Matchable($\tilde{I}$) $= \phi$. Therefore, Matchable($\tilde{I}$) $\cap$ Used($\tilde{K}$) $= \phi$.

If $\tilde{I} = \tilde{a}@H?\tilde{v}$, then the transition $\tilde{K} \overset{\mathcal{RA}}{\rightarrow} \tilde{K}'$ must be a use of the A-ReceiveExternal rule. Therefore, there exist $H'$, $H''$, and $\tilde{v}'$ such that

- Used($\tilde{K}$) $= H'$. 

If $\tilde{I} = \tilde{a}@H!\tilde{v}$, then the same steps apply as in the previous case. Therefore, Matchable($\tilde{I}$) $\cap$ Used($\tilde{K}$) $= \phi$. 

For the case where $\tilde{I}$ is a list dict expression, matchable($\tilde{I}$) $\cap$ used($\tilde{K}$) $= \phi$. If $\tilde{I}$ is a list expression, then $\tilde{I}$ is a list dict expression, so matchable($\tilde{I}$) $\cap$ used($\tilde{K}$) $= \phi$. Therefore, there is a contradiction, so Matchable($\tilde{I}$) $\cap$ Used($\tilde{K}$) $= \phi$.

Thus, we have a contradiction, so Matchable($\tilde{I}$) $\cap$ Used($\tilde{K}$) $= \phi$. \hfill \square
Matchable in pattern-matching relation, so we have marker not in Mark.

Corollary K.7.1. For all $\hat{\alpha}'H''''''$ in $\hat{\nu}'$, $H'''' = \emptyset$ (because $\hat{\nu}'$ contains only external addresses).

By the definition of $Mark$, every address in $\hat{\nu}'$ not under a list or dict gets a fresh marker not in $H'$, and $H'''' - H'$ is the set of newly added markers in $\hat{\nu}$. These are precisely the pattern-matchable markers in $\hat{\nu}$ by the definition of the input-pattern-matching relation, so we have $H'''' - H' = Matchable(\hat{\nu})$, and therefore $Matchable(\hat{\nu}) \cap Used(\hat{\nu}) = \emptyset$. Because each such new marker is fresh, no marker in $Matchable(\hat{\nu})$ appears more than once in $\hat{\nu}$. Finally, because $Mark$ adds at most one marker to each address in $\hat{\nu}'$, for each $\langle \hat{\alpha}' \rangle H'''$ in $\hat{\nu}$, $|H'''| \leq 1$.

On the other hand, if $\langle \hat{\nu} \rangle = \hat{\alpha}@H\hat{\nu}$, then $\hat{\nu} \xrightarrow{R_{RA}} \hat{\nu}'$ is a use of the A-SENDEXTERNAL rule. By the definition of that rule, $\hat{\alpha}@H$ appears in $\hat{\nu}$ and $\hat{\alpha}$ is external. Because $\hat{\nu}$ is an externals-only configuration, $|H| \leq 1.$

Also by the definition of this rule, there exists some $\hat{\nu}'$ in $\hat{\nu}$ such that either

- $\hat{\nu}' = \hat{\nu},$ or
- there exist $H'$ and $H''$ such that $Used(\hat{\nu}) = H'$, $Used(\hat{\nu}'') = H''$, and $Mark(\hat{\nu}', H') = \langle \hat{\nu}, H'' \rangle$.

Because $\hat{\nu}$ is an externals-only configuration, for all $\hat{\alpha}@H'''''$ appearing in $\hat{\nu}'$, $H'''''' = \emptyset$ if $\hat{\alpha}'$ is internal, and $|H'''''| \leq 1$ otherwise.

If $\hat{\nu}' = \hat{\nu}$, we know that the output-pattern-matching relation allows only markers on internal addresses to be matched. Therefore in this case, $Matchable(\hat{\nu}) = \emptyset$, and therefore $Matchable(\hat{\nu}) \cap Used(\hat{\nu}) = \emptyset$.

Otherwise, by the definition of $Mark$, every address in $\hat{\nu}'$ not under a list or dict gets a fresh marker not in $H'$, and $H'''''' - H'$ is the set of newly added markers in $\hat{\nu}$. In this case, those fresh markers applied to internal addresses are the matchable ones, so we have $Matchable(\hat{\nu}) \subseteq H'''''' - H'$. Therefore, $Matchable(\hat{\nu}) \cap Used(\hat{\nu}) = \emptyset$. Again because each such new marker is fresh, no marker in $Matchable(\hat{\nu})$ appears more than once in $\hat{\nu}$. Finally, because $H'''''' = \emptyset$ for every internal marked address $\hat{\alpha}@H'''''$ appearing in $\hat{\nu}'$, $|H'''''| \leq 1$ for every internal marked address $\hat{\alpha}''@H''''''$ appearing in $\hat{\nu}$.

Corollary K.7.1. For all $\hat{\nu}, \hat{\nu}'$, and $\hat{\nu}_1, \ldots, \hat{\nu}_n$, if $\hat{\nu}$ is an externals-only configuration and $\hat{\nu} \xrightarrow{\hat{l}_1, \ldots, \hat{l}_n} \hat{\nu}'$, then for all $i \in 1 \ldots n$,

- $Matchable(\hat{\nu}_i) \cap Used(\hat{\nu}) = \emptyset$ and
- if $\hat{\nu}_i = \hat{\alpha}@H?\hat{\nu}$ or $\hat{\nu}_i = \hat{\alpha}@H!\hat{\nu}$ for some $\hat{\alpha}, H,$ and $\hat{\nu}$, then
  - $|H| \leq 1,$
  - for all $\hat{\alpha}'@H'$ in $\hat{\nu}$, if $\hat{\alpha}$ is internal or $\hat{\nu}_i = \hat{\alpha}@H?\hat{\nu}$, then $|H'| \leq 1, and$
  - no marker in $Matchable(\hat{\nu}_i)$ appears more than once in $\hat{\nu}$. 

\qed
Proof. By the definition of an event step, there exist \( \tilde{K}'^n \), \( \tilde{K}'^n \) such that 
\[
\tilde{K}'^n \xrightarrow{\text{RA}} \tilde{K}'^{n-1} \xrightarrow{\text{RA}} \tilde{K}'^n = \tilde{K},
\]
and the Used Marker lemma, \( \tilde{K}'^n \) is an externals-only configuration for all \( i \in 1 \ldots n \). By induction on \( i \) and the Used Marker lemma, \( \text{Used}(\tilde{K}) \subseteq \text{Used}(\tilde{K}'^n) \) for all \( i \in 1 \ldots n \). Then by a final induction on \( i \) and the Externals-Only Label lemma, the necessary conditions hold.

\[ \square \]

Corollary K.7.2. For all event-step executions \( \tilde{K}'^n \xrightarrow{\tilde{l}_{1,i}} \ldots \xrightarrow{\tilde{l}_{i,j}} \tilde{K}'^n \), if \( \tilde{K}'^n \) is an externals-only configuration, then for all labels \( \tilde{l}_{i,j} \) in the execution,

- Matchable(\( \tilde{l}_{i,j} \)) \( \cap \text{Used}(\tilde{K}'^n) = \emptyset \) and
- if \( \tilde{l}_{i,j} = \tilde{a}@H\tilde{v} \) or \( \tilde{l}_{i,j} = \tilde{a}@H!\tilde{v} \) for some \( \tilde{a} \), \( H \), and \( \tilde{v} \), then
  - \( |H| \leq 1 \),
  - for all \( \tilde{a}'@H' \) in \( \tilde{v} \), if \( \tilde{a} \) is internal or \( \tilde{l}_{i,j} = \tilde{a}@H\tilde{v} \), then \( |H'| \leq 1 \), and
  - no marker in Matchable(\( \tilde{l}_{i,j} \)) appears more than once in \( \tilde{v} \).

Proof. By induction on \( i \) and corollary I.42.1 to the Abstract Externals-Only Preservation lemma, every configuration \( \tilde{K}'^n \) in the execution is an externals-only configuration. Also by induction on \( i \) and corollary K.9.1 to the Used Marker lemma, \( \text{Used}(\tilde{K}'^n) \subseteq \text{Used}(\tilde{K}'^n) \) for all \( i \in 1 \ldots n \). Then by a final induction on \( i \) and corollary K.7.1, the necessary conditions hold for all labels in the execution.

\[ \square \]

Corollary K.7.3. For all \( \tilde{K} \), \( L \), \( \tilde{l}_{1,i} \), \( \tilde{l}_{i,j} \), and \( \tilde{K}' \), if

- \( \tilde{K} \) is an externals-only configuration,
- \( \tilde{K} \xrightarrow{L} \tilde{K}' \), and
- \( L \) summarizes \( \tilde{l}_{1,i} \), \( \tilde{l}_{i,j} \),

then for all \( i \in 1 \ldots n \),

- Matchable(\( \tilde{l}_{i,j} \)) \( \cap \text{Used}(\tilde{K}) = \emptyset \) and
- if \( \tilde{l}_{i,j} = \tilde{a}@H\tilde{v} \) or \( \tilde{l}_{i,j} = \tilde{a}@H!\tilde{v} \) for some \( \tilde{a} \), \( H \), and \( \tilde{v} \), then
  - \( |H| \leq 1 \),
  - for all \( \tilde{a}'@H' \) in \( \tilde{v} \), if \( \tilde{a} \) is internal or \( \tilde{l}_{i,j} = \tilde{a}@H\tilde{v} \), then \( |H'| \leq 1 \), and
  - no marker in Matchable(\( \tilde{l}_{i,j} \)) appears more than once in \( \tilde{v} \).

Proof. Let \( \tilde{l}_{i,j} \) be a label in \( \tilde{l}_{1,i} \), \( \tilde{l}_{i,j} \). If \( \tilde{l}_{i,j} = \bullet \), then \( \text{Matchable}(\tilde{l}_{i,j}) = \emptyset \), so we’re done.

If \( \tilde{l}_{i,j} = \tilde{a}@H\tilde{v} \) for some \( \tilde{a} \), \( H \), and \( \tilde{v} \), then \( \tilde{l}_{i,j} \) must be a \text{rcv-ext} label, so \( i = 1 \).

By the definition of \( \tilde{K} \xrightarrow{L} \tilde{K}' \), there exist \( \tilde{p}_{1,i} \ldots \tilde{p}_{m,i} \) such that \( \tilde{K} \xrightarrow{\tilde{p}_{1,i} \ldots \tilde{p}_{m,i}} \tilde{K}' \) and \( \tilde{l}_{i,j} = \tilde{l}_{1,i} \). Then the result holds by corollary K.7.1.
Otherwise, \( |\tilde{l}_i| = \hat{a}@H!\hat{v} \) for some \( \hat{a}, H, \) and \( \hat{v} \). By the definition of summarization, there exists some \( \tilde{v}' \) such that Merge(\( \hat{v}, \tilde{v}' \)) is defined and \( (\hat{a}@H, \tilde{v}') \in \text{dom}(\hat{\mu}) \). If \( \hat{\mu}(\hat{a}@H, \tilde{v}') = \text{single} \), then it must be the case that \( \tilde{v}' = \hat{v} \) and for any sequence \( \tilde{l}'_1, \ldots, \tilde{l}'_m \) summarized by \( L \), there is some \( j \in 1 \ldots m \) such that \( |\tilde{l}'_j| = |\tilde{l}_i| \). We again know that there exist \( \tilde{l}'_1, \ldots, \tilde{l}'_m \) such that \( \tilde{K}_s \longrightarrow \tilde{K}' \), so the results on \( \tilde{l}'_j \) hold by corollary K.7.1. On the other hand, if \( \hat{\mu}(\hat{a}@H, \tilde{v}') = \text{many} \), then by the Empty Matchable Output lemma, \( \text{Matchable}(|\tilde{l}_i|) = \emptyset \).

\[ \blacksquare \]

### K.8 Monitored Matchable Markers Lemma

**Lemma (Monitored Matchable Markers).** For all \( S, S' \), and \( \hat{\lambda} \), if \( S \) is well-formed and \( S \longrightarrow S' \), then \( \text{Mon}(S') \subseteq \text{Mon}(S) \cup \text{Matchable}(\hat{\lambda}) \).

**Proof.** The proof is by case on \( \hat{\lambda} \). If \( \hat{\lambda} = \ast \), the transition \( S \rightarrow S' \) must be a use of the S-FREETRANSITION rule. Then at most one PSM in \( S \) transitions with the rule P-FREETRANSITION, with a transition of the form \( s \longrightarrow s' \). By the definition of that rule, the monitored markers in \( S'' \) and \( s' \) come from \( \text{Mon}(s) \) and the state arguments in \( s \). Because \( S \) is well-formed, so is \( s \), so all of its state arguments are in \( \text{Mon}(s) \). Therefore \( \text{Mon}([s'] \cup S'') \subseteq \text{Mon}(s) \). The other PSMs do not transition, so we have \( \text{Mon}(S') \subseteq \text{Mon}(S) \cup \text{Matchable}(\hat{\lambda}) \).

If \( \hat{\lambda} = \hat{a}@H?\hat{v} \), the transition \( S \longrightarrow S' \) must be a use of the S-SENDORRECEIVE rule. Then every PSM \( s \) in \( S \) has a transition \( s \longrightarrow s' \) via either the P-MONITOREDRECEIVE or P-UNMONITOREDRECEIVE rule. In the P-MONITOREDRECEIVE rule, the monitored markers in \( s' \) and \( S'' \) come from the state arguments of \( s \) and the pattern-matched markers from \( \hat{v} \). As in the previous case, because \( S \) is well-formed, so is \( s \), so all of its state arguments are in \( \text{Mon}(s) \). Therefore \( \text{Mon}([s'] \cup S'') \subseteq \text{Mon}(s) \cup \text{Matchable}(\hat{\lambda}) \). In the P-MONITOREDRECEIVE rule, \( S'' = \emptyset \) and \( s' = s \), so we again have \( \text{Mon}([s'] \cup S'') \subseteq \text{Mon}(s) \cup \text{Matchable}(\hat{\lambda}) \). Therefore, for the entire new configuration \( \hat{S}' \), we have \( \text{Mon}(S') \subseteq \text{Mon}(S) \cup \text{Matchable}(\hat{\lambda}) \).

Finally, if \( \hat{\lambda} = \hat{a}@H!\hat{v} \), the transition \( S \longrightarrow S' \) must be a use of the S-SENDORRECEIVE rule, and every PSM \( s \) in \( S \) has a transition \( s \longrightarrow s' \) via the P-SEND rule. In this case, the new monitored markers in \( S'' \) and \( s' \) come from the pattern-matched markers on \( \hat{v} \), so we have \( \text{Mon}([s'] \cup S'') \subseteq \text{Mon}(s) \cup \text{Matchable}(\hat{\lambda}) \). Therefore, for the entire new configuration \( \hat{S}' \), we have \( \text{Mon}(S') \subseteq \text{Mon}(S) \cup \text{Matchable}(\hat{\lambda}) \).

\[ \blacksquare \]

**Corollary K.8.1.** For all \( S, S' \), and \( \hat{\lambda} \), if \( S \) is well-formed and \( S \longrightarrow S' \), then \( \text{Mon}(S') \subseteq \text{Mon}(S) \cup \text{Matchable}(\hat{\lambda}) \).

**Proof.** If \( \hat{\lambda} = \ast \), then by the definition of the weak-step relation \( \longrightarrow, S \longrightarrow \ldots \longrightarrow S' \). We know that \( \text{Matchable}(\hat{\lambda}) = \emptyset \), so by the Monitored Matchable Mark-
ers lemma and induction on the length of that transition sequence, we have $\Mon(S') \subseteq \Mon(S)$, and therefore $\Mon(S') \subseteq \Mon(S) \cup \Matchable(\lambda)$.

If $\lambda \neq \bullet$, then by the definition of the weak-step relation $\rightarrow$, there exist $S''$ and $S'''$ such that $S \xrightarrow{\lambda} S'' \xrightarrow{\bullet} S'$. By an argument similar to the one for the previous case, $\Mon(S'') \subseteq \Mon(S)$, and therefore $\Mon(S'') \subseteq \Mon(S) \cup \Matchable(\lambda)$. By corollary K.5.2 to the Specification Well-Formed Preservation lemma, $\Mon(S'')$ is well-formed. Then by the Monitored Matchable Markers lemma, $\Mon(S''') \subseteq \Mon(S'') \cup \Matchable(\lambda)$.

By corollary K.5.1 to the Specification Well-Formed Preservation lemma, $\Mon(S'')$ is well-formed. Then again by an argument to the previous case, $\Mon(S') \subseteq \Mon(S''')$. Then we have $\Mon(S') \subseteq \Mon(S'') \subseteq \Mon(S''') \cup \Matchable(\lambda) \subseteq \Mon(S) \cup \Matchable(\lambda)$. Therefore, $\Mon(S') \subseteq \Mon(S) \cup \Matchable(\lambda)$.

$\square$

**K.9 Used Marker Lemma**

**Lemma** (Used Marker). For all $\bar{K}, \bar{K}'$, and $\bar{l}$, if $\bar{K}$ is an externals-only configuration and $\bar{K} \xrightarrow{\bar{l}}_{\text{RA}} \bar{K}'$, then $\Used(\bar{K}) \cup \Matchable(\bar{l}) \subseteq \Used(\bar{K}')$.

**Proof.** The proof is by cases on $\bar{l}$. If $\bar{l}$ is not a **rcv-ext** or **send-ext** label, then $[\bar{l}] = \bullet$, and $\Matchable([\bar{l}]) = \emptyset$. By the definition of the relation enabling $\bar{K} \xrightarrow{\bar{l}}_{\text{RA}} \bar{K}'$, $\Used(\bar{K}) \subseteq \Used(\bar{K}')$ regardless of which transition rule enables the transition, so $\Used(\bar{K}) \cup \Matchable(\bar{l}) \subseteq \Used(\bar{K}')$.

If $\bar{l} = \bar{a} : \text{rcv-ext}(H, \bar{e}, \tau)$, then there exist $H', H''$, and $\bar{v}'$ such that $\Used(\bar{K}) = H'$, $\Used(\bar{K}') = H''$, and $\Mark(\bar{v}', H') = \langle \bar{v}, H'' \rangle$. By the definition of $\Mark$, every address in $\bar{v}'$ not under a **list** or **dict** gets a fresh marker not in $H'$, and $H'' - H'$ is the set of newly added markers in $\bar{v}$. These are precisely the pattern-matchable markers in $\bar{v}$ by the definition of the input-pattern-matching relation, so we have $H'' - H' = \Matchable([\bar{l}])$. By the definition of $\Mark$, we know that $H' \subseteq H''$. Therefore, $\Used(\bar{K}) \cup \Matchable([\bar{l}]) \subseteq \Used(\bar{K}')$.

In the last case, $\bar{l} = \bar{a} : \text{send-ext}(\bar{a}@H, \bar{v}, \tau)$. By the definition of the rule enabling the step $\bar{K} \xrightarrow{\bar{l}}_{\text{RA}} \bar{K}'$, there exists some $\bar{v}'$ in $\bar{K}$ such that either

- $\bar{v}' = \bar{v}$, or

- there exist $H'$ and $H''$ such that $\Used(\bar{K}) = H'$, $\Used(\bar{K}') = H''$, and $\Mark(\bar{v}', H') = \langle \bar{v}, H'' \rangle$.

In the former subcase, because $\bar{K}$ is an externals-only configuration, for all $\bar{d}@H''$ appearing in $\bar{v}'$ where $\bar{d}'$ is internal, $H'' = \emptyset$. The output-pattern-matching relation allows only markers on internal addresses to be matched, so in that case, $\Matchable([\bar{l}]) = \emptyset$. In this case of the **A-SEND-EXTERNAL** rule, we have $\Used(\bar{K}) = \Used(\bar{K}')$. Therefore, $\Used(\bar{K}) \cup \Matchable([\bar{l}]) \subseteq \Used(\bar{K}')$. 


In the latter subcase, by the definition of \( \text{Mark} \), every address in \( \hat{v}' \) not under a list or dict gets a fresh marker not in \( H' \), and \( H'' - H' \) is the set of newly added markers in \( \hat{v} \). In this case, those fresh markers applied to internal addresses are the matchable ones, so we have \( \text{Matchable}(\hat{I}) \subseteq H'' - H' \). By the definition of \( \text{Mark} \), \( H' \subseteq H'' \). Therefore, \( \text{Used}(\hat{K}) \cup \text{Matchable}(\hat{I}) \subseteq \text{Used}(\hat{K}') \). \( \square \)

**Corollary K.9.1.** For all \( \hat{K}, \hat{K}', \hat{I}_1, \ldots, \hat{I}_n \), if \( \hat{K} \) is an externals-only configuration and \( \hat{K} \xrightarrow{\hat{I}_1, \ldots, \hat{I}_n} \hat{K}' \), then \( \text{Mon}((\hat{I}_1) \cup \text{Matchable}(\hat{I}_1)) \subseteq \text{Used}(\hat{K}') \).

**Proof.** By the definition of the event-step relation, there exist \( \hat{K}_1'' \ldots, \hat{K}_n'' \) such that \( \hat{K}_1'' \xrightarrow{\hat{I}_1} \hat{K}_2'' \ldots \xrightarrow{\hat{I}_n} \hat{K}_n'' = \hat{K} \), and \( \hat{K}_n'' = \hat{K}' \). By induction on \( i \) and the Used Marker lemma, \( \text{Used}(\hat{K}_1) \cup \text{Matchable}(\hat{I}_1) \subseteq \text{Used}(\hat{K}_1) \). Then because \( \hat{K}_n'' = \hat{K}' \), \( \text{Used}(\hat{K}_1) \cup \text{Matchable}(\hat{I}_1) \subseteq \text{Used}(\hat{K}) \). \( \square \)

**K.10  Used/Monitored Marker Lemma**

**Lemma** (Used/Monitored Marker). For all \( \hat{K}, \hat{K}', \hat{I}, S, \) and \( S' \), if

- \( \text{Mon}(S) \subseteq \text{Used}(\hat{K}) \),
- \( \hat{K} \) is an externals-only configuration,
- \( S \) is well-formed,
- \( \hat{K} \xrightarrow{\hat{I}} \hat{K}' \), and
- \( S \xrightarrow{\hat{I}} S' \),

then \( \text{Mon}(S') \subseteq \text{Used}(\hat{K}') \).

**Proof.** By the corollary to the Monitored Matchable Markers lemma, \( \text{Mon}(S') \subseteq \text{Mon}(S) \cup \text{Matchable}(\hat{I}) \). By the Used Marker lemma, \( \text{Used}(\hat{K}) \cup \text{Matchable}(\hat{I}) \subseteq \text{Used}(\hat{K}') \). Then because \( \text{Mon}(S) \subseteq \text{Used}(\hat{K}) \), we have \( \text{Mon}(S') \subseteq \text{Used}(\hat{K}') \). \( \square \)

**Corollary K.10.1.** For all \( \hat{K}, \hat{K}', L, O, S, \) and \( S' \), if

- \( \text{Mon}(S) \subseteq \text{Used}(\hat{K}) \),
- \( \hat{K} \) is an externals-only configuration,
- \( S \) is well-formed,
- \( \hat{K} \xrightarrow{L} \hat{K}' \), and
- \( S \xrightarrow{(L,O)} S' \),
then \( \text{Mon}(S') \subseteq \text{Used}(\tilde{K}') \).

**Proof.** By the definition of the summary-transition relation, there exist \( \hat{1}, \ldots, \hat{n} \) such that \( \hat{K} \hspace{0.5em} \hat{1} \ldots \hat{n} \hspace{0.5em} \hat{K}' \). Then by the definition of the event-step relation, there exist \( \hat{K}_1', \ldots, \hat{K}_{n+1}' \) such that

- \( \hat{K}_1' = \hat{K} \),
- \( \hat{K}_{n+1}' = \hat{K}' \), and
- \( \hat{K}_1' \xrightarrow{\text{RA}} \cdots \xrightarrow{\text{RA}} \hat{K}_{n+1}' \).

By induction on \( i \) and the Abstract Externals-Only Preservation lemma, \( \hat{K}_i'' \) is an externals-only configuration for all \( i \in 1 \ldots n+1 \).

By the definitions of the summary-transition relation for specification configurations and the \( \rightarrow \) relation, there exist \( S_1'', \ldots, S_{n+1}'' \) such that

- \( S_1'' = S \),
- \( S_{n+1}'' = S' \), and
- \( S_1'' \xrightarrow{\hat{1}} \cdots \xrightarrow{\hat{n}} S_{n+1}'' \).

Then by induction on \( i \) and the Used/Monitored Marker lemma, for all \( i \in 1 \ldots n+1 \), \( \text{Mon}(S_i'') \subseteq \text{Used}(\hat{K}_i'') \). Therefore, \( \text{Mon}(S') \subseteq \text{Used}(\hat{K}') \). \( \square \)

**K.11 Distinct-Marker Silent Preservation Lemma**

**Lemma** (Distinct-Marker Silent Preservation). For all \( S \) and \( S' \), if

- \( S \rightarrow S' \),
- \( S \) is well-formed, and
- all of the PSMs in \( S \) monitor distinct markers,

then all of the PSMs in \( S' \) monitor distinct markers and \( \text{Mon}(S') = \text{Mon}(S) \).

**Proof.** The transition \( S \rightarrow S' \) must be a use of the S-FREETRANSITION rule from section 3.6.7. Therefore, there exist \( s, s', S'' \), and \( S''' \) such that

- \( S = \{s\} \cup S'' \),
- \( s \xrightarrow{s''} s' \),
- \( S' = \{s'\} \cup S'' \cup S''' \).
APPENDIX K. PROOFS FOR TRANSFORMATION CONFORMANCE

That PSM transition must correspond to a use of the P-FREE_TRANSITION rule from section 3.6.5. By the definition of that rule and the PerformAll function, none of the forked PSMs in $S''$ have any input-monitored markers, and by the definition of the Dist function, the output-monitored sets for all of the PSMs in $\{s\} \cup S$ are disjoint. Therefore, all of the PSMs in $\{s\} \cup S''$ monitor distinct markers.

Also by the definition of the P-FREE_TRANSITION rule and PerformAll, the output-monitored markers in $\{s\} \cup S''$ come from $\text{Mon}(s)$ and the state arguments in $s$. Because $S$ is well-formed, so is $s$, so therefore $s$ output-monitors all of its state arguments. As a result, the PSMs $\{s\} \cup S''$ monitor only markers that $s$ monitors, so all of the PSMs in $\{s\} \cup S'' \cup S''$ (and therefore $S'$) monitor distinct markers.

By the above arguments, the markers appearing in $S'$ are the same as those appearing in $S$. □

K.12 Distinct-Marker Preservation Lemma

Lemma (Distinct-Marker Preservation). For all $S$, $S'$, and $\hat{\lambda}$, if

- $S \xrightarrow{\hat{\lambda}} S'$,
- $S$ is well-formed,
- all of the PSMs in $S$ monitor distinct markers,
- if there exist $\hat{a}$, $H$, and $\hat{v}$ such that $\hat{\lambda} = \hat{a} @ H? \hat{v}$ or $\hat{\lambda} = \hat{a} @ H! \hat{v}$, then $|H| \leq 1$ and no marker in Matchable($\hat{\lambda}$) appears more than once in $\hat{v}$,
- $\text{Matchable}(\hat{\lambda}) \cap \text{Mon}(S) = \emptyset$

then all of the PSMs in $S'$ monitor distinct markers.

Proof. The proof is by cases on the shape of $\hat{\lambda}$.

Case: $\hat{\lambda} = \bullet$

By the Distinct-Marker Silent Preservation lemma, all of the PSMs in $S'$ monitor distinct markers.

Case: $\hat{\lambda} = \hat{a} @ H? \hat{v}$

The transition $S \xrightarrow{\hat{\lambda}} S'$ must be a use of the S-SEND_OR_RECEIVE rule from section 3.6.7, so there exist $s_1, \ldots, s_n$, $s'_1, \ldots, s'_n$, $s''_1, \ldots, s''_n$, and $O_1, \ldots, O_n$ such that

- $S = \{s_1, \ldots, s_n\}$,
- $S' = \{s'_1, \ldots, s'_n\} \cup S''_1 \cup \ldots \cup S''_n$, and
Because each of the PSMs in $S$ monitor distinct markers, and because $|H| \leq 1$, there is at most one $k \in 1 \ldots n$ such that $H \cap \text{InMon}(s_k) \neq \emptyset$. If there is no such $k$, then each of the $s_i \xrightarrow{\hat{a} @ H \cap \text{O}_i, S''} s'_i$ transitions must correspond to a use of the $\text{P-UNMONITORED-RECEIVE}$ rule. Therefore, for all $i \in 1 \ldots n$, $s'_i = s_i$ and $S''_i = \emptyset$. As a result, $S' = S$, so all PSMs in $S'$ monitor distinct markers.

Otherwise, all of the PSMs in $S$ except $s_k$ transition with $\text{P-UNMONITORED-RECEIVE}$ as in the previous case, while $s_k$ transitions with the $\text{P-MONITORED-RECEIVE}$ rule. By the definition of that rule and the $\text{PerformAll}$ function, none of the forked PSMs in $S''_k$ have any input-monitored markers, and by the definition of the $\text{Dist}$ function, the output-monitored sets for all of the PSMs in $\{s'_k\} \cup S''_k$ are disjoint. Therefore, all of the PSMs in $\{s'_k\} \cup S''_k$ monitor distinct markers.

By the definition of the $\text{P-MONITORED-RECEIVE}$ rule, any new monitored markers come from the matched markers on $\hat{v}$, and therefore $\text{Mon}(S''_k \cup \{s'_k\}) - \text{Mon}(s_k) \subseteq \text{Matchable}(\hat{\lambda})$. Because $\text{Matchable}(\hat{\lambda}) \cap \text{Mon}(S) = \emptyset$, $\text{Mon}(S''_k \cup \{s'_k\}) \cap \text{Mon}(\{s_1, \ldots, s_{k-1}, s_{k+1}, \ldots, s_n\}) = \emptyset$. Therefore, all of the PSMs in $S'$ monitor distinct markers.

**Case:** $\hat{\lambda} = \hat{a} @ H \cap \hat{v}$

The transition $S \xrightarrow{\hat{\lambda}} S'$ must be a use of the $\text{S-SEND-OR-RECEIVE}$ rule from section 3.6.7, so there exist $s_1, \ldots, s_n, s'_1, \ldots, s'_n, S'_1, \ldots, S'_n$, and $O_1, \ldots, O_n$ such that

- $S = \{s_1, \ldots, s_n\}$,
- $S' = \{s'_1, \ldots, s'_n\}$, and
- $s_i \xrightarrow{\hat{a} @ H \cap \text{O}_i, S''} s'_i$ for all $i \in 1 \ldots n$.

Furthermore, each transition $s_i \xrightarrow{\hat{a} @ H \cap \text{O}_i, S''} s'_i$ must be a use of the rule $\text{P-SEND}$. Because each of the PSMs in $S$ monitor distinct markers and because $|H| \leq 1$, there is at most one $k \in 1 \ldots n$ such that $H \cap \text{OutMon}(s_k) \neq \emptyset$. If there is no such PSM, then for all $i \in 1 \ldots n$, $s'_i = s_i$ and $S''_i = \emptyset$. As a result, $S' = S$, so all PSMs in $S'$ monitor distinct markers.

Otherwise, by the definition of the $\text{P-SEND}$ rule, there exists some $po$ and $H'$ such that $\hat{v} \sim po \Rightarrow H', S''_k$ and $H' = \text{Mon}(s'_k) - \text{Mon}(s_k)$. Therefore, $\text{Mon}(S''_k \cup \{s'_k\}) - \text{Mon}(s_k) \subseteq \text{Matchable}(\hat{\lambda})$. Because $\text{Matchable}(\hat{\lambda}) \cap \text{Mon}(S) = \emptyset$, we also have that $\text{Mon}(S''_k \cup \{s'_k\}) \cap \text{Mon}(\{s_1, \ldots, s_{k-1}, s_{k+1}, \ldots, s_n\}) = \emptyset$.

By the definition of the pattern-matching rules, each of the forked PSMs in $S''_k$ has no output-monitored markers. Also by the definition of those rules, each forked PSM’s input monitored marker as well as the markers in $H'$ must come from a distinct location in $\hat{v}$. No marker in $\text{Matchable}(\hat{\lambda})$ appears more than once in $\hat{v}$, so all the PSMs in $S''_k \cup \{s'_k\}$ monitor distinct markers.
For all of the other \( i \in 1 \ldots n \) such that \( i \neq k \), \( s'_i = s_i \) and \( S''_i = \emptyset \) as in the previous case, so we have \( S' = \{ s_1, \ldots, s_{k-1}, s'_k, s_{k+1}, \ldots, s_n \} \cup S''_k \). Then because \( \text{Mon}(S''_k \cup \{ s'_k \}) \cap \text{Mon}(\{ s_1, \ldots, s_{k-1}, s_{k+1}, \ldots, s_n \}) = \emptyset \), all of the PSMs in \( S' \) monitor distinct markers.

**Corollary K.12.1.** For all \( S, S' \), and \( \hat{\lambda} \), if

- \( S \xrightarrow{\hat{\lambda}} S' \),
- \( S \) is well-formed,
- all of the PSMs in \( S \) monitor distinct markers,
- if there exist \( \hat{\alpha}, H, \) and \( \hat{\nu} \) such that \( \hat{\lambda} = \hat{\alpha}@H?\hat{\nu} \) or \( \hat{\lambda} = \hat{\alpha}@H!\hat{\nu} \), then \( |H| \leq 1 \) and no marker in \( \text{Matchable}(\hat{\lambda}) \) appears more than once in \( \hat{\nu} \),
- \( \text{Matchable}(\hat{\lambda}) \cap \text{Mon}(S) = \emptyset \)

then all of the PSMs in \( S' \) monitor distinct markers.

**Proof.** There are two cases, depending on the value of \( \hat{\lambda} \). If \( \hat{\lambda} = \bullet \), then by the definition of the weak-step relation \( \rightarrow \), there exist \( S''_1, \ldots, S''_n \) such that \( S''_1 = S, S''_n = S' \), and \( S''_1 \xrightarrow{\bullet} \ldots \xrightarrow{\bullet} S''_n \). By induction and corollary K.5.1 of the Specification Well-Formed Preservation lemma, \( S''_i \) is well-formed for all \( i \in 1 \ldots n \). Then by induction and the Distinct-Marker Silent Preservation lemma, all of the PSMs in \( S''_i \) monitor distinct markers for all \( i \in 1 \ldots n \).

In the other case, \( \hat{\lambda} \neq \bullet \). Then by the definition of the weak-step relation \( \rightarrow \), there exist \( S''_1, \ldots, S''_n \) and some \( k \) such that \( S''_1 = S, S''_n = S' \), and \( S''_1 \xrightarrow{\hat{\lambda}} \ldots \xrightarrow{\hat{\lambda}} S''_k \xrightarrow{\hat{\lambda}} \ldots \xrightarrow{\hat{\lambda}} S''_n \). Again by induction and corollary K.5.1 of the Specification Well-Formed Preservation lemma, \( S''_i \) is well-formed for all \( i \in 1 \ldots n \).

By induction and the Distinct-Marker Silent Preservation lemma, all of the PSMs in \( S''_i \) monitor distinct markers and \( \text{Mon}(S''_i) = \text{Mon}(S) \) for all \( i \in 1 \ldots k \). Then by the Distinct-Marker Preservation lemma, all of the PSMs in \( S''_k \) monitor distinct markers. Finally, again by induction and the Distinct-Marker Silent Preservation lemma, all of the PSMs in \( S''_i \) monitor distinct markers for all \( i \in k + 1 \ldots n \). Then because \( S''_n = S' \), all the PSMs in \( S' \) monitor distinct markers.

**Corollary K.12.2.** For all \( S, S', \hat{K}, \hat{K}', L, \) and \( O \), if

- \( S \xrightarrow{\langle L, O \rangle} S' \),
- \( S \) is well-formed,
- all of the PSMs in \( S \) monitor distinct markers,
- \( \hat{K} \) is an externals-only configuration,
K.12. DISTINCT-MARKER PRESERVATION LEMMA

• Mon(S) ⊆ Used(\(\hat{K}\)), and
• \(\hat{K} \xrightarrow{L} \hat{K}'\),

then all of the PSMs in \(S'\) monitor distinct markers.

Proof. By the definition of summary transitions and abstract event steps, there exist \(\hat{K}'_1, \ldots, \hat{K}'_{n+1}\) and \(\hat{l}_1, \ldots, \hat{l}_n\) such that

• \(\hat{K}'_1 = \hat{K}\),
• \(\hat{K}'_{n+1} = \hat{K}'\), and
• \(\hat{K}'_1 \xrightarrow{\text{RA}} \cdots \xrightarrow{\text{RA}} \hat{K}'_{n+1}\).

Similarly, by the definitions of summary transitions for specification configurations and the \(\xrightarrow{L}\) relation, there exist \(S''_1, \ldots, S''_{n+1}\) such that

• \(S''_1 = S\),
• \(S''_{n+1} = S'\), and
• \(S''_1 \xrightarrow{\lfloor \hat{l}_1 \rfloor} \cdots \xrightarrow{\lfloor \hat{l}_n \rfloor} S''_{n+1}\).

By induction and the Abstract Externals-Only Preservation lemma, \(\hat{K}'_i\) is an externals-only configuration for all \(i \in 1 \ldots n + 1\). Then by the Externals-Only Label lemma, for all \(i \in 1 \ldots n\), Matchable(\(\lfloor \hat{l}_i \rfloor\)) ∩ Used(\(\hat{K}'_i\)) = φ, and if there exist \(\hat{a}\), \(H\), and \(\hat{v}\) such that \(\lfloor \hat{l}_i \rfloor = \hat{a}@H?\hat{v}\) or \(\lfloor \hat{l}_i \rfloor = \hat{a}@H!\hat{v}\), then |\(H\)| ≤ 1 and no marker in Matchable(\(\lfloor \hat{l}_i \rfloor\)) appears more than once in \(\hat{v}\). Also, by induction and corollary K.5.2 of the Specification Well-Formed Preservation lemma, \(S''_i\) is well-formed for all \(i \in 1 \ldots n + 1\).

By induction and the Used/Monitored Marker lemma, for all \(i \in 1 \ldots n + 1\), Mon(\(S''_i\)) ⊆ Used(\(\hat{K}'_i\)), and therefore Matchable(\(\lfloor \hat{l}_i \rfloor\)) ∩ Mon(\(S''_i\)) = φ. Then finally, by induction and corollary K.12.1 of this lemma, all of the PSMs in \(S''_i\) monitor distinct markers for all \(i \in 1 \ldots n + 1\). Because \(S''_{n+1} = S'\), all of the PSMs in \(S'\) monitor distinct markers, which completes the proof.

Corollary K.12.3. For all summary executions \(\hat{K}_1 \xrightarrow{L_1} \cdots \) and \(S_1 \xrightarrow{(L_1, O_1)} \cdots\), if

• \(S_1\) is well-formed,
• all the PSMs in \(S_1\) monitor distinct markers,
• \(\hat{K}\) is an externals-only configuration,
• Mon(\(S_1\)) ⊆ Used(\(\hat{K}\)), and
• \(\text{len}\left(\hat{K}_1 \xrightarrow{L_1} \cdots\right) = \text{len}\left(S_1 \xrightarrow{(L_1, O_1)} \cdots\right)\),
then for all configurations \( S_i \) in \( S_1 \xrightarrow{(L_1,O_1)} \ldots \), all the PSMs in \( S_1 \) monitor distinct markers.

**Proof.** By induction and the corollary to the Abstract Externals-Only Preservation lemma, every configuration \( \bar{K}_j \) in \( \bar{K}_1 \xrightarrow{L_1} \ldots \) is an externals-only configuration. Also by induction and corollary K.5.4 of the Specification Well-Formed Preservation lemma, every configuration \( S_j \) in \( S_1 \xrightarrow{(L_1,O_1)} \ldots \) is well-formed. Finally, by induction and the corollary to the Used/Monitored Marker lemma, \( \text{Mon}(S_j) \subseteq \text{Used}(\bar{K}_j) \) for each corresponding \( S_j \) and \( \bar{K}_j \) in each execution.

The rest of the proof is by induction on \( i \). In the base case where \( i = 1 \), we already know that all the PSMs in \( S_1 \) monitor distinct markers. For the inductive case, let there be \( i > 1 \) such that \( S_i \) is a configuration in the execution, and assume by the induction hypothesis that all the PSMs in \( S_{i-1} \) monitor distinct markers. By corollary K.12.2 to this lemma, all the PSMs in \( S_i \) monitor distinct markers. \( \square \)

### K.13 Silent-Step Partition Lemma

**Lemma** (Silent-Step Partition). For all \( S_1, S_2, \) and \( S' \) such that

- \( S_1 \cup S_2 \xrightarrow{s,\varnothing} S' \),
- \( S_1 \cup S_2 \) is well-formed, and
- \( \text{Mon}(S_1) \cap \text{Mon}(S_2) = \varnothing \),

there exist \( S_1'' \) and \( S_2'' \) such that

- \( S' = S_1'' \cup S_2'' \),
- \( \text{Mon}(S_1'') \cap \text{Mon}(S_2'') = \varnothing \), and
- either \( S_i = S_i'' \) or \( S_i \xrightarrow{s,\varnothing} S_i'' \) for \( i \in 1,2 \).

**Proof.** The transition \( S_1 \cup S_2 \xrightarrow{s,\varnothing} S' \) must be an instance of the S-FREETRANSITION rule, so there must exist \( S'', S''' \), \( s \), and \( s' \) such that

- \( S'' \cup \{s\} = S_1 \cup S_2 \),
- \( s \xrightarrow{\varnothing, S''} s' \), and
- \( S' = S'' \cup \{s'\} \cup S''' \).

The transition \( s \xrightarrow{\varnothing, S''} s' \) must be an instance of the P-FREETRANSITION rule. By the definition of that rule, the monitored markers in \( S''' \) and \( s' \) come from \( \text{Mon}(s) \) and the state arguments in \( s \). Because \( S_1 \cup S_2 \) is well-formed, so is \( s \), so all of its state arguments are in \( \text{Mon}(s) \) and therefore \( \text{Mon}(\{s'\} \cup S'''') \subseteq \text{Mon}(s) \).
Also by the definition of that rule, the transitioned PSM does not remove any of its monitored markers, so we in fact have $\text{Mon}(\{s\} \cup S^m) = \text{Mon}(s)$.

Suppose that $s \in S_1 \cap S_2$. Then because $\text{Mon}(S_1) \cap \text{Mon}(S_2) = \emptyset$, it must be the case that $\text{Mon}(s) = \emptyset$, and therefore $\text{Mon}(\{s\} \cup S^m) = \emptyset$. Let $S'' = (S_1 - \{s\}) \cup \{s\} \cup S^m$ for $i \in \{1, 2\}$. Then we have that $\text{Mon}(S''_i) = \text{Mon}(S_i)$ for $i \in \{1, 2\}$, so $\text{Mon}(S''_1) \cap \text{Mon}(S''_2) = \emptyset$. Furthermore, we have $S' = S''_1 \cup S''_2$, and $S_i \xrightarrow{\hat{\lambda}} S''_i$ for $i \in \{1, 2\}$ by the definition of the S-FREE transition rule.

Otherwise, either $s \in S_1 - S_2$ or $s \in S_2 - S_1$. We prove the lemma for the first case here; the other is symmetrical. In this case, $S' = S''_1 \cup S_2 \cup \{s\} \cup S^m$. Let $S''_1 = S'' \cup \{s\} \cup S^m$, and let $S''_2 = S_2$; then we have $S'' \cup S''_2 = S'$, and by the definition of the S-FREE transition rule, $S_1 \xrightarrow{\cdots \hat{\lambda}} S''_1$. Finally, we have $S' = S'' \cup \{s\}$, so $\text{Mon}(S_1) = \text{Mon}(S'' \cup \{s\}) = \text{Mon}(S'' \cup \{s\} \cup S^m) = \text{Mon}(S''_1)$. Then because $\text{Mon}(S_1) \cap \text{Mon}(S_2) = \emptyset$, we have $\text{Mon}(S''_1) \cap \text{Mon}(S''_2) = \emptyset$, which completes the proof. □

### K.14 Communication-Step Partition Lemma

**Lemma** (Communication-Step Partition). For all $S_1$, $S_2$, $S'$, $\hat{\lambda}$, and $O$, if

- $S_1 \cup S_2 \xrightarrow{\hat{\lambda}, O} S'$,
- $S_1 \cup S_2$ is well-formed,
- $\text{Mon}(S_1) \cap \text{Mon}(S_2) = \emptyset$,
- $\hat{\lambda} \neq \ast$,
- there exist $\hat{\alpha}$, $H$, and $\hat{\nu}$ such that $|H| \leq 1$ and either $\hat{\lambda} = \hat{\alpha}@H?\hat{\nu}$ or $\hat{\lambda} = \hat{\alpha}@H!\hat{\nu}$,
- Matchable($\hat{\lambda}$) $\cap$ Mon($S_1 \cup S_2$) = $\emptyset$,

then there exist $O'_1$, $O'_2$, $S''_1$, and $S''_2$ such that

- $O = O'_1 \uplus O'_2$,
- $S' = S''_1 \cup S''_2$,
- $\text{Mon}(S''_1) \cap \text{Mon}(S''_2) = \emptyset$, and
- $S_i \xrightarrow{\hat{\lambda}, O'} S''_i$ for $i \in \{1, 2\}$.

**Proof.** Because $\hat{\lambda} \neq \ast$, the transition $S_1 \cup S_2 \xrightarrow{\hat{\lambda}, O} S'$ must be a use of the S-SEND or S-RECEIVE rule. By the definition of this rule, there exist $s_1, \ldots, s_n$, $s'_1, \ldots, s'_n$, $O'_1, \ldots, O'_n$, and $S''_1, \ldots, S''_n$ such that

- $S_1 \cup S_2 = \{s_1, \ldots, s_n\}$,
• \( s_i \xrightarrow{\lambda, O''_i, S''} s'_i \) for all \( i \in 1 \ldots n \),

• \( O = \bigcup_{i=1}^n O''_i \), and

• \( S' = \{ s'_1, \ldots, s'_n \} \cup S''_1 \cup \ldots \cup S''_n \).

If \( H \cap \text{Mon}(S_1 \cup S_2) = \emptyset \), then for all \( i \in 1 \ldots n \), the transition \( s_i \xrightarrow{\lambda, O''_i, S''} s'_i \) must be a use of either the P-\textsc{UnmonitoredReceive} or P-\textsc{Send} rule. Therefore for all \( i \in 1 \ldots n \), \( O''_i = \emptyset \), \( S''_i = \emptyset \), and \( s'_i = s_i \). Let \( O'_j = \emptyset \) and \( S''_j = S_j \) for \( j \in \{1, 2\} \). Then we have

• \( O = O'_1 \cup O'_2 \),

• \( S' = S''_1 \cup S''_2 \),

• \( \text{Mon}(S''_1) \cap \text{Mon}(S''_2) = \emptyset \), and

• \( S_j \xrightarrow{\lambda, O'_j} S''_j \) for \( j \in \{1, 2\} \) (by the S-\textsc{SendOrReceive} rule).

Otherwise, \( H \cap \text{Mon}(S_1 \cup S_2) \neq \emptyset \). We know \( |H| \leq 1 \), so there exists some \( \eta \) such that \( H = \{ \eta \} \). Because \( \text{Mon}(S_1) \cap \text{Mon}(S_2) = \emptyset \), either \( \eta \in \text{Mon}(S_1) \) or \( \eta \in \text{Mon}(S_2) \), but not both. Assume that \( \eta \in \text{Mon}(S_1) \); the proof for the other case is symmetric.

Because \( \eta \notin \text{Mon}(S_2) \), for all \( s_i \in S_2 \), as in the previous we have \( O''_i = \emptyset \), \( S''_i = S_2 \), \( s'_i = s_i \). Let \( S''_2 = S_2 \) and let \( O'_2 = \emptyset \). Then we have \( S_2 \xrightarrow{\lambda, O'_2} S''_2 \) by the S-\textsc{SendOrReceive} rule.

Let \( O'_1 = O \) and let \( S''_1 \) be the set containing all \( s'_i \) such that \( s_i \in S_1 \). Then we have

• \( O = O'_1 \cup O'_2 \),

• \( S' = S''_1 \cup S''_2 \), and

• \( S_1 \xrightarrow{\lambda, O'_1} S''_1 \) (by the S-\textsc{SendOrReceive} rule).

It remains to show that \( \text{Mon}(S''_1) \cap \text{Mon}(S''_2) = \emptyset \). By the Monitored Matchable Markers lemma, \( \text{Mon}(S''_1) \subseteq \text{Mon}(S_1) \cup \text{Matchable}(\lambda) \). We already know that \( \text{Mon}(S_1) \cap \text{Mon}(S_2) = \emptyset \) and \( \text{Matchable}(\lambda) \cap \text{Mon}(S_2) = \emptyset \). Then because \( S''_2 = S_2 \), we have \( \text{Mon}(S''_1) \cap \text{Mon}(S''_2) = \emptyset \). \[ \square \]

K.15 Weak-Silent-Step Partition Lemma

Lemma (Weak-Silent-Step Partition). For all \( S_1, S_2, \) and \( S' \), if

• \( S_1 \cup S_2 \xrightarrow{\beta} S' \),

• \( S_1 \cup S_2 \) is well-formed, and
Proof: By the definition of $S_1 \cup S_2 \stackrel{\cdot \phi}{\longrightarrow} S'$, there exist $S_1''$ such that

- $S_1'' = S_1 \cup S_2$,
- $S_n'' = S'$, and
- $S_1'' \stackrel{\cdot \phi}{\longrightarrow} \cdots \stackrel{\cdot \phi}{\longrightarrow} S_n''$.

By induction and corollary K.5.1 to the Specification Well-Formed Preservation lemma, $S_n''$ is well-formed for all $i \in 1 \ldots n$.

Let $S_i'' = S_i$ for $i \in [1, 2]$. By induction and the Silent-Step Partition lemma, there exist $S_{i,1}''$, $S_{i,2}''$, $S_{1,n}''$ and $S_{2,n}''$ such that for all $j \in 2 \ldots n$,

- $S_j'' = S_{i,j}'' \cup S_{2,j}''$,
- $\text{Mon}(S_{i,j}'') \cap \text{Mon}(S_{2,j}'') = \emptyset$, and
- either $S_{i,j}''' = S_{i,j-1}'''$ or $S_{i,j-1}''' \stackrel{\cdot \phi}{\longrightarrow} S_{i,j}'''$ for $i \in 1, 2$.

Let $S_1'' = S_{1,1}''$ and $S_2'' = S_{2,1}''$. Then we have $S' = S_1'' \cup S_2''$ and $\text{Mon}(S_1'') \cap \text{Mon}(S_2'') = \emptyset$. It remains to show that $S_i \stackrel{\cdot \phi}{\longrightarrow} S_i''$ for $i \in [1, 2]$.

For $i \in [1, 2]$, we know that there exists a sequence of configurations $S_{i,1}''$, $S_{i,2}''$ such that for all $j \in 2 \ldots n$, either $S_{i,j}''' = S_{i,j-1}'''$ or $S_{i,j-1}''' \stackrel{\cdot \phi}{\longrightarrow} S_{i,j}'''$. Then for $i \in [1, 2]$, because $S_{i,1}''' = S_i$ and $S_{i,2}''' = S_i''$, we have $S_i \stackrel{\cdot \phi}{\longrightarrow} \cdots \stackrel{\cdot \phi}{\longrightarrow} S_i'''$. Therefore, $S_i \stackrel{\cdot \phi}{\longrightarrow} S_i''$ for $i \in [1, 2]$.

\[ \Box \]

K.16 Specification Weak-Step Partition Lemma

Lemma (Specification Weak-Step Partition). For all $S_1$, $S_2$, $S'$, $\lambda$, and $O$, if

- $S_1 \cup S_2 \stackrel{\lambda, O}{\longrightarrow} S'$,
- $S_1 \cup S_2$ is well-formed,
- $\text{Mon}(S_1) \cap \text{Mon}(S_2) = \emptyset$,
• if there exist \( \hat{a}, H, \) and \( \hat{v} \) such that \( \hat{\lambda} = \hat{a}@H?\hat{v} \) or \( \hat{\lambda} = \hat{a}@H!\hat{v} \), then \( |H| \leq 1 \),

• Matchable(\( \hat{\lambda} \)) \( \cap \) Mon(\( S_1 \cup S_2 \)) = \( \emptyset \),

then there exist \( O'_1, O'_2, S''_1, \) and \( S''_2 \) such that

• \( O = O'_1 \cup O'_2 \),

• \( S' = S''_1 \cup S''_2 \),

• Mon(S''_1) \( \cap \) Mon(S''_2) = \( \emptyset \), and

• \( S_i \xrightarrow{\hat{\lambda}} S''_i \) for \( i \in \{1, 2\} \).

Proof. If \( \hat{\lambda} = \ast \), then \( O = \emptyset \) by the definition of the weak-step transition relation \( \rightarrow \). By the Weak-Silent-Step Partition lemma, there exist \( S''_1 \) and \( S''_2 \) such that

• \( S' = S''_1 \cup S''_2 \),

• Mon(S''_1) \( \cap \) Mon(S''_2) = \( \emptyset \), and

• \( S_i \xrightarrow{\ast} S''_i \) for \( i \in \{1, 2\} \).

Let \( O'_1 = \emptyset \) and \( O'_2 = \emptyset \) to complete the proof.

Otherwise, if \( \hat{\lambda} \neq \ast \), then by the definition of the weak-step transition relation, there exist \( S''' \) and \( S'''' \) such that \( S_1 \cup S_2 \xrightarrow{} S''' \xrightarrow{O} S'''' \xrightarrow{} S' \). By corollaries K.5.1 and K.5.2 to the Specification Well-Formed Preservation lemma, \( S''' \) and \( S'''' \) are well-formed. By the Weak-Silent-Step Partition lemma, there exist \( S''_1 \) and \( S''_2 \) such that

• \( S'' = S''_1 \cup S''_2 \),

• Mon(S''_1) \( \cap \) Mon(S''_2) = \( \emptyset \), and

• \( S''' \xrightarrow{\hat{\lambda}} S''_i \) for \( i \in \{1, 2\} \).

Then by the Communication-Step Partition lemma, there exist \( O'_1, O'_2, S'''_1 \) and \( S'''_2 \) such that

• \( O = O'_1 \cup O'_2 \),

• \( S''' = S'''_1 \cup S'''_2 \),

• Mon(S'''_1) \( \cap \) Mon(S'''_2) = \( \emptyset \), and

• \( S'''_i \xrightarrow{\hat{\lambda}} S''_i \) for \( i \in \{1, 2\} \).

Then by the Weak-Silent-Step Partition lemma again, there exist \( S''_1 \) and \( S''_2 \) such that...
K.17. MATCHABLE LABELS LEMMA

Lemma (Matchable Labels). For all $\hat{K}, \hat{K}'$, $L$, $\hat{l}_1, \ldots, \hat{l}_n$, and $i \in 1 \ldots n$, if

- $\hat{K} \overset{L}{\rightarrow} \hat{K}'$,
- $\hat{K}$ is an externals-only configuration,
- $L$ summarizes $\hat{l}_1, \ldots, \hat{l}_n$, and
- $\text{Matchable}([\hat{l}_i]) \neq \emptyset$,

then there exist $\hat{K}_n', \ldots, \hat{K}_m'$, $\hat{l}_1, \ldots, \hat{l}_n$, and $j \in 1 \ldots m$ such that

\[
\hat{K}_1' \overset{\hat{l}_1}{\rightarrow}_{\text{RA}} \cdots \overset{\hat{l}_m}{\rightarrow}_{\text{RA}} \hat{K}_m' = K, \text{ and } \hat{l}_j = \hat{l}_i.
\]

Proof. Because $\text{Matchable}([\hat{l}_i]) \neq \emptyset$, it must be the case that there exist some $\hat{a}$, $H$, and $\hat{v}$ such that either $[\hat{l}_i] = \hat{a} @ H \hat{v}$ or $[\hat{l}_i] = \hat{a} @ @ H \hat{v}$. In the former case, $\hat{l}_i$ must be a cv-ext label, so by the definition of what it means for $L$ to summarize $\hat{l}_1, \ldots, \hat{l}_n$, $i = 1$. Then by the definition of the summary-transition and event-step relations, there exists a transition sequence $\hat{K}_1' \overset{\hat{l}_1}{\rightarrow}_{\text{RA}} \cdots \overset{\hat{l}_m}{\rightarrow}_{\text{RA}} \hat{K}_m'$ such that $\hat{K}_1' = \hat{K}$ and $\hat{l}_1 = \hat{l}_1$. Let $j = 1$ and we're done.

Otherwise, $[\hat{l}_i] = \hat{a} @ H \hat{v}$. Let $\langle \hat{l}_i, \hat{\mu} \rangle = L$. By the definition of what it means for $L$ to summarize $\hat{l}_1, \ldots, \hat{l}_n$, there must be some $\hat{v}'$ such that $\text{Merge}(\hat{v}, \hat{v}')$ is defined and $\langle \hat{a} @ H, \hat{v}' \rangle \in \text{dom}(\hat{\mu})$. It cannot be the case that $\mu(\hat{a} @ H, \hat{v}') \neq \text{many}$, because by the Empty Matchable Output lemma, that would imply that $\text{Matchable}([\hat{l}_i]) = \emptyset$. Therefore, $\mu(\hat{a} @ H, \hat{v}') = \text{single}$.

Again by the definition of the summary-transition and event-step relations, there exists a transition sequence $\hat{K}_1' \overset{\hat{l}_1}{\rightarrow}_{\text{RA}} \cdots \overset{\hat{l}_m}{\rightarrow}_{\text{RA}} \hat{K}_m'$ such that $\hat{K}_1' = \hat{K}$. Because $\mu(\hat{a} @ H, \hat{v}') = \text{single}$, by the definition of what it means for $L$ to summarize $\hat{l}_1, \ldots, \hat{l}_n$, $\hat{v}' = \hat{v}$ and there must exist some $j \in 1 \ldots m$ such that $\hat{l}_j = \hat{a} @ H \hat{v}$. \qed
K.18 Fresh Matchables Lemma

Lemma (Fresh Matchables). For all \( L, \hat{I}_1, \ldots, \hat{I}_n, \hat{K}, \) and \( \hat{K}' \), if

- \( L \) summarizes \( \hat{I}_1, \ldots, \hat{I}_n \),
- \( \hat{K} \xrightarrow{L} \hat{K}' \),
- \( \hat{K} \) is an externals-only configuration,

then for all \( i \in 1 \ldots n \), \( \text{Matchable}(\hat{I}_i) \cap \text{Used}(\hat{K}) = \emptyset \).

Proof. Let \( \hat{I}_i \) be a label in \( \hat{I}_1, \ldots, \hat{I}_n \). If \([\hat{I}_i] = \ast\), then \( \text{Matchable}(\hat{I}_i) = \emptyset \), so we’re done.

If \([\hat{I}_i] = \tilde{a}@H\tilde{v} \) for some \( \tilde{a}, H, \) and \( \tilde{v} \), then \( \hat{I}_i \) must be a \( \text{rcv-ext} \) label, and therefore by the definition of what it means for \( L \) to summarize \( \hat{I}_1, \ldots, \hat{I}_n, i = 1 \). Then by the definition of the summary-transition and event-step relations, there exists a transition sequence \( \hat{K}'_1 \xrightarrow{\tilde{a}@H} \hat{K}'_2 \xrightarrow{\tilde{a}@H} \hat{K}'_3 \xrightarrow{\tilde{a}@H} \hat{K}'_4 \) such that \( \hat{K}'_1 \) = \( \hat{K} \) and \( \hat{I}_i = \hat{I}_1 \). Then by the Externals-Only Label lemma, \( \text{Matchable}(\hat{I}_1) \cap \text{Used}(\hat{K}) = \emptyset \).

Otherwise, \([\hat{I}_i] = \tilde{a}@H\tilde{v} \) for some \( \tilde{a}, H, \) and \( \tilde{v} \). Let \( (\hat{I}_i, \tilde{u}) = L \). By the definition of what it means for \( L \) to summarize \( \hat{I}_1, \ldots, \hat{I}_n \), there must be some \( \tilde{v}' \) such that \( \text{Merge}(\tilde{v}, \tilde{v}') \) is defined and \( \langle \tilde{a}@H, \tilde{v}' \rangle \in \text{dom}(\tilde{u}) \). If \( 
\mu(\tilde{a}@H, \tilde{v}') = \text{many} \), then by the Empty Matchable Output lemma, \( \text{Matchable}(\hat{I}_i) = \emptyset \), so we’re done.

Otherwise, \( \mu(\tilde{a}@H, \tilde{v}') = \text{single} \). Again by the definition of the summary-transition and event-step relations, there exists a transition sequence \( \hat{K}'_1 \xrightarrow{\tilde{a}@H} \hat{K}'_2 \xrightarrow{\tilde{a}@H} \hat{K}'_3 \xrightarrow{\tilde{a}@H} \hat{K}'_4 \) such that \( \hat{K}'_1 \) = \( \hat{K} \). Because \( \mu(\tilde{a}@H, \tilde{v}') = \text{single} \), by the definition of what it means for \( L \) to summarize \( \hat{I}_1, \ldots, \hat{I}_n, \tilde{v}' = \tilde{v} \) and there must exist some \( j \in 1 \ldots m \) such that \( [\hat{I}_j] = \tilde{a}@H\tilde{v} \). By induction on \( j \) and the Abstract Externals-Only Preservation lemma, \( \hat{K}'_1 \) is an externals-only configuration. Then by the the Externals-Only Label lemma, \( \text{Matchable}(\hat{I}_j) \cap \text{Used}(\hat{K}'_1) = \emptyset \). Again by induction on \( j \) and the Used Marker lemma, \( \text{Used}(\hat{K}'_1) \subseteq \text{UsedMarkers}(\hat{K}'_1) \).

Therefore, \( \text{Matchable}(\hat{I}_i) \cap \text{Used}(\hat{K}) = \emptyset \).

\( \square \)

K.19 Distinct Matchables Lemma

Lemma (Distinct Matchables). For all \( L, \hat{I}_1, \ldots, \hat{I}_n, \hat{K}, \) and \( \hat{K}' \), if

- \( L \) summarizes \( \hat{I}_1, \ldots, \hat{I}_n \),
- \( \hat{K} \xrightarrow{L} \hat{K}' \),
- \( \hat{K} \) is an externals-only configuration,

then for all \( i, j \) in \( 1 \ldots n \) such that \( i \neq j \), \( \text{Matchable}(\hat{I}_i) \cap \text{Matchable}(\hat{I}_j) = \emptyset \).
K.19. DISTINCT MATCHABLES LEMMA

Proof. Let \( \tilde{l}_1 \) and \( \tilde{l}_j \) be two distinct labels in the sequence \( \tilde{l}_1, \ldots, \tilde{l}_n \). If either Matchable(\( \tilde{l}_1 \)) = \( \emptyset \) or Matchable(\( \tilde{l}_j \)) = \( \emptyset \), then we’re done. Otherwise, let \( \tilde{K}^1_1 \xrightarrow{\text{RA}} \cdots \xrightarrow{\text{RA}} \tilde{K}^n_{m+1} \) be a transition sequence summarized by \( \tilde{K} \xrightarrow{L} \tilde{K}' \). We will show that that sequence has distinct labels \( \tilde{l}_p \) and \( \tilde{l}_q \) such that \( [\tilde{l}_1] = [\tilde{l}_p] \) and \( [\tilde{l}_j] = [\tilde{l}_q] \). Then we can use the Externals-Only Label and Used Marker lemmas to show that the matchable markers from those labels must be distinct.

Because Matchable(\( \tilde{l}_1 \)) \( \neq \emptyset \) and Matchable(\( \tilde{l}_j \)) \( \neq \emptyset \), there exist \( \hat{a}_i, H_i, \hat{v}_i, \hat{a}_j, H_j, \) and \( \hat{v}_j \) such that

- either \( [\tilde{l}_1] = \hat{a}_i@H_i?\hat{v}_i \) or \( [\tilde{l}_1] = \hat{a}_i@H_i!\hat{v}_i \), and

- either \( [\tilde{l}_j] = \hat{a}_j@H_j?\hat{v}_j \) or \( [\tilde{l}_j] = \hat{a}_j@H_j!\hat{v}_j \), and

If \( [\tilde{l}_1] = \hat{a}_i@H_i?\hat{v}_i \), then \( \tilde{l}_1 \) must be a \texttt{rcv-ext} label, and therefore by the definition of what it means for \( L \) to summarize \( \tilde{l}_1, \ldots, \tilde{l}_n \), \( i = 1 \). Therefore, \( \tilde{l}_1 = \tilde{l}_i \), so let \( p = 1 \).

Otherwise, \( [\tilde{l}_1] = \hat{a}_i@H_i!\hat{v}_i \). Let \( \langle \hat{v}, \hat{v}' \rangle = L \). By the definition of what it means for \( L \) to summarize \( \tilde{l}_1, \ldots, \tilde{l}_n \), there must be some \( \hat{v}' \) such that \( \text{Merge}(\hat{v}_i, \hat{v}') \) is defined and \( \langle \hat{a}_i@H_i, \hat{v}' \rangle \in \text{dom}(\hat{\mu}) \). It cannot be the case that \( \hat{\mu}(\hat{a}_i@H_i, \hat{v}') \neq \text{many} \), because by the Empty Matchable Output lemma, that would imply that Matchable(\( \tilde{l}_1 \)) = \( \emptyset \). Therefore, \( \hat{\mu}(\hat{a}_i@H_i, \hat{v}') = \text{single} \). Then by the definition of what it means for \( L \) to summarize \( \tilde{l}_1, \ldots, \tilde{l}_n \), \( \hat{v}' = \hat{v}_i \) and there must exist some \( p \in 1 \ldots m \) such that \( \tilde{l}_p = \hat{a}_i@H_i!\hat{v}_i \).

The label \( \tilde{l}_q \) is defined similarly.

Next, we must show that \( p \neq q \). If \( [\tilde{l}_p] = \hat{a}_i@H_i?\hat{v}_i \), then we know from the argument above that \( i = 1 \). We know that \( i \neq j \), so \( j \neq 1 \), and therefore \( \tilde{l}_j \) is not a \texttt{rcv-ext} label. Therefore \( [\tilde{l}_1] \neq [\tilde{l}_j] \), so \( \tilde{l}_p \neq \tilde{l}_q \), and therefore \( p \neq q \). A similar argument holds if \( [\tilde{l}_p] = \hat{a}_i@H_j?\hat{v}_j \).

Otherwise, we are left with the case where \( [\tilde{l}_p] = \hat{a}_i@H_i!\hat{v}_i \) and \( [\tilde{l}_q] = \hat{a}_j@H_j!\hat{v}_j \). If it were the case that \( p = q \), then we would have \( \langle \hat{a}_i@H_i, \hat{v}_i \rangle = \langle \hat{a}_j@H_j, \hat{v}_j \rangle \), and therefore by the definition of what it means for \( L \) to summarize \( \tilde{l}_1, \ldots, \tilde{l}_n \), there would exist there must be some \( \hat{v}' \) such that \( \text{Merge}(\hat{v}_i, \hat{v}') \) is defined and \( \hat{\mu}(\hat{a}_i@H_i, \hat{v}') = \text{many} \). We have already established that that is not the case, however, so \( p \neq q \).

Now, by induction and the Abstract Externals-Only Preservation lemma, every configuration in \( \tilde{K}^1_1, \ldots, \tilde{K}^n_m \) is an externals-only configuration. Suppose that \( p < q \). By induction on \( p - m \) and the Used Marker lemma, Matchable(\( \tilde{l}_p \)) \subseteq Used(\( \tilde{K}^n \)). By the Externals-Only Label lemma, Matchable(\( \tilde{l}_p \)) \cap Used(\( \tilde{K}^n \)) = \( \emptyset \). Therefore, Matchable(\( \tilde{l}_p \)) \cap Matchable(\( \tilde{l}_q \)) = \( \emptyset \), and therefore Matchable(\( \tilde{l}_p \)) \cap Matchable(\( \tilde{l}_q \)) = \( \emptyset \). The proof is symmetric in the case where \( q < p \).
K.20 Specification Summary Partition Lemma

Lemma (Specification Summary Partition). For all $S_1, S_2, S', \hat{K}, \hat{K}', L, \text{ and } O$, if

- $S_1 \sqcup S_2 \xrightarrow{\langle L, O \rangle} S'$,
- $S_1 \sqcup S_2$ is well-formed,
- $\hat{K}$ is an externals-only configuration,
- $\text{Mon}(S_1) \cup \text{Mon}(S_2) \subseteq \text{Used}(\hat{K})$, and
- $\hat{K} \xrightarrow{L} \hat{K}'$,

then there exist $O'_1, O'_2, S''_1$, and $S''_2$ such that

- $O = O'_1 \cup O'_2$,
- $S' = S''_1 \cup S''_2$, and
- $\text{Mon}(S''_1) \cap \text{Mon}(S''_2) = \emptyset$, and
- $S_i \xrightarrow{\langle L, O \rangle} S'_i$ for $i \in \{1, 2\}$.

Proof. We first define $S''_1, S''_2, O'_1$, and $O'_2$ in terms of the markers used in each one. Next, we will show that for every event step $\hat{K} \xrightarrow{L} \hat{K}'$ summarized by $\hat{K} \xrightarrow{L} \hat{K}'$ and for $i \in \{1, 2\}$, there exists a simulating transition sequence that starts at $S_i$, ends at $S''_i$, and fulfills a superset of the obligations $O'_i$. Finally, we will show that $O = O'_1 \cup O'_2$ and $S' = S''_1 \cup S''_2$.

Definitions of Partitioned Configurations and Obligations

For every new marker $\eta$ introduced by the summary transition $\hat{K} \xrightarrow{L} \hat{K}'$, it must be the case that $\eta$ is carried in a message sent to or received from an address marked with at most one marker $\eta'$. We can think of $\eta'$ as the “parent” marker for $\eta$. That parent must be either a marker that already exists in $\hat{K}$, or itself is part of some previous message in the transition $\hat{K} \xrightarrow{L} \hat{K}'$ and therefore has its own parent marker $\eta''$. Thus, every new marker introduced by the transition $\hat{K} \xrightarrow{L} \hat{K}'$ can trace its “ancestry” back to some marker that does not have a parent defined in $\hat{K} \xrightarrow{L} \hat{K}'$. We define $S''_1, S''_2, O'_1$, and $O'_2$ by tracing these ancestries, as formalized below.

Let $\langle \hat{l}, \hat{\mu} \rangle = L$. The parent of a marker $\eta$ in that label is a marker $\eta'$, written $\text{Parent}(\eta) = \eta'$, if and only if there exist $\hat{a}, \hat{a}'$, and $\hat{v}$ such that

- $\hat{a}@\{\eta\}$ appears in $\hat{v}$ outside of any list or dict expression,
K.20. SPECIFICATION SUMMARY PARTITION LEMMA

- either $|\tilde{t}| = \tilde{a}@\{\eta\}?\tilde{v}$, or $\mu(\tilde{a}@\{\eta\}, \tilde{v}) = \text{single}$ and $\tilde{a}$ is internal, and
- $\eta_1 < \eta$.

Not every marker has a parent in $L$, so $\text{Parent}$ is a partial function. To show that the function is well-defined, consider that by the definition of the summary-transition relation, for all $\tilde{a}', \eta_1$, and $\tilde{v}$ such that $\mu(\tilde{a}@\{\eta\}, \tilde{v}) = \text{single}$, and for all $\hat{K} \xrightarrow{\hat{R}_1, \ldots, \hat{R}_n} \hat{K}'$, summarized by $R \xrightarrow{L} R'$, there exists some $k \in 1 \ldots n$ and $\tilde{v}'$ such that $|\tilde{t}_k| = \tilde{a}@\{\eta\}?\tilde{v}'$ and $\tilde{v}' \subseteq id(id) \tilde{v}$. Then by the definition of $\subseteq id(id)$, any internal marked address $\tilde{a}@\{\eta\}$ that appears in $\tilde{v}$ outside of any list or dict expression also appears in $\tilde{v}'$, also outside of any list or dict expression. Because $\hat{K}$ is an externals-only configuration, the marker $\eta$ must be a fresh marker at that point in the event step $\hat{K} \xrightarrow{\hat{R}_1, \ldots, \hat{R}_n} \hat{K}'$. Similarly, if some $\tilde{a}@\{\eta\}$ appears in $\tilde{t}$, the marker $\eta$ must be fresh at that point. Therefore, the above conditions for $\text{Parent}$ are satisfied only when $\eta$ is fresh within all transitions $\hat{K} \xrightarrow{\hat{R}_1, \ldots, \hat{R}_n} \hat{K}'$ summarized by $\hat{K} \xrightarrow{L} \hat{K}'$, so $\text{Parent}$ is well-defined.

The definition of $\text{Parent}$ then lets us trace the ancestry of any marker used during the transition $\hat{K} \xrightarrow{L} \hat{K}'$ back to some “ancestor” marker that was not introduced by that summary transition. Formally, the ancestor of a marker $\eta$ in the label $L$, written $\text{Ancestor}(\eta)$, is defined as follows.

$$\text{Ancestor}(\eta) = \begin{cases} 
\text{Ancestor}(\eta') & \text{if } \text{Parent}(\eta) = \eta' \\
\eta & \text{if } \text{Parent}(\eta) \text{ is undefined}
\end{cases}$$

Because every parent marker $\eta'$ must be less than its child $\eta$ (i.e., $\eta' < \eta$, by the definition of $\text{Parent}$ above), and because for a given $L$ there can be only finitely many markers with a defined $\text{Parent}$ (i.e., those appearing in the initial label $\tilde{t}$ or in single messages), the recursion eventually bottoms out for any given $\eta$ and $L$, so $\text{Ancestor}$ is total.

With these definitions in hand, we can define $O'_1$, $O'_2$, $S''_1$, and $S''_2$. For $i \in \{1, 2\}$, define $O'_i$ as the sub(multi)set of $O$ such that for all $(\eta, po) \in O'_i$, $\text{Ancestor}(\eta) \in \text{Mon}(S_i)$. If $\text{Mon}(s) \neq \emptyset$ for all $s \in S'$, then for $i \in \{1, 2\}$, define similarly $S''_i$ as follows.

$$S''_i = \{s \mid s \in S' \text{ and } \forall \eta \in \text{Mon}(s), \text{Ancestor}(\eta) \in \text{Mon}(S_i)\}$$

Otherwise, because $S_1 \sqcup S_2 \xrightarrow{(L, O)} S'$ and by the definition of a summary transition relation, there exists $s$ such that $S_1 \sqcup S_2 = \{s\}$. If $S_1 = \{s\}$, then let $S''_1 = S'$ and let $S''_2 = \emptyset$. Otherwise, let $S''_1 = \emptyset$ and let $S''_2 = S'$.

Because $\text{Mon}(S_1) \cap \text{Mon}(S_2) = \emptyset$, by the definition of $S''_1$ and $S''_2$ it must be the case that $\text{Mon}(S''_1) \cap \text{Mon}(S''_2) = \emptyset$. It remains to show that

- $S_i \xrightarrow{(L, O)} S'_i$ for $i \in 1, 2$,
- $O = O'_1 \cup O'_2$, and
• $S' = S_1'' \cup S_2''$, which we will do below.

**Summary Transitions**

To show that $S_i \xrightarrow{(L,O_i)} S''_i$ for $i \in \{1,2\}$, let $S''_i = S_i$ for $i \in \{1,2\}$, and let $\hat{l}_1, \ldots, \hat{l}_n$ be a sequence summarized by $L$. We must show that, for $i \in \{1,2\}$, there exist $S''_{i,1}, \ldots, S''_{i,n+1}$ and $O''_{i,1}, \ldots, O''_{i,n}$ such that

- $S''_{i,1} = S_i''$,
- $O''_{i} \subseteq O''_{i,1} \cup \ldots \cup O''_{i,n}$ and
- $S''_{i,1} \xrightarrow{\hat{l}_1, O''_{i,1}} \cdots \xrightarrow{\hat{l}_n, O''_{i,n}} S''_{i,n+1}$.

By the definition of $S_1 \cup S_2 \xrightarrow{(L,O)} S'$, there exist $S'''_1, \ldots, S'''_{n+1}$ and $O'', \ldots, O''_n$ such that

- $S'''_1 = S_1 \cup S_2$,
- $S'''_{n+1} = S''$, 
- $O \subseteq O''_1 \cup \ldots \cup O''_n$, and
- $S'''_1 \xrightarrow{\hat{l}_1, O''_1} \cdots \xrightarrow{\hat{l}_n, O''_n} S'''_{n+1}$.

Furthermore, we already know that $S'''_1 = S_1 \cup S_2$ is well-formed, so by the Specification Well-Formed Preservation lemma and induction on $j$, we know that $S'''_j$ is well-formed for all $j \in 1 \ldots n + 1$.

By the Fresh Matchables lemma, $\text{Matchable}(\hat{l}_j) \cap \text{Used}(\hat{K}) = \emptyset$ for all $j \in 1 \ldots n$. By the Distinct Matchables lemma, $\text{Matchable}(\hat{l}_j) \cap \text{Matchable}(\hat{l}_k) = \emptyset$ for all $j$ and $k$ in $1 \ldots n$ such that $j \neq k$. Then by induction on $j$ and the Monitored Matchables Markers lemma, $\text{Mon}(S'''_i) \subseteq \text{Used}(\hat{K}) \cup \text{Matchable}(\hat{l}_j) \cup \ldots \cup \text{Matchable}(\hat{l}_{j-1})$ for all $j \in 1 \ldots n + 1$. Therefore, for all $j \in 1 \ldots n + 1$, $\text{Mon}(S'''_j) \cap \text{Matchable}(\hat{l}_j) = \emptyset$.

Next, we will show that for all $j \in 1 \ldots n + 1$, there exist $S''''_{1,j}$ and $S''''_{2,j}$ such that

- $S''''_{1,j} = S''_{1,j} \cup S''_{2,j}$,
- $\text{Mon}(S''''_{1,j}) \cap \text{Mon}(S''''_{2,j}) = \emptyset$, and
- if $j > 1$, then there exist $O''_{1,j-1}$ and $O''_{2,j-1}$ such that
  - $O''_{j-1} = O''_{1,j-1} \cup O''_{2,j-1}$ and
  - $S''''_{i,j-1} \xrightarrow{\hat{l}_{i,j-1}, O''_{i,j-1}} S''''_{i,j}$ for $i \in 1,2$. 


The proof is by induction on \( j \). In the base case where \( j = 1 \), we already know that \( S''_1 = S_1 \cup S_2 \) and \( \text{Mon}(S_1) \cap \text{Mon}(S_2) = \emptyset \). Let \( S''_{1,1} = S_1 \) and \( S''_{2,1} = S_2 \) to complete the proof.

In the inductive case, \( j > 1 \). By the induction hypothesis, we know that there exist \( S''_{1,j-1} \) and \( S''_{2,j-1} \) such that \( S''_{j-1} = S''_{1,j-1} \cup S''_{2,j-1} \) and \( \text{Mon}(S''_{1,j-1}) \cap \text{Mon}(S''_{2,j-1}) = \emptyset \).

We also know that \( S'''_{i,j-1} \rightarrow S''_i \), and that \( S'''_{1,j-1} \cup S'''_{2,j-1} \) is well-formed. By corollary K.7.3 to the Externals-Only Label lemma, if there exist \( \hat{\lambda}_{j-1} = \hat{a}@\hat{H}\hat{\nu} \) or \( \hat{\lambda}_{j-1} = \hat{a}@\hat{H}\hat{\nu} \), then \( |H| \leq 1 \). We know from the argument above that \( \text{Mon}(S'''_{1,j-1} \cup S'''_{2,j-1}) \cap \text{Matchable}(\hat{I}_{j-1}) = \emptyset \). Then by the Specification Weak-Step Partition lemma, there exist \( S'''_{1,j}, S'''_{2,j}, O''_{1,j-1}, \) and \( O''_{2,j-1} \) such that

- \( S'''_{1,j} = S''_{1,j} \cup S''_{2,j} \),
- \( \text{Mon}(S'''_{1,j}) \cap \text{Mon}(S'''_{2,j}) = \emptyset \), and
- \( O''_{1,j-1} = O''_{1,j-1} \), and
- \( S'''_{i,j-1} \rightarrow S'''_{i,j} \) for \( i \in 1, 2 \).

It remains to show that for \( i \in \{1, 2\} \), \( S'''_{i,n+1} = S'''_i \) and \( O_{n} \subseteq O_{1,n} \cup \ldots \cup O_{i,n} \).

**Matching Specification Configurations**

To show \( S'''_{1,n+1} = S'''_1 \) for \( i \in \{1, 2\} \), first consider the case where there exists some \( s' \in S' \) such that \( \text{Mon}(s') = \emptyset \). By the definition of the specification summary transition relation, there exists \( s \) such that \( S_1 \cup S_2 = \{s\} \). Assume that \( S_1 = \{s\} \); the case for \( S_2 = \{s\} \) is similar. In this case, \( S''_1 = S' \) and \( S''_2 = \emptyset \). Because \( S_2 = \emptyset \), it must be the case that \( S_1 = \emptyset \). Then by the specification transition semantics, because \( S_{2,1}'' \rightarrow \ldots \rightarrow S_{2,n+1}'' \) and \( S_{2,1}'' = S_2, S_{2,n+1}'' = \emptyset \). We also know that \( S_{1,n+1}'' \cup S_{2,n+1}'' = S_{n+1}'' \) and \( S'''_{n+1} = S' \). Therefore, \( S'''_{1,n+1} = S', \) and therefore \( S'''_{1,n+1} = S''_1 \).

Otherwise, \( \text{Mon}(s) \neq \emptyset \) for all \( s \in S' \). We will show that for all \( i \in \{1, 2\} \), all \( j \in 1 \ldots n+1 \), and all \( \eta \in \text{Mon}(S'''_{i,j}), \text{Ancestor}(\eta) \in \text{Mon}(S_j) \). Then because every PSM in \( S' \) monitors at least one marker by the definition of summary transitions for specification configurations, and \( \text{Mon}(S_1) \cap \text{Mon}(S_2) = \emptyset \), \( S_{i,n+1}'' = \text{Mon}(S_j) \) is exactly the set of PSMs from \( S' \) whose monitored markers all have ancestors in \( S_1 \). Therefore \( S'''_{i,n+1} = S''_i \) for \( i \in \{1, 2\} \).

The proof is by induction on \( j \). We prove the case for \( i = 1 \) below, the case for \( i = 2 \) is symmetrical. In the base case whre \( j = 1 \), \( S'''_{1,1} = S_1 \). Let \( \eta \) be a member of \( \text{Mon}(S_1) \). By the precondition to this lemma, we know that \( \eta \in \text{Used}(\hat{K}) \). Then by corollary K.7.3 to the Externals-Only Label lemma, \( \eta \in \text{Matchable}(\hat{I}_j) \) for all \( j \in 1 \ldots n \). By the definition of \textit{Parent}, any marker with a defined parent in \( L \)
would have to be in \( \text{Matchable}(\{\tilde{i}_j\}) \) for some \( j \in 1 \ldots n \), so \( \text{Parent}(\eta) \) is undefined. Therefore, \( \text{Ancestor}(\eta) = \eta \), and therefore \( \text{Ancestor}(\eta) \in \text{Mon}(S_1) \).

In the inductive case, \( j > 1 \). Let \( \eta \) be a marker in \( \text{Mon}(S''_{1,j-1}) \). If \( \eta \in \text{Mon}(S''_{1,j-1}) \), then by the induction hypothesis, \( \text{Ancestor}(\eta,L) \in \text{Mon}(S_1) \). Otherwise, by the Monitored Matchable Markers lemma, \( \eta \in \text{Matchable}(\{\tilde{i}_{j-1}\}) \). Then by the definition of \( \text{Matchable} \) (and the pattern-matching rules), there exist \( \tilde{a}, \tilde{a}', H, H' \), and \( \tilde{v} \) such that

- either \( [\tilde{i}_{j-1}] = \tilde{a}@H?\tilde{v} \) or \( [\tilde{i}_{j-1}] = \tilde{a}@H!\tilde{v} \),
- \( \tilde{a}@H' \) appears in \( \tilde{v} \) outside of a \text{list} or \text{dict} expression,
- \( \tilde{a}' \) is internal if \( [\tilde{i}_{j-1}] = \tilde{a}@H!\tilde{v} \), and
- \( \eta \in H' \).

Because \( \text{Mon}(S''_{1,j-1}) \neq \text{Mon}(S''_{1,j-1}) \), there must exist some \( \eta' \) in \( H \) that \( S''_{1,j-1} \) monitors, i.e., \( \eta' \in H \cap \text{Mon}(S''_{1,j-1}) \). By the induction hypothesis, \( \text{Ancestor}(\eta') \in \text{Mon}(S_1) \). We will show that \( \text{Parent}(\eta) = \eta' \), which implies that \( \text{Ancestor}(\eta) \in \text{Mon}(S_1) \). To do so, it remains to show that \( H' = \{\eta\} \), \( H = \{\eta'\} \), \( \tilde{\mu}(\tilde{a}@H, \tilde{v}) = \text{single} \) if \( [\tilde{i}_{j-1}] = \tilde{a}@H!\tilde{v} \), and \( \eta' < \eta \).

By the Matchable Labels lemma, there exists some transition sequence

\[
\tilde{K}_1 \xrightarrow{\tilde{l}_1} \text{RA} \ldots \xrightarrow{\tilde{l}_n} \tilde{K}_n \xrightarrow{\tilde{i}_{j}} \text{RA} \tilde{K}_n+1,
\]

and some \( k \in 1 \ldots m \) such that \( \tilde{K}_1 = K \) and \( [\tilde{i}_k] = [\tilde{i}_{j-1}] \). By induction and the Abstract Externals-Only Preservation lemma, \( \tilde{K}_n \) is an externals-only configuration. By the Externals-Only Label lemma, \( |H| \leq 1 \), so \( H = \{\eta'\} \). Because \( \eta \) is a fresh marker in the step \( \tilde{K}_n \xrightarrow{\tilde{i}_k} \text{RA} \tilde{K}_n+1 \), by the definition of the \text{Mark} function it must be the case that \( \eta' < \eta \).

If \( [\tilde{i}_{j-1}] = \tilde{a}@H!\tilde{v} \), then by the definition of the A-ReceiveExternal rule that must enable this transition, \( |H'| = 1 \). Therefore, \( H' = \{\eta\} \).

Otherwise, \( [\tilde{i}_{j-1}] = \tilde{a}@H!\tilde{v} \). By the definition of the A-SendExternal rule that must enable this transition, there must be some \( \tilde{v}' \) appearing in \( \tilde{K}_n \) and some \( H'' \) such that \( (\tilde{v},H'') \in \text{Markings}(\tilde{v}',H''') \). Then by the definition of \text{Markings} there exist some \( H''' \) and \( \eta'' \) such that \( H' = \text{tpoints}'''' \cup \{\eta''\} \) and \( \tilde{a}@H'''' \) appears in \( \tilde{v} \). Because \( \tilde{a} \) is internal and \( \tilde{K}_n \) is an externals-only configuration, \( H'''' \) = \( \emptyset \), and therefore \( H' = \{\eta\} \). By the Empty Matchable Output lemma, there is no \( \tilde{v}' \) mergeable with \( \tilde{v} \) such that \( \tilde{\mu}(\tilde{a}@H, \tilde{v}') = \text{many} \), so because \( L \) summarizes \( \tilde{I}_1, \ldots, \tilde{I}_n \), it must be the case that \( \tilde{\mu}(\tilde{a}@H, \tilde{v}) = \text{single} \).

Matching Obligations

Next, we must show that for \( i \in \{1, 2\} \), \( O'_i \subseteq O''_{i,1} \cup \ldots \cup O''_{i,n} \). First, we show that for \( i \in \{1, 2\} \), for all \( \langle \eta, po \rangle \in O''_{i,1} \cup \ldots \cup O''_{i,n}, \text{Ancestor}(\eta) \in \text{Mon}(S_i) \).

Let there be \( j \in 1 \ldots n \), and let \( \langle \eta, po \rangle \) be a member of \( O''_{i,j} \). By the definition of the transition \( S''_{i,j} \xrightarrow{[\tilde{i}_j]} O''_{i,j} \rightarrow S''_{i,j+1} \), because \( \langle \eta, po \rangle \in O''_{i,j} \), there exists some
PSM in $S''_i$ that transitioned and fulfilled the obligation, and continues to monitor the marker $\eta$. Therefore, $\eta \in \text{Mon}(S''_{i,j+1})$. Then by the arguments above, $\text{Ancestor}(\eta) \in \text{Mon}(S_i)$.

Next, we know that for all $j \in 1 \ldots n$, $O''_j = O''_{1,j} \cup O''_{2,j}$. Thus, for $i \in \{1,2\}$, $O''_{i,1} \cup \ldots \cup O''_{i,n}$ contains all obligations $\langle \eta, po \rangle$ from $O''_{1} \cup \ldots \cup O''_{n}$ such that $\text{Ancestor}(\eta) \in \text{Mon}(S_i)$. We also know that $O \subseteq O''_i \cup \ldots \cup O''_n$, and for $i \in \{1,2\}$, $O'_i$ is the sub(multi)set of $O$ such that for all $\langle \eta, po \rangle \in O'_i$, $\text{Ancestor}(\eta) \in \text{Mon}(S_i)$. Therefore, $O'_i \subseteq O''_{i,1} \cup \ldots \cup O''_{i,n}$ for $i \in \{1,2\}$.

### Disjoint Configurations and Obligations

It remains only to show that $O = O'_1 \cup O'_2$, $S' = S''_1 \cup S''_2$. Let $\bar{I}_1, \ldots, \bar{I}_n$ be a sequence summarized by $L$. By the arguments from the Summary Transitions subsection above, we know that

- $O = O'_1 \cup O'_2$.
- $S' = S''_1 \cup S''_2$.
- $\text{Mon}(S''_1) \cap \text{Mon}(S''_2) = \emptyset$, and
- for all $s \in S'$, $\text{Mon}(s) \neq \emptyset$.

If every PSM in $S''_1 \cup S''_2$ monitors at least one marker, then because $S''_1$ and $S''_2$ monitor disjoint sets of markers, $S''_1$ and $S''_2$ are disjoint. Otherwise, it must be the case that either $S''_1 = S'$ and $S''_2 = \emptyset$, or $S''_1 = \emptyset$ and $S''_2 = S'$, so $S''_1$ and $S''_2$ are disjoint. Therefore, $S' = S''_1 \cup S''_2$.

### K.21 Monitored Marker Permanence Lemma

**Lemma** (Monitored Marker Permanence). For all $s$, $\hat{\lambda}$, $O$, $S$, and $s'$ such that $s \xrightarrow{\hat{\lambda}, O, S} s'$, $\text{InMon}(s) \subseteq \text{InMon}(s')$ and $\text{OutMon}(s) \subseteq \text{OutMon}(s')$.

**Proof.** By a simple inspection of the rules enabling $s \xrightarrow{\hat{\lambda}, O, S} s'$.

**Corollary K.21.1.** For all $S$, $\hat{\lambda}$, and $S'$ such that $S \xrightarrow{\hat{\lambda}} S'$, $\text{InMon}(S) \subseteq \text{InMon}(S')$ and $\text{OutMon}(S) \subseteq \text{OutMon}(S')$.

**Proof.** Let $s$ be a member of $S$; we must show that there exists $s' \in S'$ such that $\text{InMon}(s) \subseteq \text{InMon}(s')$ and $\text{OutMon}(s) \subseteq \text{OutMon}(s')$. By the definition of the rules for the transition $S \xrightarrow{\hat{\lambda}} S'$, either $s \in S'$ or there exist $O$ and $S''$ such that $s \xrightarrow{\hat{\lambda}, O, S''} s'$ and $s' \in S'$. In the former case, let $s' = s$ and the condition holds. In the latter case, the condition holds by the above lemma.

**Corollary K.21.2.** For all $S$, $\hat{\lambda}$, and $S'$ such that $S \xrightarrow{\hat{\lambda}} S'$, $\text{InMon}(S) \subseteq \text{InMon}(S')$ and $\text{OutMon}(S) \subseteq \text{OutMon}(S')$.  

Proof. By the definition of \( S \xrightarrow{\lambda} S' \), there exist \( \lambda_1', \ldots, \lambda_n' \) such that \( S \xrightarrow{\lambda'_1} \cdots \xrightarrow{\lambda'_n} S' \). By corollary K.21.1 and induction on \( n \), \( \text{InMon}(S) \subseteq \text{InMon}(S') \) and \( \text{OutMon}(S) \subseteq \text{OutMon}(S') \).

\[ \square \]

Corollary. For all \( S, \lambda_1, \ldots, \lambda_n, \) and \( S' \), such that \( S \xrightarrow{\lambda_1 \ldots \lambda_n} S' \), \( \text{InMon}(S) \subseteq \text{InMon}(S') \) and \( \text{OutMon}(S) \subseteq \text{OutMon}(S') \).

Proof. By the definition of the \( \rightarrow \) relation, \( S \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_n} S' \). Then by induction on the length of that execution and corollary K.21.2 to this lemma, \( \text{InMon}(S) \subseteq \text{InMon}(S') \) and \( \text{OutMon}(S) \subseteq \text{OutMon}(S') \).

\[ \square \]

Corollary K.21.3. For all \( S, L, O, \) and \( S' \), if \( S \xrightarrow{(L,O)} S' \) and there exist \( \hat{K} \) and \( \hat{K}' \) such that \( \hat{K} \xrightarrow{L} \hat{K}' \), then \( \text{InMon}(S) \subseteq \text{InMon}(S') \) and \( \text{OutMon}(S) \subseteq \text{OutMon}(S') \).

Proof. Let \( \hat{K} \xrightarrow{\hat{l}_1} \cdots \xrightarrow{\hat{l}_n} \hat{K}' \) be an event step summarized by \( \hat{K} \xrightarrow{L} \hat{K}' \). By the definition of \( S \xrightarrow{\lambda} S' \) and the \( \xrightarrow{\cdot} \) relation, there exist \( S''_1, \ldots, S''_{n+1} \) and \( O'_1, \ldots, O'_n \) such that

- \( S''_1 = S' \)
- \( S''_{n+1} = S'' \)
- \( S''_1 \xrightarrow{[\hat{l}_1],O'_1} \cdots \xrightarrow{[\hat{l}_n],O'_n} S''_{n+1} \), and
- \( O \subseteq \bigcup_{i=1 \ldots n} O'_i \).

Let there be \( \langle \eta, po \rangle \in O \). Because \( O \subseteq \bigcup_{i=1 \ldots n} O'_i \), there exists some \( i \in 1 \ldots n \) such that \( \langle \eta, po \rangle \in O'_i \). By the definition of the transition relations for specification configurations and PSMs, there must exist some \( \tilde{a}, H \), and \( \tilde{v} \) such that \( [\hat{l}_i] = \tilde{a}@H!\tilde{v} \) and \( \eta \in H \). Furthermore, there exist \( S'''_1, \ldots, S'''_n \) such that

- \( S'''_1 = S''_i \)
- \( S'''_n = S''_{i+1} \), and

\[ \square \]
K.23. FAIR SPECIFICATION SUFFIX LEMMA

Lemma (Fair Specification Suffix). For all executions $S_1 \overset{\langle L_1, O_1 \rangle}{\rightarrow} \ldots$, if

- $S_1$ is fair,
- $S_1$ is well-formed,
- all PSMs in $S_1$ monitor distinct markers, and
- there exists an execution $\bar{K}_1 \overset{L_1}{\rightarrow} \ldots$ such that
  - $\bar{K}_1$ is an externals-only configuration,
  - $\text{len} \left( \bar{K}_1 \overset{L_1}{\rightarrow} \ldots \right) = \text{len} \left( S_1 \overset{\langle L_1, O_1 \rangle}{\rightarrow} \ldots \right)$, and
  - $\text{Mon}(S_1) \subseteq \text{Used}(\bar{K}_1)$,

then for all $S_k$ in that execution, and all $s \in S_k$, there exists a summary specification execution $S'_k \overset{\langle L_k, O'_k \rangle}{\rightarrow} \ldots$ such that

- $S'_k = \{ s \}$,
- if $S_1 \overset{\langle L_1, O_1 \rangle}{\rightarrow} \ldots$ is infinite, then $S'_k \overset{\langle L_k, O'_k \rangle}{\rightarrow} \ldots$ is infinite,
- if $S_1 \overset{\langle L_1, O_1 \rangle}{\rightarrow} \ldots$ is finite, then $\text{len} \left( S'_k \overset{\langle L_k, O'_k \rangle}{\rightarrow} \ldots \right) = \text{len} \left( S_k \overset{\langle L_k, O_k \rangle}{\rightarrow} \ldots \right)$,
- $S'_i \subseteq S_i$ for all $S'_i$ in the execution, and
- $S'_k \overset{\langle L_k, O'_k \rangle}{\rightarrow} \ldots$ is fair.

Proof. By corollary K.12.3 to the Distinct-Marker Preservation lemma, for all $S_j$ in $S_1 \overset{\langle L_1, O_1 \rangle}{\rightarrow} \ldots$, all PSMs in $S_k$ monitor distinct markers. Therefore, $\text{Mon}(s) \cap \text{Mon}(S_k - \{ s \}) = \emptyset$. By induction and the corollary to the Abstract
Because the original execution is fair, there exists some $O$ in the execution, and let there be $\langle L_1, O_k \rangle \rightarrow \ldots$ is well-formed. Finally, by induction and the corollary to the Used/monitored Marker lemma, $\text{Mon}(S_j) \subseteq \text{Used}(\hat{K}_j)$ for all corresponding configurations $S_j$ and $K_j$ in the two executions. Then by induction and the Specification Summary Partition lemma, there exist summary specification executions $S'_k \rightarrow \langle L_k, O'_k \rangle \rightarrow \ldots$ and $S''_k \rightarrow \langle L_k, O''_k \rangle \rightarrow \ldots$ such that

- $S'_k = \{s\}$,

- if $S_1 \rightarrow \ldots$ is infinite, then $S'_k \rightarrow \langle L_k, O'_k \rangle \rightarrow \ldots$ and $S''_k \rightarrow \langle L_k, O''_k \rangle \rightarrow \ldots$ are infinite,

- if $S_1 \rightarrow \ldots$ is finite, then $\text{len}(S'_k \rightarrow \ldots) = \text{len}(S''_k \rightarrow \ldots)$,

- $S'_i \subseteq S''_i$ for all corresponding $S'_i$ and $S''_i$ in the executions,

- $O'_i \subseteq O''_i$ for all corresponding $O'_i$ and $O''_i$ in the executions, and

- $\text{Mon}(S'_i) \cap \text{Mon}(S''_i) = \emptyset$ for all corresponding $S'_i$ and $S''_i$ in the executions.

It remains to show that $S'_k \rightarrow \langle L_k, O'_k \rangle \rightarrow \ldots$ is fair. Let there be $i$ such that $S'_{k+i}$ is in the execution, and let there be $\langle \eta, po \rangle \in \text{Obls}(S'_{k+i})$. We must show that there exists $j$ such that there is a step $S'_{k+i+j} \rightarrow \langle L_{k+i+j}, O'_{k+i+j} \rangle \rightarrow S'_{k+i+j+1}$ and $\langle \eta, po \rangle \in O'_{k+i+j}$.

By the definition of $S'_{k+i}$, there exists $S_{k+i}$ in the original execution such that $S'_{k+i} \subseteq S_{k+i}$. Then by the definition of the $\text{Obls}$ function, $\langle \eta, po \rangle \in \text{Obls}(S_{k+i})$. Because the original execution is fair, there exists $j$ such that there is a step $S_{k+i+j} \rightarrow \langle L_{k+i+j}, O_{k+i+j} \rangle \rightarrow S_{k+i+j+1}$ and $\langle \eta, po \rangle \in O_{k+i+j}$. We will show that $\langle \eta, po \rangle \in O'_{k+i+j}$.

Because $S_{k+i}$ is well-formed, $S'_{k+i}$ is well-formed. Therefore, there exists some $s \in S'_{k+i}$ such that $\eta \in \text{Mon}(s)$. Therefore, $\eta \in \text{Mon}(S'_{k+i})$. Then by induction and corollary K.21.3 to the Monitored Marker Permanence lemma, $\eta \in \text{Mon}(S'_{k+i+j+1})$. We know that $\text{Mon}(S'_{k+i+j+1}) \cap \text{Mon}(S''_{k+i+j+1}) = \emptyset$, so $\eta \notin \emptyset$. By the contrapositive of the Post-Fulfillment Monitoring lemma, $\langle \eta, po \rangle \in O''_{k+i+j}$. Then because $\langle \eta, po \rangle \in O''_{k+i+j}$ and $O_{k+i+j} = O'_{k+i+j} \cup O''_{k+i+j}$, $\langle \eta, po \rangle \in O'_{k+i+j}$.
K.24 SimExecs Simulation Lemma

Lemma (SimExecs Simulation). For all $\widehat{a\bar{a}} = \widehat{K}_1 \xrightarrow{\widehat{f}_{1,1},...\widehat{f}_{1,n}} ...$, $s$, $T$, and $Z$, if $\widehat{K}_1$ and $s$ are well-formed, $\widehat{K}_1$ is an externals-only configuration, $\text{Mon}(s) \subseteq \text{Used}(\widehat{K}_1)$, and $T$ is conformance-reflecting, then for all $S_1 \xrightarrow{(L_1,O_1)} ... \in \text{SimExecs}(\widehat{a\bar{a}}, s, T, Z)$, $\text{Simulates}(S_1 \xrightarrow{(L_1,O_1)} ... , \widehat{a\bar{a}})$.

Proof: We will show that the set of inputs to SimExecs with a well-formed initial configuration and PSM and a conformance-reflecting $T$ are a subset of the inputs to SimExecs that produce only simulations of the given execution $\widehat{a\bar{a}}$. Formally, define the sets $I_{CR}$ and $I_{sim}$ as follows; we will show that $I_{CR} \subseteq I_{sim}$.

$$I_{CR} = \left\{ \left\langle \widehat{K}_1 \xrightarrow{\widehat{f}_{1,1},...\widehat{f}_{1,n}} ... , s, T, Z \right\rangle \middle| \begin{array}{c}
\text{• } \widehat{K}_1 \text{ and } s \text{ are well-formed,} \\
\text{• } \widehat{K}_1 \text{ is an externals-only configuration,} \\
\text{• } \text{Mon}(s) \subseteq \text{Used}(\widehat{K}_1), \text{ and} \\
\text{• } T \text{ is conformance-reflecting}
\end{array} \right\}$$

$$I_{sim} = \left\{ \langle \widehat{a\bar{a}}, s, T, Z \rangle \middle| \forall S_1 \xrightarrow{(L_1,O_1)} ... \in \text{SimExecs}(\widehat{a\bar{a}}, s, T, Z). \text{Simulates}(S_1 \xrightarrow{(L_1,O_1)} ... , \widehat{a\bar{a}}) \right\}$$

By unrolling the definition of SimExecs by one step, we can define $I_{sim}$ as follows.
\[ I_{\text{sim}} = \begin{cases} \left\langle \hat{\mathcal{K}}_1 \xrightarrow{\hat{L}_1, \ldots, \hat{L}_m} s, T, Z \right\rangle \mid \text{either} \end{cases} \]

- \( \text{len} \left( \hat{\mathcal{K}}_1 \xrightarrow{\hat{L}_1, \ldots, \hat{L}_m} \ldots \right) = 0 \) and \( \text{Simulates} \left( \{s\}, \hat{\mathcal{K}}_1 \xrightarrow{\hat{L}_1, \ldots, \hat{L}_m} \ldots \right) \), or
- \( \text{len} \left( \hat{\mathcal{K}}_1 \xrightarrow{\hat{L}_1, \ldots, \hat{L}_m} \ldots \right) > 0 \) and for all \( S_1 \xrightarrow{L_1} \ldots \) and \( X \), if
  - \( S_1 = \{s\} \),
  - \( L_1 \) is the (lexicographically) least summary-transition label such that
    \( \hat{\mathcal{K}}_1 \xrightarrow{L_1} \hat{\mathcal{K}}_2 \) summarizes \( \hat{\mathcal{K}}_1 \xrightarrow{\hat{L}_1, \ldots, \hat{L}_m} \hat{\mathcal{K}}_2 \) and \( Z(\hat{\mathcal{K}}_1 \xrightarrow{L_1} \hat{\mathcal{K}}_2, s) \neq \emptyset \),
  - \((O_1, S_2) \in Z(\hat{\mathcal{K}}_1 \xrightarrow{L_1} \hat{\mathcal{K}}_2, s) \),
  - \( \text{dom}(X) = \bigcup_{s' \in S_2} \text{TransExec}_T(\hat{\mathcal{K}}_2 \xrightarrow{\hat{L}_2, \ldots, \hat{L}_m} \ldots, s') \),
  - \( \forall (\overline{a\bar{e}}, s'', A, M) \in \text{dom}(X). \ X(\overline{a\bar{e}}, s'', A, M) \in \text{SimExecs}(\overline{a\bar{e}}, s'', T, Z) \), and
  - \( S_2 \xrightarrow{L_2, O_2} \ldots = \bigcup_{s' \in S_2} \text{UntransExec}_T(\hat{\mathcal{K}}_2 \xrightarrow{\hat{L}_2, \ldots, \hat{L}_m} \ldots, s', X) \)
then \( \text{Simulates} \left( S_1 \xrightarrow{L_1, O_1} \ldots, \hat{\mathcal{K}}_1 \xrightarrow{\hat{L}_1, \ldots, \hat{L}_m} \ldots \right) \)

For all \( \hat{\mathcal{K}}_1 \xrightarrow{\hat{L}_1, \ldots, \hat{L}_m} \ldots \) and \( s \) such that \( \text{len} \left( \hat{\mathcal{K}}_1 \xrightarrow{\hat{L}_1, \ldots, \hat{L}_m} \ldots \right) = 0 \), it is trivially the case that \( \text{Simulates} \left( \{s\}, \hat{\mathcal{K}}_1 \xrightarrow{\hat{L}_1, \ldots, \hat{L}_m} \ldots \right) \). Thus, the above definition simplifies to the following.
K.24. SIMEXEC SIMULATION LEMMA

\[ I_{\text{sim}} = \begin{cases} \langle \tilde{K}_1 \xrightarrow{\tilde{t}_{1,1} \ldots \tilde{t}_{1,m}} \ldots, s, T, Z \rangle \mid \text{if } \text{len} \left( \tilde{K}_1 \xrightarrow{\tilde{t}_{1,1} \ldots \tilde{t}_{1,m}} \ldots \right) > 0, \text{then for all } S_1 \xrightarrow{(L_1, O_1)} \ldots \text{ and } X, \text{if} & \\
\quad \bullet S_1 = \{s\},
\quad \bullet L_1 \text{ is the (lexicographically) least summary-transition label such that } \tilde{K}_1 \xrightarrow{L_1} \tilde{K}_2 \text{ summarizes } \tilde{K}_1 \xrightarrow{\tilde{t}_{1,1} \ldots \tilde{t}_{1,m}} \tilde{K}_2 \text{ and } Z(\tilde{K}_1 \xrightarrow{L_1} \tilde{K}_2, s) \neq \emptyset, \\
\quad \bullet (O_1, S_2) \in Z(\tilde{K}_1 \xrightarrow{L_1} \tilde{K}_2, s), \\
\quad \bullet \text{dom}(X) = \bigcup_{s' \in S_2} \text{TransExec}(\tilde{K}_2 \xrightarrow{\tilde{t}_{2,1} \ldots \tilde{t}_{2,m}} \ldots, s'), \\
\quad \bullet \forall \langle \tilde{aex}, s''', A, M \rangle \in \text{dom}(X). X(\tilde{aex}, s'', A, M) \in \text{SimExecs}(\tilde{aex}, s''', T, Z), \text{and} \\
\quad \bullet S_2 \xrightarrow{(L_2, O_2)} \ldots \subseteq \bigcup_{s' \in S_2} \text{UntransExec}(\tilde{K}_2 \xrightarrow{\tilde{t}_{2,1} \ldots \tilde{t}_{2,m}} \ldots, s', X) \\
\text{then } \text{Simulates} \left( S_1 \xrightarrow{(L_1, O_1)} \ldots, \tilde{K}_1 \xrightarrow{\tilde{t}_{1,1} \ldots \tilde{t}_{1,m}} \ldots \right) \end{cases} \]

Now define \( I'_{\text{sim}} \) as the subset of the tuples \( \langle \tilde{aex}, s, T, Z \rangle \) in \( I_{\text{sim}} \) in which the initial configuration \( \tilde{K}_1 \) of \( \tilde{aex} \) and the PSM \( s \) are well-formed, \( \tilde{K}_1 \) is an externals-only configuration, \( \text{Mon}(s) \subseteq \text{Used}(\tilde{K}_1) \), and \( T \) is conformance-reflecting. For such a set, we can simplify the last few bullets of the above definition. By the Abstract Well-Formed Preservation lemma (appendix I), the Abstract Externals-Only Preservation lemma (also appendix I), and induction on the number of labels on the step \( \tilde{K}_1 \xrightarrow{\tilde{t}_{1,1} \ldots \tilde{t}_{1,m}} \tilde{K}_2 \), \( \tilde{K}_2 \) is a well-formed, externals-only configuration. Similarly, by corollary K.5.4 to the Specification Well-Formed Preservation lemma, every PSM \( s' \in S_2 \) is well-formed. By corollary K.10.1 to the Used/Monitored Marker lemma, for every PSM \( s' \in S_2 \), \( \text{Mon}(s') \subseteq \text{Used}(\tilde{K}_2) \). By the definition of a summary transition, no actor in \( \tilde{K}_2 \) is handling an event. Then by the Execution Simulation property for conformance reflection, we know that if \( X \) contains a simulation for all members of \( \text{TransExec}(\tilde{K}_2 \xrightarrow{\tilde{t}_{2,1} \ldots \tilde{t}_{2,m}} \ldots, s') \), then \( \text{Simulates} \left( \text{UntransExec}(\tilde{K}_2 \xrightarrow{\tilde{t}_{2,1} \ldots \tilde{t}_{2,m}} \ldots, s', X), \tilde{K}_2 \xrightarrow{\tilde{t}_{2,1} \ldots \tilde{t}_{2,m}} \ldots \right) \). Then as a result, \( \text{Simulates} \left( S_2 \xrightarrow{(L_2, O_2)} \ldots, \tilde{K}_2 \xrightarrow{\tilde{t}_{2,1} \ldots \tilde{t}_{2,m}} \ldots \right) \) by the definition of the union operation on specification summary executions, and therefore \( \text{Simulates} \left( S_1 \xrightarrow{(L_1, O_1)} \ldots, \tilde{K}_1 \xrightarrow{\tilde{t}_{1,1} \ldots \tilde{t}_{1,m}} \ldots \right) \). Thus, we can rewrite the definition of \( I'_{\text{sim}} \) as follows.
\[ I'_\text{sim} = \left\{ \left( \hat{K}_1 \xrightarrow{\hat{l}_1, \ldots, \hat{l}_m} \ldots, s, T, Z \right) \right\} \]

- \( \hat{K}_1 \) and \( s \) are well-formed,
- \( \hat{K}_1 \) is an externals-only configuration,
- \( \text{Mon}(s) \subseteq \text{Used}(\hat{K}_1) \),
- \( T \) is conformance-reflecting, and
- if \( \text{len}(\hat{K}_1 \xrightarrow{\hat{l}_1, \ldots, \hat{l}_m} \ldots) > 0 \), then for all \( L_1, O_1, S_2, s', a\text{ex}, s'', A, M, \) and \( S'_1 \xrightarrow{\Lambda_1} \ldots \),
  - \( L_1 \) is the (lexicographically) least summary-transition label such that \( \hat{K}_1 \xrightarrow{L_1} \hat{K}_2 \) summarizes \( \hat{K}_1 \xrightarrow{\hat{l}_1, \ldots, \hat{l}_m} \hat{K}_2 \) and \( Z(\hat{K}_1 \xrightarrow{L_1} \hat{K}_2, s) \neq \emptyset \),
  - \( \langle O_1, S_2 \rangle \in Z(\hat{K}_1 \xrightarrow{L_1} \hat{K}_2, s) \),
  - \( s' \in S_2 \),
  - \( \langle a\text{ex}, s'', A, M \rangle \in \text{TransExec}(\hat{K}_2 \xrightarrow{\hat{l}_2, \ldots, \hat{l}_m} \ldots, s') \), and
  - \( S'_1 \xrightarrow{\Lambda_1} \ldots \in \text{SimExec}(a\text{ex}, s'', T, Z) \),
then \( S'_1 = \{s''\} \) and \( \text{Simulates}(S'_1 \xrightarrow{\Lambda_1} \ldots, a\text{ex}) \)

Because \( I'_\text{sim} \subseteq I_\text{sim} \), we know that for all \( \langle a\text{ex}, s, T, Z \rangle \in I'_\text{sim} \) and all \( S_1 \xrightarrow{\Lambda_1} \ldots \in \text{SimExec}(a\text{ex}, s, T, Z) \), \( \text{Simulates}(S_1 \xrightarrow{\Lambda_1} \ldots, a\text{ex}) \). Furthermore, by the definition of \( \text{SimExec} \), we know that \( S_1 = \{s\} \). Thus, we can define a subset \( I''_\text{sim} \) of \( I'_\text{sim} \) as a greatest fixed point \( I''_\text{sim} = \nu Y. F(Y) \), where \( F(Y) \) is defined as follows.
K.24. SIMEXECS SIMULATION LEMMA

\[
F(Y) = \left\langle \hat{K}_1 \xrightarrow{L_1} \hat{K}_2^{l_1, \ldots, l_m} \ldots, s, T, Z \right\rangle
\]

- \( \hat{K}_1 \) and \( s \) are well-formed,
- \( \hat{K}_1 \) is an externals-only configuration,
- \( \text{Mon}(s) \subseteq \text{Used}(\hat{K}_1) \),
- \( T \) is conformance-reflecting, and
- if \( \text{len}\left( \hat{K}_1 \xrightarrow{L_1} \hat{K}_2^{l_1, \ldots, l_m} \ldots \right) > 0 \), then for all \( L_1, O_1, S_2, s', \bar{a}\bar{e}, s'', A, \) and \( M \), if
  - \( L_1 \) is the (lexicographically) least summary-transition label such that \( \hat{K}_1 \xrightarrow{L_1} \hat{K}_2 \)
    summarizes \( \hat{K}_1 \xrightarrow{L_1} \hat{K}_2^{l_1, \ldots, l_m} \hat{K}_2 \) and \( Z(\hat{K}_1 \xrightarrow{L_1} \hat{K}_2, s) \neq \emptyset \) and
  - \( \langle O_1, S_2 \rangle \in Z(\hat{K}_1 \xrightarrow{L_1} \hat{K}_2, s) \),
  - \( s' \in S_2 \), and
  - \( \langle \bar{a}\bar{e}, s'', A, M \rangle \in \text{TransExec}_T(\hat{K}_2 \xrightarrow{L_1} \hat{K}_2^{l_1, \ldots, l_m} \ldots, s') \),
    then \( \langle \bar{a}\bar{e}, s'', T, Z \rangle \in Y \)

Because \( I''_{\text{sim}} \subseteq I'_{\text{sim}} \subseteq I_{\text{sim}} \), to show that \( I_C \subseteq I_{\text{sim}} \), it suffices to show that \( I_C \subseteq I''_{\text{sim}} \). By coinduction, it suffices to show that \( I_C \) is \( F \)-dense: that is, that
\( I_C \subseteq F(I_C) \).

Let \( \langle \hat{K}_1 \xrightarrow{L_1} \hat{K}_2^{l_1, \ldots, l_m} \ldots, s, T, Z \rangle \) be a member of \( I_C \); we must show that
\( \langle \hat{K}_1 \xrightarrow{L_1} \hat{K}_2^{l_1, \ldots, l_m} \ldots, s, T, Z \rangle \in F(I_C) \). By the definition of \( I_C \), we know that \( \hat{K}_1 \)
and \( s \) are well-formed, \( \hat{K}_1 \) is an externals-only configuration, \( \text{Mon}(s) \subseteq \text{Used}(\hat{K}_1) \),
and that \( T \) is conformance-reflecting. If \( \text{len}\left( \hat{K}_1 \xrightarrow{L_1} \hat{K}_2^{l_1, \ldots, l_m} \ldots \right) = 0 \), then we’re done. Otherwise, let there be \( L_1, O_1, S_2, s', \bar{a}\bar{e}, s'', A, \) and \( M \) such that
- \( L_1 \) is the (lexicographically) least summary-transition label such that \( \hat{K}_1 \xrightarrow{L_1} \hat{K}_2 \)
  summarizes \( \hat{K}_1 \xrightarrow{L_1} \hat{K}_2^{l_1, \ldots, l_m} \hat{K}_2 \) and \( Z(\hat{K}_1 \xrightarrow{L_1} \hat{K}_2, s) \neq \emptyset \),
- \( \langle O_1, S_2 \rangle \in Z(\hat{K}_1 \xrightarrow{L_1} \hat{K}_2, s) \),
- \( s' \in S_2 \), and
- \( \langle \bar{a}\bar{e}, s'', A, M \rangle \in \text{TransExec}_T(\hat{K}_2 \xrightarrow{L_1} \hat{K}_2^{l_1, \ldots, l_m} \ldots, s') \).
As argued above, by the Abstract Well-Formed Preservation lemma, the Abstract Externals-Only Preservation lemma, and induction on the number of labels on the step $\hat{K}_1 \overset{\hat{f}_{1,1} \ldots \hat{f}_{1,m}}{\longrightarrow} \hat{K}_2$, $\hat{K}_2$ is a well-formed, externals-only configuration. By corollary K.5.4 to the Specification Well-Formed Preservation lemma, every PSM $s' \in S_2$ is well-formed. By corollary K.10.1 to the Used/Monitored Marker lemma, for every PSM $s' \in S_2$, $\text{Mon}(s') \subseteq \text{Used}(\hat{K}_2)$. By the definition of a summary transition, no actor in $\hat{K}_2$ is handling an event. Let $\hat{K}'_1$ be the first configuration of $\hat{aex}$; then by the Initial Pair Correctness, Well-Formed Preservation, Externals-Only Preservation, and Used/Monitored Preservation conformance-reflection properties, $\hat{K}'_1$ is an externals-only configuration, and $\text{Mon}(s'') \subseteq \text{Used}(\hat{K}'_1)$. Then because $T$ is conformance-reflecting, we have that $\langle \hat{aex}, s'', T, Z \rangle \in I_{CR}$, and therefore $\langle \hat{K}_1 \overset{\hat{f}_{1,1} \ldots \hat{f}_{1,m}}{\longrightarrow} \ldots, s, T, Z \rangle \in F(I_{CR})$.

\[ \square \]

K.25 \textit{SimExecs Non-Emptiness Lemma}

\textbf{Lemma (SimExecs Non-Emptiness).} For all $\hat{K}_1 \overset{\hat{f}_{1,1} \ldots \hat{f}_{1,m}}{\longrightarrow} \ldots, s, T,$ and $Z,$ if there exists $R$ such that

- $\hat{K}_1$ and $s$ are well-formed,

- $\hat{K}_1$ is an externals-only configuration,

- $\text{Mon}(s) \subseteq \text{Used}(\hat{K}),$

- $R$ is a transformation-conformance-dense relation for $T$ with witness $Z,$ and

- $\langle \hat{K}_1, s \rangle \in R,$

then $\text{SimExecs}(\hat{K}_1 \overset{\hat{f}_{1,1} \ldots \hat{f}_{1,m}}{\longrightarrow} \ldots, s, T, Z) \neq \emptyset.$

\textit{Proof.} Somewhat similar to the proof of the SimExecs Simulation lemma, we will define the set of inputs to SimExecs that come from a transformation-conformance-dense relation $R$ are a subset of those inputs that produce at least one result. Formally, define the sets $I_{TCD}$ and $I_{NE}$ as follows; we will show that $I_{TCD} \subseteq I_{NE}$. 


I_{TCD} = \{ \langle \hat{K}_1 \xrightarrow{\hat{l}_1,\ldots,\hat{l}_{1,m}} \ldots, s, T, Z \rangle \mid \exists R \text{ such that} \\
\cdot \hat{K}_1 \text{ and } s \text{ are well-formed}, \\
\cdot \hat{K}_1 \text{ is an externals-only configuration}, \\
\cdot \text{Mon}(s) \subseteq \text{Used}(\hat{K}_1), \\
\cdot R \text{ is a transformation-conformance-dense relation for } T \\
\text{ with witness } Z, \text{ and} \\
\cdot \langle \hat{K}_1, s \rangle \in R, \}

I_{NE} = \{ \langle \hat{aex}, s, T, Z \rangle \mid \text{SimExecs}(\hat{aex}, s, T, Z) \neq \emptyset \}

By unrolling the definition of SimExecs by one step, we can define $I_{NE}$ as follows.

$I_{NE} = \{ \langle \hat{K}_1 \xrightarrow{\hat{l}_1,\ldots,\hat{l}_{1,m}} \ldots, s, T, Z \rangle \mid \text{either} \\
\cdot \text{len} \left( \hat{K}_1 \xrightarrow{\hat{l}_1,\ldots,\hat{l}_{1,m}} \ldots \right) = 0, \text{ or} \\
\cdot \text{len} \left( \hat{K}_1 \xrightarrow{\hat{l}_1,\ldots,\hat{l}_{1,m}} \ldots \right) > 0 \text{ and there exist } S_1 \xrightarrow{\langle l_1, O_1 \rangle} \ldots \text{ and } X \text{ such that} \\
\quad - S_1 = \{ s \}, \\
\quad - L_1 \text{ is the (lexicographically) least summary-transition label such that} \\
\quad \hat{K}_1 \xrightarrow{L_1} \hat{K}_2 \text{ summarizes } \hat{K}_1 \xrightarrow{\hat{l}_1,\ldots,\hat{l}_{1,m}} \hat{K}_2 \text{ and } Z(\hat{K}_1 \xrightarrow{L_1} \hat{K}_2, s) \neq \emptyset, \\
\quad - \langle O_1, S_2 \rangle \in Z(\hat{K}_1 \xrightarrow{L_1} \hat{K}_2, s), \\
\quad - \text{dom}(X) = \bigcup_{s' \in S_2} \text{TransExec}_T(\hat{K}_2 \xrightarrow{\hat{l}_{2,1},\ldots,\hat{l}_{2,m}} \ldots, s'), \\
\quad - \forall \langle \hat{aex}, s'', A, M \rangle \in \text{dom}(X). \ X(\hat{aex}, s'', A, M) \in \text{SimExecs}(\hat{aex}, s'', T, Z), \text{ and} \\
\quad - S_2 \xrightarrow{\langle l_2, O_2 \rangle} \ldots = \bigcup_{s' \in S_2} \text{UntransExec}_T(\hat{K}_2 \xrightarrow{\hat{l}_{2,1},\ldots,\hat{l}_{2,m}} \ldots, s', X) \}

That definition simplifies to the following.
\[ I_{\text{NE}} = \begin{cases} \left\{ \left( \hat{K}_1 \xrightarrow{f_{1,1}, \ldots, f_{1,m}} \ldots, s, T, Z \right) \right\} \\
\text{if } \text{len}\left( \hat{K}_1 \xrightarrow{f_{1,1}, \ldots, f_{1,m}} \ldots \right) > 0, \text{ then there exist } S_1 \xrightarrow{(L_1, O_1)} \ldots \text{ and } X \text{ such that} \\
\begin{align*}
\bullet & \quad S_1 = \{s\}, \\
\bullet & \quad L_1 \text{ is the (lexicographically) least summary-transition label such that } \\
& \quad \hat{K}_1 \xrightarrow{L_1} \hat{K}_2 \text{ summarizes } \hat{K}_1 \xrightarrow{f_{1,1}, \ldots, f_{1,m}} \hat{K}_2 \text{ and } Z(\hat{K}_1 \xrightarrow{L_1} \hat{K}_2, s) \neq \emptyset, \\
\bullet & \quad (O_1, S_2) \in Z(\hat{K}_1 \xrightarrow{L_1} \hat{K}_2, s), \\
\bullet & \quad \text{dom}(X) = \bigcup_{s' \in S_2} \text{TransExec}(\hat{K}_2 \xrightarrow{f_{2,1}, \ldots, f_{2,m}} \ldots, s'), \\
\bullet & \quad \forall \langle aex, s'', A, M \rangle \in \text{dom}(X), X(aex, s'', A, M) \in \text{SimExecs}(aex, s'', T, Z), \text{ and} \\
\bullet & \quad S_2 \xrightarrow{(L_2, O_2)} \ldots = \bigcup_{s' \in S_2} \text{UntransExec}(\hat{K}_2 \xrightarrow{f_{2,1}, \ldots, f_{2,m}} \ldots, s', X) 
\end{align*} \end{cases} \]

Suppose we restrict these tuples to those in which \( \hat{K}_1 \) and \( s \) are well-formed, \( \hat{K}_1 \) is an externals-only configuration, \( \text{Mon}(s) \subseteq \text{Used}(\hat{K}_1) \), and \( T \) is conformance-reflecting. By the Abstract Well-Formed Preservation lemma (appendix I), Abstract Externals-Only Preservation lemma (appendix I), and induction on the number of labels on the step \( \hat{K}_1 \xrightarrow{f_{1,1}, \ldots, f_{1,m}} \hat{K}_2 \), \( \hat{K}_2 \) is a well-formed, externals-only configuration. By corollary K.5.4 to the Specification Well-Formed Preservation lemma, every PSM \( s' \in S_2 \) is well-formed. By corollary K.10.1 to the Used/Monitored Marker lemma, for every PSM \( s' \in S_2, \text{Mon}(s') \subseteq \text{Used}(\hat{K}_2) \). By the definition of a summary transition, no actor in \( \hat{K}_2 \) is handling an event. Then by the Execution Simulation property for conformance reflection, if \( X \) contains a simulation for every member of \( \text{TransExec}(\hat{K}_2 \xrightarrow{f_{2,1}, \ldots, f_{2,m}} \ldots, s') \), then \( \text{UntransExec}(\hat{K}_2 \xrightarrow{f_{2,1}, \ldots, f_{2,m}} \ldots, s', X) \) is an execution \( S'_2 \xrightarrow{(L_2, O'_2)} \ldots \) such that \( S'_2 = \{s'\} \) and \( \text{Simulates}\left( S'_2 \xrightarrow{(L_2, O'_2)} \ldots, \hat{K}_2 \xrightarrow{f_{2,1}, \ldots, f_{2,m}} \ldots \right) \). In that case, the union \( S_2 \xrightarrow{(L_2, O_2)} \ldots \) of all such executions for \( s' \in S_2 \) is defined. Therefore, the set \( I'_{\text{NE}} \) defined below is a subset of \( I_{\text{NE}} \).
Let \((\tilde{a} \tilde{e}, s'', A, M)\) be one of the results of \(\text{TransExec}_T\), and let \(\tilde{K}_1^t\) be the first configuration of \(\tilde{a} \tilde{e}\). By the Initial Pair Correctness, Well-Formed Preservation, Externals-Only Preservation, and Used/Monitored Preservation conformance-reflection properties, \(\tilde{K}_1^t\) and \(s''\) are well-formed, \(\tilde{K}_1^t\) is an externals-only configuration, and \(\text{Mon}(s'') \subseteq \text{Used}(\tilde{K}_1^t)\). By the \(\text{SimExecs}\) Simulation lemma, every resulting specification execution from \(\text{SimExecs}\) simulates the given program execution. By the definition of \(\text{SimExecs}\), every one of those resulting executions begins with the configuration \(\{s''\}\), where \(s''\) is the PSM given to \(\text{SimExecs}\). Thus, we need only look for those inputs in which the next-level calls to \(\text{SimExecs}\) produce at least one result. We can define that set \(I''_{\text{NE}}\) as follows, and we have that \(I''_{\text{NE}} \subseteq I'_{\text{NE}}\).
\[
I''_{\text{NE}} = \left\{ \hat{K}_1 \xrightarrow{\hat{t}_{1,1}, \ldots, \hat{t}_{1,m}} \ldots, s, T, Z \right\}
\]

if \( \text{len}(\hat{K}_1 \xrightarrow{\hat{t}_{1,1}, \ldots, \hat{t}_{1,m}} \ldots) > 0, \hat{K}_1 \) and \( s \) are well-formed, \( \hat{K}_1 \) is an externals-only configuration, \( \text{Mon}(s) \subseteq \text{Used}(\hat{K}_1) \), and \( T \) is conformance-reflecting, then there exist \( L_1, O_1, \) and \( S_2 \) such that

- \( L_1 \) is the (lexicographically) least summary-transition label such that \( \hat{K}_1 \xrightarrow{L_1} \hat{K}_2 \) summarizes \( \hat{K}_1 \xrightarrow{\hat{t}_{1,1}, \ldots, \hat{t}_{1,m}} \hat{K}_2 \) and \( Z(\hat{K}_1 \xrightarrow{L_1} \hat{K}_2, s) \neq \emptyset \),

- \( \langle O_1, S_2 \rangle \in Z(\hat{K}_1 \xrightarrow{L_1} \hat{K}_2, s) \), and

- for all \( s' \in S_2 \) and all \( \langle \hat{a} \hat{e} \hat{x}, s'', A, M \rangle \in \text{TransExec}(\hat{K}_2 \xrightarrow{\hat{t}_{2,1}, \ldots, \hat{t}_{2,m}} \ldots, s') \),

\( \langle \hat{a} \hat{e} \hat{x}, s'', T, Z \rangle \neq \emptyset \)

Of course, because \( I''_{\text{NE}} \subseteq I'_{\text{NE}} \subseteq I_{\text{NE}} \), we know that \( I''_{\text{NE}} \) contains only tuples for which \( \text{SimExec} \) returns at least one result. Thus, to simplify the last bullet, we can define a subset \( I''_{\text{NE}} \) of \( I''_{\text{NE}} \) as a greatest fixed point \( I''_{\text{NE}} = Y.F(Y) \), where \( F(Y) \) is defined as follows.

\[
F(Y) = \left\{ \hat{K}_1 \xrightarrow{\hat{t}_{1,1}, \ldots, \hat{t}_{1,m}} \ldots, s, T, Z \right\}
\]

if \( \text{len}(\hat{K}_1 \xrightarrow{\hat{t}_{1,1}, \ldots, \hat{t}_{1,m}} \ldots) > 0, \hat{K}_1 \) and \( s \) are well-formed, \( \hat{K}_1 \) is an externals-only configuration, \( \text{Mon}(s) \subseteq \text{Used}(\hat{K}_1) \), and \( T \) is conformance-reflecting, then there exist \( L_1, O_1, \) and \( S_2 \) such that

- \( L_1 \) is the (lexicographically) least summary-transition label such that \( \hat{K}_1 \xrightarrow{L_1} \hat{K}_2 \) summarizes \( \hat{K}_1 \xrightarrow{\hat{t}_{1,1}, \ldots, \hat{t}_{1,m}} \hat{K}_2 \) and \( Z(\hat{K}_1 \xrightarrow{L_1} \hat{K}_2, s) \neq \emptyset \),

- \( \langle O_1, S_2 \rangle \in Z(\hat{K}_1 \xrightarrow{L_1} \hat{K}_2, s) \), and

- for all \( s' \in S_2 \) and all \( \langle \hat{a} \hat{e} \hat{x}, s'', A, M \rangle \in \text{TransExec}(\hat{K}_2 \xrightarrow{\hat{t}_{2,1}, \ldots, \hat{t}_{2,m}} \ldots, s') \),

\( \langle \hat{a} \hat{e} \hat{x}, s'', T, Z \rangle \in Y \)

We know that \( I''_{\text{NE}} \subseteq I''_{\text{NE}} \subseteq I'_{\text{NE}} \subseteq I_{\text{NE}} \), so to show that \( I_{\text{TCD}} \subseteq I_{\text{NE}} \), it suffices to show that \( I_{\text{TCD}} \subseteq I''_{\text{NE}} \). By coinduction, it suffices to show that \( I_{\text{TCD}} \) is \( F \)-dense: that is, that \( I_{\text{TCD}} \subseteq F(I_{\text{TCD}}) \).

Let \( \hat{K}_1 \xrightarrow{\hat{t}_{1,1}, \ldots, \hat{t}_{1,m}} \ldots, s, T, Z \) be a member of \( I_{\text{TCD}} \); We must show that

\( \langle \hat{K}_1 \xrightarrow{\hat{t}_{1,1}, \ldots, \hat{t}_{1,m}} \ldots, s, T, Z \rangle \in F(I_{\text{TCD}}) \). If \( \text{len}(\hat{K}_1 \xrightarrow{\hat{t}_{1,1}, \ldots, \hat{t}_{1,m}} \ldots) = 0 \), then we're
done. Otherwise, by the definition of $I_{TCD}$, we know that there exists $R$ such that

- $\hat{K}_1$ and $s$ are well-formed,
- $\hat{K}_1$ is an externals-only configuration,
- $\text{Mon}(s) \subseteq \text{Used}(\hat{K}_1)$,
- $R$ is a transformation-conformance-dense relation for $T$ with witness $Z$, and
- $\langle \hat{K}_1, s \rangle \in R$.

Furthermore, as argued above, $\hat{K}_2$ is a well-formed, externals-only configuration by the Abstract Well-Formed Preservation lemma, Abstract Externals-Only Preservation lemma, and induction on the number of labels on the step $\hat{K}_1 \xrightarrow{L_1} \hat{K}_2$. By the definition of a summary transition, no actor in $\hat{K}_2$ is handling an event. By the definition of a transformation-conformance-dense relation, $T$ is conformance-reflecting. By the Simulation condition for a transformation-conformance-dense relation, there exists $L_1, O_1, S_2$ such that

- $\hat{K}_1 \xrightarrow{L_1} \hat{K}'$ summarizes $\hat{K}_1 \xrightarrow{L_1, \ldots, L_1, aex} \hat{K}_2$,
- $\langle O_1, S_2 \rangle \in Z(\hat{K}_1 \xrightarrow{L_1} \hat{K}_2, s)$, and
- for all transformation steps $\langle \hat{K}_1, s \rangle \xrightarrow{L_1, \hat{K}_2, Z, O_1, S_2, T, A, M, \hat{K}_2, s''} \langle \hat{K}_2, s'' \rangle$, $\langle \hat{K}_2, s'' \rangle \in R$.

Assume $L_1$ is selected such that it is the lexicographically least summary-transition label such that $\hat{K}_1 \xrightarrow{L_1} \hat{K}_2$ summarizes $\hat{K}_1 \xrightarrow{L_1, \ldots, L_1, aex} \hat{K}_2$ and $Z(\hat{K}_1 \xrightarrow{L_1} \hat{K}_2, s) \neq \emptyset$.

Let $s'$ be a member of $S_2$, and let $\langle a\hat{a}x, s'', A, M \rangle$ be a member of $\text{TransExec}(\hat{K}_2 \xrightarrow{L_2, \ldots, L_2, a\hat{a}x} \ldots, s')$, with $\hat{a\hat{a}x} = \hat{K}_2 \xrightarrow{L_2, \ldots, L_2, a\hat{a}x} \ldots$. By corollary K.5.4 to the Specification Well-Formed Preservation lemma, $s'$ is well-formed. By corollary K.10.1 to the Used/Monitored Marker lemma, $\text{Mon}(s') \subseteq \text{Used}(\hat{K}_2)$. It remains to show that $\langle a\hat{a}x, s'', T, Z \rangle \in I_{TCD}$. We show each of the points for membership in $I_{TCD}$ below, using the same relation $R$ as the necessary witness. In several of the below bullets, we rely on the fact that $\langle \hat{K}_2, s'', A, M \rangle \in T(\hat{K}_2, s')$ by the Initial Pair Correctness conformance-reflection property.

- By the Well-Formed Preservation conformance-reflection property, $\hat{K}_2$ and $s''$ are well-formed.
- By the Externals-Only Preservation conformance-reflection property, $\hat{K}_2$ is an externals-only configuration.
• By the Used/Monitored Preservation conformance-reflection property, \( Mon(s'') \subseteq \text{Used(}K'_2\text{)} \).

• We already know that \( R \) is a transformation-conformance-dense relation for \( T \) with witness \( Z \).

• By the definition of the transformation-step relation, there exists a transformation step \( \langle \hat{K}_1, s \rangle \xrightarrow{L_1, \hat{K}_2, Z, O_1, S_2, T, A, M} \langle \hat{K}_2, s'' \rangle \). Then by the results of the Simulation condition for a transformation-conformance-dense relation above, we know that \( \langle \hat{K}_2', s'' \rangle \in R \).

\[ \Box \]

**K.26 SimExecs Determinism Lemma**

**Lemma (SimExecs Determinism).** For all \( \hat{K}_1 \xrightarrow{I_1, \ldots, I_n} \ldots, s, T, \text{ and } Z \), if \( Z \) is deterministic, then \( |\text{SimExecs}(\hat{K}_1 \xrightarrow{I_1, \ldots, I_n} \ldots, s, T, Z)| \leq 1 \).

**Proof.** If \( \text{SimExecs}(\hat{K}_1 \xrightarrow{I_1, \ldots, I_n} \ldots, s, T, Z) = \emptyset \), then we're done. Otherwise, let \( S_1 \xrightarrow{\langle L_1, O_1 \rangle} \ldots \) and \( S_1' \xrightarrow{\langle L_1', O_1' \rangle} \ldots \) be members of \( \text{SimExecs}(\hat{K}_1 \xrightarrow{I_1, \ldots, I_n} \ldots, s, T, Z) \). We will show that \( S_1 \xrightarrow{\langle L_1, O_1 \rangle} \ldots = S_1' \xrightarrow{\langle L_1', O_1' \rangle} \ldots = \langle s \rangle \). Otherwise, there exists \( X \) such that

• \( S_1 = \langle s \rangle \),

• \( L_1 \) is the (lexicographically) least summary-transition label such that \( \hat{K}_1 \xrightarrow{L_1} \hat{K}_2 \) summarizes \( \hat{K}_1 \xrightarrow{I_1, \ldots, I_m} \hat{K}_2 \) and \( Z(\hat{K}_1 \xrightarrow{L_1} \hat{K}_2, s) \neq \emptyset \),

• \( \langle O_1, S_2 \rangle \in Z(\hat{K}_1 \xrightarrow{L_1} \hat{K}_2, s) \),

• \( \text{dom}(X) = \bigcup_{s' \in S_2} \text{TransExec}_T(\hat{K}_2 \xrightarrow{I_2, \ldots, I_m} \ldots, s') \),

• \( \forall \langle aex, s'', A, M \rangle \in \text{dom}(X), X(aex, s'', A, M) \in \text{SimExecs}(aex, s'', T, Z) \), and

• \( S_2 \xrightarrow{\langle L_2, O_2 \rangle} \ldots \cup_{s' \in S_2} \text{UntransExec}_T(\hat{K}_2 \xrightarrow{I_2, \ldots, I_m} \ldots, s', X) \).

Similarly, there exists \( X' \) such that

• \( S_1' = \langle s \rangle \),

• \( L_1' \) is the (lexicographically) least summary-transition label such that \( \hat{K}_1 \xrightarrow{L_1'} \hat{K}_2 \) summarizes \( \hat{K}_1 \xrightarrow{I_1, \ldots, I_m} \hat{K}_2 \) and \( Z(\hat{K}_1 \xrightarrow{L_1'} \hat{K}_2, s) \neq \emptyset \),

...
We have UntransExec definitions, \(X\) which completes the proof.

Because \(L_1\) and \(L'_1\) are both the least label satisfying the listed conditions, \(L_1 = L'_1\). Then because \(Z\) is deterministic, \(O_1 = O'_1\) and \(S_2 = S'_2\). That in turn implies that \(\text{dom}(X) = \text{dom}(X')\), by the above definitions.

Next, let \(\langle \alpha\xi, s'', A, M \rangle\) be a member of \(\text{dom}(X)\). By the above definitions, \(X(\alpha\xi, s'', A, M) \in \text{SimExecs}(\alpha\xi, s'', T, Z)\) and \(X'(\alpha\xi, s'', A, M) \in \text{SimExecs}(\alpha\xi, s'', T, Z)\). By coinduction, \(|\text{SimExecs}(\alpha\xi, s'', T, Z)| \leq 1\), so \(X(\alpha\xi, s'', A, M) = X'(\alpha\xi, s'', A, M)\). Therefore, \(X = X'\).

Finally, for all \(s' \in S_2\), because \(X = X'\), we have
\[
\text{UntransExec}_{\bar{K}_2}(\langle l_2, o_2 \rangle) = \text{UntransExec}_{\bar{K}_2}(\langle l_2, o_2 \rangle)
\]
Next, let \(\langle \alpha\xi, s'', A, M \rangle\) be a member of \(\text{dom}(X)\). By the above definitions, \(X(\alpha\xi, s'', A, M) \in \text{SimExecs}(\alpha\xi, s'', T, Z)\) and \(X'(\alpha\xi, s'', A, M) \in \text{SimExecs}(\alpha\xi, s'', T, Z)\). By coinduction, \(|\text{SimExecs}(\alpha\xi, s'', T, Z)| \leq 1\), so \(X(\alpha\xi, s'', A, M) = X'(\alpha\xi, s'', A, M)\). Therefore, \(X = X'\).

Finally, for all \(s' \in S_2\), because \(X = X'\), we have
\[
\text{UntransExec}_{\bar{K}_2}(\langle l_2, o_2 \rangle) = \text{UntransExec}_{\bar{K}_2}(\langle l_2, o_2 \rangle)
\]
which completes the proof.

\[\square\]

**K.27 SimExecs Common Prefix Lemma**

**Lemma (SimExecs Common Prefix).** For all \(\bar{K}_1 \xrightarrow{l_{1,1}, \ldots, l_{1,n}} \ldots, \bar{K}'_1 \xrightarrow{l'_{1,1}, \ldots, l'_{1,n}} \ldots\), \(S_1 \xrightarrow{(l_1, o_1)} \ldots, s, T, Z, R, and i, if**

- \(\bar{K}_1\) and \(s\) are well-formed,
- \(\bar{K}_1\) is an externals-only configuration,
- \(\text{Mono}(s) \subseteq \text{Used}(\bar{K}_1)\),
- \(\bar{K}_1 \xrightarrow{l_{1,1}, \ldots, l_{1,n}} \ldots\) and \(\bar{K}_1' \xrightarrow{l'_{1,1}, \ldots, l'_{1,n}} \ldots\) share a prefix of length \(i\),
- \(Z\) is deterministic,
- \(S_1 \xrightarrow{(l_1, o_1)} \ldots \in \text{SimExecs}(\bar{K}_1 \xrightarrow{l_{1,1}, \ldots, l_{1,n}} \ldots, s, T, Z)\),
- \(R\) is a transformation-conformance-dense relation for \(T\) with witness \(Z\), and
- \(\langle \bar{K}_1, s \rangle \in R\),
then there exists $S'_1 \langle L'_1, O'_1 \rangle \ldots \in \text{SimExecs}(\bar{K}_1 \xrightarrow{\vec{p}_{1,1} \ldots \vec{p}_{1,n}} \ldots, s, T, Z)$ such that $S_1 \langle L_1, O_1 \rangle \ldots$ and $S'_1 \langle L'_1, O'_1 \rangle \ldots$ share a prefix of length $i$.

Proof. The proof is by induction on $i$. In the base case when $i = 0$, $S_1 = \{s\}$ by the definition of SimExecs. We also have that $\bar{K}_1 = \bar{K}'_1$, so $\langle \bar{K}'_1, s \rangle \in R$ and $\bar{K}'_1$ is well-formed, $\bar{K}'_1$ is an externals-only configuration, and $\text{Mon}(s) \subseteq \text{Used}(\bar{K}'_1)$.

Then by the SimExecs Non-Emptyness lemma, there exists $S'_1 \langle L'_1, O'_1 \rangle \ldots \in \text{SimExecs}(\bar{K}'_1 \xrightarrow{\vec{p}_{1,1} \ldots \vec{p}_{1,n}} \ldots, s, T, Z)$. Again $S'_1 = \{s\}$ by the definition of SimExecs, so $S_1 \langle L_1, O_1 \rangle \ldots$ and $S'_1 \langle L'_1, O'_1 \rangle \ldots$ share a prefix of length 0.

In the inductive case, it must be the case that both $\text{len}\left(\bar{K}_1 \xrightarrow{\vec{p}_{1,1} \ldots \vec{p}_{1,n}} \ldots\right) > 0$ and $\text{len}\left(\bar{K}'_1 \xrightarrow{\vec{p}_{1,1} \ldots \vec{p}_{1,n}} \ldots\right) > 0$. Then by the definition of SimExecs, there exists some $X$ such that

- $S_1 = \{s\}$,
- $L_1$ is the (lexicographically) least summary-transition label such that $\bar{K}_1 \xrightarrow{L_1} \bar{K}_2$ summarizes $\bar{K}_1 \xrightarrow{\vec{p}_{1,1} \ldots \vec{p}_{1,m}} \bar{K}_2$ and $Z(\bar{K}_1 \xrightarrow{L_1} \bar{K}_2, s) \neq \emptyset$,
- $(O_1, S_2) \in Z(\bar{K}_1 \xrightarrow{L_1} \bar{K}_2, s)$,
- $\text{dom}(X) = \bigcup_{s' \in S_2} \text{TransExec}_T(\bar{K}_2 \xrightarrow{\vec{l}_{2,1} \ldots \vec{l}_{2,m}} \ldots, s')$,
- $\forall \langle \bar{a}\bar{e}, s'', A, M \rangle \in \text{dom}(X), X(\bar{a}\bar{e}, s'', A, M) \in \text{SimExecs}(\bar{a}\bar{e}, s'', T, Z)$, and
- $(O_2, S_2) \xrightarrow{\langle L_2, O_2 \rangle} \ldots = \bigcup_{s' \in S_2} \text{UntransExec}_T(\bar{K}_2 \xrightarrow{\vec{l}_{2,1} \ldots \vec{l}_{2,m}} \ldots, s', X)$.

Because $i > 0$, we know that $\bar{K}_1 = \bar{K}'_1$, $\bar{K}_2 = \bar{K}'_2$, and $\vec{l}_{1,j} = \vec{l}'_{1,j}$ for all $j \in 1 \ldots n$. Therefore, we have

- $L_1$ is the (lexicographically) least summary-transition label such that $\bar{K}'_1 \xrightarrow{L_1} \bar{K}'_2$ summarizes $\bar{K}'_1 \xrightarrow{\vec{p}_{1,1} \ldots \vec{p}_{1,m}} \bar{K}'_2$ and $Z(\bar{K}'_1 \xrightarrow{L_1} \bar{K}'_2, s) \neq \emptyset$ and
- $(O_1, S_2) \in Z(\bar{K}'_1 \xrightarrow{L_1} \bar{K}'_2, s)$.

Let $L'_1 = L_1$, $O'_1 = O_1$, $S'_1 = \{s\}$, and $S'_2 = S_2$. It remains to show that there exists some $X'$ and $S'_2 \langle L'_2, O'_2 \rangle \ldots$ such that

- $\text{dom}(X') = \bigcup_{s' \in S_2} \text{TransExec}_T(\bar{K}'_2 \xrightarrow{\vec{p}_{2,1} \ldots \vec{p}_{2,m}} \ldots, s')$,
- $\forall \langle \bar{a}\bar{e}, s'', A, M \rangle \in \text{dom}(X'), X'(\bar{a}\bar{e}, s'', A, M) \in \text{SimExecs}(\bar{a}\bar{e}, s'', T, Z)$,
K.27. SIMEXEC COMMON PREFIX LEmMA

\[ S'_2 \xrightarrow{\langle L'_2, O'_2 \rangle} \ldots = \bigcup_{s' \in S'_2} \text{UntransExec}_T(\hat{K}_2 \xrightarrow{\hat{I}_2, \ldots, \hat{I}_{2,m}} \ldots, s', X'), \text{ and} \]

\[ S_2 \xrightarrow{\langle L_2, O_2 \rangle} \ldots \text{ and } S'_2 \xrightarrow{\langle L'_2, O'_2 \rangle} \ldots \text{ share a prefix of length } i - 1. \]

Let \( s' \) be a member of \( S'_2 \), let \( \langle \bar{a}\bar{e}x, s'', A, M \rangle \) be a member of \( \text{SimExecs}(\hat{K}_2) \), and let \( \hat{K}'' \) be the first configuration in \( \bar{a}\bar{e}x \).

Because \( \hat{K}' = \hat{K}_1 \), \( \hat{K}_1 \) is well-formed. By the Abstract Well-Formed Preservation lemma (appendix I), Abstract Externals-Only Preservation lemma (also appendix I), and induction on \( m \), \( \hat{K}_1 \) is a well-formed, externals-only configuration. Because \( S'_1 = \{ s \} \) and \( s \) is well-formed, by corollary K.5.4 to the Specification Well-Formed Preservation lemma, \( s' \) is well-formed. By corollary K.10.1 to the Used/Monitored Marker lemma, \( \text{Mon}(s') \subseteq \text{Used}(\hat{K}_2) \).

Therefore, by the SimExecs Non-Emptyness lemma, \( \text{SimExecs}(\bar{a}\bar{e}x, s'', T, Z) \neq \emptyset \). Thus, define \( X' \) such that

\[ \text{dom}(X') = \bigcup_{s' \in S'_2} \text{SimExecs}(\hat{K}_2 \xrightarrow{I_2, \ldots, I_{2,m}} \ldots, s') \]

\[ \forall \langle \bar{a}\bar{e}x, s'', A, M \rangle \in \text{dom}(X'). X'(\bar{a}\bar{e}x, s'', A, M) \in \text{SimExecs}(\bar{a}\bar{e}x, s'', T, Z). \]

It remains to show that there exists \( S'_2 \) such that

\[ S'_2 \xrightarrow{\langle L'_2, O'_2 \rangle} \ldots = \bigcup_{s' \in S'_2} \text{UntransExec}_T(\hat{K}_2 \xrightarrow{I_2, \ldots, I_{2,m}} \ldots, s', X') \]

\[ S_2 \xrightarrow{\langle L_2, O_2 \rangle} \ldots \text{ and } S'_2 \xrightarrow{\langle L'_2, O'_2 \rangle} \ldots \text{ share a prefix of length } i - 1. \]

By the definition of \( S_2 \), it is sufficient to show that for all \( s' \in S_2 \), \( \text{UntransExec}_T(\hat{K}_2 \xrightarrow{I_2, \ldots, I_{2,m}} \ldots, s', X) \) and \( \text{UntransExec}_T(\hat{K}_2 \xrightarrow{I_2, \ldots, I_{2,m}} \ldots, s', X') \) share a prefix of length \( i - 1 \). We will show that this holds through a use of the Prefix Consistency property for conformance reflection.

Let \( s' \) be a member of \( S_2 \). We already know that \( \hat{K}_2 \xrightarrow{I_2, \ldots, I_{2,m}} \ldots \) and \( \hat{K}_2 \xrightarrow{I_2, \ldots, I_{2,m}} \ldots \) share a prefix of length \( i - 1 \). We have that \( \text{SimExecs}(\hat{K}_1 \xrightarrow{I_1, \ldots, I_{1,m}} \ldots, s') \subseteq \text{dom}(X') \), and by the SimExecs Simulation lemma, \( \text{Simulates}(X'(\bar{a}\bar{e}x, s'', A, M), \hat{K}_2 \xrightarrow{I_2, \ldots, I_{2,m}} \ldots) \) for all \( \langle \bar{a}\bar{e}x, s'', A, M \rangle \in \).
\( \text{TransExec}(\hat{R}_2 \xrightarrow{\hat{P}_{2,1} \ldots \hat{P}_{2,m}} \ldots, s') \). Furthermore, by the definition of \( \text{SimExecs} \), \( \{s''\} \) is the first configuration of \( X'(\hat{a}\hat{e}\hat{x}, s'', A, M) \) for all \( \langle \hat{a}\hat{e}\hat{x}, s'', A, M \rangle \in \text{TransExec}(\hat{R}_2 \xrightarrow{\hat{P}_{2,1} \ldots \hat{P}_{2,m}} \ldots, s') \). Then by the definition of conformance-reflection, \( \text{UntransExec}(\hat{R}_2 \xrightarrow{\hat{P}_{2,1} \ldots \hat{P}_{2,m}} \ldots, s', X') \) is defined.

Finally, let there be \( \hat{a}\hat{e}\hat{x}, \hat{a}\hat{e}\hat{x}', s'', A, M \), and \( j \) such that

- \( \langle \hat{a}\hat{e}\hat{x}, s'', A, M \rangle \in \text{TransExec}(\hat{R}_2 \xrightarrow{\hat{P}_{2,1} \ldots \hat{P}_{2,m}} \ldots, s') \),
- \( \langle \hat{a}\hat{e}\hat{x}', s'', A, M \rangle \in \text{TransExec}(\hat{R}_2' \xrightarrow{\hat{P}_{2,1} \ldots \hat{P}_{2,m}} \ldots, s') \),
- \( \hat{a}\hat{e}\hat{x} \) and \( \hat{a}\hat{e}\hat{x}' \) share a prefix of length \( j \), and
- \( j \leq i - 1 \).

By the definition of \( X \) and \( X' \), \( X(\hat{a}\hat{e}\hat{x}, s'', A, M) \in \text{SimExecs}(\hat{a}\hat{e}\hat{x}, s'', T, Z) \) and \( X'(\hat{a}\hat{e}\hat{x}', s'', A, M) \in \text{SimExecs}(\hat{a}\hat{e}\hat{x}', s'', T, Z) \). By the \( \text{SimExecs} \) Determinism lemma, \( \text{SimExecs}(\hat{a}\hat{e}\hat{x}, s'', T, Z) = \{X(\hat{a}\hat{e}\hat{x}, s'', A, M)\} \) and \( \text{SimExecs}(\hat{a}\hat{e}\hat{x}', s'', T, Z) = \{X'(\hat{a}\hat{e}\hat{x}', s'', A, M)\} \). Then by the induction hypothesis for this lemma, \( X(\hat{a}\hat{e}\hat{x}, s'', A, M) \) and \( X'(\hat{a}\hat{e}\hat{x}', s'', A, M) \) share a prefix of length \( j \). Therefore, the conditions for the Prefix Consistency property hold, and \( \text{UntransExec}(\hat{R}_2 \xrightarrow{\hat{P}_{2,1} \ldots \hat{P}_{2,m}} \ldots, s', X) \) and \( \text{UntransExec}(\hat{R}_2' \xrightarrow{\hat{P}_{2,1} \ldots \hat{P}_{2,m}} \ldots, s', X') \) share a prefix of length \( i - 1 \), which completes the proof.

\[ \square \]

**K.28 SimExecs Fulfillment Lemma**

**Lemma** (SimExecs Fulfillment). For all \( \hat{a}\hat{e}\hat{x} = \hat{K}_1 \xrightarrow{\hat{L}_{1,1} \ldots \hat{L}_{1,n}} \ldots, s, R, T, Z \), and \( S_1 \xrightarrow{(L_1, O_1)} \ldots \), if

- \( R \) is a transformation-conformance-dense relation for \( T \) with witness \( Z \),
- \( \langle \hat{K}_1, s \rangle \in R \),
- \( \hat{K}_1 \) and \( s \) are well-formed,
- \( \hat{K}_1 \) is an externals-only configuration,
- \( \text{Mon}(s) \subseteq \text{Used}(\hat{K}_1) \),
- \( \hat{a}\hat{e}\hat{x} \) is fair,
- \( S_1 \xrightarrow{(L_1, O_1)} \ldots \in \text{SimExecs}(\hat{a}\hat{e}\hat{x}, s, T, Z) \), and
- \( T \) is conformance-reflecting,
then $S_1 \xrightarrow{\langle L_1, O_1 \rangle} \ldots$ is fair.

**Proof.** Let $S_i$ be a configuration in $S_1 \xrightarrow{\langle L_i, O_i \rangle} \ldots$. We will show that for all $\langle \eta, po \rangle \in \text{Obls}(S_i)$, there exists some step $S_{i+j} \xrightarrow{\langle L_{i+j}, O_{i+j} \rangle} S_{i+j+1}$ in that execution such that $\langle \eta, po \rangle \in O_{i+j}$. The proof is by induction on $i$.

**Base Case**

In this case, $i = 1$, and therefore by the definition of $\text{SimExecs}$, $S_1 = \{s\}$. Let $\langle \eta, po \rangle$ be a member of $\text{Obls}(S_1)$. Assume that there is no $O_j$ in $S_1 \xrightarrow{\langle L_1, O_1 \rangle} \ldots$ such that $\langle \eta, po \rangle \in O_j$. We will show that this implies the existence of a transformation-step execution from $\langle \hat{K}_1, s \rangle$ that is program-fair but not specification-fair. That would in turn imply that $R$ is not a transformation-conformance-dense relation, which contradicts a premise of this lemma.

Define $N$ to be the set of tuples of the form $\langle \hat{aex}', S_1' \xrightarrow{\langle L_1', O_1' \rangle} \ldots, \eta' \rangle$, with $\hat{aex}' = \hat{K}_1' \xrightarrow{p_{1,1}} \ldots$ and $S_1' = \{s'\}$, such that

- $S_1' \xrightarrow{\langle L_1', O_1' \rangle} \ldots \in \text{SimExecs}(\hat{aex}', s', T, Z),$
- $\hat{K}_1'$ and $s'$ are well-formed,
- $\hat{K}_1'$ is an externals-only configuration,
- $\text{Mon}(s') \subseteq \text{Used}(\hat{K}_1'),$
- $\hat{aex}'$ is fair,
- $\langle \eta', po \rangle \in \text{Obls}(s'),$
- there exists no $O_j$ in $S_1' \xrightarrow{\langle L_1', O_1' \rangle} \ldots$ such that $\langle \eta', po \rangle \in O_j$.

Let $\langle \hat{aex}', S_1' \xrightarrow{\langle L_1', O_1' \rangle} \ldots, \eta' \rangle$ be a member of $N$, with $\hat{aex}' = \hat{K}_1' \xrightarrow{p_{1,1}} \ldots$ and $S_1' = \{s'\}$. If $\text{len}(\hat{K}_1' \xrightarrow{p_{1,1}} \ldots) > 0$, then by the definition of $\text{SimExecs}$, there exists $X$ such that

- $L_1'$ is the (lexicographically) least summary-transition label such that $\hat{K}_1' \xrightarrow{L_1'} \hat{K}_2'$ summarizes $\hat{K}_1' \xrightarrow{p_{1,1}} \ldots \hat{K}_2'$ and $Z(\hat{K}_1' \xrightarrow{L_1'} \hat{K}_2', s') \neq \emptyset$,
- $\langle O_1', S_2' \rangle \in Z(\hat{K}_1' \xrightarrow{L_1'} \hat{K}_2', s'),$
- $\text{dom}(X) = \bigcup_{s \in S_2} \text{TransExec}(\hat{K}_2' \xrightarrow{p_{2,1}} \ldots, s''),$
APPENDIX K. PROOFS FOR TRANSFORMATION CONFORMANCE

452

• ∀(aex"", s"", A, M) ∈ dom(X). X(aex"", s"", A, M) ∈ SimExecs(aex"", s"", T, Z), and

• S_2 \xrightarrow{(L_2', O_2')} \ldots \xrightarrow{\bigcup_{s'' \in S_2} \text{UntransExec}_T(\tilde{K}_2'^2 \xrightarrow{p_{2,1}, \ldots, p_{2,m}} \ldots, s'', X)}.

By the Abstract Well-Formed Preservation lemma (appendix I), the Abstract Externals-Only Preservation lemma (also appendix I), and induction on m, K_2' is a well-formed, externals-only configuration. By corollary K.5.4 to the Specification Well-Formed Preservation lemma, every PSM s'' ∈ S_2' is well-formed. By corollary K.10.1 to the Used/Monitored Marker lemma, Mon(s'') ⊆ Used(\tilde{K}_2') for all s'' ∈ S_2'. By the definition of N, \langle η', po⟩ ∈ O_1'. By the definition of a summary transition, no actor in \tilde{K}_2' is handling an event. Then by the definition of a specification summary transition, there exists some s'' ∈ S_2' such that \langle η', po⟩ ∈ Obls(s'').

By the SimExecs Simulation lemma and the definition of SimExecs, X contains a simulation for every member of TransExec_T(\tilde{K}_2' \xrightarrow{p_{2,1}, \ldots, p_{2,m}} \ldots, s''). Therefore, by the Fulfillment Reflection 1 property for conformance reflection, there exists some (aex"", s"", A, M) ∈ TransExec_T(\tilde{K}_2' \xrightarrow{p_{2,1}, \ldots, p_{2,m}} \ldots, s'') and S_1' \xrightarrow{\text{SimExecs}(aex"", s"", T, Z)} \xrightarrow{\ldots} such that

• S_1' \xrightarrow{(L_1', O_1')} \ldots \xrightarrow{\text{SimExecs}(aex"", s"", T, Z)} \xrightarrow{\ldots} \langle M(\eta'), po⟩ ∈ Obls(s''), and

• there exists no O_2'' in S_1' \xrightarrow{(L_1', O_1')} \ldots \xrightarrow{\text{SimExecs}(aex"", s"", T, Z)} \xrightarrow{\ldots} such that \langle M(\eta'), po⟩ ∈ O_2''.

Furthermore, let \tilde{K}_1'' be the initial configuration in aex"": by the Well-Formed Preservation, Externals-Only Preservation, Used/Monitored Preservation, and Fairness Preservation 1 properties for conformance reflection, \tilde{K}_1'' and s'' are well-formed, K_1'' is an externals-only configuration, Mon(s'') ⊆ Used(K_1''), and aex"" is fair. Therefore, \langle aex"", S_1' \xrightarrow{Λ_1} \ldots, M(\eta') \rangle ∈ N. Furthermore, by the Initial Pair Correctness property, \langle \tilde{K}_1'', s'', A, M \rangle ∈ T(\tilde{K}_2', s'').

From the above argument, we can conclude that for all \langle \tilde{K}_1' \xrightarrow{p_{1,1}, \ldots, p_{1,n}} S_1' \xrightarrow{(L_1', O_1')} \ldots, η' \rangle ∈ N (with S_1' = \{s'\}), either

len(\tilde{K}_1' \xrightarrow{p_{1,1}, \ldots, p_{1,n}} \ldots) = 0, or there exist \tilde{K}_1'' \xrightarrow{p_{n,1}, \ldots, p_{n,n}} \ldots, S_1'' \xrightarrow{(L_1', O_1')} \ldots, s'', s'', A, and M such that

• s'' ∈ S_2',

• \langle \tilde{K}_1'' \xrightarrow{p_{n,1}, \ldots, p_{n,n}} \ldots, s'', A, M \rangle ∈ TransExec_T(\tilde{K}_2' \xrightarrow{p_{2,1}, \ldots, p_{2,m}} \ldots, s''),

• \langle \tilde{K}_1', s'' \rangle \xrightarrow{L_1', \tilde{K}_1'', Z, O_1', S_2', T, A, M} (\tilde{K}_1'', s'').
K.28. SIMEXECS FULFILLMENT LEMMA

\[ S''_1 = \{ s'' \}, \quad \text{and} \]
\[ \left\langle \hat{K}'', \hat{K}'', \hat{s}', \hat{s}' \right\rangle \in N. \]

Therefore, there exists a transformation-step execution
\[ \langle \hat{K}'_1, s'_1 \rangle \xrightarrow{L_1, \hat{K}, Z, O_1, S_1, T, A_1, M_1} \langle \hat{K}'_2, s'_2 \rangle \]

such that
\[ \hat{K}'_1 = \hat{K}_1 \text{ and } s'_1 = s, \]
\[ \text{each consecutive pair of configuration/PSM pairs } \langle \hat{K}'_i, s'_i \rangle \text{ and } \langle \hat{K}'_{i+1}, s'_{i+1} \rangle \]
\[ \text{is related as described in the above bullet points, and} \]
\[ \text{either the execution is infinite or it ends in some pair } \langle \hat{K}'_n, s'_n \rangle \text{ such that} \]
\[ \langle \hat{K}'_n, \{ s'_n \}, M_{n-1} \circ \cdots \circ M_1(\eta) \rangle \in N. \]

We will show that this execution is program-fair but not specification-fair.

To show that the execution is program-fair, let \( \hat{K}'_i \) be a configuration in the execution, and let \( \hat{a} \) identify a necessarily-enabled actor in \( \hat{K}'_i \). By the definition of the execution, there exists some fair execution \( \hat{K}'_1 \xrightarrow{\hat{I}_{11}, \cdots, \hat{I}_{1m}} \cdots \)
\[ \text{such that } \hat{K}'_1 = \hat{K}_1. \]
\[ \text{Because that execution is fair, there exists some } j \text{ such that the} \]
actor at \( \hat{a} \) in \( \hat{K}'_j \) is not necessarily enabled, or \( \hat{K}'_j \xrightarrow{\hat{I}_{j1}, \cdots, \hat{I}_{jm}} \hat{K}_{j+1} \) is a step in the execution and \( \hat{a} \) is the active actor for \( \hat{I}_{j1} \).
\[ \text{We will show by induction on } j \text{ that the transformation-step execution also either reaches a point where the} \]
\[ \text{corresponding actor is not necessarily enabled, or where that actor is the active actor for the next transition.} \]

In the base case, \( j = 1 \). Then either there is no necessarily-enabled actor at \( \hat{a} \) in \( \hat{K}'_1 \), or \( \hat{a} \) identifies the active actor for \( \hat{I}_{11} \), and therefore \( \hat{a} \) identifies the active actor for \( \hat{L}'_1 \). In the inductive case, let \( \hat{K}'_j \xrightarrow{\hat{I}_{j1}, \cdots, \hat{I}_{jm}} \cdots \)
\[ \text{be the execution associated with } \hat{K}'_{j+1} \text{ in the construction of the transformation-step execution.} \]
\[ \text{By the Fairness Preservation 2 property for conformance reflection, there exists some } k < j \text{ such that either the actor at } \]
\[ \hat{A}_i(\hat{a}) \text{ in } \hat{K}'_k \text{ is not necessarily enabled, or} \]
\[ \hat{K}'_k \xrightarrow{\hat{I}_{k1}, \cdots, \hat{I}_{km}} \hat{K}_{k+1} \text{ is a step in the execution and } \hat{A}_i(\hat{a}) \text{ is the active actor for } \hat{I}_{k1}. \]
\[ \text{Then by the induction hypothesis, there exists some } p \text{ such that either } \hat{K}'_{i+1+p} \text{ is} \]
a configuration in the transformation-step execution and there is no necessarily-enabled actor at \( \hat{A}_{i+p} \circ \cdots \circ \hat{A}_i(\hat{a}) \), or \( L_{i+1+p} \) is a label in the transformation-step execution and \( A_{i+p} \circ \cdots \circ \hat{A}_i(\hat{a}) \) identifies the active actor for \( L_{i+1+p} \).

The proof for the message-reception aspect of program-fairness is similar, using the Fairness Preservation 3 property in the inductive case. Thus, the execution
\[ \langle \hat{K}'_1, s'_1 \rangle \xrightarrow{L_1, \hat{K}, Z, O_1, S_1, T, A_1, M_1} \langle \hat{K}'_2, s'_2 \rangle \]
\[ \text{is program-fair.} \]

By the definition of the transformation-step execution and the definition of \( N \), for every PSM \( s'_i \) in the execution, \( \langle M_{i-1} \circ \cdots \circ M_1(\eta), p_o \rangle \in Obls(s'_i) \), and for every
$O_j'$ is in the execution, $\langle M_{i-1} \circ \ldots \circ M_1(\eta), po \rangle \in O_j'$. Therefore, the transformation-step execution is not specification-fair.

However, this leads to a contradiction. We know that $\langle \hat{K}_1, s \rangle \in R$, and because $R$ is transformation-conformance-dense, every program-fair transformation-step execution from $\langle \hat{K}_1, s \rangle$ using the transformation $T$ and strategy $Z$ is specification-fair. Then because $\hat{K}_1' = \hat{K}_1$ and $s_1' = s$, the above transformation-step execution is specification-fair. Therefore, by contradiction, there exists some $O_j$ in the given specification execution $S_1 \xrightarrow{(L_1,O_1)} \ldots$ such that $\langle \eta, po \rangle \in O_j$.

**Inductive Case**

In this case where $i > 1$, by the definition of $SimExecs$, there exists $X$ such that

- $L_1$ is the (lexicographically) least summary-transition label such that $\hat{K}_1 \xrightarrow{L_1} \hat{K}_2$ summarizes $\hat{K}_1 \xrightarrow{f_{1,1}\ldots f_{1,m}} \hat{K}_2$ and $Z(\hat{K}_1 \xrightarrow{L_1} \hat{K}_2, s) \neq \emptyset$,

- $\langle O_1, S_2 \rangle \in Z(\hat{K}_1 L_1 \hat{K}_2, s)$,

- $dom(X) = \bigcup_{s' \in S_2} TransExec_T(\hat{K}_2 \xrightarrow{f_{2,1}\ldots f_{2,m}} \ldots, s')$,

- $\forall \langle \hat{a}\hat{e}, s'', A, M \rangle \in dom(X), X(\hat{a}\hat{e}, s'', A, M) \in SimExecs(\hat{a}\hat{e}, s'', T, Z)$, and

- $S_2 \xrightarrow{(L_2,O_2)} \ldots = \bigcup_{s' \in S_2} UntransExec_T(\hat{K}_2 \xrightarrow{f_{2,1}\ldots f_{2,m}} \ldots, s', X)$.

Let $\langle \hat{a}\hat{e}, s'', A, M \rangle$ be a member of $dom(X)$, let $\hat{K}_1'$ be the first configuration in $\hat{a}\hat{e}$, and let $S_1' \xrightarrow{(L_1',O_1)} \ldots = X(\hat{a}\hat{e}, s'', A, M)$. By the Abstract Well-Formed Preservation lemma (appendix I), Abstract Externals-Only Preservation lemma (also appendix I), and induction on the number of labels in the first event step of $\hat{a}\hat{e}$, $\hat{K}_2$ is a well-formed, externals-only configuration. By corollary K.5.4 to the Specification Well-Formed Preservation lemma, every PSM $s' \in S_2$ is well-formed. By corollary K.10.1 to the Used/Monitored Marker lemma, $Mon(s') \subseteq Used(\hat{K}_2)$ for all $s' \in S_2$. By the definition of a summary transition, no actor in $\hat{K}_2$ is handling an event. By the Initial Pair Correctness property for conformance reflection, $\langle \hat{K}_1', s'', A, M \rangle \in T(\hat{K}_2, s')$. Therefore, $\langle \hat{K}_1', s'' \rangle \in R$. By the Well-Formed Preservation property for conformance reflection, $\hat{K}_1'$ and $s''$ are well-formed. By the Externals-Only Preservation property for conformance reflection, $\hat{K}_1'$ is an externals-only configuration. By the Used/Monitored Preservation property for conformance reflection, $Mon(s'') \subseteq Used(\hat{K}_1')$. By the Fairness Preservation 1 property, $a\hat{e}$ is fair. By the above definition, $S_1' \xrightarrow{(L_1',O_1)} \ldots \in SimExecs(a\hat{e}, s'', T, Z)$. Therefore, by the induction hypothesis, for all $j < i$ and all $\langle \eta', po' \rangle \in Obs(S_j')$, there exists some step $S_{j+k}' \xrightarrow{(L_{j+k},O_{j+k})} S_{j+k+1}'$ in that execution such that $\langle \eta', po' \rangle \in O_{j+k}'$. 


Let $s'$ be a member of $S_2$, and let \( S''_2 \overset{\langle L''_2,O''_2 \rangle}{\longrightarrow} \ldots = UntransExec_T(\vec{K}_2, I_{3,1}, I_{3,2}, \ldots, s', X) \). By the \textit{SimExecs} Simulation lemma and the definition of \textit{SimExecs}, $X$ contains a simulation for every member of \textit{TransExecT}(\vec{K}_2, I_{3,1}, I_{3,2}, \ldots, s'). Then by the Fulfillment Reflection 2 property for conformance reflection, for all \( \langle \eta, po \rangle \in Obls(S_1) \), there exists some step \( S''_{i+j} \overset{\langle L''_{i+j},O''_{i+j} \rangle}{\longrightarrow} S''_{i+j+1} \) in that execution such that \( \langle \eta, po \rangle \in O''_{i+j} \).

Then by the definition of a union over specification summary executions, for all \( \langle \eta, po \rangle \in Obls(S_1) \), there exists some step \( S_{i+j} \overset{\langle L_{i+j},O_{i+j} \rangle}{\longrightarrow} S_{i+j+1} \) in that execution such that \( \langle \eta, po \rangle \in O_{i+j} \).

\[ \square \]

### K.29 Transformation Conformance Theorem

**Theorem** (Transformation Conformance). For all $P$ and $\Sigma$, if $P \models_{TR} \Sigma$, then $P \models_{S} \Sigma$.

**Proof.** Let there be $P$ and $\Sigma$ such that $P \models_{TR} \Sigma$. By the definition of $\models_{TR}$, there must exist some maximal instantiation $\langle \vec{K}_{\text{init}}, \{ s_{\text{init}} \} \rangle$ of $P$ and $\Sigma$ such that $\vec{K}_{\text{init}} \models_{TR} s_{\text{init}}$. We will show that $\vec{K}_{\text{init}} \models_{S} s_{\text{init}}$, which implies $P \models_{S} \Sigma$.

By the definition of $\models_{TR}$, there must exist some $R_{TR}$, $T$, and $Z$ such that $R_{TR}$ is a transformation-conformance-dense relation for some $T$ with witness $Z$, and $\langle \vec{K}_{\text{init}}, s_{\text{init}} \rangle \in R_{TR}$. We will use this relation to define a separate summary-conformance-dense relation that proves $\vec{K}_{\text{init}} \models_{S} s_{\text{init}}$.

Let $Z'$ be a deterministic version of $Z$. That is, for all $\langle \vec{K} \overset{L}{\longrightarrow} \vec{K}', s \rangle \in \text{dom}(Z)$, either $Z(\vec{K} \overset{L}{\longrightarrow} \vec{K}', s) = Z'(\vec{K} \overset{L}{\longrightarrow} \vec{K}', s) = \varnothing$, or there exist some $O$ and $S$ such that $Z'(\vec{K} \overset{L}{\longrightarrow} \vec{K}', s) = \langle \langle O, S \rangle \rangle$ and $\langle O, S \rangle \in Z(\vec{K} \overset{L}{\longrightarrow} \vec{K}', s)$. It is easy to see from the definition of a transformation-conformance-dense relation that $R_{TR}$ is also a transformation-conformance-dense relation for $T$ with witness $Z'$.

Define the relation $R_S$ as the set of pairs $\langle \vec{K}, s \rangle$ such that either

- \( \vec{K} = \vec{K}_{\text{init}} \) and $s = s_{\text{init}}$, or

- there exist executions $\vec{K}_1 \overset{I_{1,1}, \ldots, I_{1,n}}{\longrightarrow} \ldots$ and $S_1 \overset{\Lambda_1}{\longrightarrow} \ldots$ such that
  - \( \vec{K}_1 = \vec{K}_{\text{init}} \),
  - \( \vec{K}_1 \overset{I_{1,1}, \ldots, I_{1,n}}{\longrightarrow} \ldots \) is fair,
  - $S_1 \overset{\Lambda_1}{\longrightarrow} \ldots \in \text{SimExecs}(\vec{K}_1 \overset{I_{1,1}, \ldots, I_{1,n}}{\longrightarrow} \ldots, s_{\text{init}}, T, Z')$, and
  - there exist $\vec{K}_i$ and $S_i$ in the two executions such that $\vec{K} = \vec{K}_i$ and $s \in S_i$.  

It remains to show that \( R_S \) is summary-conformance-dense, which would imply that \( \bar{K}_{\text{init}} \models \bar{s}_{\text{init}} \). Let there be \( \langle \bar{K}_k, s \rangle \in R_S \), and let \( \bar{K}'_k \xrightarrow{\bar{I}_{k,1} \ldots \bar{I}_{k,n}} \ldots \) be a fair, non-stuck event-step execution where \( \bar{K}'_k = \bar{K}_k \). We must show that there exists a fair summary execution \( \bar{K}'_k \xrightarrow{\bar{I}_l} \ldots \) that summarizes it and a fair specification summary execution \( S'_k \xrightarrow{(L_k, O_k)} S_{k+1} \xrightarrow{(L_{k+1}, O_{k+1})} \ldots \) such that \( S'_k = \{ s \} \) and \( \langle \bar{K}'_i, s' \rangle \in R_S \) for all \( \bar{K}'_i \) and \( S_i \) in the respective executions and all \( s' \in S_i \).

First, to use some of the above lemmas, we must show that \( \bar{K}_{\text{init}} \) and \( \bar{s}_{\text{init}} \) are well-formed. By the definition of instantiation for a program configuration (section 3.5.2),

- every internal address in \( \bar{K}_{\text{init}} \) corresponds to an internal actor in \( \bar{K}_{\text{init}} \),
- \( \text{Used}(\bar{K}_{\text{init}}) \) contains all markers in the configuration,
- every external address is marked with one marker, and
- every receptionist is marked with a unique marker.

Therefore, \( \bar{K}_{\text{init}} \) is well-formed. Then from the definition of \( \mid \cdot \mid \) and because \( \bar{K}_{\text{init}} \) is well-formed, it is easy to see that \( \mid \bar{K}_{\text{init}} \mid \) is well-formed (the message-map component \( \bar{\mu} \) in \( \mid \bar{K}_{\text{init}} \mid \) must be fully merged by the definition of \( \mid \cdot \mid \)). By the definition of instantiation for a specification configuration, it is easy to see that \( \bar{s}_{\text{init}} \) output-monitors all of its state arguments and obligation destinations, so \( \bar{s}_{\text{init}} \) is well-formed.

We must also show that \( \mid \bar{K}_{\text{init}} \mid \) is an externals-only configuration and that \( \text{Mon}(\bar{s}_{\text{init}}) \subseteq \text{Used}(\mid \bar{K}_{\text{init}} \mid) \). By the definition of instantiation, \( \bar{K}_{\text{init}} \) is an externals-only configuration and \( \text{Mon}(\bar{s}_{\text{init}}) \subseteq \text{Used}(\bar{K}_{\text{init}}) \). Abstraction does not add markers to any address, so \( \mid \bar{K}_{\text{init}} \mid \) is an externals-only configuration. By the definition of the abstraction function \( \mid \cdot \mid \), \( \text{Used}(\mid \bar{K}_{\text{init}} \mid) = \text{Used}(\bar{K}_{\text{init}}) \), so \( \text{Mon}(\bar{s}_{\text{init}}) \subseteq \text{Used}(\mid \bar{K}_{\text{init}} \mid) \).

Now, if \( \langle \bar{K}_k, s \rangle \neq \langle \mid \bar{K}_{\text{init}} \mid, \bar{s}_{\text{init}} \rangle \), then we know that there exist executions \( \bar{a} \bar{e} \bar{x} = \bar{K}_1 \xrightarrow{\bar{I}_{1,1} \ldots \bar{I}_{1,n}} \ldots \) and \( S_1 \xrightarrow{\Lambda_1} \ldots \) such that

- \( \bar{K}_1 = \mid \bar{K}_{\text{init}} \mid \),
- \( \bar{K}_1 \xrightarrow{\bar{I}_{1,1} \ldots \bar{I}_{1,n}} \ldots \) is fair,
- \( S_1 \xrightarrow{\Lambda_1} \ldots \in \text{SimExecs}(\bar{K}_1 \xrightarrow{\bar{I}_{1,1} \ldots \bar{I}_{1,n}} \ldots, s_{\text{init}}, T, Z') \), and
- there exist \( \bar{K}_i \) and \( S_i \) in the two executions such that \( \bar{K} = \bar{K}_i \) and \( s \in S_i \).

Otherwise, let \( \bar{a} \bar{e} \bar{x} = \bar{K}_1 \xrightarrow{\bar{I}_{1,1} \ldots \bar{I}_{1,n}} \ldots = \bar{K}'_k \xrightarrow{\bar{I}_{1,1} \ldots \bar{I}_{1,n}} \ldots \), which we know is fair. Then we also have that \( \bar{K}_1 = \mid \bar{K}_{\text{init}} \mid \). By the \( \text{SimExecs} \) Non-Emptiness lemma, \( \text{SimExecs}(\bar{K}_1 \xrightarrow{\bar{I}_{1,1} \ldots \bar{I}_{1,n}} \ldots, s, T, Z) \neq \emptyset \), so let \( S_1 \xrightarrow{\Lambda_1} \ldots \) be an arbitrary member.
of SimExecs(\(R_1 \xrightarrow{\hat{t}_{1,1}, \ldots, \hat{t}_{1,n}} \ldots, s, T, Z\)). By the definition of SimExecs, \(S_1 = \{s\}\). Let \(i = 1\) and we have \(\hat{K} = \hat{K}_{i}\) and \(s \in S_i\).

Next, let \(\hat{aex}'\) be \(\hat{K}_1 \xrightarrow{\hat{t}_{1,1}, \ldots, \hat{t}_{1,n}} \ldots \xrightarrow{\hat{t}_{k-1,1}, \ldots, \hat{t}_{k-1,n}} \hat{K}_k \xrightarrow{\hat{t}_{k,1}, \ldots, \hat{t}_{k,n}} \ldots\), i.e., the execution obtained by taking the first \(k - 1\) steps of \(\hat{aex}\) and prepending them to the given fair execution. By the Fair Suffix lemma, \(\hat{aex}'\) is fair, and \(\hat{aex}\) and \(\hat{aex}'\) by definition share a prefix of length \(k - 1\). Therefore by the SimExecs Common Prefix lemma, there exists \(S'_1 \xrightarrow{(L_1, O_1)} \ldots \in \text{SimExecs}(\hat{aex}', s_{init}, T, Z')\) such that \(S_1 \xrightarrow{\hat{L}_1} \ldots\) and \(S'_1 \xrightarrow{(L_1, O_1)} \ldots\) share a prefix of length \(k - 1\) (and therefore \(s \in S_k\)).

We will show that a “suffix” of \(S'_1 \xrightarrow{(L_1, O_1)} \ldots\) satisfies the necessary criteria to show that \(R_S\) is summary-conformance-dense.

By the definition of instantiation and the abstraction function \(|\cdot|\), \(\hat{K}_{init}\) is an externals-only configuration and \(\text{Mon}(s_{init}) \subseteq \text{Used}(\hat{K}_{init})\). By the SimExecs Simulation lemma, \(S'_1 \xrightarrow{(L_1, O_1)} \ldots\) has the same length as \(\hat{aex}'\), and \(\hat{K}_1 \xrightarrow{L_1} \ldots \xrightarrow{L_{k-1}} \hat{K}'_k \xrightarrow{L_k} \ldots\) summarizes \(\hat{aex}'\). Then by corollary K.12.3 to the Distinct-Marker Preservation lemma, for all configurations \(S'_i\) in \(S'_1 \xrightarrow{(L_1, O_1)} \ldots\), all the PSMs in \(S'_i\) monitor distinct markers. Also, by the SimExecs Fulfillment lemma, \(S'_1 \xrightarrow{(L_1, O_1)} \ldots\) is fair. Therefore by the Fair Specification Suffix lemma, there exists a summary specification execution \(S''_k \xrightarrow{(L_k, O'_k)} \ldots\) such that

- \(S''_k = \{s\}\),
- \(\text{len}\left(S''_k \xrightarrow{(L_k, O'_k)} \ldots\right) = \text{len}\left(\hat{K}'_k \xrightarrow{\hat{t}_{k,1}, \ldots, \hat{t}_{k,n}} \ldots\right)\),
- \(S''_i \subseteq S'_i\) for all \(S''_i\) in the execution, and
- \(S''_k \xrightarrow{(L_k, O'_k)} \ldots\) is fair.

We have shown that \(S''_k \xrightarrow{(L_k, O'_k)} \ldots\) is fair, so it remains to show that \(\hat{K}'_k \xrightarrow{L_k} \ldots\) summarizes \(\hat{K}'_k \xrightarrow{\hat{t}_{k,1}, \ldots, \hat{t}_{k,n}} \ldots\), and that all corresponding program-configuration/PSM pairs from the two executions are in \(R_S\). Because the summary labels \(L_k, L_{k+1}, \ldots\) in the summary execution did not change, and \(S''_k \xrightarrow{(L_k, O'_k)} \ldots\) has the same length as \(\hat{K}'_k \xrightarrow{\hat{t}_{k,1}, \ldots, \hat{t}_{k,n}} \ldots\), we know that \(\hat{K}'_k \xrightarrow{L_k} \ldots\) summarizes \(\hat{K}'_k \xrightarrow{\hat{t}_{k,1}, \ldots, \hat{t}_{k,n}} \ldots\).

Finally, let there be corresponding configurations \(\hat{K}'_{k+i}\) and \(S''_{k+i}\) from \(\hat{K}'_k \xrightarrow{\hat{t}_{k,1}, \ldots, \hat{t}_{k,n}} \ldots\) and \(S''_k \xrightarrow{(L_k, O'_k)} \ldots\), respectively, and let \(s'\) be a member of \(S''_{k+i}\). We know that \(S''_{k+i} \subseteq S'_{k+i}\), so we have that
APPENDIX K. PROOFS FOR TRANSFORMATION CONFORMANCE

- $|\bar{K}_{\text{init}}|$ is the first configuration in $\bar{a}e\bar{x}'$,
- $\bar{a}e\bar{x}'$ is fair,
- $S_1' \xrightarrow{(L_1, O_1)} \ldots \in \text{SimExec} (\bar{a}e\bar{x}', s_{\text{init}}, T, Z')$, and
- there exists $i \leq 1 + \text{len} (\bar{K}_1 \xrightarrow{i_1, \ldots, i_{1,n}} \ldots)$ such that $\bar{K} = \bar{K}_i$ and $s \in S_i$.

Therefore, $\langle \bar{K}_{k+1}', s' \rangle \in R_S$, which completes the proof.
Appendix L

Conformance-Reflection Proofs

This appendix proves that the various transformations used in this dissertation (including the optimizations) are conformance-reflecting. It starts by introducing some definitions used in the proofs, then proves the various necessary lemmas and theorems in a bottom-up fashion.

Similar to the proof of the soundness of abstract CSA in appendix I, several of the proofs in this appendix are dedicated to showing that whenever a configuration \( \hat{K}_G \) approximates a configuration \( \hat{K}_L \), \( \hat{K}_C \) can simulate every transition from \( \hat{K}_L \). As a helpful mnemonic, the subscript \( G \) in those proofs stands for the “greater” term (i.e., the approximating one), and the subscript \( L \) stands for the “lesser” term (i.e., the approximated one).

L.1 Definitions

The transformation \textit{Split} is defined as a function that creates a separate PSM for each of the independent markers of the given PSM. In some of the proofs to show that \textit{Split} is conformance-reflecting, it will be helpful to generalize this idea to a partial function \textit{SplitSelected}, defined below, that creates separate PSMs for only a given subset of a PSM’s independent markers. Note that the function is not defined if any member of the given set \{\( \eta'_1, \ldots, \eta'_m \)\} is not an independent marker of the given PSM.
Definition. The function \( \text{AddrLoc} \) to retrieve the location for a given address is defined as follows.

\[
\text{AddrLoc}(\hat{a}) = \begin{cases} 
\ell & \text{if } \hat{a} = (\text{addr } \ell \text{ n}) \\
\ell' & \text{if } \hat{a} = (\text{collective-addr } \ell')
\end{cases}
\]

Definition. The set \( \text{ExtAbsAddr} \) is the set of external abstract addresses; i.e., \( \text{ExtAbsAddr} = \{\hat{a} \mid \hat{a} \text{ is external}\} \).

Definition. An abstract program configuration \( \langle \{\hat{\beta} \mid \hat{\mu} \mid H\} \rangle^\beta \) is a single-handler configuration if and only if for all \( \hat{a} \) and \( \hat{b} \) such that \( \hat{b} \in \hat{\beta}(\hat{a}) \) and \( \hat{b} \) is handling an event, there is no \( \hat{b}' \in \hat{\beta}(\hat{a}) \) such that \( \hat{b}' \neq \hat{b} \) and \( \hat{b}' \) is handling an event, and there is no \( \hat{a}' \neq \hat{a} \) such that \( \text{AddrLoc}(\hat{a}) = \text{AddrLoc}(\hat{a}') \) and some behavior \( \hat{b}' \in \hat{\beta}(\hat{a}') \) is handling an event.

Definition. An address-correspondence function \( A \) is approximating if for all \( \hat{a} \in \text{dom}(A) \), \( \hat{a} \subseteq_A A(\hat{a}) \).

Definition. A transformation \( T \) is well-formed-preserving if and only if for all \( K \) and \( s \), and all \( \langle K', s', A, M \rangle \in T(K, s) \), if \( K \) and \( s \) are well-formed, then so are \( K' \) and \( s' \).

Definition. A transformation \( T \) is externals-only-preserving if and only if for all \( K \) and \( s \), and all \( \langle K', s', A, M \rangle \in T(K, s) \), if \( K \) is an externals-only configuration, then so is \( K' \).

Definition. A transformation \( T \) is single-message-reflecting if and only if for all \( K \) and \( s \), all \( \langle K', s', A, M \rangle \in T(K, s) \), and all messages \( \langle \hat{a}@H, \hat{v}\rangle \) in \( K' \) with a quantity of single, there exists a message \( \langle \hat{a}@H, \hat{v}\rangle \) in \( K \) with a quantity of single such that \( A(\hat{a}) = \hat{a}' \), \( M(H) = H' \), and \( \text{Remap}(\hat{a}, A, M) = \hat{v}' \).

Definition. A transformation \( T \) is an approximating transformation if and only if for all \( K \) and \( s \), if \( K \) and \( s \) are well-formed, no actor in \( K \) is handling an event, and \( T(K, s) \) terminates, then there exist \( K' \), \( s' \), \( A \), and \( M \) such that

- \( T(K, s) = \{\langle K', s', A, M \rangle\} \),
- \( K \subseteq_{A, M} K' \), and
L.2. NO MONITORED MARKERS LEMMA

Lemma (No Monitored Markers). For all $S, S', s, \text{ and } \hat{\lambda}$, if $s \in S$, $\text{Mon}(s) = \emptyset$, and $S \xrightarrow{\hat{\lambda}} S'$, then there exists $s' \in S'$ such that $\text{Mon}(s') = \emptyset$.

Proof. If $\hat{\lambda} = \cdot$, then the transition $S \xrightarrow{\hat{\lambda}} S'$ must be a use of the S-FREE transition rule. By the definition of that rule, either $s \in S'$, or there exists $s'$ such that $s \xrightarrow{\cdot, \emptyset, \emptyset} s'$ and $s' \in S'$. In the former case, let $s' = s$ and we're done. In the latter case, the transition $s \xrightarrow{\cdot, \emptyset, \emptyset} s'$ would be a use of the P-FREE transition rule. By the definition of that rule, $\text{Mon}(s') = \text{Mon}(s)$, so $\text{Mon}(s') = \emptyset$.

Otherwise, $\hat{\lambda} \neq \cdot$, so the transition $S \xrightarrow{\hat{\lambda}} S'$ must be a use of the S-SEND OR RECEIVE rule. By the definition of this rule, there exist $s', S''$, and $O$ such that $s \xrightarrow{\hat{\lambda}, O, S''} s'$ and $s' \in S'$. The transition $s \xrightarrow{\hat{\lambda}, O, S''} s'$ must be a use of either the P-UNMONITORED RECEIVE or P-SEND rule. Then because $\text{Mon}(s) = \emptyset$, $s' = s$. Therefore, $\text{Mon}(s') = \emptyset$. 

\[ \square \]

Corollary L.2.1. For all $S, S', s, \text{ and } s', \text{ if } s \in S, \text{ Mon}(s) = \emptyset, \text{ and } S \xrightarrow{\hat{\lambda}} S', \text{ then there exists } s' \in S' \text{ such that } \text{Mon}(s') = \emptyset.

Proof. By the definition of $\xrightarrow{\cdot}$, there exist $S'_1, \ldots, S''_n$ such that $S'_1 = S$, $S''_n = S'$, and $S'_1 \xrightarrow{\cdot} \cdots \xrightarrow{\cdot} S''_n$. By induction on $i$ and the No Monitored Markers lemma, for all $i \in 1 \ldots n$, there exists $s' \in S'_i$ such that $\text{Mon}(s') = \emptyset$. Therefore, there exists $s' \in S'$ such that $\text{Mon}(s') = \emptyset$. 

\[ \square \]

Corollary L.2.2. For all $S, S', s, \text{ and } s', \text{ if } s \in S, \text{ Mon}(s) = \emptyset, \text{ and } S \xrightarrow{\hat{\lambda}} S', \text{ then there exists } s' \in S' \text{ such that } \text{Mon}(s') = \emptyset.$
Proof. If $\lambda = \bullet$, then by corollary L.2.1, there exists $s' \in S'$ such that $\text{Mon}(s') = \emptyset$. Otherwise, by the definition of the $\rightsquigarrow$ relation, there exist $S''$ and $S'''$ such that $S \rightsquigarrow S'' \overset{\lambda}{\rightarrow} S''' \overset{\lambda}{\rightarrow} S'$. By corollary L.2.1, there exists $s'' \in S''$ such that $\text{Mon}(s'') = \emptyset$. By the No Monitored Markers lemma, there exists $s''' \in S'''$ such that $\text{Mon}(s''') = \emptyset$. Then by corollary L.2.1 again, there exists $s' \in S'$ such that $\text{Mon}(s') = \emptyset$. \hfill \square

L.3 No Silent Transitions Lemma

Lemma (No Silent Transitions). For all $s$, $\lambda$, and $S$, if

- $\text{Mon}(s) = \emptyset$,
- there does not exist $s'$ such that $s \overset{\hat{\lambda} \cdot, \bar{\nu}, \bar{v}}{\rightarrow} s'$, and
- $[s] \overset{\hat{\lambda}}{\rightarrow} S$,

then $S = [s]$.

Proof. If $\lambda = \bullet$, then the transition $[s] \overset{\hat{\lambda}}{\rightarrow} S$ must be a use of the S-FREETRANSITION rule. By the definition of that case, it must be the case that there exists $s'$ such that $s \overset{\hat{\lambda} \cdot, \bar{\nu}, \bar{v}}{\rightarrow} s'$ and $s' \in S'$. We know by the precondition to this lemma that no such $psm'$ exists, however, so we can ignore this case.

Otherwise, $\lambda \neq \bullet$, so the transition $[s] \overset{\hat{\lambda}}{\rightarrow} S$ must be a use of the S-SENDORRECEIVE rule. By the definition of this rule, there exist $s'$, $S'$, and $O$ such that $s \overset{\hat{\lambda}, O, S'}{\rightarrow} s'$ and $S' = \{s'\} \cup S'$. The transition $s \overset{\hat{\lambda}, O, S'}{\rightarrow} s'$ must be a use of either the P-UNMONITOREDRECEIVE or P-SEND rule. Then because $\text{Mon}(s) = \emptyset$, $s' = s$ and $S' = \emptyset$. Therefore, $S = [s]$.

Corollary L.3.1. For all $s$ and $S$, if

- $\text{Mon}(s) = \emptyset$,
- there does not exist $s'$ such that $s \overset{\hat{\lambda} \cdot, \bar{\nu}, \bar{v}}{\rightarrow} s'$, and
- $[s] \overset{\hat{\lambda}}{\rightarrow} S$,

then $S = [s]$.

Proof. By the definition of $\rightsquigarrow$, there exist $S'_1, \ldots, S'_n$ such that

- $S'_1 = [s]$,
- $S'_n = S$, and
- $S'_1 \overset{\cdot}{} \rightsquigarrow \ldots \overset{\cdot}{} \rightsquigarrow S'_n$. 

L.3. NO SILENT TRANSITIONS LEMMA

Assume for a contradiction that \( n > 1 \). The transition \( S'_1 \rightarrow S'_2 \) would have to be a use of the S-FreeTransition rule. By the definition of that case, it must be the case that there exists \( s' \) such that \( s \xrightarrow{\cdot,\cdot,\cdot} s' \) and \( s' \in S' \). We know by the precondition to this lemma that no such \( s' \) exists, however, so this is a contradiction. Therefore, \( n = 1 \) and \( S = S'_n = S'_1 = \{s\} \).

\[
\text{Corollary L.3.2. For all } s, \hat{\lambda}, \text{ and } S, \text{ if}
\]
\[
\begin{align*}
& \cdot \text{ Mon}(s) = \emptyset, \\
& \text{ there does not exist } s' \text{ such that } s \xrightarrow{\cdot,\cdot,\cdot} s', \text{ and} \\
& \{s\} \xrightarrow{\hat{\lambda}} S,
\end{align*}
\]
then \( S = \{s\} \).

\[
\text{Proof. If } \hat{\lambda} = \cdot, \text{ then } S = \{s\} \text{ by corollary L.3.1. Otherwise, there exist } S' \text{ and } S'' \text{ such that } (s) \xrightarrow{\hat{\lambda}} S' \xrightarrow{\cdot} S'' \xrightarrow{\cdot} S. \text{ By corollary L.3.1, } S' = \{s\}. \text{ By the No Silent Transitions lemma, } S'' = \{s\}. \text{ Then by corollary L.3.1 again, } S = \{s\}. \]

\[
\text{Corollary L.3.3. For all } s, \hat{\lambda}_1, \ldots, \hat{\lambda}_n, O_1, \ldots, O_n, \text{ and } S, \text{ if}
\]
\[
\begin{align*}
& \cdot \text{ Mon}(s) = \emptyset, \\
& \text{ there does not exist } s' \text{ such that } s \xrightarrow{\cdot,\cdot,\cdot} s', \text{ and} \\
& \{s\} \xrightarrow{\hat{\lambda}_1, O_1, \ldots, \hat{\lambda}_n, O_n} S,
\end{align*}
\]
then \( S = \{s\} \).

\[
\text{Proof. By the definition of the specification event-step relation, there exist } S'_1, \ldots, S'_{n+1} \text{ such that}
\]
\[
\begin{align*}
& S'_1 = \{s\}, \\
& S'_{n+1} = S, \text{ and} \\
& S'_1 \xrightarrow{\hat{\lambda}_1, O_1} \ldots \xrightarrow{\hat{\lambda}_n, O_n} S'_{n+1}.
\end{align*}
\]
We will show that for all \( i \in 1 \ldots n, S'_i = \{s\} \). The base case where \( i = 1 \) is trivial. In the inductive case, we know that \( S'_{i-1} = \{s\} \). Then by corollary L.3.2, \( S'_i = \{s\} \). Therefore, because \( S'_{n+1} = S \), we have \( S = \{s\} \).

\[
\text{Corollary L.3.4. For all } s, L, O, S, \hat{K}, \text{ and } \hat{K}', \text{ if}
\]
\[
\begin{align*}
& \cdot \text{ Mon}(s) = \emptyset, \\
& \text{ there does not exist } s' \text{ such that } s \xrightarrow{\cdot,\cdot,\cdot} s',
\end{align*}
\]
Distinct Monitored Markers Lemma

**Lemma (Distinct Monitored Markers).** For all $S_1, S_2, S'_1, S'_2$, and $\hat{\lambda}$, if

- $S_1$ and $S_2$ are well-formed,
- $\text{Mon}(S_1) \cap \text{Mon}(S_2) = \emptyset$,
- $S_i \xrightarrow{\hat{\lambda}} S'_i$ for all $i \in \{1, 2\}$,
- either $\hat{\lambda} = \bullet$ or there exist $\hat{a}, H$, and $\hat{v}$ such that $|H| \leq 1$ and either $\hat{\lambda} = \hat{a} @ H? \hat{v}$ or $\hat{\lambda} = \hat{a} @ H! \hat{v}$, and
- $\text{Matchable}(\hat{\lambda}) \cap \text{Mon}(S_i) = \emptyset$ for all $i \in \{1, 2\}$,

then $\text{Mon}(S'_1) \cap \text{Mon}(S'_2) = \emptyset$.

**Proof.** If $\hat{\lambda} = \bullet$, then $\text{Matchable}(\hat{\lambda}) = \emptyset$. Then by the Monitored Matchable Markers lemma (appendix K), $\text{Mon}(S'_i) \subseteq \text{Mon}(S_i)$ for all $i \in \{1, 2\}$. Therefore $\text{Mon}(S'_1) \cap \text{Mon}(S'_2) = \emptyset$.

Otherwise, there exist $\hat{a}, H$, and $\hat{v}$ such that $|H| \leq 1$ and either $\hat{\lambda} = \hat{a} @ H? \hat{v}$ or $\hat{\lambda} = \hat{a} @ H! \hat{v}$. Then because $\text{Mon}(S_1) \cap \text{Mon}(S_2) = \emptyset$, there is at least one $i \in \{1, 2\}$ such that $\text{Mon}(S_i) \cap H = \emptyset$. Suppose $i = 1$; the case for $i = 2$ is similar.

Because $\hat{\lambda} \neq \bullet$, the transition $S_1 \xrightarrow{\hat{\lambda}} S'_1$ must be a use of the $\text{SENDORRECEIVE}$ rule. By the definition of this rule, there exist $s_1, \ldots, s_n$, $O_1, \ldots, O_n$, and $S'_1, \ldots, S''_n$ such that

- $S_1 = \{s_1, \ldots, s_n\}$,
- $s_j \xrightarrow{\hat{\lambda}_j, O_j, S''_j} s'_j$ for all $j \in 1 \ldots n$, and
- $S'_1 = \{s'_1, \ldots, s'_n\} \cup S''_1 \cup \ldots \cup S''_n$.

For all $j \in 1 \ldots n$, the transition $s_j \xrightarrow{\hat{\lambda}_j, O_j, S''_j} s'_j$ must be a use of either the $\text{P-UNMONITOREDRECEIVE}$ or $\text{P-SEND}$ rule. Then because $\text{Mon}(S_1) \cap H = \emptyset$, $S'_j = \emptyset$ and $s'_j = s_j$ for all $i \in 1 \ldots n$. Therefore, $S'_1 = S_1$. 

\[\square\]
By the Monitored Matchable Markers lemma, $\text{Mon}(S'_2) \subseteq \text{Mon}(S_2) \cup \text{Matchable}(\lambda)$. We already know that $\text{Mon}(S_1) \cap \text{Mon}(S_2) = \emptyset$ and $\text{Matchable}(\lambda) \cap \text{Mon}(S_1) = \emptyset$. Then because $S'_1 = S_1$, we have $\text{Mon}(S'_1) \cap \text{Mon}(S'_2) = \emptyset$.

\[\square\]

**Corollary L.4.1.** For all $S_1, S_2, S'_1, S'_2$, and $\lambda$, if

- $S_1$ and $S_2$ are well-formed,
- $\text{Mon}(S_1) \cap \text{Mon}(S_2) = \emptyset$,
- $S_i \stackrel{\lambda}{\longrightarrow} S'_i$ for all $i \in \{1, 2\}$,
- either $\lambda = \bullet$ or there exist $\bar{a}, H, \text{ and } \bar{v}$ such that $|H| \leq 1$ and either $\lambda = \bar{a}@H?\bar{v}$ or $\lambda = \bar{a}@H!\bar{v}$, and
- $\text{Matchable}(\lambda) \cap \text{Mon}(S_i) = \emptyset$ for all $i \in \{1, 2\}$,

then $\text{Mon}(S'_1) \cap \text{Mon}(S'_2) = \emptyset$.

**Proof:** If $\lambda = \bullet$, then $\text{Matchable}(\lambda) = \emptyset$. Then by corollary K.8.1 to the Monitored Matchable Markers lemma, $\text{Mon}(S'_i) \subseteq \text{Mon}(S_i)$ for all $i \in \{1, 2\}$. Therefore, $\text{Mon}(S'_1) \cap \text{Mon}(S'_2) = \emptyset$.

Otherwise, by the definition of the $\longrightarrow$ relation, there exist $S''_1, S''_1', S''_2$, and $S''_2'$ such that $S_i \stackrel{\lambda}{\longrightarrow} S''_i \stackrel{\lambda}{\longrightarrow} S'''_i \stackrel{\lambda}{\longrightarrow} S'_i$ for all $i \in \{1, 2\}$. By corollaries K.5.1 and K.5.2 to the Specification Well-Formed Preservation lemma, $S''_1, S''_1', S''_2$, and $S''_2'$ are well-formed.

By corollary K.8.1 to the Monitored Matchable Markers lemma, $\text{Mon}(S'_i) \subseteq \text{Mon}(S_i)$ for all $i \in \{1, 2\}$. Therefore, $\text{Mon}(S''_1) \cap \text{Mon}(S''_2) = \emptyset$ and $\text{Matchable}(\lambda) \cap \text{Mon}(S''_i) = \emptyset$ for all $i \in \{1, 2\}$. By the Distinct Monitored Markers lemma, $\text{Mon}(S''_1) \cap \text{Mon}(S''_2) = \emptyset$. By corollary K.8.1 again, $\text{Mon}(S'_i) \subseteq \text{Mon}(S''_i)$ for all $i \in \{1, 2\}$. Therefore, $\text{Mon}(S'_1) \cap \text{Mon}(S'_2) = \emptyset$.

\[\square\]

**Corollary L.4.2.** For all $S_1, S_2, S'_1, S'_2, L, O_1, O_2, \hat{K}$, and $\hat{K}'$, if

- $S_1$ and $S_2$ are well-formed,
- $\text{Mon}(S_1) \cap \text{Mon}(S_2) = \emptyset$,
- $S_i \stackrel{(L, O_i)}{\longrightarrow} S'_i$ for all $i \in \{1, 2\}$,
- $\hat{K}$ is an externals-only configuration,
- $\text{Mon}(S_i) \subseteq \text{Used}(\hat{K})$ for all $i \in \{1, 2\}$, and
- $\hat{K} \stackrel{L}{\longrightarrow} \hat{K}'$,

then $\text{Mon}(S'_1) \cap \text{Mon}(S'_2) = \emptyset$.
Proof. Let \( \hat{K} \xrightarrow{\hat{I}_1, \ldots, \hat{I}_n} \hat{K}' \) be an event step summarized by \( K \xrightarrow{L} \hat{K}' \). By corollary K.7.3 to the Externals-Only Label lemma, for all \( j \in 1 \ldots n \), Matchable(\( \hat{I}_j \)) \cap Used(\( \hat{K} \)) = \emptyset and if there exist \( \hat{a}, H, \) and \( \hat{v} \) such that \( [\hat{I}_j] = \hat{a}H\hat{v} \) or \( [\hat{I}_j] = \hat{a}H\hat{v} \), then \( |H| = \emptyset \).

By the definition of the specification summary transition relation, there exist \( S''_{1,1}, \ldots, S''_{1,n+1} \) and \( S''_{2,1}, \ldots, S''_{2,n+1} \) such that for all \( i \in \{1,2\} \),

- \( S''_{i,1} = S_i \),
- \( S''_{i,n+1} = S'_i \), and
- \( S''_{i,n} \xrightarrow{[\hat{I}_1], \ldots, [\hat{I}_n]} S''_{i,n+1} \).

We know that \( S''_{1,1} \) and \( S''_{2,1} \) are well-formed. Then by induction on \( j \) and corollary K.5.2 to the Specification Well-Formed Preservation lemma, \( S''_{1,j} \) is well-formed for all \( i \in \{1,2\} \) and all \( j \in 1 \ldots n + 1 \).

By the Distinct Matchables lemma (appendix K), Matchable(\( \hat{I}_j \)) \cap Matchable(\( \hat{I}_k \)) = \emptyset for all \( j \) and \( k \) in \( 1 \ldots n \) such that \( j \neq k \). Then by induction on \( j \) and the Monitored Matchable Markers lemma, Mon(\( S''_{1,j} \)) \subseteq Used(\( \hat{K} \)) \cup Matchable(\( \hat{I}_j \)) \cup \ldots \cup Matchable(\( \hat{I}_{j-1} \)) for all \( i \in \{1,2\} \) and all \( j \in 1 \ldots n + 1 \). Therefore, for all \( i \in \{1,2\} \) and all \( j \in 1 \ldots n + 1 \), Mon(\( S''_{1,j} \)) \cap Matchable(\( \hat{I}_j \)) = \emptyset.

Finally, by induction and corollary L.4.1 to the Distinct Monitored Markers lemma, Mon(\( S''_{1,j} \)) \cap Mon(\( S''_{2,j} \)) = \emptyset for all \( j \in 1 \ldots n + 1 \). Therefore, Mon(\( S''_{1,j} \)) \cap Mon(\( S''_{2,j} \)) = \emptyset.

\[ \square \]

### L.5 Summary Synchronization Lemma

This lemma says that when two specification configurations take different summary transitions to simulate the same event step, there is some common summary transition they can both take to simulate that event step and fulfill the same obligations.

**Lemma (Summary Synchronization).** For all \( S_1, S'_1, S_2, S'_2, L_1, L_2, O_1, O_2, \hat{K}, \hat{K}', \) and \( \hat{I}_1, \ldots, \hat{I}_n, \) if

- \( S_i \xrightarrow{\langle L_i, O_i \rangle} S'_i \) for \( i \in \{1,2\} \),
- \( \hat{K} \xrightarrow{\hat{I}_1, \ldots, \hat{I}_n} \hat{K}' \), and
- \( L_1 \) and \( L_2 \) both summarize \( \hat{I}_1, \ldots, \hat{I}_n, \)

then there exists \( L' \) such that

- \( S_i \xrightarrow{\langle L'_i, O'_i \rangle} S'_i \) for \( i \in \{1,2\} \) and
- \( L' \) summarizes \( \hat{I}_1, \ldots, \hat{I}_n \).
Proof. Let \( \tilde{a}_1 : \text{send-ext} (\tilde{a}_1' \otimes H_1, v_1), \ldots, \tilde{a}_m : \text{send-ext} (\tilde{a}_m' \otimes H_m, v_m) \) be the send-ext labels in \( \tilde{I}_1, \ldots, \tilde{I}_n \), then let \( \tilde{\mu}'' = \emptyset \cup \langle \tilde{a}_1' \otimes H_1, v_1 \rangle \cup \langle \tilde{a}_m' \otimes H_m, v_m \rangle \). By the Message-Map Reflexivity lemma, \( \tilde{\mu}'' \subseteq_{\text{id, id}} \tilde{\mu}' \). Because \( \tilde{L}_1 \) summarizes \( \tilde{I}_1, \ldots, \tilde{I}_n \), we know there is no rcv-ext label in \( \tilde{I}_{2}, \ldots, \tilde{I}_n \). Let \( \tilde{L}' = \langle \tilde{I}_1, \tilde{\mu}' \rangle \); then \( \tilde{L}' \) summarizes \( \tilde{I}_1, \ldots, \tilde{I}_n \).

It remains to show that \( S_i \xrightarrow{\langle L', O_i \rangle} S'_i \) for \( i \in \{1, 2\} \). Let there be \( i \in \{1, 2\} \).

Because \( S_i \xrightarrow{\langle L_i, O_i \rangle} S'_i \), we know that

- for all \( \langle \eta, po \rangle \in \text{Obls}(S_i), \langle \eta, po \rangle \in \text{Obls}(S'_i) \cup O_i \), and
- either \( \text{Mon}(s) \neq \emptyset \) for all \( s \in S'_i \), or there exists \( s \) such that \( S_i = \{s\}, \text{Mon}(s) = \emptyset \), and there is no \( s' \) such that \( s \rightarrow^*, \emptyset, \emptyset \rightarrow^* s' \).

Let \( \tilde{I}_1', \ldots, \tilde{I}_p' \) be a sequence summarized by \( \tilde{L}' \). Let \( \tilde{a}_1' : \text{send-ext}(\tilde{a}_1'' \otimes H'_1, v'_1), \ldots, \tilde{a}_m' : \text{send-ext}(\tilde{a}_m'' \otimes H'_m, v'_m) \) be the send-ext labels in \( \tilde{I}_1', \ldots, \tilde{I}_p' \), then let \( \tilde{\mu}''' = \emptyset \cup \langle \tilde{a}_1'' \otimes H'_1, v'_1 \rangle \cup \langle \tilde{a}_m'' \otimes H'_m, v'_m \rangle \). Because \( \tilde{L}' \) summarizes \( \tilde{I}_1', \ldots, \tilde{I}_p' \), we know that \( \tilde{I}' = \tilde{I}_1 \) is no rcv-ext label in that sequence, and \( \tilde{\mu}'' \subseteq_{\text{id, id}} \tilde{\mu}' \).

Let \( \langle \tilde{I}_i', \tilde{\mu}_i \rangle = L_i \). Because \( L_i \) summarizes \( \tilde{I}_1, \ldots, \tilde{I}_n \), \( \tilde{I}_i' = \tilde{I}_n \). Therefore, \( \tilde{I}_i'' = \tilde{I}_1 \). Also because \( L_i \) summarizes \( \tilde{I}_1, \ldots, \tilde{I}_n, \tilde{\mu}_i' \subseteq_{\text{id, id}} \tilde{\mu}_i \). Then by the Message-Map Transitivity lemma, \( \tilde{\mu}_i'' \subseteq_{\text{id, id}} \tilde{\mu}_i \). Therefore, \( L_i \) summarizes \( \tilde{I}_1', \ldots, \tilde{I}_p' \). Then because \( S_i \xrightarrow{\langle L_i, O_i \rangle} S'_i \), there exist \( O_1', \ldots, O_p' \) such that \( S_i \xrightarrow{\langle \tilde{I}_1', O_1' \rangle, \ldots, \langle \tilde{I}_p', O_p' \rangle} S'_i \) and \( O_i \subseteq \text{Obls}(S_i) \cup \cup O_p' \).

We have shown that all of the conditions for the specification summary transition relation are satisfied, so \( S_i \xrightarrow{\langle L', O_i \rangle} S'_i \).

\[ \text{Corollary L.5.1.} \text{ For all summary specification executions } S_1 \xrightarrow{\langle L_1, O_1 \rangle} \ldots \text{ and } S'_1 \xrightarrow{\langle L_1', O_1' \rangle} \ldots \text{ of the same length, and all event-step executions } \tilde{K}_1 \xrightarrow{\tilde{I}_1, \ldots, \tilde{I}_n} \ldots \text{ such that} \]

- Simulates \( S_1 \xrightarrow{\langle L_1, O_1 \rangle} \ldots, \tilde{K}_1 \xrightarrow{\tilde{I}_1, \ldots, \tilde{I}_n} \ldots \) and
- Simulates \( S'_1 \xrightarrow{\langle L_1', O_1' \rangle} \ldots, \tilde{K}_1 \xrightarrow{\tilde{I}_1, \ldots, \tilde{I}_n} \ldots \),

there exists a sequence \( L'' \) with the same length as \( L_1, \ldots \) such that

- Simulates \( S_1 \xrightarrow{\langle L_1', O_1' \rangle} \ldots, \tilde{K}_1 \xrightarrow{\tilde{I}_1, \ldots, \tilde{I}_n} \ldots \) and
- Simulates \( S'_1 \xrightarrow{\langle L_1', O_1' \rangle} \ldots, \tilde{K}_1 \xrightarrow{\tilde{I}_1, \ldots, \tilde{I}_n} \ldots \).
Proof. For all $i$ such that $L_i$ is a label in the first execution, by the Summary Synchronization lemma, there exists $L''_i$ such that $S_i \xrightarrow{L''_i, O_i} S'_i$, and $L''_i$ summarizes $\hat{L}_{i,1}, \ldots, \hat{L}_{i,n}$. Therefore, the sequence $L''_1, \ldots$ exists, and we have

- Simulates $S_1 \xrightarrow{L''_1, O_1} \cdots, \hat{L}_{1,1}, \ldots, \hat{L}_{1,n}$ and
- Simulates $S'_1 \xrightarrow{L''_1, O'_1} \cdots, \hat{L}_{1,1}, \ldots, \hat{L}_{1,n}$.

L.6 Split Silent Transitions Lemma

Lemma (Split Silent Transitions). For all $s, s_1, \ldots, s_n, S, S_1, \ldots, S_n, S'_1, \ldots, S'_n, \text{ and } H$, if

- $\text{SplitSelected}(s, H) = \{s_1, \ldots, s_n\}$,
- $S = \{s\} \cup (S_1 - \{s_1\}) \cup \ldots \cup (S_n - \{s_n\})$,
- $S_1 \xrightarrow{\phi} S'_1$, and
- for all $i \in 1 \ldots n$,
  - $s_i \in S_i$,
  - either $\text{Mon}(s'') \neq \emptyset$ for all $s'' \in S_i$ or $S_i = \{s_i\}$,
  - $S_i$ is well-formed,
  - all of the PSMs in $S_i$ monitor distinct markers, and
  - $\text{Mon}(S_i) \cap \text{Mon}(S_j) = \emptyset$ for all $j \in 1 \ldots n$ such that $i \neq j$,

then there exist $S', s'$, and $s'_1$ such that

- $S \xrightarrow{\phi} S'$,
- $\text{SplitSelected}(s', H) = \{s'_1, s_2, \ldots, s_n\}$,
- $S' = \{s'\} \cup (S'_1 - \{s'_1\}) \cup (S_2 - \{s_2\}) \cup \ldots \cup (S_n - \{s_n\})$,
- $s'_1 \in S'_1$, and
- $\text{Mon}(s'_1) = \text{Mon}(s'_1)$.

Proof. The transition $S_1 \xrightarrow{\phi} S'_1$ must be a use of the S-FREETRANSITION rule. By the definition of that rule, there exist $s''_1, s''_1, S''_1$, and $S''_1$ such that $S_1 = \{s''_1\} \cup S''_1, S'_1 \xrightarrow{\phi, S''_1} S''_1$, and $S'_1 = S''_1 \cup S''_1 \cup S''_1$. There are two cases, depending on whether $s''_1 = s_1$. 

Case: $s''_1 = s_1$

In this case, let $s'_1 = s''_1$ so that we have $s_1 \xrightarrow{\varphi, S''_1} s'_1$ and $s'_1 \in S'_1$. By the definition of $SplitSelected$, for all but at most one $i \in 1 \ldots n$, there exist $\eta$ and $O_s$ such that $s_i = \langle \varphi, \{\eta_i\}, \text{Dummy} : \eta', (\text{define-state} (\text{Dummy})), O_s \rangle$. Because there are no \textit{free} transitions from that \texttt{Dummy} state, there is no transition labeled with $\bullet$ from any such $s_i$. Therefore, $s_1$ does not have that form, so for all $i \in 2 \ldots n$, there exist $\eta_i$ and $O_i$ such that $s_i = \langle \varphi, \{\eta_i\}, \text{Dummy} : \eta_i, (\text{define-state} (\text{Dummy})), O_i \rangle$.

Let $\langle H', H'', \varphi : \eta'_1 \ldots \eta'_m, \overline{\Phi}, O_1 \rangle = s_1$. The transition $s_1 \xrightarrow{\varphi, S''_1} s'_1$ must be a use of the P-\texttt{FREETRANSITION} rule, so there exist $\varphi'$, $\eta''_1 \ldots \eta''_p$, and $O'_1$ such that $s'_1 = \langle H', H'', \varphi' : \eta''_1 \ldots \eta''_p, \overline{\Phi}, O_1 \cup O'_1 \rangle$. Therefore, $\text{Mon}(s_1) = \text{Mon}(s'_1)$. By the definition of $SplitSelected$, it must be the case that $s = \langle H', h \cup \overline{H}'', \varphi' : \eta''_1 \ldots \eta''_m, \overline{\Phi}, O_1 \cup \ldots \cup O_n \rangle$. The extra markers and obligations do not affect the P-\texttt{FREETRANSITION} rule, so there exists $s'$ such that $s_1 \xrightarrow{\varphi, S''_1} s'$ and $s' = \langle H', h \cup \overline{H}'', \varphi' : \eta''_1 \ldots \eta''_p, \overline{\Phi}, O_1 \cup O_2 \cup \ldots \cup O_n \rangle$. Then by the definition of $SplitSelected$, $\text{SplitSelected}(s') = \{s'_1, \ldots, s'_n\}$.

Let $S' = \{s' \cup (S - \{s\}) \cup S''\}$. By the S-\texttt{FREETRANSITION} rule, we have $S \xrightarrow{\varphi, S} S'$.

It remains to show that $S' = \{s' \cup (S'_1 - \{s'_1\}) \cup (S_2 - \{s_2\}) \cup \ldots \cup (S_n - \{s_n\})\}$. To do so, first we will show that $s \not\in S_i - \{s_i\}$ for all $i \in 1 \ldots n$, and therefore $S - \{s\} = (S_1 - \{s_1\}) \cup \ldots \cup (S_n - \{s_n\})$. If $\text{Mon}(s_i) = \emptyset$, then by the precondition to this lemma, $S_i - \{s_i\} = \emptyset$, so $s \not\in S_i - \{s_i\}$. Otherwise, there exists some $\eta \in \text{Mon}(s_i)$. By the definition of $SplitSelected$, $s \not\in \text{Mon}(s)$. Then because all of the PSMs in $S_i$ monitor distinct markers, $s \not\in \text{Mon}(S_i - \{s_i\})$, so $s \not\in S_i - \{s_i\}$.

Next, we will show that $S'_1 - \{s'_1\} = S''_1 \cup S''_1$. We already know that $S'_1 = S''_1 \cup \{s'_1\} \cup S''_1$, so $S'_1 - \{s'_1\} = (S''_1 \cup S''_1) - \{s'_1\}$. Thus, it is sufficient to show that $s'_1 \not\in S''_1 \cup S''_1$. If $\text{Mon}(s'_1) = \emptyset$, then by the precondition to this lemma, $S_1 = \{s_1\}$ and therefore $S'_1 = \emptyset$, so $s'_1 \not\in S''_1$. Otherwise, there exists some $\eta \in \text{Mon}(s'_1)$. We know that $\text{Mon}(s'_1) = \text{Mon}(s_1)$, so $\eta \in \text{Mon}(s_1)$, and because all of the PSMs in $S_1$ monitor distinct markers, $\eta \not\in \text{Mon}(S''_1)$. Therefore, $s'_1 \not\in S''_1$. Finally, let $s''_1$ be a member of $S''_1$. By the P-\texttt{FREETRANSITION} rule, $s''_1$ must be a result of a \texttt{fork} effect or a \texttt{fork-addr} pattern. Therefore, all of the state definitions for $s''_1$ must be syntactically nested inside the state definitions for $s_1$. The state definitions for $s'_1$ are the same as those of $s_1$, by the P-\texttt{FREETRANSITION} rule, so $s''_1 \neq s'_1$. Therefore, $s'_1 \not\in S''_1$.

By definition, we have $S' = \{s' \cup (S - \{s\}) \cup S''\}$. By the above argument about $S - \{s\}$, $S' = \{s' \cup S''_1 \cup (S_1 - \{s_1\}) \cup \ldots \cup (S_n - \{s_n\})\}$. We also know that $S_1 - \{s_1\} = S''_1$, so $S' = \{s' \cup S''_1 \cup (S_2 - \{s_2\}) \cup \ldots \cup (S_n - \{s_n\})\}$. Finally, we have shown that $S'_1 - \{s'_1\} = S''_1 \cup S''_1$, so $S' = \{s' \cup (S'_1 - \{s'_1\}) \cup (S_2 - \{s_2\}) \cup \ldots \cup (S_n - \{s_n\})\}$.
Case: $s''_1 \neq s_1$

In this case, $s''_1 \in (S'_1 - \{s_1\})$, and so $s'_1 \in S$. Let $S' = (S - \{s''_1\}) \cup \{s''_1\} \cup S''_1$, then by the S-FREETRANSITION rule we have that $S \rightarrow^\epsilon S'$. Furthermore, let $s' = s$ and $s'_1 = s_1$; then we also have that $\text{SplitSelected}(s', H) = \{s'_1, s_2, \ldots, s_n\}$, $s'_1 \in S'_1$, and $\text{Mon}(s_1) = \text{Mon}(s'_1)$.

It remains to show that $S' = \{s'_1\} \cup (S'_1 - \{s'_1\}) \cup (S_2 - \{s_2\}) \cup \ldots \cup (S_n - \{s_n\})$. First, we will show that $S - \{s''_1\} = \{s\} \cup (S_1 - \{s_1, s''_1\}) \cup (S_2 - \{s_2\}) \cup \ldots \cup (S_n - \{s_n\})$. We know that $S = \{s\} \cup (S_1 - \{s_1\}) \cup (S_2 - \{s_2\}) \cup \ldots \cup (S_n - \{s_n\})$, so it suffices to show that $s'' \notin \{s\} \cup (S_2 - \{s_2\}) \cup \ldots \cup (S_n - \{s_n\})$. Because $s''_1 \in S_1$ and $s''_1 \neq s_1$, we know that $S_1 \neq \{s_1\}$. Then by the precon of this lemma, every PSM in $S_1$ monitors at least one marker, so there exists some $\eta \in \text{Mon}(s''_1)$. We also know that all PSMs in $S_1$ monitor distinct markers, so $\eta \notin \text{Mon}(s_1)$. By another precon of this lemma, $\text{Mon}(s''_1) \cap \text{Mon}(S_1) = \emptyset$ for all $i \in 2 \ldots n$, so $\eta \notin \text{Mon}(S_1)$ for all $i \in 2 \ldots n$. Therefore, $s''_1 \notin S_i$ for all $i \in 2 \ldots n$. That also implies that $\eta \notin \text{Mon}(s''_1)$ for all $i \in 2 \ldots n$ by the definition of SplitSelected, $\eta \notin \text{Mon}(s)$, so $s''_1 \neq s$.

We now have that $S' = \{s\} \cup (S_1 - \{s_1\}) \cup (S_2 - \{s_2\}) \cup \ldots \cup (S_n - \{s_n\})$. We also know that $s = s$ and $S_1 - \{s_1\} = S''_1$, so $S' = \{s\} \cup (s''_1) \cup (S_2 - \{s_2\}) \cup \ldots \cup (S_n - \{s_n\})$. Furthermore, we have $S'_1 = \{s''_1\} \cup S''_1$, so it remains to show that $s_1 \notin \{s''_1\} \cup S''_1$.

We have already shown that there exists some $\eta$ such that $\eta \in \text{Mon}(s''_1)$ and $\eta \notin \text{Mon}(s_1)$. By corollary K.21.1 to the Monitored Marker Permanence lemma, $\text{Mon}(s''_1) \subseteq \text{Mon}(s''_1')$, $\eta \in \text{Mon}(s''_1)$, and therefore $s_1 \neq s''_1$. By the S-FREETRANSITION rule, every PSM $s''' \in S''_1$ must be a result of a fork effect or fork-addr pattern in the transition $s''_1 \rightarrow^* \emptyset, S''' \rightarrow^* s''_1$. By the definition of that rule, every member of $\text{Mon}(s''')$ must be a state argument on $s''_1$. Because $S_1$ is well-formed, so is $s''_1$, and therefore all of its state arguments are members of $\text{Mon}(s''_1)$, so $\text{Mon}(s''') \subseteq \text{Mon}(s''_1)$. By the definition of the Dist function used in that rule, $\text{Mon}(s''') \cap \text{Mon}(s''_1) = \emptyset$, so $\text{Mon}(s''') = \emptyset$. Because every PSM in $S_1$ monitors at least one marker, $\text{Mon}(s_1) \neq \emptyset$, so $s_1 \neq s'''$, and therefore $s_1 \notin S''_1$, which completes the proof.

\[\square\]

Corollary L.6.1. For all $s$, $s_1, \ldots, s_n$, $S$, $S_1, \ldots, S_n$, $S'_1$, and $H$, if

- $\text{SplitSelected}(s, H) = \{s_1, \ldots, s_n\}$,
- $S = \{s\} \cup (S_1 - \{s_1\}) \cup \ldots \cup (S_n - \{s_n\})$,
- $S_1 \rightarrow^* S'_1$,
- either $\text{Mon}(s'') \neq \emptyset$ for all $s'' \in S_1$, or $S_1 = \{s_1\}$, $\text{Mon}(s_1) = \emptyset$, and $\exists s''$ such that $s_1 \rightarrow^* s''$,
- for all $i \in 2 \ldots n$, either $\text{Mon}(s'') \neq \emptyset$ for all $s'' \in S_i$ or $S_i = \{s_i\}$, and
- for all $i \in 1 \ldots n$,
L.6. SPLIT SILENT TRANSITIONS LEMMA

- \( s_1 \in S_i \),
- \( S_i \) is well-formed,
- all PSMs in \( S_i \) monitor distinct markers, and
- \( \text{Mon}(S_i) \cap \text{Mon}(S) = \emptyset \) for all \( j \in 1 \ldots n \) such that \( i \neq j \),

then there exist \( S', s', \) and \( S'_i \) such that

- \( S \overset{\varnothing}{\rightarrow} S' \),
- \( \text{SplitSelected}(s', H) = \{s'_1, s_2, \ldots, s_n\} \),
- \( S' = \{s'_1\} \cup (S'_1 - \{s'_1\}) \cup (S_2 - \{s_2\}) \cup \ldots \cup (S_n - \{s_n\}) \),
- \( s'_1 \in S'_1 \), and
- \( \text{Mon}(s_1) = \text{Mon}(s'_1) \).

Proof. By the definition of \( \rightarrow \), there exist \( S''_1, \ldots, S''_m \) such that

- \( S''_1 = S_1 \),
- \( S''_m = S'_1 \), and
- \( S''_1 \overset{\varnothing}{\rightarrow} \ldots \overset{\varnothing}{\rightarrow} S''_m \).

First, we will show that for all \( i \in 1 \ldots m \),

- if \( S_1 = \{s_1\} \), \( \text{Mon}(s_1) = \emptyset \), and \( \nexists s'' \) such that \( s_1 \overset{\varnothing, \varnothing}{\rightarrow} s'' \), then \( S''_i = \{s_1\} \),
- \( S''_i \) is well-formed,
- all of the PSMs in \( S''_i \) monitor distinct markers, and
- \( \text{Mon}(S''_i) = \text{Mon}(S) \).

The proof is by induction on \( i \). In the base case where \( i = 1 \), the conditions hold by the preconditions on this lemma. In the inductive case where \( i > 1 \), if \( S_1 = \{s_1\} \), \( \text{Mon}(s_1) = \emptyset \), and \( \nexists s'' \) such that \( s_1 \overset{\varnothing, \varnothing}{\rightarrow} s'' \), then by the induction hypothesis, \( S''_{i-1} = \{s_1\} \). Then by the No Silent Transitions lemma, \( S''_i = \{s_1\} \). The other properties also hold on \( S''_{i-1} \) by the induction hypothesis. We also know that \( \text{Matchable}(\bullet) = \emptyset \). Then by corollary K.5.1 to the Specification Well-Formed Preservation lemma, the Distinct-Marker Preservation lemma, the Monitored Matchable Markers lemma, and corollary K.21.1 to the Monitored Marker Permanence lemma, the remaining properties hold for \( S''_i \).

Next, we will show that for all \( i \in 1 \ldots m \), if \( \text{Mon}(s'') \neq \emptyset \) for all \( s'' \in S'_1 \), then \( \text{Mon}(s') \neq \emptyset \) for all \( s' \in S''_i \). The proof is by induction on \( m - i \). In the base case, \( i = m \), so \( S''_i = S'_1 \), and therefore the condition holds. In the inductive case where \( i < m \), suppose that \( \text{Mon}(s'') \neq \emptyset \) for all \( s'' \in S'_1 \). Then by the induction hypothesis, \( \text{Mon}(s''') \neq \emptyset \) for all \( s''' \in S''_{i+1} \). Then by the No Monitored Markers
lemma, \( \text{Mon}(s'') \neq \emptyset \) for all \( s'' \in S''_1 \). Thus, by the above arguments and the precondition to this lemma, we have that for all \( i \in 1 \ldots m-1 \), either \( \text{Mon}(s'') \neq \emptyset \) for all \( s'' \in S''_{i+1} \), or \( S''_i = \{s_1\} \).

Finally, by induction on \( i \) and the Split Silent Transitions lemma, for all \( i \in 1 \ldots m \) there exist \( S', s', \) and \( s'_1 \) such that

\[ S \xrightarrow{\cdot \cdot \cdot} S', \]
\[ \text{SplitSelected}(s', H) = \{s'_1, s_2, \ldots, s_n\}, \]
\[ S' = \{s'_1\} \cup (S''_1 - \{s'_1\}) \cup (S_2 - \{s_2\}) \cup \ldots \cup (S_n - \{s_n\}), \]
\[ s'_1 \in S'', \]
\[ \text{Mon}(s_1) = \text{Mon}(s'_1). \]

Then we know that \( S'_1 = S''_m \), so the \( S', s', \) and \( s'_1 \) exist to complete the proof. \( \square \)

### L.7 Split Silent Weak-Step Transitions Lemma

**Lemma** (Split Silent Weak-Step Transitions). For all \( s, s_1, \ldots, s_n, S, S_1, \ldots, S_n, S'_1, \ldots, S'_n, \) and \( H \), if

- \( \text{SplitSelected}(s, H) = \{s_1, \ldots, s_n\} \),
- \( S = \{s\} \cup (S_1 - \{s_1\}) \cup \ldots \cup (S_n - \{s_n\}) \), and
- for all \( i \in 1 \ldots n, \)
  - \( s_i \in S_i, \)
  - either \( \text{Mon}(s'') \neq \emptyset \) for all \( s'' \in S'_i \), or \( S_i = \{s_i\} \), \( \text{Mon}(s_i) = \emptyset \), and \( \exists s'' \)
    such that \( s_i \xrightarrow{\cdot \cdot \cdot} s'' \),
  - \( S_i \) is well-formed,
  - all PSMs in \( S_i \) monitor distinct markers, and
  - \( \text{Mon}(S_i) \cap \text{Mon}(S_j) = \emptyset \) for all \( j \in 1 \ldots n \) such that \( i \neq j \),

then there exist \( S', s', \) and \( s'_1, \ldots, s'_n \) such that

- \( S \xrightarrow{\cdot \cdot \cdot} S' \),
- \( \text{SplitSelected}(s', H) = \{s'_1, \ldots, s'_n\} \),
- \( S' = \{s'_1\} \cup (S'_1 - \{s'_1\}) \cup \ldots \cup (S'_n - \{s'_n\}) \), and
- for all \( i \in 1 \ldots n, \)
  - \( s'_i \in S'_i \).
Then the necessary conditions are satisfied by the preconditions to this lemma.

Proof: The idea of this proof is to chain the transitions of each of the $i$ separate configurations one after another, using corollary L.6.1 to the Split Silent Transitions lemma.

To use corollary L.6.1, we must first show that for all $i \in 1\ldots n$,

- either $\text{Mon}(s^\prime) \neq \emptyset$ for all $s^\prime \in S_i$ or $S_i = \{s_i\}$, and

- either $\text{Mon}(s^\prime) \neq \emptyset$ for all $s^\prime \in S_i^\prime$ or $S_i^\prime = \{s_i\}$.

Let there be some $i \in 1\ldots n$. If $S_i \neq \{s_i\}$, then by the precondition to this lemma, $\text{Mon}(s^\prime) \neq \emptyset$ for all $s^\prime \in S_i^\prime$. Then by the No Monitored Markers lemma, $\text{Mon}(s^\prime) \neq \emptyset$ for all $s^\prime \in S_i$. Also, if there exists some $s^\prime \in S_i^\prime$ such that $\text{Mon}(s^\prime) = \emptyset$, then by the precondition to this lemma, $S_i = \{s_i\}$, $\text{Mon}(s_i) = \emptyset$, and $\exists s^\prime$ such that $s_i \xrightarrow{s_i,\emptyset} s^\prime$. Then by the No Silent Transitions lemma, $S_i^\prime = \{s_i\}$.

Next, we will show that for all $i \in 1\ldots n + 1$, there exist $S'$, $s'$, and $s_1', \ldots, s_{i-1}'$ such that

- $S \xrightarrow{s_i} S'$,

- $\text{SplitSelected}(s',H) = \{s_1', \ldots, s_{i-1}', s_i, \ldots, s_n\}$,

- $S' = \{s_1'\} \cup (\{s_1'\} \cup \ldots \cup (S_i^\prime - \{s_i\}) \cup (S_i - \{s_i\}) \cup \ldots \cup (S_n - \{s_n\}))$, and

- for all $j < i$,
  - $s_j' \in S_j'$,
  - $S_j'$ is well-formed,
  - all PSMs in $S_j'$ monitor distinct markers,
  - $\text{Mon}(s_j) = \text{Mon}(s_j')$, and
  - $\text{Mon}(S_j) = \text{Mon}(S_j')$.

The proof is by induction on $i$. In the base case, $i = 1$. Let $S' = S$ and $s' = s$. Then the necessary conditions are satisfied by the preconditions to this lemma.

In the inductive case, $i > 1$, and by the induction hypothesis, there exist $S''$, $s''$, and $s_1', \ldots, s_{i-2}'$ such that

- $S \xrightarrow{s_i} S''$,

- $\text{SplitSelected}(s'',H) = \{s_1', \ldots, s_{i-2}', s_{i-1}, \ldots, s_n\}$,

- $S'' = \{s''\} \cup (S_1^\prime - \{s_1\}) \cup \ldots \cup (S_{i-2}^\prime - \{s_{i-2}\}) \cup (S_{i-1} - \{s_{i-1}\}) \cup \ldots \cup (S_n - \{s_n\})$, and

- for all $j < i - 1$,
APPENDIX L. CONFORMANCE-REFLECTION PROOFS

by corollary K.21.2 to the Monitored Marker Permanence lemma, such that

\[ \text{which completes the proof.} \]

Distinct-Marker Preservation lemma, all PSMs in \( S'_j \) is well-formed. We know that \( \text{Matchable}(\star) = \emptyset \), so by corollary K.12.1 to the Distinct-Marker Preservation lemma, all PSMs in \( S'_{i-1} \) monitor distinct markers.

Because \( S \xrightarrow{\cdot \emptyset} S'' \) and \( S'' \xrightarrow{\cdot \emptyset} S' \), we have \( S \xrightarrow{\cdot \emptyset} S'' \). We must also show that \( \text{Mon}(S_{i-1}) = \text{Mon}(S'_{i-1}) \). We know that \( \text{Matchable}(\star) = \emptyset \), so by corollary K.8.1 to the Monitored Matchable Markers lemma, \( \text{Mon}(S'_{i-1}) \subseteq \text{Mon}(S_{i-1}) \). Then by corollary K.21.2 to the Monitored Marker Permanence lemma, \( \text{Mon}(S_{i-1}) \subseteq \text{Mon}(S'_{i-1}) \). Therefore, \( \text{Mon}(S_{i-1}) = \text{Mon}(S'_{i-1}) \).

As a result of the inductive argument above, there exist \( S', s' \), and \( s'_1, \ldots, s'_n \) such that

\[ S \xrightarrow{\cdot \emptyset} S', \]

\[ \text{SplitSelected}(s', H) = \{s'_1, \ldots, s'_{i-1}, s_i, \ldots, s_n\}, \]

\[ S' = \{s'\} \cup (S'_1 - \{s'_1\}) \cup \ldots \cup (S'_{i-1} - \{s'_{i-1}\}) \cup (S_i - \{s_i\}) \cup \ldots \cup (S_n - \{s_n\}), \]

\[ s'_{i-1} \in S'_{i-1}, \text{ and} \]

\[ \text{Mon}(s_{i-1}) = \text{Mon}(s'_{i-1}). \]

\[ S_i = \{s_i\} \cup (S'_i - \{s'_i\}) \cup \ldots \cup (S_n - \{s_n\}), \]

\[ \text{for all } i \in 1 \ldots n, \]

\[ - s'_i \in S'_i, \]

\[ - \text{Mon}(s_i) = \text{Mon}(s'_i), \text{ and} \]

\[ - \text{Mon}(S_i) = \text{Mon}(S'_i). \]

It remains to show that \( \text{Mon}(S'_i) \cap \text{Mon}(S'_j) = \emptyset \) for all \( i \) and \( j \) in \( 1 \ldots n \) such that \( i \neq j \). Let there be such an \( i \) and \( j \). We know that \( \text{Mon}(S_i) \cap \text{Mon}(S_j) = \emptyset \), \( \text{Mon}(S_i) = \text{Mon}(S'_i) \), and \( \text{Mon}(S_j) = \text{Mon}(S'_j) \). Therefore \( \text{Mon}(S'_i) \cap \text{Mon}(S'_j) = \emptyset \), which completes the proof. \( \square \)
L.8  Split Transitions Lemma

Lemma (Split Transitions). For all \( s, s_1, \ldots, s_n, S, S_1, \ldots, S_n, S'_1, \ldots, S'_n, \hat{\lambda}, O_1, \ldots, O_n, \) and \( H, \) if

- \( \text{SplitSelected}(s, H) = \{s_1, \ldots, s_n\} \),
- \( S = \{s\} \cup (S_1 - \{s_1\}) \cup \ldots \cup (S_n - \{s_n\}) \),
- there exist \( \hat{a}, H', \) and \( \hat{v} \) such that
  - \( \hat{\lambda} = \hat{a} \cdot H'?\hat{v} \) or \( \hat{\lambda} = \hat{a} \cdot H'!\hat{v} \),
  - \( |H'| \leq 1 \), and
  - every marker in \( \text{Matchable}(\hat{\lambda}) \) appears at most once in \( \hat{v} \), and
- for all \( i \in 1 \ldots n \),
  - \( S_i \xrightarrow{\hat{\lambda}, O_i} S'_i \)
  - \( s_i \in S_i \),
  - either \( \text{Mon}(s'') \neq \emptyset \) for all \( s'' \in S'_i \), or \( S_i = \{s_i\} \), \( \text{Mon}(s_i) = \emptyset \), and \( \hat{\lambda} \) s''
  - such that \( s_i \xrightarrow{\cdot, \cdot, \cdot} s'' \),
  - \( S_i \) is well-formed,
  - all PSMs in \( S_i \) monitor distinct markers,
  - \( \text{Mon}(S_i) \cap \text{Matchable}(\hat{\lambda}) = \emptyset \), and
  - \( \text{Mon}(S_i) \cap \text{Mon}(S_j) = \emptyset \) for all \( j \in 1 \ldots n \) such that \( i \neq j \),

then there exist \( S', s', s'_1, \ldots, s'_n, \) and \( O \) such that

- \( S \xrightarrow{\hat{\lambda}, O} S' \),
- \( O = \bigcup_{i \in 1 \ldots n} O_i \),
- \( \text{SplitSelected}(s', H) = \{s'_1, \ldots, s'_n\} \),
- \( S' = \{s'_1\} \cup (S'_1 - \{s'_1\}) \cup \ldots \cup (S'_n - \{s'_n\}) \), and
- for all \( i \in 1 \ldots n \)
  - \( s'_i \in S'_i \),
  - \( \text{Mon}(s_i) \subseteq \text{Mon}(s'_i) \), and
  - \( \text{Mon}(S'_i) \cap \text{Mon}(S'_j) = \emptyset \) for all \( j \in 1 \ldots n \) such that \( i \neq j \).

Proof. We will first show the properties for each of the \( s'_i \) and \( S'_i \), then define \( s' \) and show a transition to it from \( s \) and show \( \text{SplitSelected}(s', H) = \{s'_1, \ldots, s'_n\} \), and finally define \( S' \) and show that \( S \xrightarrow{\hat{\lambda}, O} S' \) and the property for the construction of \( S' \).
Properties for Split Components

For all $i \in 1 \ldots n$, the transition $S_i \rightarrow S'_i$ must be a use of the S-SENDOR-RECEIVE rule. Therefore, for all $i \in 1 \ldots n$, there exist $s''_{i,1}, \ldots, s''_{i,m}$, $s''_{i,1}, \ldots, s''_{i,m}$, $S''_{i,1}, \ldots, S''_{i,m}$, and $O'_{i,1}, \ldots, O'_{i,m}$ such that

- $S_i = \{s''_{i,1}, \ldots, s''_{i,m}\}$,
- $s''_{i,j} \rightarrow s''_{i,j}$ for all $j \in 1 \ldots m$,
- $O_i = O'_{i,1} \cup \ldots \cup O'_{i,m}$, and
- $S'_i = s''_{i,1} \cup \ldots \cup s''_{i,m} \cup S''_{i,1} \cup \ldots \cup S''_{i,m}$.

Without loss of generality, assume that for all $i \in 1 \ldots n$, $s_i = s''_{i,1}$. For all $i \in 1 \ldots n$, let $s'_i = s''_{i,1}$. Then for all $i \in 1 \ldots n$, we have $s'_i \in S'_i$. By corollary K.21.1 to the Monitored Marker Permanence lemma, we also have $Mon(s_i) \subseteq Mon(s'_i)$.

Next, let there be $i$ and $j$ in $1 \ldots n$ such that $i \neq j$; we will show that $Mon(S'_i) \cap Mon(S'_j) = \emptyset$. We already know that $Mon(S_i) \cap Mon(S_j) = \emptyset$. Because $|H| \leq 1$, either $Mon(S_i) \cap H = \emptyset$ or $Mon(S_j) \cap H = \emptyset$. Assume the former is the case; the proof for the other case is similar. Because $Mon(S_i) \cap H = \emptyset$, for all $k \in 1 \ldots m$, the transition $s''_{i,k} \rightarrow s''_{i,k}$ must be an instance of either the P-UNMONITORED-RECEIVE or P-SEND rule, and by the definitions of those rules, $Mon(s''_{i,k}) = Mon(s''_{i,k})$. Therefore $Mon(S'_i) = Mon(S_i^i)$. Then by the Monitored Matchable Markers lemma, $Mon(S'_i) \subseteq Mon(S_i) \cup \text{Matchable}(\hat{\lambda})$. We know that $Mon(S_i) \cap \text{Matchable}(\hat{\lambda}) = \emptyset$ for all $i \in 1 \ldots n$, so $Mon(S'_i) \cap Mon(S'_j) = \emptyset$.

Transition From $s$

Next, let $O' = \{O_{i,1} \in 1 \ldots n\}$, and let $S'' = \bigcup_{i \in 1 \ldots n} S''_{i,1}$. We will show that there exists $s'$ such that $s \rightarrow S''_{i,1}$ and $\text{SplitSelected}(s',H) = \{s'_1, \ldots, s'_n\}$. The proof is by case analysis on the shape of $\hat{\lambda}$.

First, consider the case where $\hat{\lambda} = \hat{\alpha}@H?\hat{\nu}$. By the definition of SplitSelected, there is at most one $i \in 1 \ldots n$ such that $\text{InMon}(s_i) \neq \emptyset$. If there does not exist such an $i$, or $\text{InMon}(s_i) \cap H = \emptyset$, then for all $i \in 1 \ldots n$, the transition $s_i \rightarrow S''_{i,1}$ is a use of the P-UNMONITORED-RECEIVE rule. In that case, $O'_{i,1} = \emptyset$, $S''_{i,1} = \emptyset$, and $s'_i = s_i$ for all $i \in 1 \ldots n$. Therefore, By the definition of SplitSelected, it is similarly the case that $\text{InMon}(s) \cap H = \emptyset$, so we have $s \rightarrow s'$. Because $O'_{i,1} = \emptyset$ and $S''_{i,1} = \emptyset$, we have $O' = \emptyset$ and $S'' = \emptyset$. Let $s' = s$; then $s \rightarrow S''_{i,1}$ and $\text{SplitSelected}(s',H) = \{s'_1, \ldots, s'_n\}$.
Without loss of generality, assume $i = 1$. Let $\langle H', H''', \varphi : \eta'_1 \ldots \eta'_p, \Phi, O''_1 \rangle = s_1$. In that case, the transition $s_1 \xrightarrow{\lambda, O'_{1,1}, S''_{1,1}} s'_1$ must be a use of the P-MONITOREDRECEIVE rule, so there exist $H''', \varphi', \eta_1'' \ldots \eta_q''$, and $O''_1$ such that $s'_1 = \langle H''', H''' \cup H''', \varphi' : \eta_1'' \ldots \eta_q'' \Phi, O''_1 \rangle \cup O''_1 \rangle$. By the definition of SplitSelected, it must be the case that $s = \langle H''', H \cup H''' \cup H''', \varphi : \eta_1'' \ldots \eta_q'' \Phi, O''_1 \cup O''_1 \rangle$, where $O''_1 = \text{Obls}(s_1)$ for all $i \in 2 \ldots n$. The extra monitored markers and obligations do not affect the P-MONITOREDRECEIVE rule, so there exists $s'$ such that $\lambda, O'_{1,1}, S''_{1,1}) \xrightarrow{\lambda, O'_{1,1}, S''_{1,1}} s'$ and $s' = \langle H''', H \cup H''' \cup H''', \varphi : \eta_1'' \ldots \eta_q'' \Phi, O''_1 \cup O''_1 \rangle$. Then by the definition of SplitSelected, SplitSelected($s'$) = \{s'_1, \ldots, s'_n\}. We still have that $O'_{i,1} = \varnothing$, $S''_{i,1} = \varnothing$, and $s'_i = s_i$ for all $i \in 2 \ldots n$. Therefore $O' = O'_{1,1}$ and $S'' = S''_{1,1}$, so $\lambda, O', S'' \xrightarrow{\lambda, O', S''} s'$. 

Second, consider the case where $\hat{\lambda} = \hat{\lambda} \hat{\lambda} H' \hat{\lambda}$. In this case, for all $i \in 1 \ldots n$, the transition $\lambda, O'_{i,1}, S''_{i,1} \xrightarrow{\lambda, O'_{i,1}, S''_{i,1}} s'_i$ is an instance of the P-SEND rule. Because $|H'| \leq 1$ and $\text{Mon}(S_i) \cap \text{Mon}(S_j) = \varnothing$ for all $i$ and $j$ in $1 \ldots n$ such that $i \neq j$, there is at most one $i \in 1 \ldots n$ such that $\text{OutMon}(s_i) \cap H' \neq \varnothing$. If there is no such $i$, then for all $i \in 1 \ldots n$, $O'_{i,1} = \varnothing$, $S''_{i,1} = \varnothing$, and $s'_i = s_i$. Then by the definition of SplitSelected, SplitSelected($s'$) = \{s'_1, \ldots, s'_n\}. Because $O'_{i,1} = \varnothing$ and $S''_{i,1} = \varnothing$, we have $O' = \varnothing$ and $S'' = \varnothing$. Let $s' = s$; then $\lambda, O', S'' \xrightarrow{\lambda, O', S''} s'$ and SplitSelected($s'$, $H$) = \{s'_1, \ldots, s'_n\}$. 

Otherwise, there exists exactly one $i \in 1 \ldots n$ such that $\text{OutMon}(s_i) \cap H' \neq \varnothing$. Without loss of generality, assume $i = 1$. By the definition of the P-SEND rule, there exist $O'_{1,1}$ and $H''$ such that $\text{Obls}(s_1) = O'_{1,1} \cup O''_1$ and $s'_1$ is exactly like $s_i$, except $\text{InMon}(s'_1) = \text{InMon}(s_i) \cup H''$ and $\text{Obls}(s_1) = O''_1$. By the definition of SplitSelected, $\text{InMon}(s_i) \cap H' = \text{InMon}(s_1) \cap H'$, and $\text{Obls}(s_1) = O'_{1,1} \cup O''_1 \cup O''_1 \cup \ldots \cup O''_1$, where $O''_1 = \text{Obls}(s_1)$ for all $i \in 2 \ldots n$. Therefore, there exists $s'$ such that $s \xrightarrow{\lambda, O'_{1,1}, S''_{1,1}} s'$ and $s'$ is exactly like $s$, except $\text{InMon}(s') = \text{InMon}(s) \cup H''$ and $\text{Obls}(s) = O'_{1,1} \cup O''_1 \cup \ldots \cup O''_1 \cup O''_1$. We still have that $O'_{1,1} = \varnothing$, $S''_{i,1} = \varnothing$, and $s'_i = s_i$ for all $i \in 2 \ldots n$. Therefore $O' = O'_{1,1}$ and $S'' = S''_{1,1}$, so $\lambda, O', S'' \xrightarrow{\lambda, O', S''} s'$. 

In that case, if $H'' \neq \varnothing$, those markers must have come from a self-addr pattern in an obligation in $s_1$. The marker for such an obligation is by definition not an independent marker of $s$, so $s_1$ is the PSM that gets the input-marked markers from splitting $s$. Therefore, regardless of whether $H' = \varnothing$, SplitSelected($s'$, $H$) = \{s'_1, \ldots, s'_n\} by the definition of SplitSelected.
Transition From $S$

It remains to show there exist $O$ and $S'$ such that $S \xrightarrow{\lambda,O} S'$, $O = \bigcup_{i \in 1..n} O_i$, and $S' = \{s'\} \cup (S'_1 - \{s'_1\}) \cup \ldots \cup (S'_n - \{s'_n\})$. We have that

- $S = \{s\} \cup (S_1 - \{s_1\}) \cup \ldots \cup (S_n - \{s_n\})$,
- $S_i - \{s_i\} = \{s''_{i,2}, \ldots, s''_{i,m}\}$,
- \[ s \xrightarrow{\lambda,O,S''} s' \]
- \[ s''_{i,j} \xrightarrow{O'_{i,j},S''_{i,j}} s''_{i,j} \text{ for all } j \in 2..m, \]
- $O_i = O'_{i,1} \cup \ldots \cup O'_{i,m}$, and

Let $O = O' \cup (O_1 - O''_{1,1}) \cup \ldots \cup (O_n - O''_{n,1})$, and let $S' = \{s'\} \cup S'' \cup \left( \bigcup_{i \in 1..n} \bigcup_{j \in 2..m} \{s''_{i,j}\} \cup S''_{i,j} \right)$. Then by the S-SENDORRECEIVE rule, we have $S \xrightarrow{\lambda,O} S'$.

We know that $O' = \bigcup_{i \in 1..n} O'_{i,1}$. Therefore, $O = \bigcup_{i \in 1..n} O_i$.

We also know that $S'' = \bigcup_{i \in 1..n} S''_{i,1}$. Therefore, $S' = \{s'\} \cup \left( \bigcup_{i \in 1..n} \{s''_{i,2}, \ldots, s''_{i,m}\} \cup S''_{i,1} \cup \ldots \cup S''_{i,m} \right)$. It remains to show that for all $i \in 1..n$, $S'_i - \{s'_i\} = \{s''_{i,2}, \ldots, s''_{i,m}\} \cup S''_{i,1} \cup \ldots \cup S''_{i,m}$. We know that $s'_i = s''_{i,1}$ and $S'_i = \{s''_{i,2}, \ldots, s''_{i,m}\} \cup S''_{i,1} \cup \ldots \cup S''_{i,m}$ for all $i \in 1..n$, so it suffices to show that $s'_i \notin \{s''_{i,2}, \ldots, s''_{i,m}\} \cup S''_{i,1} \cup \ldots \cup S''_{i,m}$.

Let there be some $i \in 1..n$. Suppose it is the case that $s_i = \{s_i\}$, $\text{Mon}(s_i) = \emptyset$, and there does not exist $s''$ such that $s_i \xrightarrow{\lambda,O,S''} s''$. Then by the No Silent Transitions lemma, $S'_i = \{s_i\}$, and therefore $\{s''_{i,2}, \ldots, s''_{i,m}\} \cup S''_{i,1} \cup \ldots \cup S''_{i,m} = \emptyset$. Therefore, $s'_i \notin \{s''_{i,2}, \ldots, s''_{i,m}\} \cup S''_{i,1} \cup \ldots \cup S''_{i,m}$.

Otherwise, by the precondition to this lemma, every PSM in $S_i$ monitors at least one marker. so $\text{Mon}(s'_i) \neq \emptyset$. Then by the No Monitored Markers lemma, every PSM in $S_i$ monitors at least one marker. By the Monitored Matchable Markers lemma, $\text{Mon}(\{s''_{i,j}\} \cup S''_{i,j}) \subseteq \text{Mon}(s''_{i,j}) \cup \text{Matchable}(\lambda)$ for all $j \in 1..m$. Furthermore, all of the PSMs in $S_i$ monitor distinct markers and $|H'| \leq 1$, so there is at most one $j \in 1..m$ such that $\text{Mon}(s''_{i,j}) \cap H' \neq \emptyset$. Therefore, for all but at most one $j \in 1..m$, the transition $s''_{i,j} \xrightarrow{\lambda,O',S''_{i,j}} s_{i,j}''$, must be an instance of either the P-UNMONITOREDRECEIVE or P-SEND rule, and by the definitions of those rules, $\text{Mon}(s''_{i,j}) = \text{Mon}(s''_{i,j})$ and $\text{Mon}(S''_{i,j}) = \emptyset$. Therefore for all $j \in 2..m$, $\text{Mon}(s''_{i,j}) \cap \text{Mon} \{s''_{i,j} \cup S''_{i,j}\}$, so $s'_i \notin \{s''_{i,j}\} \cup S''_{i,j}$. Finally, if $S''_{i,j} \neq \emptyset$, then $s'_i \xrightarrow{\lambda,O',S''_{i,j}} s''_{i,j}$ must be an instance of the P-MONITOREDRECEIVE or P-SEND rule, and each
L.9. SPLIT WEAK-STEP TRANSITIONS LEMMA

Lemma (Split Weak-Step Transitions). For all $s, s_1, \ldots, s_n, S, S_1, \ldots, S_n, S'_{i,1}, \ldots, S'_{i,n}, \lambda, O_1, \ldots, O_n$, and $H$, if

- $\text{SplitSelected}(s, H) = \{s_1, \ldots, s_n\}$,

- $S = \{s\} \cup (S_1 - \{s_1\}) \cup \ldots \cup (S_n - \{s_n\})$,

- if there exist $\hat{a}^i, H', \hat{\nu}^i$, and $\nu^i$ such that $\lambda = \hat{a}@H'?\hat{\nu}^i$ or $\lambda = \hat{a}@H'\nu^i$, then $|H'| \leq 1$ and every marker in $\text{Matchable}(\lambda)$ appears at most once in $\hat{\nu}$, and

- for all $i \in 1 \ldots n$,
  
  - $S_i \xrightarrow{\lambda, O_i} S'_i$
  
  - $s_i \in S_i$,

  - either $\text{Mon}(s''_i) \neq \emptyset$ for all $s''_i \in S'_i$, or $S_i = \{s_i\}$, $\text{Mon}(s_i) = \emptyset$, and $\exists s''$ such that $s_i \xrightarrow{\nu^i, \emptyset} s''$,

  - $S_i$ is well-formed,

  - all PSMs in $S_i$ monitor distinct markers,

  - $\text{Mon}(S_i) \cap \text{Matchable}(\lambda) = \emptyset$, and

  - $\text{Mon}(S_i) \cap \text{Mon}(S_j) = \emptyset$ for all $j \in 1 \ldots n$ such that $i \neq j$,

then there exist $S', s', s'_{1,1}, \ldots, s'_{1,n}$, and $O$ such that

- $S \xrightarrow{\lambda, O} S'$,

- $O = \bigcup_{i \in 1 \ldots n} O_i$,

- $\text{SplitSelected}(s', H) = \{s'_{1,1}, \ldots, s'_{1,n}\}$,

- $S' = \{s'\} \cup (S'_{1,1} - \{s'_{1,1}\}) \cup \ldots \cup (S'_{1,n} - \{s'_{1,n}\})$, and

- for all $i \in 1 \ldots n$
  
  - $s' \in S'_{i,1}$,

  - $\text{Mon}(s_i) \subseteq \text{Mon}(s'_i)$, and
- \( Mon(S'_i) \cap Mon(S'_j) = \varnothing \) for all \( j \in 1 \ldots n \) such that \( i \neq j \).

**Proof.** If \( \hat{\lambda} = \bullet \), then \( O_i = \varnothing \) for all \( i \in 1 \ldots n \), so the necessary \( S', s', \) and \( s'_1, \ldots, s'_n \) exist by the Split Silent Weak-Step Transitions lemma. Let \( O = \varnothing \) to complete this case.

Otherwise, \( \hat{\lambda} \neq \bullet \), so for all \( i \in 1 \ldots n \) there exist \( S''_i \) and \( S'''_i \) such that \( S_i \xrightarrow{\varphi} S''_i \xrightarrow{\lambda, O} S'''_i \xrightarrow{\varphi} S'_i \). By corollaries K.5.1 and K.5.2 to the Specification Well-Formed Preservation lemma, \( S''_i \) and \( S'''_i \) are well-formed. By corollary K.12.1 to the Distinct-Marker Preservation lemma, all PSMs in \( S''_i \) monitor distinct markers, and by that lemma itself, all PSMs in \( S'''_i \) monitor distinct markers.

Suppose that \( Mon(s''_i) \neq \varnothing \) for all \( i \). Then by corollary L.2.1 to the No Monitored Markers lemma, \( Mon(s'''_i) \neq \varnothing \) for all \( i \). Then by the No Monitored Markers lemma, \( Mon(s'''_i) \neq \varnothing \) for all \( i \).

On the other hand, if it is not the case that \( Mon(s''_i) \neq \varnothing \) for all \( i \), then by the Split Silent Weak-Step Transitions lemma, there exist \( S''_i \), \( s''_i \), and \( s''_{i_1}, \ldots, s''_{i_n} \) with properties similar to those of \( S, s, \) and \( s_{i_1}, \ldots, s_{i_n} \). By corollary L.3.1 to the No Silent Transitions lemma, \( S''_i = \{ s_i \} \). Then by the No Silent Transitions lemma, \( S'''_i = \{ s_i \} \).

Now by the Split Silent Weak-Step Transitions lemma, there exist \( S''_i \), \( s''_i \), and \( s''_{i_1}, \ldots, s''_{i_n} \) with properties similar to those of \( S, s, \) and \( s_{i_1}, \ldots, s_{i_n} \). We know that \( Matchable(\bullet) = \varnothing \), so by corollary K.8.1 to the Monitored Matchable Markers lemma, \( Mon(s''_i) \subseteq Mon(S_i) \) for all \( i \). We also have that \( Mon(S''_i) \neq \varnothing \) for all \( i \). Finally by the Split Silent Weak-Step Transitions lemma again, there exist \( s', s'_1, \ldots, s'_n \) with properties similar to those of \( S, s, \) and \( s_{i_1}, \ldots, s_{i_n} \). Therefore, for all \( i \), \( Mon(s'_i) \subseteq Mon(s''_i) \).

**L.10 Split Event-Step Transitions Lemma**

**Lemma** (Split Event-Step Transitions). For all \( s, s_{i_1}, \ldots, s_{i_n} \), \( S, S_{i_1}, \ldots, S_{i_n}, \lambda_{i_1}, \ldots, \lambda_{i_m}, (O_{1,1}, \ldots, O_{1,m}), \ldots, (O_{n,1}, \ldots, O_{n,m}) \), and \( H \), if

- \( \text{SplitSelected}(s, H) = \{ s_{i_1}, \ldots, s_{i_n} \} \),
- \( S = \{ s \} \cup (S_{i_1} - \{ s_{i_1} \}) \cup \ldots \cup (S_{i_n} - \{ s_{i_n} \}) \),
- for all \( j \in 1 \ldots m \) if there exist \( \hat{a}, H' \), and \( \hat{v} \) such that \( \hat{\lambda}_j = \hat{a}@H' ? \hat{v} \) or \( \hat{\lambda}_j = \hat{a}@H' \hat{v} \), then \( |H'| \leq 1 \) and every marker in Matchable(\( \hat{\lambda}_j \)) appears at most once in \( \hat{v} \).
L.10. SPLIT EVENT-STEP TRANSITIONS LEMMA

- Matchable($\tilde{\lambda}_j$) ∩ Matchable($\tilde{\lambda}_k$) = $\emptyset$ for all $j$ and $k$ in $1 \ldots m$ such that $j \neq k$, and
- for all $i \in 1 \ldots n$,
  - $S_i$ $\to_{(\tilde{\lambda}_1, O_{i,1}), \ldots, (\tilde{\lambda}_m, O_{i,m})} S'_i$
  - $s_i \in S_i$,
  - either $\text{Mon}(s'') \neq \emptyset$ for all $s'' \in S'_i$, or $S_i = \{s_i\}$, $\text{Mon}(s_i) = \emptyset$, and $\exists s''$ such that $s_i \xrightarrow{\cdot, \emptyset, \emptyset} s''$,
  - $S_i$ is well-formed, and
  - all PSMs in $S_i$ monitor distinct markers,
  - $\text{Mon}(S_i) \cap \text{Matchable}(\tilde{\lambda}_j) = \emptyset$ for all $j \in 1 \ldots m$,
  - $\text{Mon}(S_i) \cap \text{Mon}(S_j) = \emptyset$ for all $j \in 1 \ldots n$ such that $i \neq j$,

then there exist $S'_i$, $s'_i$, $s'_1, \ldots, s'_n$, and $O_1, \ldots, O_m$ such that

- $S \to_{(\tilde{\lambda}_1, O_{1}), \ldots, (\tilde{\lambda}_m, O_{m})} S'$,
- $O_j = \bigcup_{i=1 \ldots n} O_{i,j}$ for all $j \in 1 \ldots m$,
- $\text{SplitSelected}(s', H) = \{s'_1, \ldots, s'_n\}$,
- $S' = \{s'_1\} \cup (S'_1 - \{s'_1\}) \cup \ldots \cup (S'_n - \{s'_n\})$, and
- for all $i \in 1 \ldots n$, $s'_i \in S'_i$ and $\text{Mon}(s_i) \subseteq \text{Mon}(s'_i)$.

Proof. By the definition of the $\to$ relation, for all $i \in 1 \ldots n$ there exist $S''_{i,1}, \ldots, S''_{i,m+1}$ such that

- $S''_{i,1} = S_i$,
- $S''_{i,m+1} = S_i'\text{, and}$
- $S''_{i,1} \to_{\tilde{\lambda}_1, O_{i,1}} \ldots \to_{\tilde{\lambda}_m, O_{i,m}} S''_{i,m+1}$.

By induction on $j$ and corollary K.5.2 to the Specification Well-Formed Preservation lemma, $S''_i$ is well-formed for all $i \in 1 \ldots n$ and all $j \in 1 \ldots m + 1$.

To use the Split Weak-Step Transitions lemma, we must show that for all $j \in 1 \ldots m + 1$ and all $i \in 1 \ldots n$, if $\text{Mon}(s'') \neq \emptyset$ for all $s'' \in S'_i$, then $\text{Mon}(s'') \neq \emptyset$ for all $s''' \in S''_{i,j}$. The proof is by induction on $m + 1 - j$. In the base case where $j = m + 1$, $S'_i = S''_{i,j}$, so the result holds by default. In the inductive case, suppose that $\text{Mon}(s'') \neq \emptyset$ for all $s'' \in S'_i$. By the induction hypothesis, we know that $\text{Mon}(s''') \neq \emptyset$ for all $s''' \in S''_{i,j+1}$. Then by corollary L.2.2 to the No Monitored Markers lemma, $\text{Mon}(s'') \neq \emptyset$ for all $s'' \in S''_{i,j}$.
We must also show that for all \( j \in 1\ldots m \), and all \( i \in 1\ldots n \), if \( S_i = \{s_i\} \), \( \text{Mon}(s_i) = \emptyset \), and \( \exists s'' \) such that \( s_i \xrightarrow{\ast,\emptyset} s'' \), then \( S''_{i,j} = \{s_i\} \). Let \( i \) be a number in \( 1\ldots n \); the proof is by induction on \( j \). In the base case where \( j = 1 \), \( S''_{i,j} = S_i \), so the conditions hold by the preconditions on this lemma. If \( j > 1 \), suppose that \( S_i = \{s_i\} \), \( \text{Mon}(s_i) = \emptyset \), and \( \exists s'' \) such that \( s_i \xrightarrow{\ast,\emptyset} s'' \). By the induction hypothesis, \( S''_{i,j-1} = \{s_i\} \). Then by corollary L.32 to the No Silent Transitions lemma, \( S''_{i,j} = \{s_i\} \).

We must also show that for all \( j \in 1\ldots m \), and all \( i \in 1\ldots n \),

- \( S''_{i,j} \) is well-formed,
- all PSMs in \( S''_{i,j} \) monitor distinct markers, and
- \( \text{Mon}(S''_{i,j}) \cap \text{Matchable}(\hat{\lambda}_k) = \emptyset \) for all \( k \in j \ldots m \)

Let there be some \( i \in 1\ldots n \); the proof is by induction on \( j \). In the base case where \( j = 1 \), the conditions hold by the preconditions on this lemma. In the inductive case where \( j > 1 \), we know from the induction hypothesis that \( S''_{i,j-1} \) is well-formed, all PSMs in \( S''_{i,j-1} \) monitor distinct markers, and \( \text{Mon}(S''_{i,j-1}) \cap \text{Matchable}(\hat{\lambda}_k) = \emptyset \) for all \( k \in j-1 \ldots m \). By corollary K.5.2 to the Specification Well-Formed Preservation lemma, \( S''_{i,j} \) is well-formed. By corollary K.12.1 to the Distinct-Marker Preservation lemma, all PSMs in \( S''_{i,j} \) monitor distinct markers. By corollary K.8.1 to the Monitored Matchable Markers lemma, \( \text{Mon}(S''_{i,j}) \subseteq \text{Mon}(S''_{i,j-1}) \cup \text{Matchable}(\hat{\lambda}_{j-1}) \). Then because \( \text{Mon}(S''_{i,j-1}) \cap \text{Matchable}(\hat{\lambda}_k) = \emptyset \) for all \( k \in j-1 \ldots m \) and \( \text{Matchable}(\hat{\lambda}_{j-1}) \cap \text{Matchable}(\hat{\lambda}_k) = \emptyset \) for all \( k \in j \ldots m \), we have \( \text{Mon}(S''_{i,j}) \cap \text{Matchable}(\hat{\lambda}_k) = \emptyset \) for all \( k \in j \ldots m \).

Next, we will show for \( j \in 1 \ldots m + 1 \) that there exist \( S''_j \), \( s''_j \), \( s''_{i,j} \), \ldots, \( s''_{n,j} \), and (when \( j > 1 \)) \( O'_{j-1} \) such that

- \( \overset{\hat{\lambda}_{j-1}, O'_{j-1}}{S''_{j-1}} \rightarrow S''_j \) if \( j > 1 \),
- \( O'_{j-1} = \bigcup_{i=1 \ldots n} O'_{i,j-1} \) if \( j > 1 \),
- \( \text{SplitSelected}(s''_j, H) = \{s''_{1,j}, \ldots, s''_{n,j}\} \),
- \( S''_j = \{s''_j\} \cup (S''_{1,j} - \{s''_{1,j}\}) \cup \ldots \cup (S''_{n,j} - \{s''_{n,j}\}) \), and
- for all \( i \in 1 \ldots n \),
  - \( s''_{i,j} \in S''_{i,j} \),
  - \( \text{Mon}(s_i) \subseteq \text{Mon}(s''_{i,j}) \), and
  - \( \text{Mon}(S''_{i,j}) \cap \text{Mon}(S''_{k,j}) = \emptyset \) for all \( k \in 1 \ldots n \) such that \( i \neq k \).
L.11. SPLIT SUMMARY TRANSITIONS LEMMA

The proof is by induction on $j$. In the base case where $j = 1$, let $S''_j = S$, $s''_j = s$, and $s''_{i,j} = s_i$ for all $i \in 1\ldots n$. The necessary conditions hold by the preconditions on this lemma, with the addition of the fact that $\text{Mon}(s_i) \subseteq \text{Mon}(s''_{i,j})$ for all $i \in 1\ldots n$ because $s''_{i,j} = s_i$.

In the inductive case where $j > 1$, the necessary $S''_j$, $s''_j$, $s''_{1,j}$, $\ldots$, $s''_{n,j}$, and $O_{j-1}$ exist by the Split Weak-Step Transitions lemma, except that for all $i \in 1\ldots n$ we have $\text{Mon}(s''_{i,j-1}) \subseteq \text{Mon}(s''_{i,j})$ rather than $\text{Mon}(s_{i,j}) \subseteq \text{Mon}(s''_{i,j})$. By the induction hypothesis, however, $\text{Mon}(s_i) \subseteq \text{Mon}(s''_{i,j-1})$, so $\text{Mon}(s_i) \subseteq \text{Mon}(s''_{i,j})$.

Let $S' = S''_{m+1}$, $s' = s''_{m+1}$, and $s'_i = s''_{i,m+1}$ for all $i \in 1\ldots n$ to complete the proof.

\section*{L.11 Split Summary Transitions Lemma}

\textbf{Lemma} (Split Summary Transitions). For all $\hat{\mathcal{K}}$, $\hat{\mathcal{K}}'$, $L$, $s$, $s_1,\ldots,s_n$, $H$, $S$, $S_1,\ldots,S_n$, $S'_1,\ldots,S'_n$, and $O_1,\ldots,O_n$, if

- $\hat{\mathcal{K}}$ is an externals-only configuration,
- $\hat{\mathcal{K}} \xrightarrow{L} \hat{\mathcal{K}}'$,
- $\text{SplitSelected}(s,H) = \{s_1,\ldots,s_n\}$,
- $S = \{s\} \cup (S_1 - \{s_1\}) \cup \ldots \cup (S_n - \{s_n\})$, and
- for all $i \in 1\ldots n$,
  - $S_i \xrightarrow{(L,O_i)} S'_i$,
  - $s_i \in S_i$,
  - either $\text{Mon}(s'') \neq \emptyset$ for all $s'' \in S'_i$, or $S_i = \{s_i\}$, $\text{Mon}(s_i) = \emptyset$, and $\exists s''$ such that $s_i \xrightarrow{\emptyset,\emptyset,\emptyset} s''$,
  - $S_i$ is well-formed,
  - all of the PSMs in $S_i$ monitor distinct markers,
  - $\text{Mon}(S_i) \subseteq \text{Used}(\hat{\mathcal{K}})$, and
  - $\text{Mon}(S_i) \cap \text{Mon}(S_j) = \emptyset$ for all $j \in 1\ldots n$ such that $i \neq j$,

then there exist $S'$, $s'$, and $s'_1,\ldots,s'_n$ such that

- $S \xrightarrow{(L,O_1,\ldots,O_n)} S'$,
- $\text{SplitSelected}(s',H) = \{s'_1,\ldots,s'_n\}$,
- $S' = \{s'_1\} \cup (S'_1 - \{s'_1\}) \cup \ldots \cup (S'_n - \{s'_n\})$, and
- for all $i \in 1\ldots n$, $s'_i \in S'_i$ and $\text{Mon}(s_i) \subseteq \text{Mon}(s'_i)$. 
Proof. To define $S'$, $s'$, and $s_1', ..., s_n'$, let $\hat{I}_1, ..., \hat{I}_m$ be a sequence summarized by $L$ (such a sequence must exist because $K \xrightarrow{L} \hat{K}'$). By corollary K.7.3 to the Externals-Only Label lemma, for all $j \in 1 ... m$,

- $\text{Matchable}(\hat{I}_j) \cap \text{Used}(\hat{K}) = \emptyset$ and
- if $[\hat{I}_j] = \hat{a}@H^?\hat{v}$ or $[\hat{I}_j] = \hat{a}@H'^?\hat{v}$ for some $\hat{a}$, $H'$, and $\hat{v}$, then $|H'| \leq 1$ and no marker in $\text{Matchable}(\hat{I}_j)$ appears more than once in $\hat{v}$.

For all $i \in 1 ... n$, because $\text{Mon}(S_i) \subseteq \text{Used}(\hat{K})$, we also have $\text{Mon}(S_i) \cap \text{Matchable}(\hat{I}_j) = \emptyset$ for all $j \in 1 ... m$. By the Distinct Matchables lemma (appendix K), for all $j$ and $k$ in $1 ... m$ such that $j \neq k$, $\text{Matchable}(\hat{I}_j) \cap \text{Matchable}(\hat{I}_k) = \emptyset$.

By the definition of the specification summary transition, for all $i \in 1 ... n$ there exist $O_{1,i}, ..., O_{m,i}$ such that $S_i \xrightarrow{\langle \hat{I}_{i,1}, \hat{O}_{1,i} \rangle, ..., \langle \hat{I}_{i,m}, \hat{O}_{m,i} \rangle} S'_i$ and $O_i \subseteq \bigcup_{j \in 1 ... m} O_{i,j}$. Then by the Split Event-Step Transitions lemma, there exist $S'$, $s'$, $s_1', ..., s_n'$, and $O_{1}', ..., O_{m}'$ such that

- $S \xrightarrow{LO_{1,\ldots,m}O_{n}} S'$
- $O'_j = \bigcup_{i \in 1 ... n} O_{i,j}'$ for all $j \in 1 ... m$,
- $\text{SplitSelected}(s', H) = \{s_1', ..., s_n'\}$,
- $S' = \{s'\} \cup (S'_1 - \{s'_1\}) \cup \ldots \cup (S'_n - \{s'_n\})$, and
- for all $i \in 1 ... n$, $s_i' \in S_i'$ and $\text{Mon}(s_i) \subseteq \text{Mon}(s_i')$.

It remains to show that $S \xrightarrow{LO_{1,\ldots,m}O_{n}} S'$. Let $\hat{I}_1, ..., \hat{I}_p$ be a sequence summarized by $L$. By the definition of the specification summary transition, for all $i \in 1 ... n$ there exist $O_{1,i,p}''$, ..., $O_{i,p}''$ such that $S_i \xrightarrow{\langle \hat{I}_{i,1}, \hat{O}_{1,i}'' \rangle, ..., \langle \hat{I}_{i,p}, \hat{O}_{i,p}'' \rangle} S_i'$ and $O_i \subseteq \bigcup_{j \in 1 ... p} O_{i,j}''$. Then by the Split Event-Step Transitions lemma, there exist $S''$, $s''$, $s_1'', ..., s_n''$, and $O_{1}', ..., O_{p}'$ such that

- $S \xrightarrow{\langle \hat{I}_{1,1}, \hat{O}_{1,1}'' \rangle, ..., \langle \hat{I}_{p,1}, \hat{O}_{p,1}'' \rangle} S''$
- $O''_j = \bigcup_{i \in 1 ... n} O_{i,j}''$ for all $j \in 1 ... p$,
- $\text{SplitSelected}(s'', H) = \{s_1'', ..., s_n''\}$,
- $S'' = \{s''\} \cup (S'_1 - \{s'_1\}) \cup \ldots \cup (S'_n - \{s'_n\})$, and
- for all $i \in 1 ... n$, $s_i'' \in S_i'$ and $\text{Mon}(s_i) \subseteq \text{Mon}(s_i')$. 


To show that $S'' = S'$, let $i$ be a number in $1 \ldots n$. If it is not the case that every PSM in $S_i'$ monitors at least one marker, then $S_i = (s_i), \text{Mon}(s_i) = \emptyset$, and there is no $s''$ such that $s_i \xrightarrow{\emptyset, \emptyset} s''$. Then by corollary L.3.3 to the No Silent Transitions lemma, $S_i' = S_i'' = \{s_i'\}$, so $s_i' = s_i''$. Otherwise, let $\eta_i$ be a member of $\text{Mon}(s_i)$. From the results of both uses of the Split Event-Step Transitions lemma, we know that $\eta_i \in \text{Mon}(s_i')$ and $\eta_i \in \text{Mon}(s_i'')$. By corollary K.12.2 to the Distinct-Marker Preservation lemma, all of the PSMs in $S_i'$ monitor distinct markers. Therefore, there is exactly one PSM in $S_i'$ that monitors $\eta_i$, so $s_i'' = s_i'$. Then because $s_i'' = s_i'$ for all $i \in 1 \ldots n$, we have $\text{SplitSelected}(s'', H) = \text{SplitSelected}(s', H)$, and so by the definition of $\text{SplitSelected}$, $s'' = s'$. Therefore, $S'' = S'$.

Next, we must show that $\bigcup_{i \in 1 \ldots n} O_i \subseteq \bigcup_{j \in 1 \ldots p} O''_{i,j}$. We know that $O''_{i,j} = \bigcup_{i \in 1 \ldots n} O''_{i,j}$ for all $j \in 1 \ldots p$, so $\bigcup_{j \in 1 \ldots p} O''_{i,j} = \bigcup_{j \in 1 \ldots p} \left( \bigcup_{i \in 1 \ldots n} O''_{i,j} \right)$. We also know that for all $i \in 1 \ldots n$, $O_i \subseteq \bigcup_{j \in 1 \ldots p} O''_{i,j}$, so $\bigcup_{i \in 1 \ldots n} O_i \subseteq \bigcup_{j \in 1 \ldots p} \left( \bigcup_{i \in 1 \ldots n} O''_{i,j} \right)$. Therefore, $\bigcup_{i \in 1 \ldots n} O_i \subseteq \bigcup_{j \in 1 \ldots p} O''_{i,j}$.

Third, let $\langle \eta, po \rangle$ be a member of $\text{Obls}(S)$. If $\langle \eta, po \rangle \in \text{Obls}(s)$, then by the definition of $\text{SplitSelected}$ there exists some $i \in 1 \ldots n$ such that $\langle \eta, po \rangle \in \text{Obls}(s_i)$, and therefore $\langle \eta, po \rangle \in \text{Obls}(S_i)$. Otherwise, because $S = (s_1) \cup (S_2 - \{s_1\}) \cup \ldots \cup (S_n - \{s_n\})$ there exists some $i \in 1 \ldots n$ such that $\langle \eta, po \rangle \in \text{Obls}(S_i)$. Then because $S_i \xrightarrow{(L, O)} S_i', \langle \eta, po \rangle \in \text{Obls}(S_i') \cup O_i$. If $\langle \eta, po \rangle \in \text{Obls}(S_i')$, then again by the definition of $\text{SplitSelected}$ and the fact that $S' = S_i' \cup (S_1' - \{s_1\}) \cup \ldots \cup (S_n' - \{s_n\})$, $\langle \eta, po \rangle \in \text{Obls}(S')$. Otherwise, $\langle \eta, po \rangle \in O_i$, and therefore $\langle \eta, po \rangle \in \bigcup_{j \in 1 \ldots p} O_{i,j}$.

Fourth, if $n = 1$, then by the definition of $\text{SplitSelected}$, $S = S_1$ and $S' = S_1'$. Then either $\text{Mon}(s'') \neq \emptyset$ for all $s'' \in S'$, or $S = (s_1)$ and $\not\exists s''$ such that $s \xrightarrow{\emptyset, \emptyset} s''$. Otherwise, let $s''$ be a member of $S'$. If $s'' = s'$, then because $n > 1$, $H \neq \emptyset$. Therefore, $\text{Mon}(s'') \neq \emptyset$ by the definition of $\text{SplitSelected}$. If $s'' \neq s'$, then there exists some $i$ such that $s'' \in (S_i' - \{s_i'\})$. We also know that $s_i' \in S_i'$, so it cannot be the case that $|S_i'| = 1$. Therefore by corollary L.3.4 to the No Silent Transitions lemma, it is not the case that $S_i = (s_i), \text{Mon}(s_i) = \emptyset$, and $\not\exists s'''$ such that $s \xrightarrow{\emptyset, \emptyset} s'''$. Therefore by the preconditions to this lemma, $\text{Mon}(s''') \neq \emptyset$ for all $s''' \in S_i'$, so $\text{Mon}(s''') \neq \emptyset$.

We have shown each of the properties necessary for a specification summary transition, so $S \xrightarrow{L, O_1, \ldots, O_n} S'$, which completes the proof. \hfill \Box

### L.12 Split Conformance Reflection Theorem

**Theorem (Split Conformance Reflection).** Split is a conformance-reflecting transformation.

**Proof.** First, we must define $\text{TransExec}_{\text{Split}}$ and $\text{UntransExec}_{\text{Split}}$. For the former, let there be $\overline{aex} = \overline{K_1} \xrightarrow{\overline{I_1},\ldots,\overline{I_m}} \overline{s}$ and $s$ such that $\overline{K_1}$ and $s$ are well-formed, $\overline{K_1}$ is an externals-only configuration, $\text{Mon}(s) \subseteq \text{Used}(\overline{K_1})$, and no actor in $\overline{K_1}$ is handling an event. By the definition of $\text{Split}$, there exist
\[ \text{s}_1', \ldots, \text{s}_n' \text{ such that } \text{Split}(\tilde{K}_1, s) = \{ (\tilde{K}_1, \text{s}_1', \text{id}, \text{id}) \ldots, (\tilde{K}_1, \text{s}_n', \text{id}, \text{id}) \}. \]

Then define \( \text{TransExec}_{\text{split}}(\tilde{aex}, s) = \{ (\tilde{aex}, \text{s}_1', \text{id}, \text{id}) \ldots, (\tilde{aex}, \text{s}_n', \text{id}, \text{id}) \}\).

Next, to define \( \text{UntransExec}_{\text{Split}} \), let there be \( \tilde{aex} = \tilde{K}_1 \xrightarrow{\bar{i}_{1,1}, \ldots, \bar{i}_{1,m}} \ldots, s \), and \( X \) such that:

- \( \tilde{K}_1 \) and \( s \) are well-formed,
- \( \tilde{K}_1 \) is an externals-only configuration,
- \( \text{Mon}(s) \subseteq \text{Used}(\tilde{K}_1) \),
- no actor in \( \tilde{K}_1 \) is handling an event, and
- \( X \) contains a simulation for every member of \( \text{TransExec}_{\text{Split}}(\tilde{K}_1, \bar{i}_{1,1}, \ldots, \bar{i}_{1,m}, \ldots, s) \).

We must define \( \text{UntransExec}_{\text{T}}(\tilde{K}_1, \bar{i}_{1,1}, \ldots, \bar{i}_{1,m}, \ldots, s, X) \). If \( \text{len}(\tilde{aex}) = 0 \), then define \( \text{UntransExec}_{\text{T}}(\tilde{K}_1, \bar{i}_{1,1}, \ldots, \bar{i}_{1,m}, \ldots, s, X) = \{ s \} \). Otherwise, define it as described below.

By the definition of \( \text{TransExec}_{\text{Split}} \), there exist some \( \text{s}_1', \ldots, \text{s}_n' \) such that \( \text{TransExec}_{\text{split}}(\tilde{aex}, s) = \{ (\tilde{aex}, \text{s}_1', \text{id}, \text{id}) \ldots, (\tilde{aex}, \text{s}_n', \text{id}, \text{id}) \} \). For \( i \in 1 \ldots n \), let \( \text{Simulates}(\text{s}_i', 1, \text{\bar{l}_{i,1}}, \ldots, \text{\bar{l}_{i,m}}) \rightarrow \ldots = X(\tilde{aex}, \text{s}_i', \text{id}, \text{id}) \). By \( n - 1 \) uses of corollary L.5.1 to the Summary Synchronization lemma, there exists a sequence \( \text{L}_1, \ldots \) such that for all \( i \in 1 \ldots n \), \( \text{s}_i' \xrightarrow{(\text{L}_1, \text{O}_{i,1})} \ldots \) is a valid execution and

- \( \text{S}_i' \) is well-formed,
- all of the PSMs in \( \text{S}_i' \) monitor distinct markers,
- \( \text{Mon}(\text{S}_i') \subseteq \text{Used}(\tilde{K}_j) \), and
- \( \text{Mon}(\text{S}_i') \cap \text{Mon}(\text{S}_k') = \emptyset \) for all \( k \in 1 \ldots n \) such that \( i \neq k \).

The proof is by induction on \( j \). In the base case, \( j = 1 \), and we know that \( \text{S}_i' = \{ \text{s}_i \} \) for all \( i \in 1 \ldots n \). It is vacuously true that all of the PSMs in \{s_i\} monitor distinct markers. By the definition of \( \text{Split} \), we know that that \{s_i\} is well-formed, \( \text{Mon}(s_i) \subseteq \text{Used}(\tilde{K}_1) \), and \( \text{Mon}(s_i) \cap \text{Mon}(s_k) = \emptyset \) for all \( k \in 1 \ldots n \) such that \( i \neq k \).
In the inductive case, the results hold by corollary K.5.4 to the Specification Well-Formed Preservation lemma, corollary K.12.2 to the Distinct-Marker Preservation lemma, corollary K.10.1 to the Used/Monitored Marker lemma, and corollary L.4.2 to the Distinct Monitored Markers lemma.

Finally, we must show that for all $i \in 1 \ldots n$ and all $j < \text{len}(\hat{a}e\hat{x}) + 2$, either $\text{Mon}(s'') \neq \emptyset$ for all $s'' \in S_{i,j+1}'$, or there exists some $s''$ such that $S_{i,j}' = \{s''\}$, $\text{Mon}(s'') = \emptyset$, and there is no $s'''$ such that $s'' \xrightarrow{\text{L}j,a\hat{x},s,s'} s'''$. Because $\text{len}(\hat{a}e\hat{x}) > 0$, for all $i \in 1 \ldots n$ and all $j < \text{len}(\hat{a}e\hat{x}) + 2$, there exists a transition $S_{i,j}' \xrightarrow{\{L_j, O_{ij}\}} S_{i,j+1}'$.

Then by the definition of the specification summary transition relation, the condition holds.

Let $H$ be the set of independent markers of $s$. It is easy to see by the definition of $\text{Split}$ and $\text{SplitSelected}$ that $\text{SplitSelected}(s,H) = \{s'_1, \ldots, s'_n\}$. We also know that for all $i \in 1 \ldots n$, $S_{i,1}' = \{s'_i\}$. Then by repeated uses of the Split Summary Transitions lemma (using the same marker set $H$ for each use), there exists an execution $S_1 \xrightarrow{(L_i, O_i)} \ldots$ with the same length as $\hat{a}e\hat{x}$ such that

- $S_1 = \{s\}$,
- for each configuration $S_j$ in that execution, there exist $s''$ and $S''$ such that $S_j = \{s''\} \cup S''$ and $\bigcup_{i \in 1 \ldots n} S_{i,j}' = \text{SplitSelected}(s'', H) \cup S''$, and
- for each $O_j$ in the execution, $O_j = \bigcup_{i \in 1 \ldots n} O_{i,j}'$.

Assume that each next transition $S_j \xrightarrow{\{L_j, O_j\}} S_{j+1}$ is chosen deterministically based on the initial configuration $S_j$ and the $n$ many transitions $S_{i,j}' \xrightarrow{\{L_j, O_{ij}\}} S_{i,j+1}'$, $S_{i,j}' \xrightarrow{\{L_j, O_{ij}'\}} S_{i,j+1}'$ used by the Split Summary Transitions lemma. Then define $\text{UntransExecSpl}(\hat{a}e\hat{x}, s, X) = S_1 \xrightarrow{(L_i, O_i)} \ldots$.

It remains to prove the necessary properties for $\text{Split}$, $\text{TransExecSpl}$, and $\text{UntransExecSpl}$.

**Properties for Split**

Let there be $\hat{K}$ and $s$ such that $\hat{K}$ and $s$ are well-formed, $\hat{K}$ is an externals-only configuration, $\text{Mon}(s) \subseteq \text{Used}(\hat{K})$, and no actor in $\hat{K}$ is handling an event. By the definition of $\text{Split}$, there exist $s'_1, \ldots, s'_n$ such that $\text{Split}(\hat{K}, s) = \{\langle \hat{K}, s'_1, \text{id}, \text{id} \rangle, \ldots, \langle \hat{K}, s'_n, \text{id}, \text{id} \rangle\}$. The proofs for each of the properties follow.

**Well-Formed Preservation** We already know that $\hat{K}$ is well-formed. We also know that $s$ is well-formed, so by the definition of $\text{Split}$, $s'_i$ is well-formed for all $i \in 1 \ldots n$.

**Externals-Only Preservation** We already know that $\hat{K}$ is an externals-only configuration.
All-Awaiting Preservation  We already know that no actor in \(\bar{K}\) is handling an event.

Used/Monitored Preservation  For all \(i \in 1...n\), \(Mon(s'_i) \subseteq Mon(s)\) by the definition of \(Split\). Therefore, \(Mon(s'_i) \subseteq Used(\bar{K})\) for all \(i \in 1...n\).

No New Enabled Actors  The transformed program configuration is the same as the given one, so the necessarily enabled actors are the same, and \(id\) provides a one-to-one mapping between them.

Atomic Address Reflection  By the definition of the identity function as the address correspondence function.

Unique Actor Correspondences  By the definition of the identity function as the address correspondence function.

No New Single Messages  The transformed program configuration is the same as the given one, so the messages with quantity \textit{single} are the same, and \(id\) provides the mapping between them.

Internal Address Reflection  By the definition of the identity function as the address correspondence function.

Unique Approximating Messages  The transformed program configuration is the same as the given one, so the message-map component \(\hat{\mu}\) is the same in both the pre- and post-transformed configurations. Because \(\bar{K}\) is well-formed, \(\bar{\mu}\) is fully merged, so by the Message-Map Reflexivity lemma (appendix I, \(\hat{\mu} \sqsubseteq id, id \hat{\mu}\). Then the rules for \(\sqsubseteq\) provide the necessary properties.

Unique Approximated Messages  The transformed program configuration is the same as the given one, so the message-map component \(\mu\) is the same in both the pre- and post-transformed configurations. Because \(\bar{K}\) is well-formed, \(\mu\) is fully merged, so by the Message-Map Reflexivity lemma (appendix I, \(\mu \sqsubseteq id, id \hat{\mu}\). Then the rules for \(\sqsubseteq\) provide the necessary properties.

Properties for \(TransExec_{Split}\)

Let there be \(\bar{\alpha} = \bar{K}_{1} \xrightarrow{f_{1,1},...f_{1,m}} \ldots\) and \(s\) such that

- \(K_{1}\) and \(s\) are well-formed,
- \(\bar{K}_{1}\) is an externals-only configuration,
- \(Mon(s) \subseteq Used(\bar{K}_{1})\), and
L.12. SPLIT CONFORMANCE REFLECTION THEOREM

... no actor in \( \hat{K}_1 \) is handling an event.

By the definition of TransExec\(_{Split} \) above, there exist \( s'_1, \ldots, s'_n \) such that TransExec\(_{Split}(\alpha \hat{e}x, s) = \{ \langle \alpha \hat{e}x, s'_1, id, id \rangle, \ldots, \langle \alpha \hat{e}x, s'_n, id, id \rangle \} \). Proofs for each of the conformance-reflection properties for TransExec\(_{Split} \) follow.

**Initial Pair Correctness**  Trivial by the definition of TransExec\(_{Split} \).

**Fairness Preservation 1**  Trivial by the definition of TransExec\(_{Split} \).

**Fairness Preservation 2**  Let there be some \( \hat{a} \) and \( j \) such that an actor at \( \hat{a} \) is necessarily enabled in \( \hat{K}_1 \) and either the actor at \( \hat{a} \) is not necessarily enabled in a configuration \( \hat{K}_j \) in the execution, or \( \hat{K}_j \xrightarrow{\iota_{j,1}, \ldots, \iota_{j,m}} \hat{K}_{j+1} \) is a step in the execution such that \( \hat{a} \) identifies the active actor for \( \iota_{j,1} \). Then for all \( i \in 1 \ldots n \), let \( k = j \): then either the actor at \( id \hat{a} \) is not necessarily enabled in a configuration \( \hat{K}_k \) in the execution, or \( \hat{K}_k \xrightarrow{\iota_{k,1}, \ldots, \iota_{k,m}} \hat{K}_{k+1} \) is a step in the execution, and \( id(\hat{a}) \) identifies the active actor for \( \iota_{k,1} \).

**Fairness Preservation 3**  Let \( \langle \langle \hat{\beta} \mid \hat{\mu} \rangle H \rangle \) be \( \hat{K}_1 \), and let there be \( \hat{a}, H', \hat{\nu}, \hat{\vartheta} \), and \( j \) such that \( \hat{\mu}(\hat{a} \hat{\vartheta} H', \hat{\nu}) = \text{single} \) and either \( \hat{K}_j = \langle \langle \hat{\beta}' \mid \hat{\mu}' \rangle H'' \rangle \) is a configuration in the execution such that \( \langle \hat{a} @ H', \hat{\vartheta} \rangle \in \text{dom}(\hat{\mu}') \) or \( \hat{\mu}'(\hat{a} @ H', \hat{\nu}) = \text{many} \), or \( \hat{K}_j \xrightarrow{\hat{\alpha} : \text{rcv-int}(H', \hat{\vartheta})} \hat{K}_{j+1} \) is a step in the execution. For all \( i \in 1 \ldots n \), let \( \hat{\alpha}' = \hat{\mu}'(\hat{a}) \), \( H'' = \text{id}(H') \), \( \hat{\vartheta}' = \text{id}(\text{id}(\hat{\vartheta})) \), and \( k = j \): then \( \hat{K}_k' = \langle \langle \hat{\beta}'' \mid \hat{\mu}'' \rangle H''' \rangle \) is a configuration in the execution such that either \( \langle \hat{a}' @ H'''', \hat{\vartheta}' \rangle \notin \text{dom}(\hat{\mu}'') \), \( \hat{\mu}''(\hat{a}' @ H'''', \hat{\vartheta}') = \text{many} \), or \( \hat{K}_k' \xrightarrow{\hat{\alpha}' : \text{rcv-int}(H'''', \hat{\vartheta}')} \hat{K}_{k+1} \) is a step in that execution such that \( \hat{I}_{k,1} = \hat{\alpha}' : \text{rcv-int}(H'''', \hat{\vartheta}'). \)

**Properties for UntransExec\(_{Split} \)**

Let there be \( \alpha \hat{e}x = \hat{K}_1 \xrightarrow{\iota_{1,1}, \ldots, \iota_{1,m}} \ldots, s \), and \( X \) such that

- \( \hat{K}_1 \) and \( s \) are well-formed,
- \( \hat{K}_1 \) is an externals-only configuration,
- \( \text{Mon}(s) \subseteq \text{Used}(\hat{K}_1) \),
- no actor in \( \hat{K}_1 \) is handling an event, and
- \( X \) contains a simulation for all members of TransExec\(_T\)(\( \hat{K}_1 \xrightarrow{\iota_{1,1}, \ldots, \iota_{1,m}} \ldots, s \)).
By the definition of TransExecSplit above, there exist \( s'_1, \ldots, s'_n \) such that 
TransExecSplit(\( \bar{a}ex, s \)) = \{ \langle \bar{a}ex, s'_1, id, id \rangle, \ldots, \langle \bar{a}ex, s'_n, id, id \rangle \}. Also, 
SplitSelected(\( s, H \)) = \{ s'_1, \ldots, s'_n \}, where \( H \) is the set of independent markers of \( s \). By the definition of what it means to contain a simulation, for all \( i \in 1 \ldots n \), 
let \( S'_{i,1} = \ldots = X(\bar{a}ex, s'_1, id, id) \). Then by the definition of UntransExecSplit above, UntransExecSplit(\( \bar{a}ex, s, X \)) is a specification execution \( S_1 \xrightarrow{(L_1, O_1)} \ldots \) with the same length as \( \bar{a}ex \) such that

- for each configuration \( S_j \) in the execution, there exist \( s'' \) and \( S'' \) such that \( S_j = \{ s'' \} \cup S'' \) and \( \bigcup_{i=1 \ldots n} S'_{i,j} = \text{SplitSelected}(s'', H) \cup S'' \), and
- for each \( O_j \) in the execution, \( O_j = \bigcup_{i=1 \ldots n} O_{i,j} \).

The proofs for each of the properties follow.

**Execution Simulation** From the above description of \( S_1 \xrightarrow{(L_1, O_1)} \ldots \), we already know that \( S_1 = s \) and \( \text{Simulates}(S_1 \xrightarrow{(L_1, O_1)} \ldots, \bar{a}ex) \).

**Prefix Consistency** Let there be \( \hat{K}_1 \xrightarrow{\hat{L}_1, \ldots, \hat{L}_1, m} \ldots \) and \( \hat{K}_1 \xrightarrow{\hat{L}_1, \ldots, \hat{L}_1, m} \ldots \) share a prefix of length \( i \),

- \( \text{UntransExecSplit}(\hat{K}_1 \xrightarrow{\hat{L}_1, \ldots, \hat{L}_1, m} \ldots, s, X') = S''_i \xrightarrow{(L''_1, O''_1)} \ldots \), and

- for all \( \bar{a}ex, \bar{a}ex', s', A, M \), and \( j \) such that
  - \( \langle \bar{a}ex, s', A, M \rangle \in \text{TransExecSplit}(\hat{K}_1 \xrightarrow{\hat{L}_1, \ldots, \hat{L}_1, m} \ldots, s) \),
  - \( \langle \bar{a}ex', s', A, M \rangle \in \text{TransExecSplit}(\hat{K}_1 \xrightarrow{\hat{L}_1, \ldots, \hat{L}_1, m} \ldots, s) \),
  - \( \bar{a}ex \) and \( \bar{a}ex' \) share a prefix of length \( j \), and
  - \( j \leq i \),

\( X(\bar{a}ex, s', A, M) \) and \( X'(\bar{a}ex', s', A, M) \) share a prefix of length \( j \).

The proof is by induction on \( i \). If \( i = 0 \), then by the definition of UntransExecSplit, \( S_1 = S''_1 = \{ s \} \), so both executions share a prefix of length 0.

If \( i > 0 \), let \( \bar{a}ex' = \hat{K}_1 \xrightarrow{\hat{L}_1, \ldots, \hat{L}_1, m} \ldots \) and let \( S''_{i,1} \xrightarrow{(L''_{i,1}, O''_{i,1})} \ldots = X(\bar{a}ex', s'_j, id, id) \) for all \( j \in 1 \ldots n \). By the above conditions, we know that \( X(\bar{a}ex, s'_j, id, id) \) and \( X(\bar{a}ex', s'_j, id, id) \) share a prefix of length \( i \) for all \( j \in 1 \ldots n \), so \( L'_i = L''_{i,1}, O'_i = O''_{i,1}, S'_{i,1} = S''_{i,1}, \) and \( S'_{j,i+1} = S''_{j,i+1} \) for all \( j \in 1 \ldots n \). Finally, by the induction hypothesis, we also have that \( S''_i = S_i \). By the definition of UntransExecSplit, the next step is chosen deterministically based only on the above configurations and labels, so \( L_i = L''_i, O_i = O''_i \), and \( S_{i+1} = S''_{i+1} \). Therefore, both executions share a prefix of length \( i \).
Fulfillment Reflection 1  Let there be some \( \langle \eta, po \rangle \in Obls(s) \) such that for all \( i \in 1 \ldots n \), either \( \langle \eta, po \rangle \notin Obls(s) \) or there is a step \( S_i^{j} \xrightarrow{(L_i,O_i,j)} S_{i,j+1} \in S_i^{j} \xrightarrow{(L_i,O_i,k)} \ldots \) such that \( \langle \eta, po \rangle \in O_{i,j} \).

By the definition of Split, there is some \( i \in 1 \ldots n \) such that \( \langle \eta, po \rangle \in Obls(s) \), so there exists some \( O_{i,j} \) in the associated execution such that \( \langle \eta, po \rangle \in O_{i,j} \).

We know that \( O_j = \bigcup_{i \in 1 \ldots n} O_{i,j} \). Therefore there exists a step \( S_j \xrightarrow{(L_j,O_j)} S_{j+1} \) in \( S_j^{(L_j,O_j)} \xrightarrow{(L_j,O_j)} \ldots \) such that \( \langle \eta, po \rangle \in O_j \).

Fulfillment Reflection 2  Let there be some \( j \) such that for all \( k \leq j \), all \( \langle \eta, po \rangle \in Obls(S'_k) \) fulfills the obligation \( \langle \eta, po \rangle \). Then let there be some \( k' \leq j \) and some \( \langle \eta', po' \rangle \in Obls(S_{k'}) \). We must show that the execution \( S_1 \xrightarrow{(L_1,O_1)} \ldots \) fulfills the obligation \( \langle \eta', po' \rangle \).

We already know that there exist \( s'' \) and \( S'' \) such that \( S_j = \{s''\} \cup S'' \) and \( \bigcup_{i \in 1 \ldots n} S'_{i,k'} = \text{SplitSelected}(s'',H) \cup S'' \). By the definition of SplitSelected, for all \( \langle \eta'', po'' \rangle \in Obls(s'') \), \( \langle \eta'', po'' \rangle \in Obls(S''(s'',H)) \). Therefore, there exists some \( i \in 1 \ldots n \) such that \( \langle \eta', po' \rangle \in Obls(S'_{i,k}). \) By the above conditions, the execution \( S'_i \xrightarrow{(L_i,O_i,k)} \ldots \) fulfills that obligation, i.e., there is a step \( S_{i,m}^{(L_i,O_i,m)} \xrightarrow{(L_i,O_i,k)} S'_{i,m+1} \) such that \( \langle \eta', po' \rangle \in O_{i,m} \). Therefore by the definition of the execution \( S_1 \xrightarrow{(L_1,O_1)} \ldots \) above, there exists a step \( S_m^{(L_m,O_m)} \xrightarrow{(L_m,O_m)} S_{m+1} \) in that execution such that \( \langle \eta', po' \rangle \in O_m \). Therefore, that execution fulfills the obligation \( \langle \eta', po' \rangle \).

\[ \square \]

### L.13 Remap Well-Formed Preservation Lemma

**Lemma (Remap Well-Formed Preservation).** For all \( \hat{K}, d, A, \) and \( M \), if

- \( \hat{K} \) is well-formed,
- every address appearing in \( \hat{K} \) is a member of \( \text{dom}(A) \), and
- for all \( \hat{a} \in \text{dom}(A) \), if \( \hat{a} \) is external, then \( A(\hat{a}) \) is external,

then \( \text{Remap}(\hat{K}, A, M) \) is well-formed.

**Proof.** Because every address appearing in \( \hat{K} \) is a member of \( \text{dom}(A) \), \( \text{Remap}(\hat{K}, A, M) \) is defined. Let \( \langle \beta | \hat{A} | H \rangle = \hat{K} \), and let \( \langle \beta' | \hat{A}' | H' \rangle = \text{Remap}(\hat{K}, A, M) \). Let \( \hat{a} \) be an internal address appearing in \( \hat{K}' \). By the definition of \( \text{Remap} \), there exists some \( \hat{a}' \) appearing in \( \hat{K} \) such that \( A(\hat{a}') = \hat{a} \). Then
by the preconditions to this lemma, \( \hat{a}' \) is internal. Because \( \hat{K} \) is well-formed, \( \hat{a}' \in \text{dom}(\hat{\beta}) \), and therefore by the definition of \( \text{Remap} \), \( \hat{a} \in \text{dom}(\hat{\beta}') \).

Next, let \( \hat{a}@H'' \) be a marked address appearing in \( \hat{K}' \). By the definition of \( \text{Remap} \), there exists some marked address \( \hat{a}@H''' \) appearing in \( \hat{K} \) such that \( M(H'''' \cap \text{dom}(M)) = H''' \). Because \( \hat{K} \) is well-formed, \( H''' \subseteq H \). By the definition of \( \text{Remap} \), \( H' = M(H \cap \text{dom}(M)) \), so \( H''' \subseteq H' \).

Finally, by the definition of \( \text{Remap} \), there exist \( \hat{a}_1, \ldots, \hat{a}_n, H'_1, \ldots, H'_n \), and \( \hat{v}_1, \ldots, \hat{v}_n \) such that \( \hat{\mu}' = \emptyset \uplus \langle \hat{a}_1@H'_1, \hat{v}_1 \rangle \uplus \ldots \uplus \langle \hat{a}_n@H'_n, \hat{v}_n \rangle \). The empty map \( \emptyset \) is fully merged, so by induction on \( n \) and the Fully Merged Preservation lemma (appendix I), \( \hat{\mu}' \) is fully merged. \( \square \)

**L.14 Remap Externals-Only Preservation Lemma**

**Lemma** (Remap Externals-Only Preservation). For all \( \hat{K}, A, M \), if \( \hat{K} \) is an externals-only configuration and \( A \) is approximating, then \( \text{Remap}(\hat{K}, A, M) \) is an externals-only configuration.

For every marked address \( \hat{a}@H \) appearing in \( \text{Remap}(\hat{K}, A, M) \), there exists some \( \hat{a}@H' \) in \( \hat{K} \) such that \( \text{Remap}(\hat{a}@H', A, M) = \hat{a}@H \). Because \( A \) is approximating, either \( \hat{a} \) and \( \hat{a}' \) are both internal or both external. If internal, \( H' = \emptyset \) by the definition of an externals-only configuration, so by the definition of \( \text{Remap} \), \( H = \emptyset \). Otherwise, \( |H'| \leq 1 \), so by the definition of \( \text{Remap} \), \( |H| \leq 1 \).

Similarly, for every receptionist \( \langle \hat{a}@H, \tau \rangle \) appearing in \( \text{Remap}(\hat{K}, A, M) \), there exists some \( \hat{a}@H' \) in \( \hat{K} \) such that \( \text{Remap}(\hat{a}@H', A, M) = \hat{a}@H \). Then \( |H'| \leq 1 \) by the definition of an externals-only configuration, so by the definition of \( \text{Remap} \), \( |H| \leq 1 \). Therefore, \( \text{Remap}(\hat{K}, A, M) \) is an externals-only configuration.

**L.15 Abstract Single-Handler Preservation Lemma**

**Lemma** (Abstract Single-Handler Preservation). For all \( \hat{K}, \hat{I}, \text{ and } \hat{K}' \), if \( \hat{K} \) is a single-handler configuration and \( \hat{K} \xrightarrow{\hat{I}}_{RA} \hat{K}' \), then \( \hat{K}' \) is a single-handler configuration.

**Proof.** The transition \( \xrightarrow{\hat{I}}_{RA} \) is defined such that \( \hat{K} \xrightarrow{\hat{I}}_{RA} \hat{K}' \) only if \( \hat{K} \xrightarrow{\hat{I}} \hat{K}' \).

Therefore, the proof proceeds by cases on the rule enabling the transition \( \hat{K} \xrightarrow{\hat{I}} \hat{K}' \).

**A-RECEIVEINTERNAL Case** For this rule, \( \hat{I} \) must be a handler-start label (see section 2.5.4) of the form \( \hat{a}: rcv-int(H, \hat{\beta}) \). The transition relation \( \xrightarrow{\tau_{SH}} \) allows a transition with a handler-start label only if no actor in \( \hat{K} \) is handling an event, and an A-RECEIVEINTERNAL transition results in just one additional actor handling an event (with just one behavior handling the event). Therefore, \( \hat{K}' \) is a single-handler configuration.
A-RECEIVEEXTERNAL Case  Same as the A-RECEIVEINTERNAL Case, with a handler-start label of the form \( \hat{a} : rcv-ext(H, \hat{v}, \tau) \).

A-TIMEOUT Case  Same as the A-RECEIVEINTERNAL Case, with a handler-start label of the form \( \hat{a} : \text{timeout} \).

A-SPAWN Case  For this case, \( \hat{l} \) must be a handler-continuation label of the form \( \hat{a} : \text{spawn}(\hat{a'}) \). The actor at \( \hat{a} \) in \( \hat{K} \) must have exactly one behavior handling a message, and by the definition of this rule, the actor at \( \hat{a} \) in \( \hat{K}' \) has at most one behavior handling a message. The spawned actor at \( \hat{a'} \) must have a different location than \( \hat{a} \), because the spawn expression for one must be lexically nested inside the spawn expression for the other, so there is no other actor with location \( \text{AddrLoc}(\hat{a}) \) handling an event in \( \hat{K}' \).

We must also show that there is no other actor with location \( \text{AddrLoc}(\hat{a'}) \) currently handling an event in \( \hat{K} \), which would imply there is at most one actor (and one behavior) with that location handling an event in \( \hat{K}' \). We will prove this by contradiction.

Assume that such an actor does exist, at some address \( \hat{a}'' \) such that \( \hat{a} < \hat{a}'' \) and \( \text{AddrLoc}(\hat{a}) = \text{AddrLoc}(\hat{a'}) \). Because the actor at \( a \) is spawning an actor at \( a' \), it must be the case that the spawn expression at location \( \ell' \) must be lexically nested inside the spawn expression at location \( \ell \). Therefore, \( \ell < \ell' \).

By the definition of \( < \) for addresses, \( \hat{a} < \hat{a}'' \). The transition relation \( \rightarrow_{\text{RA}} \) allows a transition with a handler-continuation label only if there is no \( \hat{a}'' \) such that \( \hat{a} < \hat{a}'' \) and the actor at \( \hat{a}'' \) in \( \hat{K} \) is handling an event. The transition \( \hat{K} \xrightarrow{\hat{l}}_{\text{RA}} \hat{K}' \) is possible though, so this is a contradiction. Therefore there is no actor spawned from \( \ell' \) currently handling an event in \( \hat{K} \).

An A-SPAWN transition does not otherwise change which actors are handling events, so \( \hat{K}' \) is a single-handler configuration.

Other Cases  None of the other transition rules add to the number of actors or behaviors handling an event, so \( \hat{K}' \) is a single-handler configuration in each case.

\[ L.16 \quad \text{Abstract Extra Markers Lemma} \]

**Lemma** (Abstract Extra Markers). For all \( \hat{v}, \hat{v}', H, H', \hat{v}'', A, \) and \( M \), if

- \( \text{dom}(M) \subseteq H \),
- \( \text{absv} \subseteq_{A,M} \hat{v}'', \) and
- \( \text{Mark}(\hat{v}, H) = (\hat{v}', H') \),

then \( \text{absv}' \subseteq_{A,M} \hat{v}'' \).

**Proof.** Similar to the proof of the Extra Markers lemma (appendix I).
L.17 Abstract Marker Soundness Lemma

Lemma (Abstract Marker Soundness). For all \( \vec{v}_L, \vec{v}_L', \vec{v}_G, H_L, H_L', H_G, A, \) and \( M \) such that

- \( \text{Mark}(\vec{v}_L, H_L) = \langle \vec{v}_L', H_L \cup H_L' \rangle \),
- \( \vec{v}_L \subseteq_{A,M} \vec{v}_G \),
- \( H_L \subseteq M H_G \),
- \( H_L \) includes every marker appearing in \( \vec{v}_L \),
- \( M \) is one-to-one, and
- \( \text{dom}(M) \subseteq H_L \),

there exist \( \vec{v}_G', H_G' \), and \( M' \) such that

- \( \text{Mark}(\vec{v}_G, H_G) = \langle \vec{v}_G', H_G \cup H_G' \rangle \),
- \( \vec{v}_L' \subseteq_{A,M} \vec{v}_G' \),
- \( H_L \cup H_L' \subseteq M \cup M' H_G \cup H_G' \),
- \( M \cup M' \) is one-to-one, and
- \( \text{dom}(M') \subseteq H_L' \).

Proof. Similar to the proof of the Marker Soundness lemma (appendix I). \( \square \)

L.18 Approximation Substitution Lemma

Lemma (Approximation Substitution). For all \( \vec{e}_L, \vec{e}_G, \vec{e}_L', \vec{e}_G' \) \( x \in A \), and \( M \), if

- \( \vec{e}_L \subseteq_{A,M} \vec{e}_G \),
- \( \vec{e}_L' \subseteq_{A,M} \vec{e}_G' \), and
- either there exists \( \vec{E} \) such that \( \vec{e}_L = \vec{E}[x] \) or there exists \( \vec{v} \) such that \( \vec{e}_L' = \vec{v} \),

then \( \vec{e}_L[x \leftarrow \vec{e}_L'] \subseteq_{A,M} \vec{e}_G[x \leftarrow \vec{e}_G'] \).

Proof. The proof is by structural induction on \( \vec{e}_L \). The case for a list is shown below. The case for a dictionary is similar, and the remaining cases are straightforward.
**Case:** $\hat{e}_L = \langle \text{list } \{\hat{o}v_{L,1}, \ldots, \hat{o}v_{L,n}\}\rangle$

A list expression cannot be a context $\hat{E}$, so there must exist $\hat{v}_L$ such that $\hat{e}'_L = \hat{v}_L$, and by the Value Approximation lemma there exists some $\hat{v}_G$ such that $\hat{e}'_G = \hat{v}_G$. By the definition of $\subseteq$, there exist $\hat{o}v_{G,1}, \ldots, \hat{o}v_{G,m}$ such that

- $\hat{e}_G = \langle \text{list } \{\hat{o}v_{G,1}, \ldots, \hat{o}v_{G,m}\}\rangle$,
- for all $i \in 1 \ldots n$, there exists $j \in 1 \ldots m$ such that $\hat{o}v_{L,i} \subseteq_{A,M} \hat{o}v_{G,j}$, and
- every free variable occurring in $\{\hat{o}v_{G,1}, \ldots, \hat{o}v_{G,m}\}$ occurs in $\{\hat{o}v_{L,1}, \ldots, \hat{o}v_{L,n}\}$.

Then by the induction hypothesis, for all $i \in 1 \ldots n$ there exists $j \in 1 \ldots m$ such that $\hat{o}v_{L,i}[x \leftarrow \hat{v}_L] \subseteq_{A,M} \hat{o}v_{G,j}[x \leftarrow \hat{v}_G]$.

By the definition of substitution, we have

- $\hat{e}_L[x \leftarrow \hat{v}_L] = \langle \text{list } \bigcup_{i \in 1 \ldots n} \{\hat{o}v_{L,i}[x \leftarrow \hat{v}_L]\}\rangle$ and
- $\hat{e}_G[x \leftarrow \hat{v}_G] = \langle \text{list } \bigcup_{j \in 1 \ldots m} \{\hat{o}v_{L,j}[x \leftarrow \hat{v}_G]\}\rangle$.

Because we are only substituting values into the expressions, $\hat{e}_L[x \leftarrow \hat{v}_L]$ and $\hat{e}_G[x \leftarrow \hat{v}_G]$ contain only open values, so they are each valid list expressions. Because we substitute for the same variable $x$ in each expression, every free variable occurring in $\bigcup_{j \in 1 \ldots m} \{\hat{o}v_{L,j}[x \leftarrow \hat{v}_G]\}$ occurs in $\bigcup_{i \in 1 \ldots n} \{\hat{o}v_{L,i}[x \leftarrow \hat{v}_L]\}$. Therefore, by the definition of $\subseteq$ for lists, we have $\hat{e}_L[x \leftarrow \hat{v}_L] \subseteq_{A,M} \hat{e}_G[x \leftarrow \hat{v}_G]$.

**Corollary L.18.1.** For all $\hat{E}_L$, $\hat{e}_L$, $\hat{E}_G$, $\hat{e}_G$, $A$, and $M$, if $\hat{E}_L \subseteq_{A,M} \hat{E}_G$ and $\hat{e}_L \subseteq_{A,M} \hat{e}_G$, then $\hat{E}_L[\hat{e}_L] \subseteq_{A,M} \hat{E}_G[\hat{e}_G]$.

**Proof.** Convert $\hat{E}_L$ and $\hat{E}_G$ into expressions $\hat{e}'_L$ and $\hat{e}'_G$ by replacing the hole in each context with a variable $x$ that does not appear in either $\hat{E}_L$, $\hat{E}_G$, $\hat{e}_L$, or $\hat{e}_G$. Then we have that $\hat{E}_L[\hat{e}'_L] = \hat{e}'_L[x \leftarrow \hat{e}'_L]$, $\hat{E}_G[\hat{e}'_G] = \hat{e}'_G[x \leftarrow \hat{e}'_G]$, and $\hat{e}'_L \subseteq_{A,M} \hat{e}'_G$. Furthermore, a context $\hat{E}$ cannot have a hole inside a list or dict expression, so $\hat{e}'_L[x \leftarrow \hat{e}'_L]$ and $\hat{e}'_G[x \leftarrow \hat{e}'_G]$ are valid expressions. Then by the Approximation Substitution lemma, $\hat{e}'_L[x \leftarrow \hat{e}'_L] \subseteq_{A,M} \hat{e}'_G[x \leftarrow \hat{e}'_G]$, so $\hat{E}_L[\hat{e}_L] \subseteq_{A,M} \hat{E}_G[\hat{e}_G]$.

**L.19 Abstract Internal Address Types Lemma**

**Lemma (Abstract Internal Address Types).** For all $\hat{v}_L$, $\hat{v}_G$, $\hat{\rho}_L$, $\hat{\rho}_G$, $A$, and $M$, if

- $\hat{v}_L \subseteq_{A,M} \hat{v}_G$.
- $\text{IntAddrTypes}(\hat{v}_L, \tau) = \hat{\rho}_L$, and
- $\text{IntAddrTypes}(\hat{v}_G, \tau) = \hat{\rho}_G$,

then $\hat{\rho}_L \subseteq_{A,M} \hat{\rho}_G$.

**Proof.** Straightforward structural induction on $\hat{v}_L$. 

L.20 Abstract Functional-Step Soundness Lemma

**Lemma** (Abstract Functional-Step Soundness). For all \( \hat{e}_L, \hat{e}_L', \hat{e}_G, A, \) and \( M, \) if \( \hat{e}_L \subseteq_{A,M} \hat{e}_G \) and \( \hat{e}_L \rightarrow \hat{e}_L', \) then there exists \( \hat{e}_G' \) such that \( \hat{e}_G \rightarrow \hat{e}_G' \) and \( \hat{e}_L' \subseteq_{A,M} \hat{e}_G'. \)

**Proof.** Straightforward case analysis based on the rule enabling the step \( \hat{e}_L \rightarrow \hat{e}_L'. \) The function \( EvalAbsPrimop \) that evaluates primitive operations on abstract values is defined such that for all \( o, \) \( \hat{v}_{\hat{L},1}, \ldots, \hat{v}_{\hat{L},n}, \hat{v}_{\hat{G},1}, \ldots, \hat{v}_{\hat{G},n}, A, \) and \( M, \) if \( \| \hat{v}_{\hat{L},i} \| \subseteq_{A,M} \hat{v}_{\hat{G},i} \) for all \( i \in 1 \ldots n \) and \( EvalPrimop(o, \hat{v}_{\hat{L},1}, \ldots, \hat{v}_{\hat{L},n}) = \hat{v}_{\hat{G}}', \) there exists \( \hat{v}_{\hat{G}}'' \in EvalAbsPrimop(o, \hat{v}_{\hat{G},1}, \ldots, \hat{v}_{\hat{G},n}) \) such that \( \hat{v}_{\hat{L}}' \subseteq_{A,M} \hat{v}_{\hat{G}}'' \). \( \square \)

L.21 Approximation Soundness Lemma

Very similar to the Soundness of Abstract CSA lemma in appendix I, this lemma says that if a configuration \( \hat{K}_G \) approximates another configuration \( \hat{K}_L, \) then \( \hat{K}_G \) can take any (restricted) transition that \( \hat{K}_L \) can take. As in the Soundness of Abstract CSA lemma, the correspondence functions \( A \) and \( M \) must only map addresses and markers appearing in the approximated configuration.

**Lemma** (Approximation Soundness). For all \( \hat{K}_L, \hat{K}_L', \hat{K}_G, \hat{I}_L, A, \) and \( M, \) if

- \( \hat{K}_L \) is a well-formed, externals-only, single-handler configuration,
- \( \hat{K}_L \subseteq_{A,M} \hat{K}_G, \)
- \( \hat{K}_L \xrightarrow{RA} \hat{K}_L', \)
- \( \text{dom}(A) = \text{dom}(\hat{\beta}_L) \cup \text{ExtAbsAddr}, \) where \( \hat{\beta}_L \) is the actor-behavior map for \( \hat{K}_L, \)
- \( \text{dom}(M) \subseteq \text{Used}(\hat{K}_L), \) and
- \( \text{for all } \hat{a} \in \text{ExtAddr}, A(\hat{a}) = \hat{a}, \)

then there exist \( \hat{I}_G, \hat{K}_G', A', \) and \( M' \) such that

- \( \hat{K}_G \xrightarrow{RA} \hat{K}_G', \)
- \( \hat{I}_L \subseteq_{A \oplus A', M \oplus M'} \hat{I}_G, \)
- \( \hat{K}_L' \subseteq_{A \oplus A', M \oplus M'} \hat{K}_G', \)
- \( \text{dom}(A') = \text{dom}(\hat{\beta}_L') - \text{dom}(\hat{\beta}_L), \) where \( \hat{\beta}_L \) and \( \hat{\beta}_L' \) are the actor-behavior maps for \( \hat{K}_L \) and \( \hat{K}_L', \) respectively, and
- \( \text{dom}(M') \subseteq \text{Used}(\hat{K}_L') - \text{Used}(\hat{K}_L). \)
Proof. The step $\hat{K}_L \xrightarrow{\hat{i}_L} \hat{K}'_L$ is possible only if $\hat{K}_L \xrightarrow{\hat{i}_L} \hat{K}'_L$, so the proof proceeds by case analysis on the transition-relation rule that enables that transition. Each of the below cases is divided into sections that show

1. the transition $\hat{K}_L \xrightarrow{\hat{i}_L} \hat{K}'_L$ can be matched by a transition $\hat{K}_G \xrightarrow{\hat{i}_G} \hat{K}'_G$ with the same transition rule,

2. there exist $A'$ and $M'$ that have the required properties and relate the transition labels $\hat{i}_L$ and $\hat{i}_G$ and the reached configurations $\hat{K}'_L$ and $\hat{K}'_G$, and

3. the transition from the first step is also a valid transition of the restricted abstract-transition relation $\rightarrow_{RA}$.

Case: A-RECEIVEEXTERNAL

By the definition of this rule, there exist $\hat{\beta}_L$, $\hat{\mu}_L$, $H_L$, $H'_L$, $\hat{\rho}_L$ $\hat{\alpha}_L$, $\hat{\nu}_L$, $\tau$, $Q_{L,1}, \ldots, Q_{L,n}$, $x$, $\hat{e}_L$, $\hat{f}_L$, and $B_L$ such that

- $\hat{K}_L = \left\langle \hat{\beta}_L \mid \hat{\mu}_L \mid H_L \right\rangle^{\hat{\beta}_L}$,

- $\hat{i}_L = \hat{\alpha}_L : \text{rcv-ext}(H''_L, \hat{\nu}_L, \tau)$,

- $\langle Q_{L,1} \ldots Q_{L,n}, (\text{receive } x \hat{e}_L \hat{f}_L) \rangle \in \hat{\beta}_L(\hat{\alpha}_L)$

- $\langle \hat{\alpha}_L @ H''_L, \tau \rangle$ is a receptionist on $\hat{K}_L$,

- $\varnothing, \varnothing \vdash \hat{v} : \tau$,

- $\langle \hat{v}_L, H'_L \rangle \in \text{Markings}(\hat{v}, H_L)$,

- for every marked external address $\hat{\alpha}'_L @ H'''_L$ in $\hat{v}$, $H'''_L = \varnothing$, and

- $\hat{K}'_L = \left\langle \hat{\beta}_L \oplus [\hat{\alpha}_L \rightarrow (\hat{Q}_{L,1} \ldots \hat{Q}_{L,n}, \hat{e}_L[x \leftarrow \hat{v}_L])] \mid \hat{\mu}_G \mid H'_L \right\rangle^{\hat{\rho}_G}$.

Additionally, the restrictions from the transition relation $\rightarrow_{RA}$ enforce that

- every address in $\hat{v}$ is external,

- $\hat{v} \in \text{MaxVals}(\tau)$,

- no actor in $\hat{K}_L$ is handling an event, and

- $\langle \hat{v}_L, H'_L \rangle = \text{Mark}(\hat{v}, H_L)$.
A-RECEIVEEXTERNAL Transition

We focus first on showing that \( \hat{K}_G \) can take a similar step with the rule A-RECEIVEEXTERNAL. First, by the definitions of \( \sqsubseteq \), there exist \( \bar{a}_G, H''_G, \hat{Q}_{G,1}, \ldots, \hat{Q}_{G,n}, \hat{e}_G \), and \( \hat{t}_G \) such that

- \( \bar{a}_L \sqsubseteq_A \bar{a}_G, H_L \sqsubseteq_A H_G, H''_L \sqsubseteq_A H''_G, \hat{Q}_{L,i} \sqsubseteq_A M \hat{Q}_{G,i} \) for all \( i \in 1 \ldots n \), \( \hat{e}_L \sqsubseteq_A M \hat{e}_G \)
- \( \langle \hat{Q}_{G,1} \ldots \hat{Q}_{G,n}, (\text{receive } \bar{e}_G \hat{t}_G) \rangle \in \hat{\beta}_G(\bar{a}_G) \)
- \( \langle \bar{a}_G \hat{t}_G \rangle \in \hat{\beta}_G \).

By the definition of MaxVals, \( \bar{v} \) contains no internal addresses and no markers. By the preconditions to this lemma, \( A(\bar{a}') = \bar{a}' \) for all abstract external addresses \( \bar{a}' \), so by the rules for \( \sqsubseteq \) it is easy to see that \( \bar{v} \sqsubseteq_A M \bar{v} \). Also because \( \bar{v} \) contains no internal addresses, the IntAddrTypes side-condition for A-RECEIVEEXTERNAL is vacuously true. We already know that \( \varnothing \sqsubseteq \bar{v} \sqsubseteq \bar{v} : \tau \).

No markers appear in \( \bar{v} \) by the definition of MaxVals. We already know that \( H_L \sqsubseteq_A H_G \) and \( \bar{v} \sqsubseteq_A M \bar{v} \), and \( M \) is one-to-one by the definition of \( \hat{K}_L \sqsubseteq_A M \hat{K}_G \). Then by the Abstract Marker Soundness lemma there exist \( \bar{v}_G, H''_G \), and \( M' \) such that

- \( \text{Mark}(\bar{v}, H_G) = \langle \bar{v}_G, H''_G \rangle \)
- \( \bar{v}_L \sqsubseteq_A M \sqsubseteq M' \bar{v}_G \)
- \( H'_L \sqsubseteq_A M' \sqsubseteq H'_G \)
- \( M \sqsubseteq M' \) is one-to-one, and
- \( \text{dom}(M') \subseteq H'_L - H_L \).

By the Deterministic Marking lemma, we also have that \( \langle \bar{v}_G, H''_G \rangle \in \text{Markings}(\bar{v}, H_G) \). As a result, there exist \( \hat{K}_G \) and \( \hat{I}_G = [a_G : \text{rcv-ext}(H''_G, \bar{v}_G, \tau)] \) (\text{receive } \bar{v}_G \tau) such that \( \hat{K}_G \xrightarrow{\hat{t}_G} \hat{K}'_G \) and \( \hat{K}'_G = \langle \hat{\beta}_G \hat{\beta}_G[\bar{a}_G \bar{e}_G \bar{t}_G G(x - \bar{v}_G)] \rangle \hat{\beta}_G[H'_G] \). 

Approximating Configuration

We must also show that \( \hat{I}_L \sqsubseteq_A M \sqsubseteq M' \hat{I}_G \) and \( \hat{K}'_L \sqsubseteq_A M \sqsubseteq M' \hat{K}'_G \). For the former, we have already shown the approximation for each of the components of the labels, so we have \( \hat{I}_L \sqsubseteq_A M \sqsubseteq M' \hat{I}_G \). For the latter, let \( \hat{b}_L = \langle \hat{Q}_{L,1} \ldots \hat{Q}_{L,n}, \hat{e}_L[x \leftarrow \bar{v}_L] \rangle \) and let \( \hat{b}_G = \langle \hat{Q}_{G,1} \ldots \hat{Q}_{G,n}, \hat{e}_G[x \leftarrow \bar{v}_G] \rangle \). We have already shown that \( H'_L \sqsubseteq_A M' \sqsubseteq H'_G \), so it remains to show that \( \hat{\beta}_L \hat{\beta}_G[\bar{a}_L \bar{e}_L \hat{t}_L \hat{t}_G \bar{v}_L \bar{v}_G] = [\hat{a}_L \leftarrow \hat{b}_L] \sqsubseteq_A M' \sqsubseteq [\hat{a}_G \leftarrow \hat{b}_G] \).

We know from \( \hat{K}_L \sqsubseteq_A M \hat{K}_G \) that \( \hat{\beta}_L \sqsubseteq_A M \hat{\beta}_G \). The configuration \( \hat{K}_L \) is well-formed, so every marker in \( \hat{K}_L \) appears in \( H'_L \), and therefore \( \hat{\beta}_L \sqsubseteq_A M \hat{\beta}_G \).
We also know that $\hat{e}_L \sqsubseteq_{A,M \sqsubseteq M'} \hat{e}_G$ and $\hat{v}_L \sqsubseteq_{A,M \sqsubseteq M'} \hat{v}_G$, so by the Approximation Substitution lemma, we have $\hat{e}_L[x \leftarrow \hat{v}_L] \sqsubseteq_{A,M \sqsubseteq M'} \hat{e}[x \leftarrow \hat{v}]$. Therefore, $\hat{b}_L \sqsubseteq_{A,M \sqsubseteq M'} \hat{b}_G$, and we already know that $\hat{a}_L \sqsubseteq_A \hat{a}_G$. We also know that $\hat{a}_L \in \text{dom}(\hat{b}_L)$, and because $\hat{b}_L \sqsubseteq_{A,M \sqsubseteq M'} \hat{b}_G$, no other behavior in $\hat{b}_G(\hat{a})$ is handling an event. Then by the New Behavior lemma (appendix I), we have $\hat{b}_L \oplus [\hat{a}_L \leftarrow \hat{b}_L] \sqsubseteq_{A,M \sqsubseteq M'} \hat{b}_G \oplus [\hat{a}_G \leftarrow \hat{b}_G]$.

Let $A' = \emptyset$. We have $\text{dom}(\hat{K}_L^{'}) = \text{dom}(\hat{K}_L)$, so $\text{dom}(A') = \text{dom}(\hat{K}_L^{'}) - \text{dom}(\hat{K}_L)$.

**Restricted Transition**

Having shown that $\hat{K}_G \xrightarrow{I_G} \hat{K}_G'$, we must also show that $\hat{K}_G \xrightarrow{I_G,\text{RA}} \hat{K}_G'$. That is, we must show that the restrictions imposed by the relation $\longrightarrow_{\text{RA}}$ do not prevent the transition. We argue for each of the conditions of $\longrightarrow_{\text{RA}}$ below.

1. We have already shown that $\hat{K}_G \xrightarrow{I_G} \hat{K}_G'$.

2. $I_G$ is not a handler-continuation label.

3. By the definition of $\sqsubseteq$, if an actor in $\hat{K}_G$ is handling an event, then there must be an actor in $\hat{K}_L$ also handling an event. But by the definition of $\longrightarrow_{\text{RA}}$, there is no such actor in $\hat{K}_L$. Therefore, there is no actor handling an event in $\hat{K}_G$.

4. $I_G$ is not a spawn label.

5. We have already shown that $\text{Mark}(\hat{v}_G,H_G) = \langle \hat{v}_G,H_G' \rangle$ and $\hat{v} \in \text{MaxVals}(r)$.

6. $I_G$ is not a send-ext label.

**Case: A-ReceiveInternal**

This case is similar to the previous one. The proof here focuses on the differences.

By the definition of this rule, there exist $\hat{b}_L$, $\hat{b}_L$, $H_L$, $\hat{b}_L$, $a$, $\hat{Q}_{L,1}, \ldots, \hat{Q}_{L,n}$, $x$, $\hat{e}_L$, $\hat{f}_L$, $H'_L$, $\hat{v}_L$, $\hat{v}'_L$, $H''_L$, and $\hat{b}_L$ such that

- $\hat{K}_L = \langle \hat{b}_L \mid \hat{b}_L \mid H_L \rangle$.
- $\langle \hat{Q}_{L,1}, \ldots, \hat{Q}_{L,n}, \text{(receive } x \hat{e}_L \hat{f}_L) \rangle \in \hat{b}(\hat{a}_L)$.
- $\langle \hat{a}_L \oplus \hat{H}_L', \hat{v}_L \rangle \in \text{dom}(\hat{b}_L)$.
- $\langle \hat{v}_L', H''_L \rangle \in \text{Markings}(\hat{v}_L,H_L)$ if $H_L' \neq \emptyset$, else $\langle \hat{v}_L', H''_L \rangle = \langle \hat{v}_L, H_L \rangle$.
- $\hat{I}_L = \hat{a}_L \:: \text{rcv-int}(H'_L, \hat{v}'_L)$.
- $\hat{b}_L = \langle \hat{Q}_{L,1}, \ldots, \hat{Q}_{L,n}, \hat{e}_L[x \leftarrow \hat{v}_L] \rangle$, and
\[K'_L = \left\langle \left( \beta_L \uplus [a] \rightarrow \phi \right) \left| \mu_L \uplus (a \uplus H'_L \uplus \delta_L) \right| H'_L \right\rangle \]

Because \(K'_L\) is an externals-only configuration, we also know that \(H'_L = \emptyset\), so \(\delta'_L = \emptyset\) and \(H''_L = H_L\).

**A-ReceiveInternal Transition**

There must exist \(\beta_G, \mu_G, H_G\), and \(\hat{\beta}_G\) such that \(\hat{K}_G = \left\langle \left( \beta_G \left| \mu_G \right| H_G \right) \right\rangle ^{\hat{\beta}_G}\). Similar to the previous case, by the definition of \(\subseteq\), there exist \(\hat{a}_G, \hat{Q}_{G,1}, \ldots, \hat{Q}_{G,n}, \hat{e}_G\), and \(\hat{\iota}_G\) such that

- \(\hat{a}_L \subseteq_L \hat{a}_G, H_L \subseteq_M H_G, \hat{Q}_{L,i} \subseteq_{A,M} \hat{Q}_{G,i}\) for all \(i \in 1 \ldots n\), \(\hat{a}_L \subseteq_{A,M} \hat{e}_G\), and
- \(\langle \hat{Q}_{G,1} \ldots \hat{Q}_{G,n}, (\text{receive } x \hat{e}_G \hat{\iota}_G) \rangle \in \beta_G(\hat{a}_G)\).

Also, by the definition of \(\subseteq\), \(\hat{a}_G\) is the only address such that \(\hat{a}_L \subseteq_L \hat{a}_G\).

Because \(\hat{\mu}_L \subseteq_{A,M} \hat{\mu}_G\), there exists a unique \(\hat{\nu}_G\) such that

- \(\langle \hat{\beta}_G \uplus \emptyset, \hat{\nu}_G \rangle \in \text{dom}(\hat{\mu}_G)\),
- \(\hat{\nu}_L \subseteq_{A,M} \hat{\nu}_G\), and
- \(\hat{\mu}_L(\hat{a}_G \uplus \emptyset, \hat{\nu}_L) \subseteq \hat{\mu}_G(\hat{a}_G \uplus \emptyset, \hat{\nu}_G)\).

Therefore, there exist \(\hat{\iota}_G, \hat{\delta}_G\), and \(\hat{K}'_G\) such that

- \(\hat{K}_G \xrightarrow{\hat{\iota}_G} \hat{K}'_G\),
- \(\hat{\iota}_G = \hat{a}_G : \text{rcv-int}(\emptyset, \hat{\nu}_G)\),
- \(\hat{\delta}_G = \langle \hat{Q}_{G,1} \ldots \hat{Q}_{G,n}, \hat{e}_G(x - \hat{\nu}_G) \rangle\), and
- \(\hat{K}'_G = \left\langle \left( \hat{\beta}_G \uplus [\hat{a}_G \rightarrow \hat{\delta}_G] \left| \hat{\mu}_G \uplus (\hat{a}_G \uplus \hat{\nu}_G) \right| H_G \right) \right\rangle ^{\hat{\beta}_G}\).

**Approximating Configuration**

Let \(A' = \emptyset\) and \(M' = \emptyset\). The transition \(\hat{K}_L \xrightarrow{\hat{\iota}_L} \hat{K}'_L\) adds no new actors or markers, so \(\text{dom}(A')\) contains the addresses of all new actors, and \(\text{dom}(M')\) contains all new markers.

It remains to show that \(\hat{\iota}_L \subseteq_{A,M} \hat{\iota}_G\) and \(\hat{K}_L \subseteq_{A,M} \hat{K}'_L\). The argument is largely similar to the argument in the previous case, using the Approximation Substitution and New Behavior lemmas to show a correspondence between the transitioned actors. It remains to show that \(\hat{\mu}_L \uplus (\hat{a}_L \uplus \emptyset, \hat{\nu}_L) \subseteq_{A,M} \hat{\mu}_G \uplus (\hat{a}_G \uplus \emptyset, \hat{\nu}_G)\).

Let \(\hat{\mu}'_L = \hat{\mu}_L \uplus (\hat{a}_L \uplus \emptyset, \hat{\nu}_L)\) and \(\hat{\mu}'_G = \hat{\mu}_G \uplus (\hat{a}_G \uplus \emptyset, \hat{\nu}_G)\). If \(\hat{\mu}_L(\hat{a}_L \uplus \emptyset, \hat{\nu}_L) = \text{many}\), then \(\hat{\mu}_G(\hat{a}_G \uplus \emptyset, \hat{\nu}_G) = \text{many}\), \(\hat{\mu}'_L = \hat{\mu}_L\), and \(\hat{\mu}'_G = \hat{\mu}_G\). We already know \(\hat{\mu}_L \subseteq_{A,M} \hat{\mu}_G\), so we’re done.

Otherwise, \(\hat{\mu}_L(\hat{a}_L \uplus \emptyset, \hat{\nu}_L) = \text{single}\). In that case, first let \(\langle \hat{a}_L \uplus \emptyset, \hat{\nu}_L \rangle\) be a member of \(\text{dom}(\hat{\mu}'_L)\) (we know there are no markers on the address because \(\hat{K}\) is an
L.21. APPROXIMATION SOUNDNESS LEMMA

External-only configuration and dom($\mu'_L$) $\subseteq$ dom($\mu_L$). We must show there is a unique $\langle a'_G @ \emptyset, \vec{v}'_L \rangle \in$ dom($\mu'_L$) such that

- $a'_L @ \emptyset \subseteq_{A,M} a'_G @ \emptyset$,
- $\vec{v}'_L \subseteq_{A,M} \vec{v}'_G$,
- $\mu'_L (a'_L @ \emptyset, \vec{v}'_L) \subseteq \mu'_G (a'_G @ \emptyset, \vec{v}'_G)$.

By the definition of $\emptyset$, it must be the case that $\langle a'_L @ \emptyset, \vec{v}'_L \rangle \in$ dom($\mu_L$) and $\langle a'_L @ \emptyset, \vec{v}'_L \rangle \neq \langle a_L @ \emptyset, \vec{v}_L \rangle$. Then because $\mu_L \subseteq_{A,M} \mu_G$, we know that there exists such a $\langle a'_G @ \emptyset, \vec{v}'_G \rangle$ in dom($\mu_G$). If $\mu_G (a_G @ \emptyset, \vec{v}_G) = \text{many}$, then $\mu'_G = \mu_G$, and therefore $\langle a'_G @ \emptyset, \vec{v}'_G \rangle$ is a unique such element in dom($\mu'_G$). Otherwise, $\mu_G (a_G @ \emptyset, \vec{v}_G) = \text{single}$, and then because $\mu_L \subseteq_{A,M} \mu_G$, $\langle a_L @ \emptyset, \vec{v}_L \rangle$ is the only message from $\mu_L$ that $(a_G @ \emptyset, \vec{v}_G)$ approximates. Therefore, $\langle a'_G @ \emptyset, \vec{v}'_G \rangle \in$ dom($\mu'_G$), and because no other messages were added to $\mu'_G$, that message is unique.

Second, let $\langle a'_G @ \emptyset, \vec{v}'_G \rangle$ be a member of dom($\mu'_G$) such that $\mu'_G (a'_G @ \emptyset, \vec{v}'_G) = \text{single}$. There are no markers on the address because $\langle a'_G @ \emptyset, \vec{v}'_G \rangle$ must be a member of dom($\mu_G$), and because $\mu_L \subseteq_{A,M} \mu_G$, the message approximates some message in $\mu_L$, which has no markers on its address because $K$ is an external-only configuration. We must show there is a unique $\langle a'_L @ \emptyset, \vec{v}'_L \rangle \in$ dom($\mu'_L$) such that

- $a'_L @ \emptyset \subseteq_{A,M} a'_G @ \emptyset$,
- $\vec{v}'_L \subseteq_{A,M} \vec{v}'_G$,
- $\mu'_L (a'_L @ \emptyset, \vec{v}'_L) = \text{single}$.

By the definition of $\emptyset$, it must be the case that $\langle a'_G @ \emptyset, \vec{v}'_G \rangle \in$ dom($\mu_G$) and $\langle a'_G @ \emptyset, \vec{v}'_G \rangle \neq \langle a_L @ \emptyset, \vec{v}_L \rangle$. Because $\mu_L \subseteq_{A,M} \mu_G$, we know that there exists such a $\langle a'_G @ \emptyset, \vec{v}'_G \rangle$ in dom($\mu_L$), and that $(a_L @ \emptyset, \vec{v}_L)$ is the only message from $\mu_L$ that $(a_G @ \emptyset, \vec{v}_G)$ approximates. Therefore, $\langle a'_L @ \emptyset, \vec{v}'_L \rangle \in$ dom($\mu'_L$), and because no other messages were added to $\mu'_L$, that message is unique.

Finally, we must show that $\mu'_G$ is fully merged. If $\mu_G (a_G @ \emptyset, \vec{v}_G) = \text{many}$, then $\mu'_G = \mu_G$, and we know that $\mu_G$ is fully merged because $\mu_L \subseteq \mu_G$. Otherwise, $\mu_G = \mu'_G \cup \{ (a_G @ \emptyset, \vec{v}_G) \rightarrow m \}$ for some $m$, and therefore $\mu'_G$ is fully merged.

**Restricted Transition**

As in the previous case, we must also show that $K_G \xrightarrow{\ell_G} K'_G$ RA. We argue for each of the conditions of RA below.
1. We have already shown that $\tilde{K}_G \xrightarrow{\tilde{I}_G} \tilde{K}'_G$.

2. $\tilde{I}_G$ is not a handler-continuation label.

3. By the definition of $\sqsubset$, if an actor in $\tilde{K}_G$ is handling an event, then there must be an actor in $\tilde{K}_L$ also handling an event. But by the definition of $\longrightarrow_{RA}$, there is no such actor in $\tilde{K}_L$. Therefore, there is no actor handling an event in $\tilde{K}_G$.

4. $\tilde{I}_G$ is not a spawn label.

5. $\tilde{I}_G$ is not a rcv-ext label.

6. $\tilde{I}_G$ is not a send-ext label.

**Case: A-TIMEOUT**

This case is nearly identical to the one for A-RECEIVEEXTERNAL. The proof here focuses on the differences.

By the definition of this rule, there exist $\hat{\beta}_L, \hat{\mu}_L, H_L, \hat{\alpha}_L, \hat{Q}_L, \ldots, \hat{Q}_L, x, \hat{e}_L, n, \hat{e}'_L, H'_L, \hat{v}_L, \hat{v}'_L$, and $H''_L$ such that

- $\tilde{K}_L = \langle \langle \hat{\beta}_L \mid \hat{\mu}_L \mid H_L \rangle \rangle^{\hat{\beta}_L}_L$,
- $\langle \langle \hat{Q}_L, \ldots, \hat{Q}_L, n, (\text{receive } x \hat{e}_L \ [ \text{timeout } n \ ) \ \hat{e}'_L \rangle \rangle \rangle \in \hat{\beta}_L(\hat{\alpha}_L)$,
- $\tilde{I}_L = \hat{\alpha}_L : \text{timeout}$, and
- $\tilde{K}'_L = \langle \langle \hat{\beta}_L \oplus \langle \hat{\alpha}_L \rightarrow \langle \langle \hat{Q}_L, \ldots, \hat{Q}_L, \hat{e}'_L \rangle \rangle \rangle^{\hat{\beta}_L}_L \mid \hat{\mu}_L \mid H_L \rangle \rangle^{\hat{\beta}_L}_L$.

**A-TIMEOUT Transition**

First, we show that a similar transition from $\tilde{K}_G$ with A-TIMEOUT is possible. There must exist $\hat{\beta}_G, \hat{\mu}_G, H_G$, and $\hat{\beta}_G$ such that $\tilde{K}_G = \langle \langle \hat{\beta}_G \mid \hat{\mu}_G \mid H_G \rangle \rangle^{\hat{\beta}_G}_G$. Similar to the previous case, by the definition of $\sqsubset$, there exist $\hat{\alpha}_G, \hat{Q}_G, \ldots, \hat{Q}_G, n, \hat{e}_G$, and $\hat{e}'_G$ such that

- $\hat{\alpha}_G, H_L \sqsubseteq_M H_G, \hat{Q}_L, \ldots, \hat{Q}_G, \hat{e}_G, \hat{e}'_G \sqsubseteq_{A,M}$ for all $i 
- \hat{\alpha}_G, H_L \sqsubseteq_{A,M} H_G, \hat{Q}_L, \ldots, \hat{Q}_G, \hat{e}_G, \hat{e}'_G \sqsubseteq_{A,M}$

Then by the rule A-TIMEOUT, there exist $\hat{I}_G$ and $\tilde{K}'_G$ such that

- $\tilde{K}_G \xrightarrow{\tilde{I}_G} \tilde{K}'_G$,
- $\tilde{I}_G = \hat{\alpha}_G : \text{timeout}$, and
- $\tilde{K}'_G = \langle \langle \hat{\beta}_G \oplus \langle \hat{\alpha}_G \rightarrow \langle \langle \hat{Q}_G, \ldots, \hat{Q}_G, \hat{e}'_G \rangle \rangle \rangle^{\hat{\beta}_G}_G \mid \hat{\mu}_G \mid H_G \rangle \rangle^{\hat{\beta}_G}_G$. 

Approximating Configuration

Let $A' = \emptyset$ and $M' = \emptyset$. The transition $\hat{K}_L \xrightarrow{\hat{I}_L} \hat{K}_L'$ adds no new actors or markers, so $\text{dom}(A')$ contains the addresses of all new actors, and $\text{dom}(M')$ contains all new markers.

The argument that $\hat{I}_L \subseteq A \cup A', M \cup M'$ and $\hat{K}_L \subseteq A \cup A', M \cup M'$ is similar to the argument in the A-RECEIVE-INTERNAL case, using the Approximation Substitution and New Behavior lemmas to show a correspondence between the transitioned actors; the only difference is that the message maps $\hat{\mu}_L$ and $\hat{\mu}_G$ do not change.

Restricted Transition

We must also show that $\hat{K}_G \xrightarrow{\hat{I}_G} \hat{K}_G'$. We argue for each of the conditions of $\xrightarrow{\text{RA}}$ below.

1. We have already shown that $\hat{K}_G \xrightarrow{\hat{I}_G} \hat{K}_G'$.
2. $\hat{I}_G$ is not a handler-continuation label.
3. By the definition of $\subseteq$, if an actor in $\hat{K}_G$ is handling an event, then there must be an actor in $\hat{K}_L$ also handling an event. But by the definition of $\xrightarrow{\text{RA}}$, there is no such actor in $\hat{K}_L$. Therefore, there is no actor handling an event in $\hat{K}_G$.
4. $\hat{I}_G$ is not a spawn label.
5. $\hat{I}_G$ is not a rcv-ext label.
6. $\hat{I}_G$ is not a send-ext label.

Case: A-SEND-EXTERNAL

By the definition of this rule, there exist $\hat{\beta}_L$, $\hat{\mu}_L$, $H_L$, $\hat{\rho}_L$, $\hat{a}_L$, $\hat{b}_L$, $\hat{d}_L'$, $\hat{Q}_{L,1}, \ldots, \hat{Q}_{L,n}$, $\hat{E}_L$, $\hat{a}'_L$, $\hat{H}'_L$, $\hat{v}_L$, $\hat{v}'_L$, $\hat{H}'_L$, $\hat{r}$, and $\hat{\rho}'_L$ such that

- $\hat{K}_L = \langle \hat{\beta}_L \mid \hat{a}_L \quad \hat{b}_L \cup \{\hat{b}_L\} \mid \hat{\mu}_L \mid H_L \rangle \hat{\rho}_L$,
- $\hat{b}_L = \langle \hat{Q}_{L,1} \ldots \hat{Q}_{L,n}, C_L \mid \hat{E}_L \mid \langle \text{send} \quad \hat{a}'_L \@ H'_L \quad \hat{v}_L \rangle \rangle$,
- $\hat{a}'_L$ is external,
- $\text{ActorType}(\hat{a}'_L) = \tau$,
- either $\langle \hat{v}'_L, H'_L \rangle \in \text{Markings}(\hat{v}_L, H_L)$ (if $\hat{E}_L \neq \hat{E}' \mid \langle \text{for/fold} \quad [x \quad \hat{E}'' \mid [x' \quad \hat{v}'' \mid \hat{c}'] \mid C = [] \rangle$ or $\langle \hat{v}'_L, H'_L \rangle = \langle \hat{v}_L, H_L \rangle$, or

\[ \text{L.21. APPROXIMATION SOUNDNESS LEMMA} \]
First, we show that a similar transition from $\tilde{K}_L$ with $\text{A-SendExternal}$ is possible. By the definition of $\sqsubset$, there exist $\tilde{\rho}_L$, $\tilde{\mu}_L$, $H_G$, $\tilde{\rho}_G$, $\tilde{\alpha}_G$, $\tilde{B}_L$, $\tilde{b}_G$, $\tilde{Q}_{G,1}, \ldots, \tilde{Q}_{G,n}$, $C_G$, and $\tilde{e}_G$ such that

1. $\tilde{K}_L = \langle \langle \tilde{\rho}_L | \tilde{\mu}_L | H_G \rangle \rangle$.
2. $\tilde{\rho}_G(\tilde{\alpha}_G) = \tilde{B}_L \cup \{ \tilde{b}_G \}$.
3. $\tilde{b}_G = \langle \tilde{Q}_{G,1}, \ldots, \tilde{Q}_{G,n} \rangle$, $C_G$, and $\tilde{e}_G$.
4. $\tilde{\alpha}_L \sqsubset A_M \tilde{\rho}_G$, $\tilde{\alpha}_G \sqsubset A_M \tilde{\alpha}_G$.

Then by the Abstract Context lemma, and the definition of $\sqsubset$, there exist $\tilde{E}_L$, $\tilde{\alpha}'_G$, $H'_G$, and $\tilde{\nu}_G$ such that

1. $\tilde{E}_L \sqsubset A_M \tilde{E}_G$.
2. $\tilde{\alpha}'_L @ H'_L \sqsubset A_M \tilde{\alpha}'_G @ H'_G$.
3. $\tilde{\nu}_L \sqsubset A_M \tilde{\nu}_G$.
4. $\tilde{\alpha}'_G$ is external, and
5. $\text{ActorType}(\tilde{\alpha}'_G) = \tau$.

If $C_G = []$ and $\tilde{E}_G \neq \tilde{E}_G' \langle \text{for/fold} [ x \tilde{E}'_G ] [ x' \tilde{\nu}'_G ] \tilde{\alpha}'_G \rangle$, then let $\langle \tilde{\nu}_G, H'_G \rangle = \text{Mark}(\tilde{\nu}_G, H_G)$. Otherwise, let $\langle \tilde{\nu}_G, H'_G \rangle = \langle \tilde{\nu}_G, H_G \rangle$. Furthermore, let $\tilde{\rho}'_G = \text{IntAddrTypes}(\tilde{\nu}_G, \tau)$. Then there exist $\tilde{I}_G$, $\tilde{K}'_G$, and $\tilde{b}_G'$ such that

1. $\tilde{K}_G \xrightarrow{\tilde{I}_G} \tilde{K}'_G$.  

• \( \hat{\beta}_G = \hat{\beta}_G : \text{send-ext}(\hat{a}_G' \oplus H_G', \hat{v}_G') \),

• \( \hat{v}_G' = \langle \hat{Q}_{G,1} \ldots \hat{Q}_{G,n}, C_G[\hat{E}_G[(\text{variant Unit})]] \rangle \), and

• \( \hat{K}_G = \langle \hat{\beta}_G : \hat{B}_G \cup \{ \hat{v}_G' \} \big| \hat{\mu}_G \big| H''_G \rangle \hat{\beta}_G \cup \hat{v}_G'. \)

**Approximation Configuration**

If \( \langle \hat{v}'_G, H''_G \rangle = \langle \hat{v}_G, H_G \rangle \), then let \( M' = \emptyset \). It is trivially the case that \( \text{dom}(M') \subseteq H''_L - H_L \). If \( \langle \hat{v}'_L, H''_L \rangle = \text{Mark}(\hat{v}_L, H_L) \), then by the Abstract Marker Soundness lemma,

\[
\hat{v}_L \subseteq_{A,M \oplus M'} \hat{v}_G'.
\]

Otherwise if \( \langle \hat{v}'_L, H''_L \rangle = \langle \hat{v}_L, H_L \rangle \), then we already have that

\[
\hat{v}_L \subseteq_{A,M} \hat{v}_G,
\]

so we also know that \( \hat{v}_L \subseteq_{A,M \oplus M'} \hat{v}_G'. \)

Otherwise, \( \langle \hat{v}'_G, H''_G \rangle = \text{Mark}(\hat{v}_G, H_G) \), so we know that \( C_G = [] \) and \( \hat{E}_G \neq \hat{E}'_G \) (for/fold \[x \hat{E}'_G \] \[x' \hat{v}'_G \] \[\hat{v}'_G \]). Then by the definition of \( \sqsubseteq \), \( \hat{E}_L \neq \hat{E}'_L \) (for/fold \[x \hat{E}'_L \] \[x' \hat{v}'_L \] \[\hat{v}'_L \]) and \( C = [] \), so \( \langle \hat{v}'_L, H''_L \rangle = \text{Mark}(\hat{v}_L, H_L) \). Then by the Abstract Marker Soundness lemma, there exists \( M' \) such that

• \( \hat{v}'_L \subseteq_{A,M \oplus M'} \hat{v}_G' \),

• \( H''_L \subseteq_{M \oplus M'} H''_L \),

• \( M \cup M' \) is one-to-one, and

• \( \text{dom}(M') \subseteq H''_L - H_L \).

Because \( \hat{K}_L \) is well-formed, \( H_L \) includes every marker appearing in \( \hat{K}_L \), so we have

• \( \hat{\beta}_L \subseteq_{A,M \oplus M'} \hat{\beta}_G \),

• \( \hat{a}_L \subseteq_A \hat{a}_G \),

• \( \hat{Q}_{L,i} \subseteq_{A,M \oplus M'} \hat{Q}_{G,i} \) for all \( i \in 1 \ldots n \),

• \( \hat{E}_L \subseteq_{A,M \oplus M'} \hat{E}_G \),

• \( \hat{\mu}_L \subseteq_{A,M \oplus M'} \hat{\mu}_G \),

• \( \hat{\beta}_L \subseteq_{A,M \oplus M'} \hat{\beta}_G \).

It is trivially the case that \( \text{(variant Unit)} \subseteq_{A,M \oplus M'} \text{(variant Unit)} \), so by corollary L.18.1 to the Approximation Substitution lemma, we have that \( \hat{E}_L[(\text{variant Unit})] \subseteq_{A,M \oplus M'} \hat{E}_G[(\text{variant Unit})] \), and therefore \( \hat{v}'_L \subseteq_{A,M \oplus M'} \hat{v}_G' \).

We also know that \( \hat{a}_L \in \text{dom}(\hat{\beta}_L) \), and because \( \hat{\beta}_L \subseteq_{A,M} \hat{\beta}_G \), no behavior in \( \hat{B}_G \) is handling an event. Also, because \( \hat{K}_L \) is a single-handler configuration, there is no actor at some \( \hat{a}_L' \neq \hat{a}_L \) approximated by \( \hat{a}_G \) that is handling an event. Therefore by the Replaced Behavior lemma, we also have

\[
\hat{\beta}_L \left[ \hat{a}_L \rightarrow \hat{B}_L \cup \{ \hat{v}_L' \} \right] \subseteq_{A,M \oplus M'} \hat{\beta}_G \left[ \hat{a}_G \rightarrow \hat{B}_G \cup \{ \hat{v}_G' \} \right].
\]
It remains to show that \( \hat{\rho}_L \cup \hat{\rho}_L' \subseteq_{A,M \in M'} \hat{\rho}_G \cup \hat{\rho}_G' \). We already know that 
\( \hat{\rho}_L \subseteq_{A,M \in M'} \hat{\rho}_G \), and by the Abstract Internal Address Types lemma we have 
\( \hat{\rho}_L' \subseteq_{A,M \in M'} \hat{\rho}_G' \), which suffices to show that 
\( \hat{\rho}_L \cup \hat{\rho}_L' \subseteq_{A,M \in M'} \hat{\rho}_G \cup \hat{\rho}_G' \).

We have already shown that \( dom(M') \subseteq H'_G - H_L \). Let \( A' = \emptyset \). Then by the above arguments, \( \hat{R}'_L \subseteq_{A,M \in M'} \hat{R}'_G \) and \( \hat{i}'_L \subseteq_{A,M \in M'} \hat{i}'_G \). The transition \( \hat{k}_L \xrightarrow{RA} \hat{k}'_L \) adds no new actors, so \( dom(A') \) contains the addresses of all new actors.

**Restricted Transition**

We must also show that \( \hat{k}_G \xrightarrow{RA} \hat{k}'_G \). We argue for each of the conditions of \( \rightarrow_{RA} \) below.

1. We have already shown that \( \hat{k}_G \xrightarrow{RA} \hat{k}'_G \).

2. By contradiction, assume there is some \( \tilde{a}'_G \) such that \( \tilde{a}_G < \tilde{a}'_G \) and the actor at \( \tilde{a}'_G \) in \( \hat{k}_G \) is handling an event. Because \( \hat{k}_L \subseteq_{A,M} \hat{k}_G \), there exists some \( \tilde{a}'_L \) such that the actor at \( \tilde{a}'_L \) in \( \hat{k}_L \) is handling an event. By the definition of \( \subseteq_{A,M} \), it must be the case that \( \tilde{a}_L < \tilde{a}'_L \). This leads to a contradiction, however, because by the definition of \( \hat{k}_L \xrightarrow{RA} \hat{k}'_L \), there can be no such \( \tilde{a}'_L \). Therefore, there is no \( \tilde{a}'_G \) such that \( \tilde{a}_G < \tilde{a}'_G \) and the actor at \( \tilde{a}'_G \) in \( \hat{k}_G \) is handling an event.

3. \( \hat{i}_G \) is not a handler-start label.

4. \( \hat{i}_G \) is not a `spawn` label.

5. \( \hat{i}_G \) is not a `rcv-ext` label.

6. We have already shown that either \( H''_G = H_G \) or \( \text{Mark}((\hat{\nu}_G,H_G)) = \langle \hat{\nu}'_G,H''_G \rangle \).

**Case: A-Send**

This case is similar to the previous one. The proof here focuses on the differences.

By the definition of this rule, there exist \( \hat{\beta}_L \), \( \hat{\mu}_L \), \( H_L \), \( \hat{\rho}_L \), \( \tilde{a}_L \), \( \tilde{b}_L \), \( \hat{Q}_{L,1} \ldots \hat{Q}_{L,n} \), \( C_L \), \( \hat{E}_L \), \( \tilde{a}'_L \), \( H'_L \), \( \hat{\nu}_L \), \( \hat{\nu}'_L \), and \( H''_L \) such that

- \( \hat{k}_L = \langle \hat{\beta}_L \mid \tilde{a}_L \rightarrow \hat{b}_L \cup \{ \hat{b}_L \} \mid \hat{\mu}_L \mid H_L \rangle \).
- \( \hat{b}_L = \langle \hat{Q}_{L,1} \ldots \hat{Q}_{L,n} \mid C_L [ \hat{E}_L \left[ (\text{send } \tilde{a}'_L @ H'_L \hat{\nu}_L) \right] ] \rangle \).
- \( \tilde{a}'_L \) is internal,
- if \( H'_L = \emptyset \), then \( \langle \hat{\nu}'_L,H''_L \rangle = \langle \hat{\nu}_L,H_L \rangle \).
First, we show that a similar transition from \( \hat{L} \) is possible. 

Because \( \hat{K}_L \) is an externals-only configuration, we also know that \( H_L' = \emptyset \), so \( \hat{v}_L' = \hat{v}_L \) and \( H_L'' = H_L \).

**A-SendInternal Transition**

First, we show that a similar transition from \( \hat{K}_G \) with A-SendInternal is possible.

By the definition of \( \sqsubseteq \), there exist \( \hat{\beta}_G, \hat{\mu}_G, H_G, \hat{\rho}_G, \hat{a}_G, \hat{B}_G, \hat{b}_G, \hat{Q}_{G,1}, \ldots, \hat{Q}_{G,n}, \hat{C}_G, \) and \( \hat{\epsilon}_G \) such that

- \( \hat{K}_G = \langle \langle \hat{\beta}_G | \hat{\mu}_G | H_G \rangle \rangle_{\hat{\rho}_G} \),
- \( \hat{\rho}_G(\hat{a}_G) = \hat{B}_G \sqcup \{ \hat{\epsilon}_G \} \),
- \( \hat{\delta}_G = \langle \hat{Q}_{G,1} \ldots \hat{Q}_{G,n}, \hat{C}_G | \hat{\epsilon}_G \rangle \), and
- \( \hat{\mu}_L \sqsubseteq_{A,\hat{M}} \hat{\beta}_G, \hat{a}_L \sqsubseteq_{A} \hat{a}_G, \hat{Q}_{L,i} \sqsubseteq_{A,\hat{M}} \hat{Q}_{G,i} \) for all \( i \in 1 \ldots n \), \( \hat{C}_L \sqsubseteq C_G \), \( \hat{B}_L \sqsubseteq_{\hat{A},M} \{ \text{send} \hat{a}_L' @ H_L' | \hat{v}_L \} \sqsubseteq_{A,\hat{M}} \hat{a}_G' \sqsubseteq_{A} \hat{a}_G, \hat{\mu}_L \sqsubseteq_{A,\hat{M}} \hat{\mu}_G, \hat{H}_L \sqsubseteq M H_G, \) and \( \hat{\rho}_L \sqsubseteq_{A,\hat{M}} \hat{\rho}_G \).

Similar to the previous case, the Abstract Context lemma gives us that there exist \( \hat{E}_G, \hat{a}_G', \) and \( \hat{v}_G \) such that

- \( \hat{\epsilon}_G = \hat{E}_G[\text{send} \hat{a}_G' \circ \emptyset \hat{v}_G] \),
- \( \hat{E}_L \sqsubseteq_{A,\hat{M}} \hat{E}_G \),
- \( \hat{a}_G' \circ \emptyset \sqsubseteq_{A,\hat{M}} \hat{a}_G' \circ \emptyset \),
- \( \hat{v}_L \sqsubseteq_{A,\hat{M}} \hat{v}_G \), and
- \( \hat{a}_G' \) is internal.

Let \( \hat{v}_G' = \hat{v}_G \) and \( H_G'' = H_G \). Then there exist \( \hat{L}_G, \hat{K}_G', \) and \( \hat{b}_G' \) such that

- \( \hat{K}_G \xrightarrow{\hat{\epsilon}_G} \hat{K}_G' \),
- \( \hat{L}_G = \hat{a}_G : \text{send-int}(\hat{a}_G' \circ \emptyset \hat{v}_G') \),
- \( \hat{b}_G' = \langle \hat{Q}_{G,1} \ldots \hat{Q}_{G,n}, \hat{C}_G | \hat{E}_G | \text{(variant Unit)} \rangle \rangle \), and
- \( \hat{K}_G' = \langle \langle \hat{\beta}_G [\hat{a}_G \leftarrow \hat{B}_G \cup \{ \hat{\epsilon}_G \} | \hat{\mu}_G \circ \{ \hat{a}_G' \circ \emptyset, \hat{v}_G' \} | H_G'' \rangle \rangle_{\hat{\rho}_G} \), where \( \hat{\beta}_G = \langle \hat{Q}_{G,1} \ldots \hat{Q}_{G,n}, \hat{C}_G | \hat{E}_G | \text{(variant Unit)} \rangle \rangle \).
Approximating Configuration

We already know that $H_L \subseteq_M H_G$, and $\hat{\rho}_L \subseteq_{A,M} \hat{\rho}_G$. Furthermore, we know that $\bar{a}_L \subseteq \hat{\bar{a}}_G$, $\bar{a}_L \otimes \emptyset \subseteq_{A,M} \bar{a}_G \otimes \emptyset$, and $\bar{b}_L' \subseteq_{A,M} \bar{b}_G'$, so we have that $\bar{t}_L \subseteq_{A,M} \bar{t}_G$.

The proof that $\hat{\beta}_G \left[ \bar{a}_G \rightarrow \hat{\bar{b}}_G \right] \left\{ \left\{ \bar{Q}_{G,1} \ldots \right\} \hat{Q}_{G,n}, \hat{E}_G \left[ \left\{ \text{variant Unit} \left( \text{Unit} \right) \right\} \right] \right\}$ approximates $\beta_L \left[ \bar{a}_L \rightarrow \bar{t}_L \right] \left\{ \left\{ \bar{Q}_{L,1} \ldots \right\} \hat{Q}_{L,n}, \hat{E}_L \left[ \left\{ \text{variant Unit} \left( \text{Unit} \right) \right\} \right] \right\}$ is identical to the one for the previous case, using corollary L.18.1 to the Approximation Substitution lemma and the Replaced Behavior lemma.

It remains to show that $\hat{\mu}_L \otimes \langle \hat{\bar{a}}_L \otimes \emptyset, \hat{\bar{b}}_L' \rangle \subseteq_{A,M} \hat{\mu}_G \otimes \langle \hat{\bar{a}}_G \otimes \emptyset, \hat{\bar{b}}_G' \rangle$. We know that

- $\bar{\mu}_L \subseteq_{A,M} \hat{\mu}_G$,
- $\bar{a}_L' \otimes \emptyset \subseteq_{A,M} \hat{\bar{a}}_L' \otimes \emptyset$, and
- $\bar{\bar{b}}_L \subseteq_{A,M} \hat{\bar{b}}_G$.

Because $\bar{\mu}_L \subseteq_{A,M} \hat{\mu}_G$, $\hat{\mu}_G$ is fully merged. Then by the Message-Addition Soundness lemma, we have $\hat{\mu}_L \otimes \langle \hat{\bar{a}}_L' \otimes \emptyset, \hat{\bar{b}}_L' \rangle \subseteq_{A,M} \hat{\mu}_G \otimes \langle \hat{\bar{a}}_G' \otimes \emptyset, \hat{\bar{b}}_G' \rangle$.

Let $A' = \emptyset$ and $M' = \emptyset$. The transition $\hat{K}_L \xrightarrow{\bar{t}_G} \hat{K}_L'$ adds no new actors or markers, so $\text{dom}(A')$ contains the addresses of all new actors, and $\text{dom}(M')$ contains all new markers.

Restricted Transition

We must also show that $\hat{K}_G \xrightarrow{\bar{t}_G} \hat{K}_G'$. We argue for each of the conditions of $\xrightarrow{\text{to-RA}}$ below.

1. We have already shown that $\hat{K}_G \xrightarrow{\bar{t}_G} \hat{K}_G'$.
2. Identical to the argument in the A-SENDEXTERNAL case above.
3. $\bar{t}_G$ is not a handler-start label.
4. $\bar{t}_G$ is not a spawn label.
5. $\bar{t}_G$ is not a rcv-ext label.
6. $\bar{t}_G$ is not a send-ext label.

Case: A-SPAWN

By the definition of this rule, there exist $\hat{\beta}_L$, $\bar{\mu}_L$, $H_L$, $\hat{\bar{b}}_L$, $\bar{\bar{b}}_L$, $\bar{\bar{Q}}_{L,1} \ldots \bar{\bar{Q}}_{L,n}$, $C_L$, $E_L$, $\ell$, $\bar{\bar{t}}_L$, $\bar{\bar{Q}}_{L,m}$, $C_L'$, $\bar{\bar{a}}_L'$, $n$, and $\bar{\bar{b}}_L''$ such that

- $\hat{K}_L = \langle \langle \hat{\beta}_L \left[ \bar{a}_L \rightarrow \hat{\bar{b}}_L \cup \{ \bar{b}_L \} \right] \rangle \bar{\mu}_L \left| H_L \right\rangle \hat{\beta}_L$,
- $\bar{\bar{b}}_L = \langle \bar{\bar{Q}}_{L,1} \ldots \bar{\bar{Q}}_{L,n} \rangle \bar{E}_L \left[ \left\{ \text{spawn' } \bar{e}_L \bar{\bar{Q}}_{L,1} \ldots \bar{\bar{Q}}_{L,m} \right\} \right]$,
- $\hat{K}_G = \langle \langle \hat{\beta}_G \left[ \bar{a}_G \rightarrow \hat{\bar{b}}_G \right] \rangle \hat{\mu}_G \left| H_G \right\rangle \hat{\beta}_G$,
- $\hat{\bar{b}}_G = \langle \bar{\bar{Q}}_{G,1} \ldots \bar{\bar{Q}}_{G,n} \rangle \hat{E}_G \left[ \left\{ \text{variant Unit} \left( \text{Unit} \right) \right\} \right]$.


L.21. APPROXIMATION SOUNDNESS LEMMA

First, we show that a similar transition from $\hat{E}_L = \hat{E}'[(\text{for/fold } [x E''_{G} ] [x' \hat{v}_G ] \hat{e}'_{G})]$ for some $E', x, E''_{G}, x', \hat{v}_L$, and $\hat{e}'_{L}$, otherwise $\hat{a}'_{L} = (\text{addr } \ell n)$ and $\hat{a}'_{L} \notin \{\hat{a}_L\} \cup \text{dom}(\hat{\mu}_L)$,

$C'_G = (\text{in-loop } [])$ if $C_G = (\text{in-loop } [])$ or $\hat{E}_L = \hat{E}'[(\text{for/fold } [x E''_{G} ] [x' \hat{v}_G ] \hat{e}'_{G})]$ for some $E', x, \hat{E}'_{L}, x', \hat{v}_L$, and $\hat{e}'_{L}$, otherwise $C'_G = \{\}$.

$\hat{b}'_L = \langle \hat{Q}_{L,1} \ldots \hat{Q}_{L,n}, \hat{C}_L \left[ \hat{E}_L \left[ \hat{a}'_{L} \hat{\varphi} \right] \right] \rangle$,

$\hat{b}'_{L'} = \langle \hat{Q}_{L',n}[\text{self } - \hat{a}'_{L} \hat{\varphi}] \ldots \hat{Q}_{L,1}[\text{self } - \hat{a}'_{L} \hat{\varphi}], \hat{C}_L[\hat{\varphi}, \hat{a}'_{L} \hat{\varphi}] \rangle$, and

$\hat{t}_L = \hat{a}_L : \text{spawn}(\hat{a}'_{L})$, and

$\hat{K}'_L = \langle \hat{b}_L \left[ \hat{a}_L \rightarrow \hat{E}_L \hat{b}'_L \right] \hat{t}_L \left[ \hat{a}'_L \rightarrow \hat{b}'_{L'} \right] \hat{\mu}_L \left[ H_L \right] \rangle$.

A-SPAWN Transition

First, we show that a similar transition from $\hat{K}_G$ with A-SPAWN is possible. By the definition of $\sqsubseteq$, there exist $\hat{\beta}_G, \hat{\mu}_G, H_G, \hat{\rho}_G, \hat{a}_G, \hat{B}_G, \hat{\bar{b}}_G, \hat{Q}_{G,1}, \ldots, \hat{Q}_{G,n}, C_G$, and $\hat{e}_G$ such that

$\hat{K}_G = \langle \hat{\beta}_G \left[ \hat{\mu}_G \left[ H_G \right] \right] \rangle$, and $\hat{\beta}_G(\hat{a}_G) = \hat{B}_G \cup \{\hat{b}_G\}$,

$\hat{b}_G = \langle \hat{Q}_{G,1} \ldots \hat{Q}_{G,n}, C_G \left[ \hat{e}_G \right] \rangle$ and

$\hat{b}_L \sqsubseteq_{A,M} \hat{\beta}_G, \hat{a}_L \sqsubseteq_{A} \hat{a}_G, \hat{Q}_{L,i} \sqsubseteq_{A,M} \hat{Q}_{G,i}$ for all $i \in 1 \ldots n$, $C_L \sqsubseteq C_G$, $\hat{E}_L \left[ \text{spawn}^\tau \hat{e}_L \hat{Q}_{L,1} \ldots \hat{Q}_{L,n} \right] \sqsubseteq_{A,M} \hat{e}_G, \hat{\mu}_L \sqsubseteq_{A,M} \hat{\mu}_G, H_L \sqsubseteq_{M} H_G$, and $\hat{\beta}_L \sqsubseteq_{A,M} \hat{\beta}_G$.

By the Abstract Context lemma and the definition of $\sqsubseteq$, there exist $\hat{E}_G, \hat{e}'_G,$ and $\hat{Q}_{G,1}, \ldots, \hat{Q}_{G,m}$ such that such that

$\hat{e}_G = \hat{E}_G[(\text{spawn}^\tau \hat{e}_G \hat{Q}_{G,1} \ldots \hat{Q}_{G,m})]$,

$\hat{E}_L \sqsubseteq_{A,M} \hat{E}_G$, and

$\hat{e}_L \sqsubseteq_{A,M} \hat{e}'_G$, and

$\hat{Q}_{L,i} \sqsubseteq_{A,M} \hat{Q}_{G,i}$ for all $i \in 1 \ldots m$.

If $\hat{a}_G$ is collective or $\hat{E}_G = \hat{E}'[(\text{for/fold } [x E''_{G} ] [x' \hat{v}_G ] \hat{e}'_{G})]$ for some $E', x, \hat{E}'_{G}, x', \hat{v}_G$, and $\hat{e}'_{G}$, then let $\hat{a}_G = (\text{collective-addr } \ell)$. Otherwise, let $n$ be the smallest natural number such that $(\text{addr } \ell n) \notin \text{dom}(\hat{\beta}_G) \cup \{\hat{a}_G\}$, and let $\hat{a}_G = (\text{addr } \ell n)$. Also, if $\hat{E}_G =$
$\hat{\beta}_G'' = \langle \hat{Q}_{G,1}'[\text{self } \to \hat{\alpha}_G'@\emptyset] \ldots \hat{Q}_{G,m}'[\text{self } \to \hat{\alpha}_G'@\emptyset], C_G'[\hat{\alpha}_G'[\text{self } \to \hat{\alpha}_G'@\emptyset]] \rangle$. Then there exist $\check{I}_G$ and $\check{R}_G'$ such that

- $\check{I}_G = \check{\alpha}_G@\emptyset$: spawn($\hat{\alpha}_G$),
- $\check{R}_G' = \langle \hat{\beta}_G[\hat{\alpha}_G \to \check{B}_G \cup \{\hat{\beta}_G\}] \uplus \hat{\alpha}_G \to \hat{\beta}_G'' \mid \check{\mu}_G \mid H_G \rangle^{\hat{\beta}_G}$.

### Approximating Configuration

Let $A' = \left[ \hat{a}_L \to \hat{\alpha}_G' \right]$. By the definition of $\check{K}_L \subseteq_{A,M} \check{K}_G$, $\text{dom}(\hat{\rho}_L) \subseteq \text{dom}(\hat{A})$. Then because $\check{K}_L$ is well-formed, every internal address in $\hat{\mu}_L$, $H_L$, and $\hat{\rho}_L$ is in $\text{dom}(\hat{\rho}_L)$. Therefore, we have $\hat{\mu}_L \subseteq_{A\cup\{A'\},M} \hat{\beta}_G$, $H_L \subseteq_{M} H_G$, and $\hat{\rho}_L \subseteq_{A\cup\{A'\},M} \hat{\beta}_G$.

It remains to show that $\hat{\beta}_G[\hat{\alpha}_G \to \check{B}_G \cup \{\hat{\beta}_G\}] \uplus \hat{\alpha}_G \to \hat{\beta}_G''$ approximates $\hat{\beta}_L[\hat{\alpha}_L \to \check{B}_L \cup \{\hat{\beta}_L\}] \uplus \hat{\alpha}_L \to \hat{\beta}_L''$. Let $\hat{\beta}_L = \hat{\rho}_L[\hat{\alpha}_L \to \check{B}_L \cup \{\hat{\beta}_L\}]$, and let $\hat{\beta}_G = \hat{\rho}_G[\hat{\alpha}_G \to \check{B}_G \cup \{\hat{\beta}_G\}]$; that is, each one is the original actor-behavior map with the spawning actor updated. By the definition of $A'$, we have that $\hat{\alpha}_G'@\emptyset \in \hat{A}_M \Rightarrow \hat{\alpha}_G'@\emptyset$. Then similarly to the previous case, we can use corollary $\check{L}.18.1$ to the Approximation Substitution lemma and the Replaced Behavior lemma to show that $\hat{\rho}_G \subseteq_{A\cup\{A'\},M} \hat{\beta}_G$.

It remains to show that $\hat{\rho}_G \uplus [\hat{\alpha}_L' \to \hat{\beta}_L''] \subseteq_{A\cup\{A'\},M} \hat{\beta}_G \uplus [\hat{\alpha}_G' \to \hat{\beta}_G'']$. We have

- $\check{B}_L = \langle \hat{Q}_{L,1}'[\text{self } \to \hat{\alpha}_L''@\emptyset] \ldots \hat{Q}_{L,m}'[\text{self } \to \hat{\alpha}_L''@\emptyset], C_G'[\hat{\alpha}_G''[\text{self } \to \hat{\alpha}_G''@\emptyset]] \rangle$ and
- $\check{B}_G = \langle \hat{Q}_{G,1}'[\text{self } \to \hat{\alpha}_G''@\emptyset] \ldots \hat{Q}_{G,m}'[\text{self } \to \hat{\alpha}_G''@\emptyset], C_G'[\hat{\alpha}_G''[\text{self } \to \hat{\alpha}_G''@\emptyset]] \rangle$.

From the above arguments, we know that $\hat{\alpha}_L \subseteq_{A,M} \hat{\alpha}_G''$ and $\hat{Q}_{L,i} \subseteq_{A,M} \hat{Q}_{G,i}$ for all $i \in 1 \ldots m$. Furthermore, because the new concrete address $\hat{\alpha}_L''$ cannot appear in $\hat{\alpha}_L$ or $\hat{Q}_{L,1}' \ldots \hat{Q}_{L,m}'$, we also have $\hat{\alpha}_L \subseteq_{A\cup\{A'\},M} \hat{\alpha}_G''$ and $\hat{Q}_{L,i} \subseteq_{A\cup\{A'\},M} \hat{Q}_{G,i}$ for all $i \in 1 \ldots m$. Then by the Approximation Substitution lemma, we have that $\hat{\alpha}_L[\text{self } \to \hat{\alpha}_L''@\emptyset] \subseteq_{A\cup\{A'\},M} \hat{\alpha}_G''[\text{self } \to \hat{\alpha}_G''@\emptyset]$ and $\hat{Q}_{L,i}[\text{self } \to \hat{\alpha}_L''@\emptyset] \subseteq_{A\cup\{A'\},M} \hat{Q}_{G,i}[\text{self } \to \hat{\alpha}_G''@\emptyset]$ for all $i \in 1 \ldots n$.

For the contexts $C_L$ and $C_G$, if $C_G' = \emptyset$, then it must be the case that neither $\check{E}_G$ does not have the form $(\text{for/fold} [x \check{E}_G'] [x' \check{\nu}_G] \check{\nu}_G''])$ and $\hat{\alpha}_G$ is atomic. Then by the definition of $\subseteq$, $\check{E}_L$ does not have such a form and $\hat{\alpha}_L$ is atomic, Therefore $C_L'' = \emptyset$, and so $C_L' \subseteq C_G'$. Otherwise, $C_G' = (\text{in-loop} [\text{local}])$, so $C_L' \subseteq C_G'$.

The above arguments give $\hat{\rho}_G \subseteq_{A\cup\{A'\},M} \hat{\beta}_G''$. We also have that either $\hat{\alpha}_G'$ is collective or $\hat{\alpha}_G'' \notin \text{dom}(\hat{\beta}_G)$.
is the highest-priority actor in \( \hat{A}_G \) handling an event, and because \( \hat{a}'_G \) has a higher priority than \( \hat{a}_G \), there is no behavior in \( \hat{\beta}_G'(\hat{a}'_G) \) handling an event. Therefore, by the New Behavior lemma, we have \( \hat{\beta}_L' \oplus [\hat{a}'_L \rightarrow \hat{b}_L''] \subseteq_{A \cup A', \hat{M}} \hat{\beta}_G' \oplus [\hat{a}'_G \rightarrow \hat{b}_G''] \).

Let \( M' = \emptyset \). The transition \( \hat{\mathcal{K}}_L \xrightarrow{\text{RA}} \hat{\mathcal{K}}_G \) adds no new markers, so \( \text{dom}(M') \) contains all new markers. We also have \( \text{dom}(A') = \{ \hat{a}_L \} \), so it contains the addresses of all new actors, and by the above arguments, \( \hat{\mathcal{I}}_L \subseteq_{A \cup A', \hat{M} \cup M'} \hat{\mathcal{I}}_G \) and \( \hat{\mathcal{K}}_L' \subseteq_{A \cup A', \hat{M} \cup M'} \hat{\mathcal{K}}_G' \).

**Restricted Transition**

We must also show that \( \hat{\mathcal{K}}_G \xrightarrow{\text{ra}} \hat{\mathcal{K}}_G \). We argue for each of the conditions of \( \xrightarrow{\text{ra}} \) below.

1. We have already shown that \( \hat{\mathcal{K}}_G \xrightarrow{\text{ra}} \hat{\mathcal{K}}_G \).
2. Identical to the argument in the A-SENDEXTERNAL case above.
3. \( \hat{\mathcal{I}}_G \) is not a handler-start label.
4. The address \( \hat{a}'_G \) was defined with the minimum possible identifier \( n \).
5. \( \hat{\mathcal{I}}_G \) is not a rcv-ext label.
6. \( \hat{\mathcal{I}}_G \) is not a send-ext label.

**Case: A-GOTO**

By the definition of this rule, there exist \( \hat{\beta}_L, \hat{\mu}_L, \hat{H}_L, \hat{\rho}_L, \hat{a}_L, \hat{b}_L, \hat{B}, \hat{b}'_L \), \( \hat{Q}_{L,1}, \ldots, \hat{Q}_{L,n}, C_L, \hat{E}_L, q, \hat{v}_{L,1}, \ldots, \hat{v}_{L,m}, x_1, \ldots, x_m, \tau_1, \ldots, \tau_m, x', \hat{e}_L, \hat{\ell}_L, k \) and \( \hat{e}'_L \) such that

- \( \hat{K}_L = \langle \hat{\beta}_L [\hat{a}_L \rightarrow \hat{b}_L \cup \{ \hat{b}_L \} \mid \hat{\mu}_L \mid \hat{H}_L] \rangle^{\hat{\beta}_L} \),
- \( \hat{b}_L = \langle \hat{Q}_{L,1} \ldots \hat{Q}_{L,n}, C_L \mid \hat{E}_L \mid \{ \text{goto} q \hat{v}_{L,1} \ldots \hat{v}_{L,m} \} \rangle \),
- \( 1 \leq k \leq n \),
- \( \hat{Q}_{L,k} = \langle \text{define-state} (q \{ x_1 \tau_1 \} \ldots (x_m \tau_m) \} x' \hat{e}_L \hat{\ell}_L) \rangle \),
- \( \hat{I}_L = \hat{a}_L : \text{goto} \),
- \( \hat{e}'_L = \langle \text{receive} x' \hat{e}_L \hat{\ell}_L \} [x_1 \ldots \hat{v}_{L,1} \ldots [x_m \hat{v}_{L,m}] \rangle \),
- \( \hat{b}'_L = \langle \hat{Q}_{L,1} \ldots \hat{Q}_{L,n}, \hat{e}'_L \rangle \), and
- \( \hat{K}_L = \langle \hat{\beta}_L [\hat{a}_L \rightarrow \hat{B} \cup \{ \hat{b}'_L \} \mid \hat{\mu}_L \mid \hat{H}_L] \rangle^{\hat{\beta}_L} \).
A-GOTO Transition

First, we show that a similar transition from $\hat{K}_G$ with A-GOTO is possible.  

By the definition of $\subseteq$, there exist $\hat{\beta}_G, \hat{\mu}_G, H_G, \hat{\rho}_G, \hat{\varrho}_G, B_G, \hat{\delta}_G, \hat{Q}_G, \ldots, \hat{Q}_{G,n}, C_G$, and $\hat{\epsilon}_G$ such that

- $\hat{K}_G = \langle \langle \hat{\beta}_G \mid \hat{\mu}_G \mid H_G \rangle \rangle^{\hat{\beta}_G}$,
- $\hat{\beta}_G(\hat{\varrho}_G) = \hat{B}_G \cup \{ \hat{\delta}_G \}$,
- $\hat{\delta}_G = \langle \hat{Q}_{G,1} \ldots \hat{Q}_{G,n}, C_G [\hat{\epsilon}_G] \rangle$, and
- $\hat{\rho}_L \subseteq_{A,M} \hat{\beta}_G$, $\hat{\alpha}_L \subseteq_A \hat{\alpha}_G$, $\hat{Q}_{L,i} \subseteq_{A,M} \hat{Q}_{G,i}$ for all $i \in 1 \ldots n$, $C_L \subseteq C_G$, $\hat{E}_L \langle (\text{goto } q \hat{v}_{L,1} \ldots \hat{v}_{L,m}) \rangle \subseteq_{A,M} \hat{\epsilon}_G$, $\hat{\mu}_L \subseteq_{A,M} \hat{\mu}_G$, $H_L \subseteq_H H_G$, and $\hat{\rho}_L \subseteq_{A,M} \hat{\rho}_G$.

By the Abstract Context lemma and the definition of $\subseteq$, there exist $\hat{E}_G$ and $\hat{v}_{G,1}, \ldots, \hat{v}_{G,m}$ such that such that

- $\hat{\epsilon}_G = \hat{E}_G \langle (\text{goto } q \hat{v}_{G,1} \ldots \hat{v}_{G,m}) \rangle$,
- $\hat{E}_L \subseteq_{A,M} \hat{E}_G$, and
- $\hat{v}_{L,j} \subseteq_{A,M} \hat{v}_{G,j}$ for all $j \in 1 \ldots m$.

Because $\hat{Q}_{L,k} \subseteq_{A,M} \hat{Q}_{G,k}$, there also exist $\hat{\epsilon}'_G$ and $\hat{f}_G$ such that

- $\hat{Q}_{G,k} = \langle \text{define-state} (q \ [x_1 \ \tau_1] \ldots \ [x_m \ \tau_m]) x' \hat{\epsilon}'_G \ {\hat{f}_G} \rangle$,
- $\hat{\epsilon}_L \subseteq_{A,M} \hat{\epsilon}'_G$, and
- $\hat{f}_L \subseteq_{A,M} \hat{f}_G$.

Then by the rule A-GOTO, there exist $\hat{\epsilon}'_G, \hat{\iota}_G, \hat{\delta}'_G$, and $\hat{K}'_G$ such that

- $\hat{K}_G \overset{\hat{\iota}_G}{\longrightarrow} \hat{K}'_G$,
- $\hat{\epsilon}'_G = \langle \text{receive} x' \hat{\epsilon}'_G \ {\hat{f}_G} \rangle [x_1 \leftarrow \hat{v}_{G,1}] \ldots [x_m \leftarrow \hat{v}_{G,m}]$,
- $\hat{\iota}_G = \hat{\alpha}_G$: goto,
- $\hat{\delta}'_G = \langle \hat{Q}_{G,1} \ldots \hat{Q}_{G,n}, \hat{\epsilon}'_G \rangle$,
- $\hat{K}'_G = \langle \langle \hat{\beta}_G [\hat{\alpha}_G \rightarrow \hat{B}_G \cup \{ \hat{\delta}_G \}] \mid \hat{\mu}_G \mid H_G \rangle \rangle^{\hat{\beta}_G}$.
L.21. APPROXIMATION SOUNDNESS LEMMA

Approximating Configuration

Most of the components of the configurations did not change, so it remains to show only that \( \hat{\beta}_L \left( \hat{\alpha}_L \mapsto \hat{B}_L \cup \{ \hat{\beta}_L' \} \right) \) approximates \( \hat{\beta}_L \left( \hat{\alpha}_L \mapsto \hat{B}_L \cup \{ \hat{\beta}_L' \} \right) \).

By repeated uses of the Approximation Substitution lemma (one for each of the \( m \) arguments to the state), we have \( \hat{\epsilon}_L \sqsubseteq A.M \hat{\epsilon}_G' \). That gives us that \( \langle \hat{Q}_{L,1}, \ldots, \hat{Q}_{L,m}, \hat{\epsilon}_L' \rangle \sqsubseteq A.M \langle \hat{Q}_{G,1}, \ldots, \hat{Q}_{G,m}, \hat{\epsilon}_G' \rangle \). We also know that \( \hat{\alpha}_L \in \text{dom}(\hat{\beta}_L) \), and because \( \hat{\beta}_L \sqsubseteq A.M \hat{\beta}_G \), no behavior in \( \hat{B}_G \) is handling an event. Also, because \( \hat{K}_L \) is a single-handler configuration, there is no actor at some \( \hat{\alpha}_L' \neq \hat{\alpha}_L \) approximated by \( \hat{\alpha}_G \) that is handling an event. Therefore by the Replaced Behavior lemma, we have \( \hat{\rho}_L \left( \hat{\alpha}_L \mapsto \hat{B}_L \cup \{ \hat{\beta}_L' \} \right) \sqsubseteq A.M \hat{\rho}_G \left( \hat{\alpha}_G \mapsto \hat{B}_G \cup \{ \hat{\beta}_G' \} \right) \).

Let \( A' = \emptyset \) and \( M' = \emptyset \). The transition \( \hat{K}_L \rightarrow_{RA} \hat{K}_L' \) adds no new actors or markers, so \( \text{dom}(A') \) contains the addresses of all new actors, and \( \text{dom}(M') \) contains all new markers. Then by the above arguments, \( \hat{I}_L \sqsubseteq A.M \hat{I}_G \) and \( \hat{K}_L' \sqsubseteq A.M \hat{K}_G' \).

Restricted Transition

We must also show that \( \hat{K}_G \rightarrow_{RA} \hat{K}_G' \). We argue for each of the conditions of \( \rightarrow_{RA} \) below.

1. We have already shown that \( \hat{K}_G \rightarrow_{RA} \hat{K}_G' \).
2. Identical to the argument in the A-SENDEXTERNAL case above.
3. \( \hat{I}_G \) is not a handler-start label.
4. \( \hat{I}_G \) is not a spawn label.
5. \( \hat{I}_G \) is not a rcv-ext label.
6. \( \hat{I}_G \) is not a send-ext label.

Case: A-FUNC

By the definition of this rule, there exist \( \hat{\rho}_L \), \( \hat{\mu}_L \), \( H_L \), \( \hat{\rho}_L \), \( \hat{\alpha}_L \), \( \hat{B}_L \), \( \hat{\beta}_L' \), \( \hat{Q}_{L,1}, \ldots, \hat{Q}_{L,n}, C_L, \hat{E}_L, \hat{\epsilon}_L \), and \( \hat{\epsilon}_L' \) such that

- \( \hat{K}_L = \left( \hat{\rho}_L \left[ \hat{\alpha}_L \mapsto \hat{B}_L \cup \{ \hat{\beta}_L' \} \right] \left| \hat{\mu}_L \left| H_L \right. \right) \hat{\rho}_L \),
- \( \hat{\beta}_L = \langle \hat{Q}_{L,1}, \ldots, \hat{Q}_{L,n}, C_L \left| \hat{E}_L \left[ \hat{\epsilon}_L \right] \right. \rangle \),
- \( \hat{\epsilon}_L \mapsto \hat{\epsilon}_L' \),
- \( \hat{I}_L = \hat{\alpha}_L : \text{func} \).
APPENDIX L. CONFORMANCE-REFLECTION PROOFS

By the Abstract Functional-Step Soundness lemma, there exists

\[ \hat{\delta}_L' = \left\langle \hat{Q}_{L,1} \ldots \hat{Q}_{L,n}, C_L \left[ \hat{E}_L \left[ \hat{e}_L' \right] \right] \right\rangle, \]

\[ \hat{R}'_L = \left\langle \left\langle \hat{\beta}_L \left[ \hat{a}_L \rightarrow B_L \cup \hat{\beta}_L' \right] \mid \hat{\mu}_L \left| H_L \right. \right\rangle \right\rangle^{\hat{\rho}_L}. \]

**A-FUNC Transition**

First, we show that a similar transition from \( \hat{R}_G \) with A-FUNC is possible.

By the definition of \( \subseteq \), there exist \( \hat{\beta}_G, \hat{\mu}_G, H_G, \hat{\rho}_G, \hat{a}_G, \hat{B}_G, \hat{\hat{b}}_G, \hat{Q}_G, \ldots, \hat{Q}_{G,n}, \)

\( C_G \), and \( \hat{e}_G \) such that

\[ \hat{K}_G = \left\langle \left\langle \hat{\beta}_G \mid \hat{\mu}_G \left| H_G \right. \right\rangle \right\rangle^{\hat{\rho}_G}, \]

\[ \hat{\beta}_G(\hat{a}_G) = \hat{B}_G \cup \{ \hat{b}_G \}, \]

\[ \hat{b}_G = \left( \hat{Q}_{G,1} \ldots \hat{Q}_{G,n}, C_G \left[ \hat{e}_G \right] \right), \]

\[ \hat{\beta}_L \subseteq A_M \hat{\beta}_G, \hat{a}_L \subseteq A_M \hat{a}_G, \hat{Q}_{L,1} \subseteq A_M \hat{Q}_{G,1}, \ldots, \text{ for all } i \in 1 \ldots n, C_L \subseteq C_G, \]

\[ \hat{E}_L \left[ \hat{e}_L \right] \subseteq A_M \hat{e}_G, \hat{\mu}_L \subseteq A_M \hat{\mu}_G, H_L \subseteq H_G, \text{ and } \hat{\rho}_L \subseteq A_M \hat{\rho}_G. \]

By the Abstract Context lemma, there exist \( \hat{E}_G \) and \( \hat{\hat{e}}_G' \) such that such that

\[ \hat{\hat{e}}_G = \hat{E}_G[\hat{\hat{e}}_G'], \]

\[ \hat{E}_L \subseteq A_M \hat{E}_G, \text{ and} \]

\[ \hat{\hat{e}}_L \subseteq A_M \hat{\hat{e}}_G'. \]

By the Abstract Functional-Step Soundness lemma, there exists \( \hat{\hat{e}}_G'' \) such that

\( \hat{\hat{e}}_G' \rightarrow \hat{\hat{e}}_G'' \) and \( \hat{\hat{e}}_L \subseteq A_M \hat{\hat{e}}_G''. \) Therefore, by the rule A-FUNC, there exist \( \hat{I}_G, \hat{\hat{b}}_G', \) and \( \hat{\hat{R}}_G' \) such that

\[ \hat{\hat{K}}_G \xrightarrow{\hat{I}_G} \hat{\hat{R}}_G', \]

\[ \hat{I}_G = \hat{\hat{a}}_G : \text{func}, \]

\[ \hat{\hat{b}}_G' = \left\langle \hat{Q}_{G,1} \ldots \hat{Q}_{G,n}, C_G \left[ \hat{E}_G \left[ \hat{\hat{e}}_G' \right] \right] \right\rangle, \text{ and} \]

\[ \hat{\hat{R}}_G' = \left\langle \left\langle \hat{\beta}_G \left[ \hat{a}_G \rightarrow B_G \cup \{ \hat{\hat{b}}_G' \} \right] \mid \hat{\mu}_G \left| H_G \right. \right\rangle \right\rangle^{\hat{\rho}_G}. \]

**Approximating Configuration**

As in the previous case, most of the components did not change, so it remains only to show an approximation relationship between the two actors. The result from the Functional-Step Soundness lemma gives us that \( \hat{\hat{e}}_L \subseteq A_M \hat{\hat{e}}_G', \) and corollary L.18.1 to the Approximation Substitution lemma \( \hat{E}_L[\hat{e}_L'] \subseteq A_M \hat{E}_G[\hat{\hat{e}}_G']. \) The rest of the proof is identical to the one for the A-SENDEXTERNAL case, using the Replaced Behavior lemma to show the relationship.
Restricted Transition

We must also show that \( \hat{K}_G \xrightarrow{\delta} \hat{K}^\prime_G \). We argue for each of the conditions of \( \rightarrow_{RA} \) below.

1. We have already shown that \( \hat{K}_G \xrightarrow{\delta} \hat{K}'_G \).
2. Identical to the argument in the A-SENDEXTERNAL case above.
3. \( \hat{I}_G \) is not a handler-start label.
4. \( \hat{I}_G \) is not a spawn label.
5. \( \hat{I}_G \) is not a rcv-ext label.
6. \( \hat{I}_G \) is not a send-ext label.

\[ \Box \)

L.22 Event-Step Approximation Soundness Lemma

Lemma (Event-Step Approximation Soundness). For all \( \hat{K}_L = \langle \hat{\beta}_L \ | \ H_L \rangle \hat{\mu}_L \), \( \hat{K}'_L = \langle \hat{\beta}'_L \ | \ H'_L \rangle \hat{\mu}'_L \), \( \hat{K}_G, \hat{I}_{L,1}, \ldots, \hat{I}_{L,n}, A, \) and \( M \), if

- \( \hat{K}_L \) is a well-formed, externals-only, single-handler configuration,
- \( \hat{K}_L \subseteq_{A,M} \hat{K}_G \),
- \( \hat{K}_L \xrightarrow{\hat{I}_{L,1}, \ldots, \hat{I}_{L,n}} \hat{K}'_L \),
- \( \text{dom}(A) = \text{dom}(\hat{\beta}_L) \cup \text{ExtAbsAddr} \),
- \( \text{dom}(M) \subseteq H_L \), and
- for all \( \hat{a} \in \text{ExtAddr} \), \( A(\hat{a}) = \hat{a} \),

then there exist \( \hat{I}_{G,1}, \ldots, \hat{I}_{G,n}, \hat{K}'_G, A', \) and \( M' \) such that

- \( \hat{K}_G \xrightarrow{\hat{I}_{G,1}, \ldots, \hat{I}_{G,n}} \hat{K}'_G \),
- \( \hat{I}_{L,i} \subseteq_{A \oplus A', M \oplus M'} \hat{I}_{G,i} \) for all \( i \in \{1 \ldots n\} \),
- \( \hat{K}'_L \subseteq_{A \oplus A', M \oplus M'} \hat{K}'_G \),
- \( \text{dom}(A') = \text{dom}(\hat{\beta}'_L) - \text{dom}(\hat{\beta}_L) \), and
516 APPENDIX L. CONFORMANCE-REFLECTION PROOFS

such that $i \in [1, n]$.

Proof. By the definition of an event step, there exist $\hat{K}'_{L,1}, \ldots, \hat{K}'_{L,n+1}$ such that $\hat{K}'_{L,1} = K_L$, $\hat{K}'_{L,n+1} = K_L$. Let $\hat{\beta}'_{L,1}, \ldots, \hat{\beta}'_{L,n+1}$ be the actor-behavior maps for $\hat{K}'_{L,1}, \ldots, \hat{K}'_{L,n+1}$, respectively. We will show that for all $i \in [1, n+1]$, there exist $\hat{K}''_{G,i}, \hat{\beta}_{L,i}$, $\hat{\beta}_{L,i} A_i'$, and $M'$ such that

- $\hat{K}''_{G,1} = K_G$,
- $\hat{K}''_{G,i} = \hat{K}_{G,i}$,
- $\hat{\beta}_{L,i} = \hat{\beta}_{L,i}$,
- $dom(A_i') = dom(\hat{\beta}_{L,i}) - dom(\hat{\beta}_{L,i})$, and
- $dom(M') = Used(\hat{K}''_{L,i}) - H_L$.

Then the case where $i = n + 1$ completes the proof.

The proof is by induction on $i$. In the base case when $n = 1$, let $\hat{K}''_{G,1} = K_G$, $A' = \emptyset$, and $M' = \emptyset$; then the proof is immediate by the preconditions to this lemma.

In the inductive case, there exist $\hat{K}''_{G,1}, \ldots, \hat{K}''_{G,i-1}, \hat{\beta}_{L,i-1}, \hat{\beta}_{L,i-1} A_i''$, and $M''$ such that

- $\hat{K}''_{G,1} = K_G$,
- $\hat{K}''_{G,i} = \hat{K}_{G,i}$,
- $\hat{\beta}_{L,i} = \hat{\beta}_{L,i}$,
- $dom(A_i'') = dom(\hat{\beta}_{L,i}) - dom(\hat{\beta}_{L,i})$, and
- $dom(M'') = Used(\hat{K}''_{L,i-1}) - H_L$.

Because $dom(A_i'') = dom(\hat{\beta}_{L,i-1}'') - dom(\hat{\beta}_{L,i})$ and transition steps never remove an entry from the actor-behavior map, $dom(A_i'') = dom(\hat{\beta}_{L,i-1}'') \cup ExtAbsAddr$. Similarly, because $dom(M'') = Used(\hat{K}''_{L,i-1}) - H_L$ and transition steps never remove markers from the used-marker set, $dom(M \cup M'') = Used(\hat{K}_{L,i-1}'')$. By induction and the Abstract Well-Formed Preservation, Abstract Externals-Only Preservation, and Abstract Single-Handler Preservation lemmas, $\hat{K}'_{L,i-1}$ is a well-formed, externals-only, single-handler configuration. Therefore, by the Approximation Soundness lemma, there exist $\hat{I}_{G,i-1}, \hat{K}''_{G,i}, A_i''$, and $M''$ such that
L.23. FAIR EXECUTION APPROXIMATION SOUNDNESS LEMMA

Lemma (Fair Execution Approximation Soundness). For all event-step executions $\hat{\tilde{K}}_1 \xrightarrow{l_1,1...l_{n_1}} \hat{\tilde{K}}_1$ and $\hat{\tilde{K}}'_1 \xrightarrow{\tilde{l}_1,1...\tilde{l}_{n_1}} \hat{\tilde{K}}'_1$, of the same length, and all $A$ and $M$, if

- no actor in $\hat{\tilde{K}}_1$ is handling an event,
- $\hat{\tilde{K}}_1 \xrightarrow{l_1,1...l_{n_1}} \hat{\tilde{K}}_1$ is fair,
- all of the $\hat{\tilde{K}}_i$ in the first execution are externals-only configurations,
- $\hat{\tilde{K}}_i \subseteq A, M \hat{\tilde{K}}'_i$ for all corresponding configurations in the two executions, and
- $\hat{l}_{i,j} \subseteq A, M \hat{\tilde{l}}_{i,j}$ for all corresponding labels in the two executions,

then $\hat{\tilde{K}}'_1 \xrightarrow{\tilde{l}_1,1...\tilde{l}_{n_1}} \hat{\tilde{K}}'_1$ is also fair.

Let $A' = A'' \cup A'''$, and let $M' = A'' \cup A'''$. By the transition rules, for all $j \in 1...i-2$, every internal address appearing in $\hat{\tilde{K}}_{l,j}$ must also be in $dom(\hat{\tilde{K}}_{L,i-1}')$, and every marker appearing in $\hat{\tilde{K}}_{l,j}$ must also be in $Used(\hat{\tilde{K}}''_{L,i-1})$. We already have that every abstract external address $\hat{\tilde{\alpha}}$ is in $dom(A)$. As a result, $A \cup A'$ and $M \cup M'$ give the exact same results as $A'' \cup A'''$ and $M'' \cup M'''$, respectively, for all addresses and markers appearing in $\hat{\tilde{K}}_{l,j}$. Therefore, for all $j \in 1...i-1$, $\hat{\tilde{K}}_{l,j} \subseteq A \cup A', M \cup M' \hat{\tilde{K}}_{G,i}$. Because $dom(A'') = dom(\hat{\tilde{\alpha}}''') - dom(\hat{\tilde{\alpha}}''')$ and $dom(A') = dom(\hat{\tilde{\alpha}}'') - dom(\hat{\tilde{\alpha}}'')$, $dom(A) = dom(\hat{\tilde{\alpha}}') = dom(\hat{\tilde{\alpha}}'') - dom(\hat{\tilde{\alpha}}'')$. By similar reasoning, $dom(M) = dom(\hat{\tilde{\alpha}}''') - dom(\hat{\tilde{\alpha}}''')$.

\[ dom(A''') = dom(\hat{\tilde{\alpha}}''') - dom(\hat{\tilde{\alpha}}''') \]
\[ dom(M'') = dom(\hat{\tilde{\alpha}}''') - dom(\hat{\tilde{\alpha}}''') \]

Therefore, $\hat{\tilde{K}}''_{L,i-1} = dom(\hat{\tilde{\alpha}}'') - \hat{\tilde{K}}''_{L,i-1}$, which completes the proof.

\[ \square \]
Proof. In the following, let \( \langle \beta_i \mid H_i \rangle^{\hat{\beta}_i} = K_i \) and \( \langle \beta_i \mid H_i \rangle^{\check{\beta}_i} = K'_i \) for all \( K_i \) and \( K'_i \) in the two executions. We must show that for every configuration in the second execution, every necessarily-enabled actor eventually runs or is no longer necessarily enabled, and every message with a quantity of \textit{single} is either eventually received or no longer has a quantity of \textit{single}.

First, let \( K'_i \) be a configuration in the execution, and let \( \hat{a}' \) be the address of a necessarily-enabled actor in \( K'_i \). By the definition of necessarily-enabled, there exists some \( \hat{p} \) such that

- an abstract step labeled with \( \hat{p} \) is enabled in \( K'_i \),
- \( \hat{p} \) is not a \textit{rcv-ext} label,
- if \( \hat{a}' \) is collective, then \( \hat{p} \) is not a \textit{rcv-int} or \textit{timeout} label, and
- if \( \hat{p} = \hat{a}' : \textit{rcv-int}(H', \hat{v}') \) for some \( \hat{v}' \) and \( H' \), then \( \mu_i'(\hat{a}'@H', \hat{v}') = \text{single} \).

By a precondition to this lemma, no actor in \( K_1 \) is handling an event, and by the definition of event steps, no actor in any configuration \( K_j \) for \( j > 1 \) in the first execution is handling an event. Therefore, no actor in \( K_i \) is handling an event, and by the definition of \( \sqsubseteq \), no actor in \( K'_i \) is handling an event.

As a result of the above, \( \hat{p} \) must be either a \textit{rcv-int} or \textit{timeout} label, and so \( \hat{a}' \) is atomic. Then by the definition of \( \sqsubseteq \) there exists some \( \hat{a} \) such that \( \hat{a} \sqsubseteq_A \hat{a}' \). Also by the definition of \( \sqsubseteq \), if \( \hat{p} \) is a \textit{timeout} label, then \( \bar{K} \) is similarly able to step with a \textit{timeout} step for that actor. If \( \hat{p} \) is a \textit{rcv-int} label, then we know the message being received has a quantity of \textit{single}. Then by the definition of \( \sqsubseteq \), there is a message to the actor at \( \hat{a} \) with quantity \textit{single}, so \( K_i \) is able to step with a \textit{rcv-int} step to receive that message. Thus, the actor at \( \hat{a} \) is necessarily enabled in \( \bar{K}_i \).

Because \( \bar{K}_i \) is a configuration in the second execution, let there be \( \hat{a}' \), \( H' \), and \( \hat{v}' \) such that \( \mu'_i(\hat{a}@H', \hat{v}') = \text{single} \). Then by the definition of \( \sqsubseteq \), there exist exactly one \( \hat{a} \), \( H \), and \( \hat{v} \) such that \( (\hat{a}@H, \hat{v}) \in \mu_i, \hat{a}@H \sqsubseteq_{A,M} \hat{a}@H' \), and \( \hat{v} \sqsubseteq_{A,M} \hat{v}' \). Because the first execution is fair, there exists some \( j > 0 \) such that \( \bar{K}_{i+j} - \bar{t}_{i+j,1} \ldots \bar{t}_{i+j,n} \bar{K}_{i+j+1} \) is a step in that execution and \( \bar{t}_{i+j,1} = a : \textit{rcv-int}(H, \hat{v}) \).

Because \( \bar{t}_{i+j,1} \sqsubseteq_{A,M} \bar{t}_{i+j,1} \), \( \bar{t}_{i+j,1} = \hat{a}' : \textit{rcv-int}(H', \hat{v}'') \) for some \( \hat{v}'' \) such that \( \hat{v} \sqsubseteq_{A,M} \hat{v}'' \).

If \( \hat{v}'' = \hat{v}' \), then we’re done. Otherwise, both transitions must be uses of the A-\textit{Receive\-Internal} rule. Because \( \bar{K}_i \) is an externals-only configuration, \( H = \emptyset \), and then by the definition of \( \sqsubseteq \), \( H' = \emptyset \). Then by the definitions of those transition rules, \( (\hat{a}@H, \hat{v}) \in \mu_{i+j} \) and \( (\hat{a}@H', \hat{v}') \in \text{dom}(\mu_{i+j}) \). Then because

...
\( \hat{R}_{i+j} \subseteq_{A,M} \hat{R}_{i+j}' \) must be the only element of \( \text{dom}(\hat{\mu}_{i+j}) \) such that \( \hat{\alpha}@H \subseteq_{A,M} \hat{\alpha}'@H' \) and \( \hat{\nu} \subseteq_{A,M} \hat{\nu}' \). Therefore, \( \langle \hat{\alpha}@H', \hat{\nu}' \rangle \in \text{dom}(\hat{\mu}) \), so we're done.

**L.24 Label Sequence Construction Lemma**

**Lemma** (Label Sequence Construction). For all \( L_L, L_G, \hat{I}_{L,1}, \ldots, \hat{I}_{L,n} \), and \( M \), if

- \( L_L \) summarizes \( \hat{I}_{L,1}, \ldots, \hat{I}_{L,n} \),
- \( M \) is one-to-one, and
- \( L_L \subseteq_{A,M} L_G \),

then there exist \( \hat{I}_{G,1}, \ldots, \hat{I}_{G,n} \) such that \( L_G \) summarizes \( \hat{I}_{G,1}, \ldots, \hat{I}_{G,n} \) and \( \hat{I}_{L,i} \subseteq_M \hat{I}_{G,i} \) for all \( i \in 1 \ldots n \).

**Proof.** We first define the labels \( \hat{I}_{G,1}, \ldots, \hat{I}_{G,n} \), then show that \( \hat{I}_{L,i} \subseteq_M \hat{I}_{G,i} \) for all \( i \in 1 \ldots n \), and finally show that \( L_G \) summarizes \( \hat{I}_{G,1}, \ldots, \hat{I}_{G,n} \).

**Definition of Labels**

Because \( L_L \subseteq_{M} L_G \), there exists some \( A \) such that \( L_L \subseteq_{A,M} L_G \). Define \( A' \) as the following total function.

\[
A'(\hat{\alpha}) = \begin{cases} 
A(\hat{\alpha}) & \text{if } \hat{\alpha} \in \text{dom}(A) \\
\hat{\alpha} & \text{otherwise}
\end{cases}
\]

By the definition of \( \subseteq \), \( A \) must contain a mapping for all addresses in \( L_L \). Therefore \( A' \) gives the same mappings as \( A \) for all addresses in \( L_L \), so \( L_L \subseteq_{A',M} L_G \).

By the definition of what it means to summarize a sequence of labels, there exists \( \hat{\mu}_L \) such that \( L_L = \langle \hat{I}_{L,1}, \hat{\mu}_L \rangle \). Also by the grammar for summary labels, there exist \( \hat{I}_{G,1} \) and \( \hat{\mu}_G \) such that \( L_G = \langle \hat{I}_{G,1}, \hat{\mu}_G \rangle \). For all \( i \in 2 \ldots n \), define \( \hat{I}_{G,i} \) as follows.

- If \( \hat{I}_{L,i} = \hat{\alpha}_L : \text{send-ext}(\hat{\alpha}_L'@H_L, \hat{\nu}_L) \) for some \( \hat{\alpha}_L, \hat{\alpha}_L', H_L, \) and \( \hat{\nu}_L \), then by the definition of what it means to summarize a sequence of labels, there exists some \( \hat{\nu}_L' \) such that \( \text{Merge}(\hat{\nu}_L, \hat{\nu}_L') = \hat{\nu}_L' \) and \( \langle \hat{\alpha}_L'@H_L, \hat{\nu}_L' \rangle \in \text{dom}(\hat{\mu}_L) \). Then because \( L_L \subseteq_{A',M} L_G \), which implies \( \hat{\mu}_L \subseteq_{A',M} \hat{\mu}_G \), there exist \( \hat{\alpha}'_G, H_G, \) and \( \hat{\nu}_G \) such that \( \langle \hat{\alpha}'_G@H_G, \hat{\nu}_G \rangle \in \text{dom}(\hat{\mu}_G) \), \( \hat{\alpha}'_L@H_L \subseteq_{A',M} \hat{\alpha}'_G@H_G \), and \( \hat{\nu}_L \subseteq_{A',M} \hat{\nu}_G \). Finally, let \( \hat{\alpha}_G = A'(\hat{\alpha}_L) \). Then define \( \hat{I}_{G,i} = \hat{\alpha}_G : \text{send-ext}(\hat{\alpha}'_G@H_G, \hat{\nu}_G) \).

- If \( \hat{I}_{L,i} \) is not a \text{send-ext} label, then define \( \hat{I}_{G,i} \) by applying the functions \( A' \) and \( M \) to each component of the label as necessary. For example, if \( \hat{I}_{L,i} = \hat{\alpha}_L : \text{send-int}(\hat{\alpha}_L@H_L, \hat{\nu}_L) \), then define \( \hat{I}_{G,i} = \hat{\alpha}_G : \text{send-ext}(\hat{\alpha}'_G@H_G, \hat{\nu}_G) \), where \( \hat{\alpha}_G = A'(\hat{\alpha}_L) \), \( \hat{\alpha}'_G = A'(\hat{\alpha}_L') \), \( H_G = M(H_L \cap \text{dom}(M)) \), and \( \hat{\nu}_G = A'(M(\hat{\nu}_G)) \).
APPENDIX L. CONFORMANCE-REFLECTION PROOFS

For each send-ext label $a_L$:\ send-ext$(a'_L@H_L, v_L)$, there is exactly one
$\langle a'_G@H_G, v_G \rangle \in \text{dom}(\mu_G)$ such that $a'_L@H_L \subseteq A', M a'_G@H_G$ and $v'_L \subseteq A', M v_G$, by
the following argument. By the definition of $\subseteq$, there is at most one $a'_G$ such that
$a'_L \subseteq A', a'_G$ and at most one $H_G$ such that $H_L \subseteq M H_G$. For a contradiction, suppose
there is some $v'_G \neq v_G$ such that $\langle a'_G@H_G, v'_G \rangle \in \text{dom}(\mu_G)$ and $v'_L \subseteq A', M v'_G$. Because
$\mu_L \subseteq A', M \mu_G$, $\mu_G$ is fully merged, and therefore $\text{Merge}(v_G, v'_G)$ is undefined. The
Approximation Mergeability lemma, however, implies that $\text{Merge}(v_G, v'_G)$ is defined. Therefore we have a contradiction, so there is no such $v'_G$, and the selection of $v_G$ is unique.

Label Approximation

Next, we will show that $\tilde{I}_{L,i} \subseteq M \tilde{I}_{G,i}$ for all $i \in 1 \ldots n$. Let $i$ be a number in $i \in 1 \ldots n$.
If $i = 1$, then we already know that $\tilde{I}_{L,1} \subseteq M \tilde{I}_{G,1}$ by the definition of $L_L \subseteq M L_G$. If $i > 1$ and $\tilde{I}_{L,i}$ is not a send-ext label, then by the above construction, $\tilde{I}_{G,i}$ is the
obvious correspondent of $\tilde{I}_{L,i}$ given by $A'$ and $M$. Therefore $\tilde{I}_{L,i} \subseteq A', M \tilde{I}_{G,i}$, and so
$\tilde{I}_{L,i} \subseteq M \tilde{I}_{G,i}$.

Otherwise, $\tilde{I}_{L,i} = \tilde{a}_L$:\ send-ext$(\tilde{a}_L@H_L, \tilde{v}_L)$ for some $\tilde{a}_L$, $\tilde{a}'_L$, $H_L$, and $\tilde{v}_L$. By
the above construction, there exist $\tilde{v}'_L$, $\tilde{a}_G$, $\tilde{a}'_G$, $H_G$, and $\tilde{v}_G$ such that

- $\text{Merge}(\tilde{v}, \tilde{v}') = \tilde{v}'_L$,
- $\tilde{a}'_L@H_L \subseteq A', M \tilde{a}'_G@H_G$,
- $\tilde{v}'_L \subseteq A', M \tilde{v}_G$,
- $A'(\tilde{a}_L) = \tilde{a}_G$, and
- $\tilde{I}_{G,i} = \tilde{a}_G$:\ send-ext$(\tilde{a}_G@H_G, \tilde{v}_G)$.

By the Expression Reflexivity lemma, $\tilde{v}_L \subseteq id, id \tilde{v}_L$, so by the Merge Result lemma,
$\tilde{v}_L \subseteq id, id \tilde{v}'_L$. Then by the Expression Transitivity lemma, $\tilde{v}_L \subseteq A', M \tilde{v}_G$. Finally,
because $A'(\tilde{a}_L) = \tilde{a}_G$, $\tilde{a}_L \subseteq A', \tilde{a}_G$. Therefore $\tilde{I}_{L,i} \subseteq A', M \tilde{I}_{G,i}$, and so $\tilde{I}_{L,i} \subseteq M \tilde{I}_{G,i}$.

Sequence Summary

It remains to show that $L_G$ summarizes $\tilde{I}_{G,1}, \ldots, \tilde{I}_{G,n}$. Let
$\tilde{a}_{G,1}$:\ send-ext$(\tilde{a}_{G,1}@H_{G,1}, \tilde{v}_{G,1})$, \ldots, $\tilde{a}_{G,m}$:\ send-ext$(\tilde{a}_{G,m}@H_{G,m}, \tilde{v}_{G,m})$ be the
send-ext labels in $\tilde{I}_{G,1}, \ldots, \tilde{I}_{G,n}$, and let $\tilde{\mu}_G = \phi \oplus \langle \tilde{a}_{G,1}@H_{G,1}, \tilde{v}_{G,1} \rangle \ldots \oplus \langle \tilde{a}_{G,m}@H_{G,m}, \tilde{v}_{G,m} \rangle$. We must show that $\tilde{\mu}_G \subseteq id, id \mu_G$. To do so, we will first show that $\text{dom}(\tilde{\mu}_G) \subseteq \text{dom}(\mu_G)$ and for all $\langle \tilde{a}'_G@H'_G, \tilde{v}'_G \rangle \in \text{dom}(\tilde{\mu}_G)$,
$\tilde{\mu}_G(\tilde{a}'_G@H'_G, \tilde{v}'_G) = \mu_G(\tilde{a}'_G@H'_G, \tilde{v}'_G)$. 


Properties for $\mu_G$

Let $\langle \tilde{a}_{G}^n @ H_G', \tilde{v}_G' \rangle$ be a member of $\text{dom}(\tilde{\mu}_G)$. By the definition of $\tilde{\mu}_G$, there exist $\tilde{v}_{G,1}, \ldots, \tilde{v}_{G,p}$ such that $\tilde{v}_G' = \text{Merge}(\tilde{v}_{G,1}, \ldots, \tilde{v}_{G,p})$ and for all $j \in 1 \ldots p$ there exists $i \in 1 \ldots m$ such that $\langle \tilde{a}_{G}^n @ H_{G,i}', \tilde{v}_{G,i}' \rangle = \langle \tilde{a}_G^n @ H_{G,i}, \tilde{v}_G' \rangle$. Because $\mu_G \sqsubseteq A', M \mu_G$, we know that $\tilde{\mu}_G$ is fully merged (i.e., no two distinct messages in $\text{dom}(\tilde{\mu}_G)$ are mergeable). Therefore $p = 1$, so there exists $i \in 1 \ldots m$ such that $\langle \tilde{a}_{G}^n @ H_G', \tilde{v}_G' \rangle = \langle \tilde{a}_{G,i}^n @ H_{G,i}, \tilde{v}_{G,i}' \rangle$. Therefore, $\langle \tilde{a}_{G}^n @ H_G', \tilde{v}_G' \rangle \in \text{dom}(\tilde{\mu}_G)$.

Next, let $\langle \tilde{a}_{G}^n @ H_G', \tilde{v}_G' \rangle$ be a member of $\text{dom}(\tilde{\mu}_L)$, and let $m = \tilde{\mu}_G(\tilde{a}_{G}^n @ H_G', \tilde{v}_G')$. If $m = \text{many}$, then $\tilde{\mu}_L(\tilde{a}_{G}^n @ H_G', \tilde{v}_G') \subseteq m$ by default. Otherwise, $m = \text{single}$, so it remains to show that $\tilde{\mu}_G(\tilde{a}_{G}^n @ H_G', \tilde{v}_G') = \text{single}$.

Because $\tilde{\mu}_L \sqsubseteq A', M \tilde{\mu}_G$, there exists a unique message $\langle \tilde{a}_{L}^m @ H_L', \tilde{v}_L' \rangle \in \text{dom}(\tilde{\mu}_L)$ such that $\tilde{a}_L^n @ H_L' \sqsubseteq A', M \tilde{a}_G^n @ H_G', \tilde{v}_L' \subseteq A', M \tilde{v}_G'$, and $\tilde{\mu}_L(\tilde{a}_L^m @ H_L', \tilde{v}_L') = \text{single}$. By the definition of what it means to summarize a sequence of labels, there exists a unique $i \in 1 \ldots n$ and some $\tilde{a}_L^n$, $\tilde{v}_L'$ such that $\tilde{L}_{L,i} = \tilde{a}_L^n : \text{send-ext}(\tilde{a}_L^m @ H_L', \tilde{v}_L')$ and $\tilde{v}_L' \sqsubseteq \text{id}_{id} \tilde{v}_L'$. Then by the Expression Transitivity lemma, $\tilde{v}_L' \sqsubseteq A', M \tilde{v}_G'$. To show that $\tilde{\mu}_G(\tilde{a}_L^m @ H_L', \tilde{v}_L') = \text{single}$, we must also show that there is no other $\text{send-ext}$ label in $\tilde{L}_{L,1}, \ldots, \tilde{L}_{L,n}$ with a message approximated by $\langle \tilde{a}_{L}^m @ H_L', \tilde{v}_L' \rangle$. For a contradiction, suppose there is some $j \neq i$ in $1 \ldots n$ and some $\tilde{a}_L^m$, $\tilde{v}_L'$, and $\tilde{v}_L''$ such that $\tilde{L}_{L,j} = \tilde{a}_L^m : \text{send-ext}(\tilde{a}_L^m @ H_L', \tilde{v}_L'')$, $\tilde{a}_L^m @ H_L' \sqsubseteq A', M \tilde{a}_G^n @ H_G'$, and $\tilde{v}_L'' \subseteq A', M \tilde{v}_G'$. By the definition of what it means to summarize a sequence of labels, there exists some $\tilde{v}_L'''$ such that $\langle \tilde{a}_L^m @ H_L'', \tilde{v}_L''' \rangle \in \text{dom}(\tilde{\mu}_L)$ and $\tilde{v}_L''' \sqsubseteq \text{id}_{id} \tilde{v}_L'''$. Because $\tilde{\mu}_L \sqsubseteq A', M \tilde{\mu}_G$, there must exist some $\tilde{v}_G''$ such that $\langle \tilde{a}_G^n @ H_G', \tilde{v}_G'' \rangle \in \text{dom}(\tilde{\mu}_G)$, $\tilde{v}_G'' \subseteq A', M \tilde{v}_G'$, and $\tilde{\mu}_G(\tilde{a}_G^n @ H_G', \tilde{v}_G'') = \tilde{\mu}_G(\tilde{a}_L^m @ H_L'', \tilde{v}_L''')$. By the Expression Transitivity lemma, $\tilde{v}_G'' \sqsubseteq A', M \tilde{v}_G'$, and then by the Approximation Mergeability lemma, $\text{Merge}(\tilde{v}_G', \tilde{v}_G'')$ is defined. The map $\tilde{\mu}_G$ is fully merged, so it must be the case that $\tilde{v}_G' = \tilde{v}_G''$. Therefore because $\tilde{\mu}_L(\tilde{a}_L^m @ H_L', \tilde{v}_L'') \subseteq \tilde{\mu}_G(\tilde{a}_L^m @ H_G', \tilde{v}_G'')$, $\tilde{\mu}_L(\tilde{a}_L^m @ H_L', \tilde{v}_L'') = \text{single}$. Again because $\tilde{\mu}_L \sqsubseteq A', M \tilde{\mu}_G$, there exists a unique message in $\tilde{\mu}_L$ with a quantity of single that is approximated by $\langle \tilde{a}_{G}^n @ H_G', \tilde{v}_G' \rangle$. We already know that message is $\langle \tilde{a}_{L}^m @ H_L', \tilde{v}_L' \rangle$, so $\langle \tilde{a}_L^m @ H_L', \tilde{v}_L''' \rangle = \langle \tilde{a}_{L}^m @ H_L', \tilde{v}_L' \rangle$. This is a contradiction, though, because it implies there are two separate $\text{send-ext}$ labels summarized by $\langle \tilde{a}_{L}^m @ H_L', \tilde{v}_L' \rangle$, but $\tilde{\mu}_L(\tilde{a}_L^m @ H_L', \tilde{v}_L') = \text{single}$. Therefore, $\tilde{L}_{L,i}$ is the only $\text{send-ext}$ label in $\tilde{L}_{L,1}, \ldots, \tilde{L}_{L,n}$ with a message approximated by $\langle \tilde{a}_{L}^m @ H_L', \tilde{v}_L' \rangle$.

We have already shown that $\tilde{L}_{L,i} \sqsubseteq A', M \tilde{L}_{G,i}$, so $\tilde{L}_{G,i} = \tilde{a}_G^n : \text{send-ext}(\tilde{a}_G^n @ H_G', \tilde{v}_G')$ for some $\tilde{a}_G^n = A'(\tilde{a}_G^n)$ and some $\tilde{v}_G'$ such that $\tilde{v}_G' \sqsubseteq A', M \tilde{v}_G'$. By the construction of $\tilde{L}_{G,i}$, $\langle \tilde{a}_G^n @ H_G', \tilde{v}_G' \rangle \in \text{dom}(\tilde{\mu}_G)$. As argued in the Definition of Labels section above, there is only one message in $\text{dom}(\tilde{\mu}_G)$ that approximates $\langle \tilde{a}_G^n @ H_G', \tilde{v}_G' \rangle$, so $\tilde{v}_G' = \tilde{v}_G$.
tion, assume there is some \( k \) \( \neq i \) in \( 1 \ldots n \) and some \( \hat{a}_G^{\prime \prime} \) and \( \hat{v}_G^{\prime \prime} \) such that \( \hat{I}_{G,k} = \hat{a}_G^{\prime \prime} \): \( \mathsf{send-ext}(\hat{a}_G^{\prime \prime} @ H_G', \hat{v}_G^{\prime \prime}) \) and \( \mathsf{Merge}(\hat{v}_G, \hat{v}_G^{\prime \prime}) \) is defined. We know that 
\[
\langle \hat{a}_G^{\prime \prime} @ H_G', \hat{v}_G^{\prime \prime} \rangle
\]
must be a member of \( \mathsf{dom}(\hat{\mu}_G) \), and because \( \hat{\mu}_G \) is fully merged, 
\( \hat{v}_G^{\prime \prime} = \hat{v}_G \). Because \( \hat{I}_L,k \subseteq A' \cup M \), \( \hat{I}_L,k = \hat{a}_L^{\prime \prime \prime} \): \( \mathsf{send-ext}(\hat{a}_L^{\prime \prime \prime} @ H_L^{\prime \prime \prime}, \hat{v}_L^{\prime \prime \prime}) \) for some \( \hat{a}_L^{\prime \prime \prime} \), \( \hat{v}_L^{\prime \prime \prime} \), \( H_L^{\prime \prime \prime} \), and \( \hat{v}_L^{\prime \prime \prime} \) such that \( \hat{a}_L^{\prime \prime \prime} @ H_L^{\prime \prime \prime} \subseteq A' \cup M \) \( \hat{a}_G^{\prime \prime} @ H_G' \) and \( \hat{v}_L^{\prime \prime \prime} \subseteq A' \cup M \) \( \hat{v}_G \). This is a contradiction, though, because we have already established that \( \hat{I}_{L,i} \) is the only \( \mathsf{send-ext} \) label in \( \hat{I}_{L,1}, \ldots, \hat{I}_{L,n} \) with a message approximated by 
\[
\langle \hat{a}_G^{\prime \prime} @ H_G', \hat{v}_G \rangle.
\]
Therefore there is no such \( k \), and so by the definition of \( \hat{\mu}_G' \), 
\( \hat{\mu}_G'(\hat{a}_G^{\prime \prime} @ H_G', \hat{v}_G) = \mathsf{single} \).

**Approximation of \( \hat{\mu}_G' \)**

It remains to show that \( \hat{\mu}_G' \subseteq \mathsf{id}, \hat{\mu}_G \). First, let \( \langle \hat{a}_G^{\prime \prime} @ H_G', \hat{v}_G \rangle \) be a member of 
\( \mathsf{dom}(\hat{\mu}_G) \). We know that 
\[
\langle \hat{a}_G^{\prime \prime} @ H_G', \hat{v}_G \rangle \in \mathsf{dom}(\hat{\mu}_G)
\]
and by two uses of the Expression Reflexivity lemma, \( \hat{a}_G^{\prime \prime} @ H_G', \hat{v}_G \subseteq A', \hat{\mu}_G \). Also, we established above that \( \hat{\mu}_G'(\hat{a}_G^{\prime \prime} @ H_G', \hat{v}_G) = \mathsf{single} \).

To show that 
\[
\langle \hat{a}_G^{\prime \prime} @ H_G', \hat{v}_G \rangle
\]
...
that $\hat{a}''@H'' \sqsubseteq_{id{id}} \hat{a}''@H'' \hat{v}'' \sqsubseteq_{id{id}} \hat{v}''$, and $\hat{\mu}'_G(\hat{a}''@H'', \hat{v}'') = \text{single}$. Because $id$ is one-to-one, by the definition of $\sqsubseteq$, $\hat{a}'' = \hat{a}''$ and $H'' = H''$. By the Expression Reflexivity lemma, $\hat{v}'' \sqsubseteq_{id{id}} \hat{v}''$, and therefore by the Approximation Mergeability lemma, $\text{Merge}(\hat{v}'', \hat{v}'')$ is defined. Because $\hat{\mu}_G$ is fully merged, it must be the case that $\hat{v}'' = \hat{v}''$.

Third, because $\hat{\mu}_L \sqsubseteq_{A',M} \hat{\mu}_G$, we know that $\hat{\mu}_G$ is fully merged. Therefore $\hat{\mu}'_G \sqsubseteq_{id{id}} \hat{\mu}_G$, so $L_G$ summarizes $\hat{l}_G, \ldots, \hat{l}_G$.

\[ \square \]

**L.25 Abstract Configuration Completeness Lemma**

**Lemma** (Abstract Configuration Completeness). For all $S_L, S_G, S'_G, M, \hat{\lambda}_L, \hat{\lambda}_G$, and $O_G$, if

- $M$ is one-to-one,
- $M(S_L) = S_G$,
- $\hat{\lambda}_L \sqsubseteq M \hat{\lambda}_G$,
- if there exist $\hat{a}, H, \hat{v}$ such that $\hat{\lambda}_L = \hat{a}@H?\hat{v}$ or $\hat{\lambda}_L = \hat{a}@H!\hat{v}$, then for all $\hat{a}'@H'$ in $\hat{v}$ outside of any list or dict expression such that $\hat{a}'$ is internal or $\hat{\lambda}_L = \hat{a}@H?\hat{v}$, $|H'| \leq 1$, and
- $S_G \xrightarrow{\hat{\lambda}_G,O_G} S'_G$,

then there exists $S'_L$ and $O_L$ such that

- $S_L \xrightarrow{\hat{\lambda}_L,O_L} S'_L$,
- $M(O_L) = O_G$, and
- $M(S'_L) = S'_G$.

**Proof.** The proof is similar to the proof of the Configuration Completeness lemma from appendix I. The only difference is that the pattern matching for is on abstract values for both transitions, rather than a concrete value for one and an abstract value for the other. Nevertheless, it is easy to prove analogs of the Input Pattern and Output Pattern lemmas from that appendix that apply when both values are abstract, so the appropriate $S'_L$ and $O_L$ exist. \[ \square \]

**Corollary L.25.1.** For all $S_L, S_G, S'_G$, and $M$, if

- $M$ is one-to-one,
- $M(S_L) = S_G$,
- $S_G \xrightarrow{\cdot} S'_G$, 

then there exists $S'_L$ and $O_L$ such that

- $S_L \xrightarrow{\hat{\lambda}_L,O_L} S'_L$,
- $M(O_L) = O_G$, and
- $M(S'_L) = S'_G$. 

**Proof.** The proof is similar to the proof of the Configuration Completeness lemma from appendix I. The only difference is that the pattern matching for is on abstract values for both transitions, rather than a concrete value for one and an abstract value for the other. Nevertheless, it is easy to prove analogs of the Input Pattern and Output Pattern lemmas from that appendix that apply when both values are abstract, so the appropriate $S'_L$ and $O_L$ exist. \[ \square \]
then there exists \( S'_L \) such that

- \( S_L \xrightarrow{a} S'_L \) and
- \( M(S'_L) = S'_G \).

**Proof.** Analogous to the proof of corollary I.34.1. \( \square \)

**Corollary L.25.2.** For all \( S_L, S_G, S'_G, M, \hat{\lambda}_L, \hat{\lambda}_G \), and \( O_G \), if

- \( M \) is one-to-one,
- \( M(S_L) = S_G \),
- \( \hat{\lambda}_L \subseteq_M \hat{\lambda}_G \),
- if there exist \( \hat{a}, H, \) and \( \hat{\nu} \) such that \( \hat{\lambda}_L = \hat{a}@H?\hat{\nu} \) or \( \hat{\lambda}_L = \hat{a}@H'\hat{\nu} \), then for all \( \hat{a}'@H' \) in \( \hat{\nu} \) outside of any list or dict expression such that \( \hat{a}' \) is internal or \( \hat{\lambda}_L = \hat{a}@H?\hat{\nu}, |H'| \leq 1 \),
- \( S_G \xrightarrow{\hat{\lambda}_G,O_G} S'_G \),

then there exists \( S'_L \) and \( O_L \) such that

- \( S_L \xrightarrow{\hat{\lambda}_L,O_L} S'_L \),
- \( M(O_L) = O_G \), and
- \( M(S'_L) = S'_G \).

**Proof.** Analogous to the proof of corollary I.34.2. \( \square \)

**Corollary L.25.3.** For all \( S_L, S_G, S'_G, M, \hat{\lambda}_{L,1}, \ldots, \hat{\lambda}_{L,n}, \hat{\lambda}_{G,1}, \ldots, \hat{\lambda}_{G,n}, \) and \( O_{G,1}, \ldots, O_{G,n} \), if

- \( M \) is one-to-one,
- \( M(S_L) = S_G \),
- \( \hat{\lambda}_{L,i} \subseteq_M \hat{\lambda}_{G,i} \) for all \( i \in 1 \ldots n \),
- for all \( i \in 1 \ldots n \), if there exist \( \hat{a}, H, \) and \( \hat{\nu} \) such that \( \hat{\lambda}_{L,i} = \hat{a}@H?\hat{\nu} \) or \( \hat{\lambda}_{L,i} = \hat{a}@H'\hat{\nu} \), then for all \( \hat{a}'@H' \) in \( \hat{\nu} \) outside of any list or dict expression such that \( \hat{a}' \) is internal or \( \hat{\lambda}_{L,i} = \hat{a}@H?\hat{\nu}, |H'| \leq 1 \), and
- \( S_G \xrightarrow{\langle \hat{\lambda}_{G,1},O_{G,1} \rangle, \ldots, \langle \hat{\lambda}_{G,n},O_{G,n} \rangle} S'_G \),

then there exist \( S'_L \) and \( O_{L,1}, \ldots, O_{L,n} \) such that

- \( S_L \xrightarrow{\langle \hat{\lambda}_{L,1},O_{L,1} \rangle, \ldots, \langle \hat{\lambda}_{L,n},O_{L,n} \rangle} S'_L \).
L.25. ABSTRACT CONFIGURATION COMPLETENESS LEMMA

- \( M(O_L) = O_G \) for all \( i \in 1 \ldots n \), and
- \( M(S'_L) = S'_G \).

Proof. By a simple induction on \( n \).

**Corollary L.25.4.** For all \( S_L, S_G, S'_G, M, L_G, \tilde{l}_{L,1}, \ldots, \tilde{l}_{L,n}, \) and \( \tilde{l}_{G,1}, \ldots, \tilde{l}_{G,n} \), if

- \( M \) is one-to-one,
- \( M(S_L) = S_G \),
- \( \tilde{l}_{L,i} \subseteq \tilde{l}_{G,i} \) for all \( i \in 1 \ldots n \),
- for all \( i \in 1 \ldots n \), if there exist \( \hat{a}, H, \) and \( \hat{v} \) such that \( |\tilde{l}_{L,i}| = \hat{a}@H?\hat{v} \) or \( |\tilde{l}_{L,i}| = \hat{a}@H!!\hat{v} \), then for all \( \hat{a}@H' \) in \( \hat{v} \) outside of any list or dict expression such that \( \hat{a}' \) is internal or \( \hat{a}' \) is fully merged by definition, so by the Message-Map Reflexivity lemma, \( \hat{a}' \leq 1 \), and
  - \( S_G \xrightarrow{(L_G,O_G)} S'_G \),
  - \( L_G \) summarizes \( \tilde{l}_{G,1}, \ldots, \tilde{l}_{G,n} \),

then there exist \( S'_L, L_L, \) and \( O_L \) such that

- \( S_L \xrightarrow{(L_L,O_L)} S'_L \),
- \( M(O_L) = O_G \),
- \( M(S'_L) = S'_G \), and
- \( L_L \) summarizes \( \tilde{l}_{L,1}, \ldots, \tilde{l}_{L,n} \).

Proof. First, we must construct the label \( L_L \). Let \( \hat{a}_1 : \text{send-ext}(\hat{a}'_1@H_1,\hat{v}_1), \ldots, \hat{a}_m : \text{send-ext}(\hat{a}'_m@H_m,\hat{v}_m) \) be the send-ext labels in \( \tilde{l}_{L,1}, \ldots, \tilde{l}_{L,n} \). Define \( \hat{\mu}_L = \phi \oplus (\hat{a}'_1@H_1,\hat{v}_1) \ldots \oplus (\hat{a}'_m@H_m,\hat{v}_m) \). Then define \( L_L = (\tilde{l}_{L,1},\hat{\mu}_L) \).

Because \( L_G \) summarizes \( \tilde{l}_{G,1}, \ldots, \tilde{l}_{G,n} \), there is no rcv-ext label in \( \tilde{l}_{G,1}, \ldots, \tilde{l}_{G,n} \), and therefore by the definition of \( \sqsubseteq \), there is no rcv-ext label in \( \tilde{l}_{L,1}, \ldots, \tilde{l}_{L,n} \). The map \( \hat{\mu}_L \) is fully merged by definition, so by the Message-Map Reflexivity lemma, \( \hat{\mu}_L \sqsubseteq \hat{\mu}_L \). Therefore, \( L_L \) summarizes \( \tilde{l}_{L,1}, \ldots, \tilde{l}_{L,n} \).

Next, we must show there exist \( O_L \) and \( S'_L \) such that \( S_L \xrightarrow{(L_L,O_L)} S'_L, M(O_L) = O_G, \) and \( M(S'_L) = S'_G \). Let there be some \( \tilde{l}_{L,1}, \ldots, \tilde{l}_{L,m} \) summarized by \( L_L \). We will show that \( L_G \) summarizes a similar label sequence \( \tilde{l}_{G,1}, \ldots, \tilde{l}_{G,m} \) by the Label Sequence Construction lemma, then use corollary L.25.3 to get the desired result.

To use that lemma, we must show that \( L_L \sqsubseteq_M L_G \). We already know that \( \tilde{l}_{L,1} \subseteq_M \tilde{l}_{G,1} \). Let \( \hat{\mu}' \) be constructed from the send-ext labels in \( \tilde{l}_{G,1}, \ldots, \tilde{l}_{G,n} \) in the same manner as \( \hat{\mu} \). By induction on the number of those labels and the Message-Addition Soundness lemma (appendix I), \( \hat{\mu} \sqsubseteq \hat{\mu}' \). Then let \( (\tilde{l}''',\hat{\mu}'') = L_G \). Because \( L_G \) summarizes \( \tilde{l}_{G,1}, \ldots, \tilde{l}_{G,n} \), it must be the case that \( \hat{\mu}' \sqsubseteq_{\text{id},ld} \hat{\mu}'' \), so
by the Message-Map Transitivity lemma (also appendix I), \( \tilde{\mu} \subset_M \tilde{\mu}'' \). Therefore, \( L_L \subset_M L_G \).

Now by the Label Sequence Construction lemma, there exist \( \hat{\pi}_{G,1}, \ldots, \hat{\pi}_{G,m} \) such that \( L_G \) summarizes \( \hat{\pi}_{G,1}, \ldots, \hat{\pi}_{G,m} \) and \( \hat{\pi}_{L,i} \subset_M \hat{\pi}_{gss,i} \) for all \( i \in 1 \ldots m \).

To use corollary L.25.3, we must show a similar restriction on the markers on \( \hat{\pi}_{L,1}, \ldots, \hat{\pi}_{L,m} \) as we have for \( \hat{\pi}_{L,1}, \ldots, \hat{\pi}_{L,n} \). Let \( \hat{\pi}_{L,j} \) be a label in \( \hat{\pi}_{L,1}, \ldots, \hat{\pi}_{L,m} \) such that there exist \( \tilde{a}, H, \) and \( \tilde{v} \) such that \( [\hat{\pi}_{L,j}] = \tilde{a}@H?\tilde{v} \) or \( [\hat{\pi}_{L,j}] = \tilde{a}@H!\tilde{v} \). In the former case, \( \hat{\pi}_{L,j} \) must be a \texttt{rcv-ext} label, and therefore \( \hat{\pi}_{L,1} = \hat{\pi}_{L,1} \) by the definition of what it means for \( L_L \) to summarize \( \hat{\pi}_{L,1}, \ldots, \hat{\pi}_{L,m} \). Then by the precondition to this lemma, for all \( \tilde{a}@H' \) in \( \tilde{v} \) outside of any \texttt{list} or \texttt{dict} expression, \( |H'| \leq 1 \). In the latter case, there exists some \( \tilde{v}' \) such that \( \text{Merge}(\tilde{v}, \tilde{v}') \) is defined and \( \langle \tilde{a}@H', \tilde{v}' \rangle \in \text{dom}(\tilde{\mu}) \). Then by the definition of \( \tilde{\mu} \), there exist \( \tilde{v}'' \) and \( j \in 1 \ldots n \) such that \( [\hat{\pi}_{L,j}] = \tilde{a}@H'?\tilde{v}'' \) and \( \text{Merge}(\tilde{v}', \tilde{v}'') \) is defined. Again by the precondition to this lemma, for all \( \tilde{a}@H' \) in \( \tilde{v}' \) outside of any \texttt{list} or \texttt{dict} expression such that \( \tilde{a} \) is internal, \( |H'| \leq 1 \), so by the definition of \( \text{Merge} \), the same holds for \( \tilde{v} \).

By the definition of \( \langle L_G, O_G \rangle \rightarrow S_G' \), there exist \( O'_{G,1}, \ldots, O'_{G,m} \) such that

- \( S_G \xrightarrow{\langle L_G, O_G \rangle} S_G' \)
- \( O_G \subset O'_{G,1} \cup \ldots \cup O'_{G,m} \)
- for all \( \langle \eta, po \rangle \in \text{Obls}(S_G) \), \( \langle \eta, po \rangle \in \text{Obls}(S_G') \cup O'_G \), and
- for all \( s \in S'_G \), \( \text{Mon}(s) \neq \emptyset \).

Thus, by corollary L.25.3, there exist \( S'_L \) and \( O'_{L,1}, \ldots, O'_{L,m} \) such that

- \( S_L \xrightarrow{\langle L_L, O_L \rangle} S'_L \)
- \( \text{Mon}(O'_{L,i}) = O'_{G,i} \) for all \( i \in 1 \ldots m \), and
- \( \text{Mon}(S'_L) = S'_G \).

Because \( O_G \subset O'_{G,1} \cup \ldots \cup O'_{G,m} \) and \( \text{Mon}(O'_{L,i}) = O'_{G,i} \) for all \( i \in 1 \ldots m \), there must exist some \( O_L \) such that \( \text{Mon}(O_L) = O_G \). Because \( M \) is one-to-one, \( O_L \) and \( S'_L \) must be the same for all label sequences \( \hat{\pi}_{L,1}, \ldots, \hat{\pi}_{L,m} \) summarized by \( L_L \). Because \( \langle \eta, po \rangle \in \text{Obls}(S'_G) \cup O'_G \) for all \( \langle \eta, po \rangle \in \text{Obls}(S_G) \), and because \( \text{Mon}(O_L) = O_G \), we have that \( \langle \eta', po' \rangle \in \text{Obls}(S'_L) \cup O'_L \) for all \( \langle \eta', po' \rangle \in \text{Obls}(S_L) \). Finally, because \( \text{Mon}(s) \neq \emptyset \) for all \( s \in S'_G \) and \( \text{Mon}(S'_L) = S'_G \), we have \( \text{Mon}(s') \neq \emptyset \) for all \( s' \in S'_L \).

Therefore, \( S_L \xrightarrow{\langle L_L, O_L \rangle} S'_L \), which completes the proof. \( \Box \)
L.26 Approximating Transformation Lemma

Several of the transformations used (in particular, Unmark, Assimilate, Canonicalize, and Accelerate) simply return an approximation of the given program configuration. It turns out that we can use the same techniques to show that every such transformation is conformance-reflecting (as long as it also preserves the well-formed and externals-only properties). This lemma formalizes and proves that fact.

**Lemma** (Approximating Transformation). For all transformations $T$, if $T$ is an approximating, well-formed-preserving, externals-only-preserving, single-message-reflecting transformation, then $T$ is conformance-reflecting.

**Proof.** We must first define the functions $\text{TransExec}_T$ and $\text{UntransExec}_T$, then show that all of the necessary properties hold.

**Definition of $\text{TransExec}_T$**

For $\text{TransExec}_T$, let there be $K_1\xrightarrow{l_{1,1},\ldots,l_{1,m}}\ldots$ and $s$ such that $K_1$ and $s$ are well-formed, $K_1$ is an externals-only configuration, $\text{Mon}(s) \subseteq \text{Used}(K_1)$, and no actor in $K_1$ is handling an event (note that this implies $K_1$ is a single-handler configuration). We define the result of $\text{TransExec}_T(K_1\xrightarrow{l_{1,1},\ldots,l_{1,m}}\ldots,s)$ by repeated uses of the Event-Step Approximation Soundness lemma as follows.

Because $T$ is an approximating transformation, there exist some $K'_1$, $s'$, $A$, and $M$ such that $T(K_1,s) = \langle (K'_1,s',A,M) \rangle$, $K_1 \subseteq_{A,M} K'_1$, and $M(s) = s'$. Let $\hat{\beta}$ be the actor-behavior component of $K_1$, then let $A' = A|_{\text{dom}(\hat{\beta}) \cap \text{ExtAbsAddr}}$, and $M' = M|_{\text{Used}(K_1)}$. Because $K_1$ is well-formed, all internal addresses in $K_1$ are members of $\text{dom}(K_1)$, and all markers in $K_1$ are members of $\text{Used}(K_1)$, so $K_1 \subseteq_{A',M'} K'_1$. Then because $s$ is well-formed, all markers in $s$ are members of $\text{Mon}(s)$, and we know that $\text{Mon}(s) \subseteq \text{Used}(K_1)$, so $M'(s) = s'$.

Therefore, define $\text{TransExec}_T(K_1\xrightarrow{l_{1,1},\ldots,l_{1,m}}\ldots,s) = \langle K'_1\xrightarrow{l'_{1,1},\ldots,l'_{1,n}}\ldots,s',A,M \rangle$, where $K'_1\xrightarrow{l'_{1,1},\ldots,l'_{1,n}}$ is the execution constructed inductively by, at each step, taking the next event-step as described by the Event-Step Approximation Soundness lemma. (Note that by the results of that lemma, each such step preserves the necessary $\sqsubseteq$ preconditions and the conditions on $\text{dom}(A)$ and $\text{dom}(M)$, and the well-formedness, externals-only, and single-handler preconditions are preserved by the Abstract Well-Formed Preservation, Abstract Externals-Only Preservation, and Abstract Single-Handler Preservation lemmas.) Assume that each next step is chosen deterministically, so that if some execution $K'_1\xrightarrow{l''_{1,1},\ldots,l''_{1,m}}\ldots$ shares a prefix of length $i$ with $K_1\xrightarrow{l_{1,1},\ldots,l_{1,m}}\ldots$, then the execution returned by $\text{TransExec}_T(K'_1\xrightarrow{l''_{1,1},\ldots,l''_{1,m}}\ldots,s)$ shares a prefix of length $i$ with the one returned by $\text{TransExec}_T(K_1\xrightarrow{l_{1,1},\ldots,l_{1,m}}\ldots,s)$. 

L.26. APPROXIMATING TRANSFORMATION LEMMA 527
By the Event-Step Approximation Soundness lemma, each such new step $\hat{R}_i' \xrightarrow{\hat{R}_{i,1},...,\hat{R}_{i,n}} \hat{R}'_{i+1}$ also defines new correspondence functions $A''_i$ and $M''_i$ such that

$$\hat{K}_{i+1} \subseteq A''_i \cup M''_i \cup \cdots \cup A''_1 \cup M''_1 \cup \cdots \cup M''_i \cup M''_{i+1} \hat{R}'_{i+1}$$

and

$$\hat{I}_{i,j} \subseteq A''_i \cup M''_i \cup \cdots \cup A''_1 \cup M''_1 \cup \cdots \cup M''_i \cup M''_{i+1} \hat{I}'_{i,j}$$

for all $j \in 1 \ldots n$.

Let $A''$ and $M''$ be the disjoint union of all such $A''_i$ and $M''_i$, respectively, defined by those steps. By the Event-Step Approximation Soundness lemma, each new $A''_i$ and $M''_i$ maps only the new internal addresses or markers, respectively, introduced in that step, so we have $\hat{K}_i \subseteq A'' \cup M'' \hat{K}_i'$ for all corresponding configurations in the two executions and $\hat{I}_{i,j} \subseteq A'' \cup M'' \hat{I}_{i,j}'$ for all corresponding labels in the two executions. By the same reasoning, we also have $(M' \cup M'')(s) = s$.

**Definition of UntransExec**

For $\text{UntransExec}_T$, let there be $\hat{K}_1 \xrightarrow{\hat{I}_{1,1},\ldots,\hat{I}_{1,m}} \ldots, s$, and $X$ such that

- $\hat{K}_1$ and $s$ are well-formed,
- $\hat{K}_1$ is an externals-only configuration,
- $\text{Mon}(s) \subseteq \text{Used}(\hat{K}_1)$,
- no actor in $\hat{K}_1$ is handling an event, and
- $X$ contains a simulation for every member of $\text{TransExec}_T(\hat{K}_1 \xrightarrow{\hat{I}_{1,1},\ldots,\hat{I}_{1,m}} \ldots, s)$.

We must define $\text{UntransExec}_T(\hat{K}_1 \xrightarrow{\hat{I}_{1,1},\ldots,\hat{I}_{1,m}} \ldots, s, X)$.

By the definition of $\text{TransExec}_T$ above, there exist some $\overline{a\bar{e}x}$, $s'$, $A$ and $M$ such that $\text{TransExec}_T(\hat{K}_1 \xrightarrow{\hat{I}_{1,1},\ldots,\hat{I}_{1,m}} \ldots, s) = \{\langle \overline{a\bar{e}x}, s', A, M \rangle \}$, and $\overline{a\bar{e}x}$ and $\hat{K}_1 \xrightarrow{\hat{I}_{1,1},\ldots,\hat{I}_{1,n}} \ldots$ have the same length. Let $\hat{R}_1' \xrightarrow{\hat{R}_{1,1},\ldots,\hat{R}_{1,n}} \ldots = \overline{a\bar{e}x}$, and let $\hat{\beta}$ be the actor-behavior map from $\hat{K}_1$. Then we furthermore know from the above definition of $\text{TransExec}_T$ that there exist $A'$, $A''$, $M'$, and $M''$ such that

- $A' = A|_{\text{dom}(\hat{\beta}) \cap \text{ExtAbsAddr}}$,
- $M' = M|_{\text{Used}(\hat{K}_1)}$,
- $\hat{K}_i \subseteq A'' \cup M'' \hat{K}_i'$ for all corresponding configurations in the two executions,
- $\hat{I}_{i,j} \subseteq A'' \cup M'' \hat{I}_{i,j}'$ for all corresponding labels in the two executions, and
• \((M' \cup M'')(s) = s'.\)

Let \(S'_1 = (L'_i, O'_i) \rightarrow \ldots = X(\bar{a}\bar{x}, s', A, M).\) Because \(X\) contains a simulation for \(\langle \bar{a}\bar{x}, s', A, M, M'\rangle\), we know that \(S'_1 = \{s'\}, S'_1 = (L'_i, O'_i) \rightarrow \ldots\) has the same length as \(\bar{a}\bar{x}\), and \(\hat{K}'_1 \rightarrow \ldots\) summarizes \(\bar{a}\bar{x}\). By the definition of \(\hat{K}'_1 \subseteq A'^{\prime} \cup M' \cup M'\) \(\hat{K}'_1, M' \cup M'\) is one-to-one. By corollary K.7.2 to the Externals-Only Label lemma, for every label \(\hat{l}_{i,j}\) in \(\hat{K}'_1 \rightarrow \hat{l}_{i_1,1} \rightarrow \ldots \rightarrow \hat{l}_{i_n,1} \rightarrow \ldots\), if \([\hat{l}_{i,j}] = \hat{a}\hat{a}H?\hat{v}\) or \([\hat{l}_{i,j}] = \hat{a}\hat{a}H!\hat{v}\) for some \(\hat{a}, H,\) and \(\hat{v}\), then for all \(\hat{a}'\hat{a}H'\) in \(\hat{v}\) such that \(\hat{a}'\) is internal or \(\hat{a}'\hat{a}H'\hat{v}\), \(|H'| \leq 1\). Then by repeated uses of corollary L.25.4 to the Abstract Configuration Completeness lemma, there exists an execution \(S_1 \rightarrow (L_i, O_i) \rightarrow \ldots\) with the same length as \(S'_1 = (L'_i, O'_i) \rightarrow \ldots\) such that

• \(S_1 = \{s]\),

• \(\hat{K}'_1 \rightarrow \ldots\) summarizes \(\hat{K}_1 \rightarrow \hat{l}_{i_1,1} \rightarrow \ldots \rightarrow \hat{l}_{i_m,1} \rightarrow \ldots\),

• \((M \cup M')(O_i) = O'_i\) for all corresponding obligation multisets in the two executions, and

• \((M \cup M')(S_1) = S'_1\) for all corresponding configurations in the two executions.

Thus, define \(\text{UntransExec}(\hat{K}_1 \rightarrow \hat{l}_{i_1,1} \rightarrow \ldots \rightarrow \hat{l}_{i_m,1} \rightarrow \ldots, s, X) = S_1 \rightarrow (L_i, O_i) \rightarrow \ldots\)

Assume that each next step from corollary L.25.4 is chosen deterministically, so that if there exist \(\hat{K}'_1 \rightarrow \hat{l}_{i_1,1} \rightarrow \ldots \rightarrow \hat{l}_{i_m,1} \rightarrow \ldots \rightarrow X', \bar{a}\bar{x}',\) and \(i\) such that

• \(\text{TransExec}(\hat{K}'_1 \rightarrow \hat{l}_{i_1,1} \rightarrow \ldots \rightarrow \hat{l}_{i_m,1} \rightarrow \ldots, s) = \{\langle \bar{a}\bar{x}', s', A, M\rangle\}\),

• \(\hat{K}_1 \rightarrow \hat{l}_{i_1,1} \rightarrow \ldots \rightarrow \hat{l}_{i_m,1} \rightarrow \ldots\) and \(\hat{K}'_1 \rightarrow \hat{l}_{i_1,1} \rightarrow \ldots \rightarrow \hat{l}_{i_m,1} \rightarrow \ldots\) share a prefix of length \(i,\) and

• \(X(\bar{a}\bar{x}, s', A, M)\) and \(X'(\bar{a}\bar{x}', s', A, M)\) share a prefix of length \(i,\)

then \(\text{UntransExec}(\hat{K}_1 \rightarrow \hat{l}_{i_1,1} \rightarrow \ldots \rightarrow \hat{l}_{i_m,1} \rightarrow \ldots, s, X)\) shares a prefix of length \(i\) with \(\text{UntransExec}(\hat{K}'_1 \rightarrow \hat{l}_{i_1,1} \rightarrow \ldots \rightarrow \hat{l}_{i_m,1} \rightarrow \ldots, s, X').\)

**Property Proofs**

It remains to prove each of the properties necessary for \(T\) to be conformance-reflecting, which we do below.

**Properties for \(T\)**

**Well-Formed Preservation** By the definition of a well-formed-preserving transformation.
**Externals-Only Preservation** By the definition of an externals-only-preserving transformation.

**All-Awaiting Preservation** Let there be \( \hat{K} \) and \( s \) such that they are both well-formed and no actor in \( \hat{K} \) is handling an event. Furthermore, let \( \langle \hat{K}', s', A, M \rangle \) be a member of \( T(\hat{K}, s) \). By the definition of an approximating transformation, \( \hat{K} \sqsubseteq_{A,M} \hat{K}' \). By the definition of the \( \sqsubseteq \) relation, every behavior that is handling an event in \( \hat{K}' \) must approximate some behavior in \( \hat{K} \). Because no actor is handling an event in \( \hat{K} \), there is no behavior handling an event in \( \hat{K} \), so there must be no behavior (and therefore no actor) handling an event in \( \hat{K}' \).

**Used/Monitored Preservation** Let there be \( \hat{K} \) and \( s \) such that they are both well-formed and \( \text{Mon}(s) \subseteq \text{Used}(\hat{K}) \). Furthermore, let \( \langle \hat{K}', s', A, M \rangle \) be a member of \( T(\hat{K}, s) \). By the definition of an approximating transformation, \( \hat{K} \sqsubseteq_{A,M} \hat{K}' \) and \( M(s) = s' \). By the definition of \( \sqsubseteq \), \( \text{Used}(\hat{K}') = \bigcup_{\eta \in \text{Used}(\hat{K}) \cap \text{dom}(M)} \{ M(\eta) \} \). By the definition of \( M(s) \), \( \text{Mon}(s') = M(\text{Mon}(s)) \), so it must be the case that \( \text{Mon}(s) \subseteq \text{dom}(M) \), and therefore \( \text{Mon}(s) \subseteq \text{Used}(\hat{K}) \cap \text{dom}(M) \). Therefore, \( \text{Mon}(s') \subseteq \text{Used}(\hat{K}') \).

**No New Enabled Actors** Let \( \langle \hat{K}', s', A, M \rangle \) be a member of \( T(\hat{K}, s) \), and let \( \hat{a} \) identify a necessarily enabled actor in \( \hat{K}' \). By the definition of a necessarily enabled actor and because no actor in \( \hat{K}' \) is handling an event, \( \hat{a} \) is atomic. By the definition of an approximating transformation, \( \hat{K} \sqsubseteq_{A,M} \hat{K}' \), and by the definition of \( \sqsubseteq \) for atomic actors and the definition of a necessarily enabled actor, there must be some \( \hat{a}' \) such that \( A(\hat{a}') = \hat{a} \). Furthermore, by the definition of \( \sqsubseteq \), \( \hat{a}' \) identifies a necessarily enabled actor in \( \hat{K} \).

**Atomic Address Reflection** Enforced by the rules for \( \sqsubseteq_{A,M} \).

**Unique Actor Correspondences** Enforced by the rules for \( \sqsubseteq_{A,M} \).

**No New Single Messages** By the definition of a single-message-reflecting transformation.

**Internal Address Reflection** Enforced by the rules for \( \sqsubseteq_{A,M} \).

**Unique Approximating Messages** Let \( \langle \hat{K}', s', A, M \rangle \) be a member of \( T(\hat{K}, s) \), and let \( \hat{\mu} \) and \( \hat{\mu}' \) be the message-map components of \( \hat{K} \) and \( \hat{K}' \), respectively. By the definition of an approximating transformation, \( \hat{K} \sqsubseteq_{A,M} \hat{K}' \), which implies that \( \hat{\mu} \sqsubseteq_{A,M} \hat{\mu}' \). Then the rules for \( \sqsubseteq \) provide the necessary properties.

**Unique Approximated Messages** Let \( \langle \hat{K}', s', A, M \rangle \) be a member of \( T(\hat{K}, s) \), and let \( \hat{\mu} \) and \( \hat{\mu}' \) be the message-map components of \( \hat{K} \) and \( \hat{K}' \), respectively. By the definition of an approximating transformation, \( \hat{K} \sqsubseteq_{A,M} \hat{K}' \), which implies that \( \hat{\mu} \sqsubseteq_{A,M} \hat{\mu}' \). Then the rules for \( \sqsubseteq \) provide the necessary properties.
Properties for $\text{TransExec}_T$

Let there be $\hat{K}_1 \xrightarrow{\hat{l}_{1,1}...\hat{l}_{1,m}} ...$ and $s$ such that

- $\hat{K}_1$ and $s$ are well-formed,
- $\hat{K}_1$ is an externals-only configuration,
- $\text{Mon}(s) \subseteq \text{Used}(\hat{K}_1)$, and
- no actor in $\hat{K}_1$ is handling an event.

By the definition of $\text{TransExec}_T$ above, there exist $\hat{K}'_1 \xrightarrow{\hat{l}'_{1,1}...\hat{l}'_{1,n}} ...$, $s'$, $A$, and $M$ such that $\text{TransExec}_T(\hat{K}_1 \xrightarrow{\hat{l}_{1,1}...\hat{l}_{1,m}} ... , s) = \{ \langle \hat{K}'_1 \xrightarrow{\hat{l}'_{1,1}...\hat{l}'_{1,n}} ... , s', A, M \rangle \}$. Let $\hat{\beta}$ be the actor-behavior map of $\hat{K}_1$; then there furthermore exist $A'$, $A''$, $M'$, $M''$ such that

- $A' = A|_{\text{dom}(\hat{\beta}) \cup \text{ExtAbsAddr}}$,
- $M' = M|_{\text{Used}(\hat{K}_1)}$,
- $\hat{K}_i \sqsubseteq A' \cup A'' \cup M' \cup M'' \hat{K}'_i$ for all corresponding configurations in the two executions,
- $\hat{l}_{i,j} \sqsubseteq A' \cup A'' \cup M' \cup M'' \hat{l}'_{i,j}$ for all corresponding labels in the two executions, and
- $(M' \cup M'')(s) = s'$.

Let $A'' = A' \cup A''$ and $M'' = M' \cup M''$. Proofs for each of the conformance-reflection properties for $\text{TransExec}_T$ follow.

Initial Pair Correctness  By the definition of $\text{TransExec}_T$, $\langle \hat{K}'_1, s', A, M \rangle \in T(\hat{K}_1, s)$.

Fairness Preservation 1  By the Fair Execution Approximation Soundness lemma.

Fairness Preservation 2  Let there be $\hat{a}$ and $i$ such that the actor at $\hat{a}$ is necessarily enabled in $\hat{K}_1$ and either the actor at $\hat{a}$ is not necessarily enabled in $\hat{K}_1$ or $\hat{K}_i \xrightarrow{\hat{l}_{i,1}...\hat{l}_{i,m}} \hat{K}'_{i+1}$ is a step in the execution and $\hat{a}$ identifies the active actor for $\hat{l}_{i,1}$. In the former case, by the definition of $\sqsubseteq$, the actor at $A''(\hat{a})$ is not necessarily enabled in $\hat{K}'_i$. In the latter case, Let $\hat{a}'$ be the address that identifies the active actor for $\hat{l}'_{i,1}$. Because $\hat{l}_{i,1} \sqsubseteq A'' \cup M'' \hat{l}'_{i,1}$, $A''(\hat{a}) = \hat{a}'$. The address $\hat{a}$ must be in $\text{dom}(\hat{\beta})$, so $A(\hat{a}) = \hat{a}'$. 
Fairness Preservation 3 For each configuration \( \hat{K}_i \) and \( \hat{K}'_i \) in each execution, let \( \mu_i \) and \( \mu'_i \) be the message map from those configurations, respectively. Let there be \( \hat{a}, H, \hat{v} \), and \( i \) such that \( \mu_i(\hat{a}@H, \hat{v}) = \text{single} \) and either \( \hat{K}_i \) is a configuration in the execution such that \( \langle \hat{a}@H, \hat{v} \rangle \notin \text{dom}(\mu_i) \) or \( \mu_i(\hat{a}@H, \hat{v}) = \text{many} \), or \( \hat{K}_i \underset{\tilde{I}_{i1,\ldots,\tilde{I}_{i,m}}}{\rightarrow} \hat{K}_{i+1} \) is a step in the execution with \( \tilde{I}_{i1,1} = \hat{a}: \text{rcv-int}(H, \hat{v}) \). Let \( \hat{a}' = A''(\hat{a}), H' = M''(H) \), and \( \hat{v}' = A''(M''(\hat{v})) \). We must show that there exists some \( j \leq i \) such that either \( \langle \hat{a}@H', \hat{v}' \rangle \notin \text{dom}(\mu'_j) \), \( \mu'_j(\hat{a}@H', \hat{v}') = \text{many} \), or \( \hat{K}'_j \underset{\tilde{I}'_{j1,1} = \tilde{I}'_{j,m}}{\rightarrow} \hat{K}'_{j+1} \) is a step in the execution with \( \tilde{I}'_{j1,1} = \hat{a}' : \text{rcv-int}(H', \hat{v}') \).

There are three cases. First, consider the case in which \( \hat{K}_i \underset{\tilde{I}_{i1,1} = \tilde{I}_{i1,m}}{\rightarrow} \hat{K}_{i+1} \) is a step in the execution with \( \tilde{I}_{i1,1} = \hat{a} : \text{rcv-int}(H, \hat{v}) \). By the definition of \( \subseteq \), \( \tilde{I}_{i1,1} = \hat{a} : \text{rcv-int}(H', \hat{v}') \) for some \( \hat{v}' \). If \( \hat{v}' = \hat{v} \), let \( j = i \) and we're done. Otherwise, we know that both labels must correspond to a use of the A-ReceiveInternal rule, and therefore \( \langle \hat{a}@H, \hat{v} \rangle \in \text{dom}(\mu_i) \) and \( \langle \hat{a}@H', \hat{v}' \rangle \in \text{dom}(\mu'_j) \). Because \( \tilde{I}_{i1,1} \subseteq A'' \cap A'' = \tilde{I}'_{i1,1} \) We also know that \( \hat{a}@H \subseteq A'' \cap M'' = \hat{a}@H' \), and \( \hat{v} \subseteq A'' \cap M'' = \hat{v}' \). It must be the case that \( \mu_i \subseteq A'' \cap M'' = \mu'_j \), and therefore \( \langle \hat{a}@H', \hat{v}' \rangle \) is the only message in \( \mu'_j \) such that \( \hat{a}@H \subseteq A'' \cap M'' = \hat{a}@H' \), and \( \hat{v} \subseteq A'' \cap M'' = \hat{v}' \). By the definition of \( \hat{v}' \), \( \hat{v} \subseteq A'' \cap M'' = \hat{v}' \). Therefore it must be the case that \( \langle \hat{a}@H', \hat{v}' \rangle \notin \text{dom}(\mu'_j) \) so let \( j = i \) and we're done.

Second, consider the case in which \( \mu_i(\hat{a}@H, \hat{v}) = \text{many} \). Because \( \mu_i \subseteq \mu'_j \), there exists exactly one \( \hat{v}' \) such that \( \langle \hat{a}@H', \hat{v}' \rangle \in \text{dom}(\mu'_j) \) and \( \hat{v} \subseteq A'' \cap M'' = \hat{v}' \). Then either \( \hat{v}' = \hat{v} \), or \( \langle \hat{a}@H', \hat{v}' \rangle \notin \text{dom}(\mu'_j) \). Then let \( j = i \) and we're done.

Finally, consider the case in which \( \langle \hat{a}@H, \hat{v} \rangle \notin \text{dom}(\mu_i) \) and neither of the previous two cases apply. Let \( \hat{K}_j \) be the first configuration in the execution such that \( \langle \hat{a}@H, \hat{v} \rangle \notin \text{dom}(\mu_j) \) (it must be the case that \( 1 \leq j \leq i \)). Because \( \tilde{I}_{j-1,1} \neq \hat{a} : \text{rcv-int}(H, \hat{v}) \), it must be the case that the configuration sent some message that merged with \( \hat{v} \); i.e., there exist \( k \in 1 \ldots m \), \( \hat{a}'' \), and \( \hat{v}'' \) such that \( \tilde{I}_{j-1,k} = \hat{a}'' : \text{send-int}(\hat{a}@H, \hat{v}'') \) and Merge(\( \hat{a}'', \hat{v}'') \) is defined. Then because \( \tilde{I}_{j-1,k} = A'' \cap M'' = \tilde{I}'_{j-1,k} \), let \( \tilde{I}'_{j-1,k} = \hat{a}'' : \text{send-int}(\hat{a}@H', \hat{v}'') \) for some \( \hat{a}'' \) and \( \hat{v}'' \) such that \( \hat{v}'' \subseteq A'' \cap M'' = \hat{v}'' \). By the Approximation Mergeability lemma (appendix I), Merge(\( \hat{v}'', \hat{v}'' \)) is defined. By the Mergeability Preservation lemma (also appendix I) any future sends of the message \( \langle \hat{a}@H', \hat{v}' \rangle \) in the event step \( \hat{K}'_{j-1} \underset{\tilde{I}'_{j-1,1} = \tilde{I}'_{j-1,m}}{\rightarrow} \hat{K}' \) must also merge with that resulting merged message. If the resulting merged message is \( \hat{v}' \), then it must be the case that \( \mu'_j(\hat{a}@H', \hat{v}') = \text{many} \). Otherwise, \( \langle \hat{a}@H', \hat{v}' \rangle \notin \text{dom}(\mu'_j) \).

Properties for UntransExec_T

Let there be \( \hat{K}_1 \underset{\tilde{I}_{1,1} = \tilde{I}_{1,m}}{\rightarrow} \ldots, s \), and \( X \) such that

- \( \hat{K}_1 \) and \( s \) are well-formed,
- \( \hat{K}_1 \) is an externals-only configuration,
- \( \text{Mon}(s) \subseteq \text{Used}(\hat{K}_1) \),
• no actor in $\hat{K}_1$ is handling an event, and
• $X$ contains a simulation for all members of

\[
\text{TransExec}(\hat{K}_1 \xrightarrow{\hat{f}_{1,1,\ldots,\hat{f}_{1,m}}} \ldots, s).
\]

By the definition of $\text{UntransExec}$ above, $\text{UntransExec}(\hat{K}_1 \xrightarrow{\hat{f}_{1,1,\ldots,\hat{f}_{1,m}}} \ldots, s, X)$ is a specification execution $S_1 \xrightarrow{(L_1, O_1)} \ldots$ with the same length as $\hat{K}_1 \xrightarrow{\hat{f}_{1,1,\ldots,\hat{f}_{1,m}}} \ldots$ such that $S_1 = \{s\}$ and $\hat{K}_1 \xrightarrow{\hat{f}_{1,1,\ldots,\hat{f}_{1,m}}} \ldots$ summarizes $\hat{K}_1 \xrightarrow{\hat{f}_{1,1,\ldots,\hat{f}_{1,n}}} \ldots$.

Furthermore, By the definition of $\text{TransExec}$ above, there exist $\overline{\text{aex}}, s'$, $A$, and $M$ such that $\text{TransExec}(\hat{K}_1 \xrightarrow{\hat{f}_{1,1,\ldots,\hat{f}_{1,m}}} \ldots, s) = \{\langle \overline{\text{aex}}, s', A, M \rangle\}$ and $\overline{\text{aex}}$ has the same length as $\hat{K}_1 \xrightarrow{\hat{f}_{1,1,\ldots,\hat{f}_{1,m}}} \ldots$. Because $X$ contains a simulation for all members of $\text{TransExec}(\hat{K}_1 \xrightarrow{\hat{f}_{1,1,\ldots,\hat{f}_{1,m}}} \ldots, s)$, let $S' \xrightarrow{(\hat{L}_i, O_i)} \ldots = X(\overline{\text{aex}}, s', A, M)$; we know that $S' \xrightarrow{\hat{L}_i, \hat{O}_i} \ldots$ has the same length as $\overline{\text{aex}}$, and therefore as $S_1 \xrightarrow{(L_1, O_1)} \ldots$. Then by the definition of $\text{UntransExec}$ above, there exist some $M'$ and $M''$ such that

• $M' = M\big|_{\text{Uned}(\hat{K}_1)}$,
• $(M' \cup M'')(O_i) = O'_i$ for all corresponding obligation multisets in the two executions, and
• $(M' \cup M'')(S_i) = S'_i$ for all corresponding configurations in the two executions.

The proofs for each of the properties follow.

**Execution Simulation** We have already shown that $S_1 = \{s\}$ and $\text{Simulates}(S_1 \xrightarrow{(L_1, O_1)} \ldots, \hat{K}_1 \xrightarrow{\hat{f}_{1,1,\ldots,\hat{f}_{1,m}}} \ldots)$.

**Prefix Consistency** Let there be $\hat{K}_1 \xrightarrow{\hat{f}_{1,1,\ldots,\hat{f}_{1,m}}} \ldots, X', S'_1 \xrightarrow{\hat{L}'_1, \hat{O}'_1} \ldots$, and $i$ such that

• $\hat{K}_1 \xrightarrow{\hat{f}_{1,1,\ldots,\hat{f}_{1,m}}} \ldots$ and $\hat{K}' \xrightarrow{\hat{f}_{1,1,\ldots,\hat{f}_{1,m}}} \ldots$ share a prefix of length $i$,
• $\text{UntransExec}(\hat{K}_1 \xrightarrow{\hat{f}_{1,1,\ldots,\hat{f}_{1,m}}} \ldots, s, X') = S'_1 \xrightarrow{\hat{L}'_1, \hat{O}'_1} \ldots$, and

• for all $\overline{\text{aex}}, \overline{\text{aex}}', s', A, M$, and $j$ such that

\[
\begin{align*}
\langle \overline{\text{aex}}, s', A, M \rangle &\in \text{TransExec}(\hat{K}_1 \xrightarrow{\hat{f}_{1,1,\ldots,\hat{f}_{1,m}}} \ldots, s), \\
\langle \overline{\text{aex}}', s', A, M \rangle &\in \text{TransExec}(\hat{K}'_1 \xrightarrow{\hat{f}_{1,1,\ldots,\hat{f}_{1,m}}} \ldots, s),
\end{align*}
\]

By the definition of $\text{UntransExec}$ above, $\text{UntransExec}(\hat{K}_1 \xrightarrow{\hat{f}_{1,1,\ldots,\hat{f}_{1,m}}} \ldots, s, X')$ is a specification execution $S_1 \xrightarrow{(L_1, O_1)} \ldots$ with the same length as $\hat{K}_1 \xrightarrow{\hat{f}_{1,1,\ldots,\hat{f}_{1,m}}} \ldots$ such that $S_1 = \{s\}$ and $\hat{K}_1 \xrightarrow{\hat{f}_{1,1,\ldots,\hat{f}_{1,m}}} \ldots$ summarizes $\hat{K}_1 \xrightarrow{\hat{f}_{1,1,\ldots,\hat{f}_{1,n}}} \ldots$.
- $\hat{aex}$ and $\hat{aex}'$ share a prefix of length $j$, and
- $j \leq i$.

$X(\hat{aex}, s', A, M)$ and $X'(\hat{aex}', s', A, M)$ share a prefix of length $j$.

By the definition of $\text{TransExec}$, there exist $\hat{aex}$, $\hat{aex}'$, $s'$, $A$, and $M$ such that $\text{TransExec}(\hat{R}_1 \xrightarrow{i_1, \ldots, i_m} \ldots, s) = \{(\hat{aex}, s', A, M)\}$, and $\text{TransExec}(\hat{R}_1' \xrightarrow{i_1', \ldots, i_m'} \ldots, s') = \{(\hat{aex}', s', A, M)\}$. Also by the definition of $\text{TransExec}$, because the two executions share a prefix of length $i$, $\hat{aex}$ and $\hat{aex}'$ also share a prefix of length $i$ (recall from that definition that each next step is chosen deterministically). Then by the above conditions, $X(\hat{aex}, s', A, M)$ and $X'(\hat{aex}', s', A, M)$ share a prefix of length $i$. Therefore by the definition of $\text{UntransExec}$, $S_1 \xrightarrow{(L_1, O_1)} \ldots$ and $S'_1 \xrightarrow{(L'_1, O'_1)} \ldots$ share a prefix of length $i$ (recall from the definition of $\text{UntransExec}$ that each next step is again chosen deterministically).

**Fulfillment Reflection 1** Let $\langle \eta, po \rangle$ be a member of $\text{Obls}(s)$, and suppose that for all $\langle \hat{aex}, s', A, M \rangle \in \text{TransExec}(\hat{R}_1 \xrightarrow{i_1, \ldots, i_m} \ldots, s)$, $X(\hat{aex}, s', A, M)$ is an execution $\langle \hat{l}_i, O_i^\prime \rangle \ldots$ such that either

- $\langle M(\eta), po \rangle \notin \text{Obls}(S''_1)$, or
- there is a step $S_i'' \xrightarrow{(L_i, O''_i)} S_{i+1}''$ in that execution such that $\langle M(\eta), po \rangle \in O''_i$.

We must show that there exists a step $S_j \xrightarrow{(L_j, O_j)} S_{j+1}$ in the execution $S_1 \xrightarrow{(L_1, O_1)} \ldots$ such that $\langle \eta, po \rangle \in O_j$.

Let $\eta' = M(\eta)$. Because $s$ is well-formed, $\eta \in \text{Mon}(s)$, and therefore $\eta \in \text{dom}(M')$, so $(M' \cup M')(\eta) = \eta'$. Therefore, $\langle \eta', po \rangle \in \text{Obls}(s')$. By the above conditions, there is a step $S_i' \xrightarrow{(L'_i, O'_i)} S_{i+1}'$ such that $\langle \eta', po \rangle \in O'_i$.

By the definition of $\text{TransExec}$ above, $\hat{R}_1' \subseteq M' \cup M'' \hat{R}_1$, where $\hat{R}_1'$ is the first configuration in $\hat{aex}$, and therefore $M' \cup M''$ is one-to-one. Therefore, because $(M' \cup M')(O_i) = O'_i$, $\langle \eta, po \rangle \in O'_i$.

**Fulfillment Reflection 2** By an argument similar to the one for Fulfillment Reflection 1. □

**L.27 Fully Merged Expansion Lemma**

**Lemma** (Fully Merged Expansion). For all $\bar{\mu}$, $\bar{\mu}'$, $\bar{a}$, $H$, $\bar{v}$, and $m$, if $\bar{\mu}$ is fully merged and $\bar{\mu} = \bar{\mu}' \cup \{\langle \bar{a} @ H, \bar{v} \rangle \rightarrow m\}$, then

$$\bar{\mu} = \begin{cases} \bar{\mu}' \cup \{\langle \bar{a} @ H, \bar{v} \rangle\} & \text{if } m = \text{single} \\ \bar{\mu}' \cup \{\langle \bar{a} @ H, \bar{v} \rangle \cup \{\langle \bar{a} @ H, \bar{v} \rangle\} & \text{if } m = \text{many} \end{cases}$$
L.28  Remap Approximation Lemma

Lemma (Remap Approximation). For all \( \hat{K}, A, \) and \( M, \) if

- no actor in \( \hat{K} \) is handling an event,
- \( A \) is approximating, and
- \( M \) is one-to-one,

then \( \hat{K} \sqsubseteq_{A,M} \text{Remap}(\hat{K}, A, M). \)

Proof. Let \( \left< \left< \hat{\beta} \mid \hat{\mu} \mid H \right> \right>^\hat{\beta} = \hat{K}, \) and let \( \left< \left< \hat{\beta}' \mid \hat{\mu}' \mid H' \right> \right>^{\hat{\beta}'} = \text{Remap}(\hat{K}, A, M). \) We will show that each of the conditions necessary for \( \hat{K} \sqsubseteq_{A,M} \text{Remap}(\hat{K}, A, M) \) hold. We have by the precondition on this lemma that \( M \) is one-to-one, so it remains to show the conditions for each of the components of \( \left< \left< \hat{\beta} \mid \hat{\mu} \mid H \right> \right>^\hat{\beta} \) and \( \left< \left< \hat{\beta}' \mid \hat{\mu}' \mid H' \right> \right>^{\hat{\beta}'}. \)

Actor-Behavior Map

First, to show \( \hat{\beta} \sqsubseteq_{A,M} \hat{\beta}', \) let \( \hat{a} \) be a member of \( \text{dom}(\hat{\beta}), \) and let \( \hat{a}' = A(\hat{a}). \) We must show that \( \hat{\beta}(\hat{a}) \sqsubseteq_{A,M} \hat{\beta}'(\hat{a}'). \) Let \( \{\hat{b}_1, \ldots, \hat{b}_n\} = \hat{\beta}(\hat{a}). \) Then by the definition of \( \text{Remap}, \) \( \hat{\beta}' = \text{Remap}(\hat{\beta}, A, M), \) and therefore \( \{\text{Remap}(\hat{b}_1, A, M), \ldots, \text{Remap}(\hat{b}_n, A, M)\} \subseteq \hat{\beta}'(\hat{a}'). \) Then for all \( i \in 1, \ldots, n, \) it is easy to see from the definition of \( \text{Remap} \) and the conditions on \( A \) and \( M \) that \( \hat{\beta}_i \sqsubseteq_{A,M} \text{Remap}(\hat{b}_i, A, M). \)

For the second property to show that \( \hat{\beta} \sqsubseteq_{A,M} \hat{\beta}', \) let \( \hat{a}' \) be an atomic address in \( \text{dom}(\hat{\beta}'). \) By the definition of \( \text{Remap} \) and because \( A \) is approximating, there exists \( \hat{a} \in \text{dom}(\hat{\beta}) \) such that \( \hat{a} \sqsubseteq_{A} \hat{a}'. \) Finally, no actor in \( \hat{K} \) is handling an event, so by the definition of \( \hat{K}', \) no actor in \( \hat{K}' \) is handling an event, and therefore \( \hat{\beta} \sqsubseteq_{A,M} \hat{\beta}'. \)

Message Map

Second, we must show that \( \hat{\mu} \sqsubseteq_{A,M} \hat{\mu}'. \) By the definition of \( \text{Remap}, \) \( \hat{\mu}' = \text{Remap}(\hat{\mu}, A, M). \) We prove by induction on the size of \( \text{dom}(\hat{\mu}) \) that \( \hat{\mu} \sqsubseteq_{A,M} \text{Remap}(\hat{\mu}, A, M). \)

In the base case, \( \hat{\mu} = \emptyset. \) Then \( \hat{\mu}' = \emptyset \) by the definition of \( \text{Remap}, \) and therefore \( \hat{\mu} \sqsubseteq_{A,M} \hat{\mu}'. \)

In the inductive case, let there be \( \hat{\mu}'', \hat{a}, H'', \hat{v}, \) and \( m \) such that \( \hat{\mu} = \hat{\mu}'' \cup (\hat{a}@H'', \hat{v}) \rightarrow m. \) Let \( \hat{a}' = \text{Remap}(\hat{a}, A, M), H'' = \text{Remap}(H'', A, M), \) and
\( \hat{v}' = \text{Remap}(\hat{v}, A, M) \). It is easy to see by the definition of \( \text{Remap} \) and the conditions on \( A \) and \( M \) that \( \hat{a}@H'' \sqsubseteq_{A,M} \hat{a}'@H''' \) and \( \hat{v} \sqsubseteq_{A,M} \hat{v}' \).

There are two cases, depending on the value of \( m \). If \( m = \text{single} \), then by the Fully Merged Expansion lemma, \( \hat{\mu} = \hat{\mu}'' \oplus (\hat{a}@H'', \hat{v}) \), and by the definition of \( \text{Remap} \), \( \hat{\mu}' = \text{Remap}(\hat{\mu}'', A, M) \oplus (\hat{a}@H''', \hat{v}') \). Also by the definition of \( \text{Remap} \), \( \text{Remap}(\hat{\mu}'', A, M) \) is fully merged. By the induction hypothesis, \( \hat{\mu}'' \sqsubseteq_{A,M} \text{Remap}(\hat{\mu}'', A, M) \). Then by the Message-Addition Soundness lemma (appendix I), \( \hat{\mu} \sqsubseteq_{A,M} \hat{\mu}' \).

On the other hand, if \( m = \text{many} \), then by the Fully Merged Expansion lemma, \( \hat{\mu} = \hat{\mu}'' \oplus (\hat{a}@H'', \hat{v}) \oplus (\hat{a}@H''', \hat{v}) \), and by the definition of \( \text{Remap} \), \( \hat{\mu}' = \text{Remap}(\hat{\mu}'', A, M) \oplus (\hat{a}@H''', \hat{v}) \oplus (\hat{a}@H''', \hat{v}') \). Again by the definition of \( \text{Remap} \), \( \text{Remap}(\hat{\mu}'', A, M) \) is fully merged. Then by the induction hypothesis, \( \hat{\mu}'' \sqsubseteq_{A,M} \text{Remap}(\hat{\mu}'', A, M) \), and by two uses of the Message-Addition Soundness lemma, \( \hat{\mu} \sqsubseteq_{A,M} \hat{\mu}' \).

**Used Markers**

By the definition of \( \text{Remap} \), \( H' = \text{Remap}(H, A, M) \). Then it is easy to see by the definition of \( \text{Remap} \) that \( H \sqsubseteq_{A,M} \text{Remap}(H, A, M) \).

**Receptionists**

Finally, we must show that \( \hat{\rho} \sqsubseteq_{A,M} \hat{\rho}' \). Let \( (\hat{a}@H'', \tau) \) be a member of \( \hat{\rho} \). By the definition of \( \text{Remap} \), \( (\text{Remap}(\hat{a}@H''), \tau) \in \hat{\rho}' \). It is easy to see by the definition of \( \text{Remap} \) and the conditions on \( A \) and \( M \) that \( \hat{a}@H'' \sqsubseteq_{A,M} \text{Remap}(\hat{a}@H'') \). Therefore, \( \hat{\rho} \sqsubseteq_{A,M} \hat{\rho}' \).

**L.29 Remap Single-Message Reflection Lemma**

**Lemma** (Remap Single-Message Reflection). For all \( \hat{K} = \langle \hat{\beta} \mid \hat{\mu} \mid H \rangle^{\hat{\rho}} \), \( \hat{K}' = \langle \langle \hat{\beta} \rangle \mid \hat{\mu}' \mid H' \rangle^{\hat{\rho}} \), \( A, \) and \( M \), if \( \text{Remap}(\hat{K}, A, M) = \hat{K}' \), then for all \( \hat{a}', H'' \), and \( \hat{v}' \) such that \( \hat{\mu}(\hat{a}@H'', \hat{v}) = \text{single} \), there exist \( \hat{a} \), \( H'' \), and \( \hat{v} \) such that \( \hat{\mu}(\hat{a}@H'', \hat{v}) = \text{single} \), such that \( A(\hat{a}) = \hat{a}' \), \( M(H) = H' \), and \( \text{Remap}(\hat{\mu}, A, M) = \hat{\mu}' \).

**Proof.** Let there be \( \hat{a}', H'' \), and \( \hat{v}' \) such that \( \hat{\mu}'(\hat{a}@H'', \hat{v}') = \text{single} \). By the definition of \( \text{Remap} \), because the quantity is \( \text{single} \), this message is exactly the \( \text{Remap} \) of a message in \( \hat{\mu} \). Thus, there exist \( \hat{a}, H'' \), and \( \hat{v} \) such that \( \hat{\mu}(\hat{a}@H'', \hat{v}) = \text{single} \), such that \( A(\hat{a}) = \hat{a}' \), \( M(H) = H' \), and \( \text{Remap}(\hat{\mu}, A, M) = \hat{\mu}' \).

**L.30 Unmark Conformance Reflection Theorem**

**Theorem** (Unmark Conformance Reflection). Unmark is a conformance-reflecting transformation.
L.31. ASSIMILATE CONFORMANCE REFLECTION THEOREM

Proof. Let there be some \( \tilde{K} \) and \( s \), and let \( H = \text{Mon}(s) \). If \( \text{Unmark}(\tilde{K}, s) \) terminates, then it returns \( \{ \langle \text{Remap}(\tilde{K}, A, M), s, A, M \rangle \} \), where \( A = \text{id} \) and \( M = \text{id} \mid_H \).

It is easy to see that \( A \) is approximating and that \( M \) is one-to-one. If \( s \) is well-formed and no actor in \( \tilde{K} \) is handling an event, then by the Remap Approximation lemma, \( \tilde{K} \sqsubseteq_{A, M} \text{Remap}(\tilde{K}, A, M) \). Because \( s \) is well-formed, every marker appearing in \( s \) is a member of \( H \). Therefore, \( M(s) = s \). Therefore, \( T \) is an approximating transformation.

If \( \tilde{K} \) is well-formed, then by the Remap Well-Formed Preservation lemma, \( \text{Remap}(\tilde{K}, A, M) \) is well-formed. Therefore, \( \text{Unmark} \) is a well-formed-preserving transformation.

Next, we have already shown that \( A \) is approximating, so if \( \tilde{K} \) is an externals-only configuration, then so is \( \text{Remap}(\tilde{K}, A, M) \) by the Remap Externals-Only Preservation lemma. Therefore, \( \text{Unmark} \) is externals-only-preserving.

Finally, for all messages \( \langle \hat{a}'@H', \hat{v} \rangle \) in \( \tilde{K}' \) with a quantity of single, by the Remap Single-Message Reflection lemma, there exists a message \( \langle \hat{a}@H, \hat{v} \rangle \) in \( \tilde{K} \) with a quantity of single such that \( A(\hat{a}) = \hat{a}' \), \( M(H) = H' \), and \( \text{Remap}(\hat{v}, A, M) = \hat{v}' \). Therefore \( \text{Unmark} \) is single-message-reflecting, and by the Approximating Transformation lemma, \( \text{Unmark} \) is conformance-reflecting.

\( \square \)

L.31 Assimilate Conformance Reflection Theorem

Theorem (Assimilate Conformance Reflection). Assimilate is a conformance-reflecting transformation.

Proof. Let there be some \( \tilde{K} \) and \( s \). If \( \text{Assimilate}(\tilde{K}, s) \) terminates, then it returns \( \{ \langle \text{Remap}(\tilde{K}, A, \text{id}), s, A, \text{id} \rangle \} \), where \( A = \text{id} \left[ (\text{addr} \; \ell_1 \; n_1) \rightarrow (\text{collective-addr} \; \ell_1) \right] \) for some internal addresses \( (\text{addr} \; \ell_1 \; n_1), \ldots, (\text{addr} \; \ell_m \; n_m) \).

It is easy to see that \( A \) is approximating and \( \text{id} \) is one-to-one. If \( s \) is well-formed and no actor in \( \tilde{K} \) is handling an event, then by the Remap Approximation lemma, \( \tilde{K} \sqsubseteq_{A, \text{id}} \text{Remap}(\tilde{K}, A, \text{id}) \). We also have that \( \text{id}(s) = s \), so \( \text{Assimilate} \) is an approximating transformation.

By the definition of \( A \), every address appearing in \( \tilde{K} \) is a member of \( \text{dom}(A) \), and for all external addresses \( \hat{a} \in \text{dom}(A) \), \( A(\hat{a}) \) is external. Therefore, by the Remap Well-Formed Preservation lemma, \( \text{Assimilate} \) is a well-formed-preserving transformation.

Next, we have already shown that \( A \) is approximating, so if \( \tilde{K} \) is an externals-only configuration, then so is \( \text{Remap}(\tilde{K}, A, M) \) by the Remap Externals-Only Preservation lemma. Therefore, \( \text{Assimilate} \) is externals-only-preserving.

Finally, for all messages \( \langle \hat{a}'@H', \hat{v} \rangle \) in \( \tilde{K}' \) with a quantity of single, by the Remap Single-Message Reflection lemma, there exists a message \( \langle \hat{a}@H, \hat{v} \rangle \) in \( \tilde{K} \) with a quantity of single such that \( A(\hat{a}) = \hat{a}' \), \( M(H) = H' \), and \( \text{Remap}(\hat{v}, A, M) = \hat{v}' \). Therefore \( \text{Assimilate} \) is single-message-reflecting, and by the Approximating
Therefore, Canonicalize is a conformance-reflecting transformation. Thus, by the Approximating Transformation lemma, Assimilate is conformance-reflecting. 

\[ \square \]

### L.32 Canonicalize Conformance Reflection Theorem

**Theorem (Canonicalize Conformance Reflection).** Canonicalize is a conformance-reflecting transformation.

**Proof.** Let there be some \( \hat{K} \) and \( s \). If Canonicalize(\( \hat{K}, s \)) terminates, then it returns \( \langle \text{Remap}(\hat{K}, A, M), M(s), A, M \rangle \), where \( A = \text{CanonicalAddrSubst}(\hat{K}) \) and \( M = \text{CanonicalMarkerSubst}(\hat{K}, s) \). By the definition of CanonicalAddrSubst, \( A \) is approximating.

To show that Canonicalize is approximating, suppose that \( \hat{K} \) and \( s \) are well-formed and no actor in \( \hat{K} \) is handling an event. To show that \( M \) is one-to-one, let \( \eta \) be a member of \( \text{rng}(M) \), and let \( \eta'_1, \ldots, \eta'_n \) be the state arguments for \( s \). There are three cases.

1. If \( \eta = ^*0^* \), then \( \eta \in \text{InMon}(s) \). Because \( s \) is well-formed, \( |\text{InMon}(s)| \leq 1 \), so \( \eta \) is unique.

2. If \( \eta = ^*i^* \) for some \( i \in 1 \ldots n \) where \( s \) has \( n \) state arguments, then the only marker \( \eta' \) such that \( M(\eta') = \eta \) is \( \eta'_i \), by the definition of CanonicalMarkerSubst.

3. Otherwise, \( \eta = ^*m+1^* \) for some \( m \) such that \( m = \left| \text{InMon}(s) \cup \{\eta'_1, \ldots, \eta'_n\} \cup H \right| \), where \( H = \{\eta'' | \eta'' < \eta\} \) for some \( \eta' \). Let \( H' = \text{InMon}(s) \cup \{\eta'_1, \ldots, \eta'_n\} \). We can rewrite the definition of \( m \) as \( m = |H'| + \left| \{\eta'' | \eta'' \notin H' \text{ and } \eta'' < \eta\} \right| \). Because history markers are totally ordered, there is no \( \eta'''' \) such that \( \left| \{\eta'''' | \eta'''' \notin H' \text{ and } \eta'''' < \eta''\} \right| = \left| \{\eta'''' | \eta'''' \notin H' \text{ and } \eta'''' < \eta''\} \right| \). Therefore, \( \eta' \) is the unique marker such that \( M(\eta') = \eta \).

Therefore, \( M \) is one-to-one.

Then by the Remap Approximation lemma, \( \hat{K} \sqsubseteq_{A,M} \text{Remap}(\hat{K}, A, M) \). Therefore, Canonicalize is an approximating transformation.

By the definition of CanonicalAddrSubst, every address appearing in \( \hat{K} \) is a member of \( \text{dom}(A) \), and for all external addresses \( \hat{a} \in \text{dom}(A) \), \( A(\hat{a}) \) is external. Therefore, by the Remap Well-Formed Preservation lemma, Canonicalize is a well-formed-preserving transformation.

Next, for all messages \( \langle \hat{a}@H', \hat{v}' \rangle \) in \( \hat{K} \) with a quantity of \text{single}, by the Remap Single-Message Reflection lemma, there exists a message \( \langle \hat{a}@H, \hat{v} \rangle \) in \( \hat{K} \) with a quantity of \text{single} such that \( A(\hat{a}) = \hat{a}' \), \( M(H) = H' \), and \( \text{Remap}(\hat{v}, A, M) = \hat{v}' \). Therefore Canonicalize is single-message-reflecting, and by the Approximating Transformation lemma, Canonicalize is conformance-reflecting.
Finally, we have already shown that \( A \) is approximating, so \( \text{Canonicalize} \) is externals-only-preserving. Thus, by the Approximating Transformation lemma, \( \text{Canonicalize} \) is conformance-reflecting.

### L.33 TryTrans Well-Formed Preservation Lemma

**Lemma (TryTrans Well-Formed Preservation).** For all \( \hat{K}, \hat{K}', \hat{K}'', \hat{K}''' \), and \( s \), if \( \hat{K} \) and \( s \) are well-formed and \( \text{TryTrans}(\hat{K}, s, L, \hat{K}') = \hat{K}''' \), then \( \hat{K}''' \) is well-formed.

**Proof.** The proof follows the definition of \( \text{TryTrans} \) in figure 8.2. By that definition, there exist some \( \hat{K}'' \), \( A \), and \( M \) such that \( \langle \hat{K}'', s, A, M \rangle \in T_{\text{NoAcc}}(\hat{K}', s) \). By the various conformance-reflection theorems in this appendix, \( T_{\text{NoAcc}} \) is conformance-reflecting. Therefore, by the Well-Formed Preservation property for conformance reflection, \( \hat{K}'' \) is well-formed.

Next, there exists some \( \hat{K}''' \) such that \( \hat{K}'' \overset{L}{\rightarrow} \hat{K}'''. \) By the definition of summary transitions and event steps, there exist some \( \hat{i}_1, \ldots, \hat{i}_n \) such that \( \hat{K}'' \overset{\hat{i}_1}{\rightarrow}_{RA} \ldots \overset{\hat{i}_n}{\rightarrow}_{RA} \hat{K}''' \). Then by the Abstract Well-Formed Preservation lemma (appendix I) and induction on \( n \), \( \hat{K}''' \) is well-formed.

Finally, it must be the case that \( \langle \hat{K}''', s, A, M \rangle \in T_{\text{NoAcc}}(\hat{K}'''', s) \) for some \( A \) and \( M \). Again by the Well-Formed Preservation property for conformance reflection, \( \hat{K}'''' \) is well-formed.

### L.34 Accelerate Conformance Reflection Theorem

**Theorem (Accelerate Conformance Reflection).** Accelerate is a conformance-reflecting transformation.

**Proof.** Let there be some \( \hat{K}_{\text{init}} \) and \( s_{\text{init}} \) such that \( \hat{K}_{\text{init}} \) and \( s_{\text{init}} \) are well-formed. The main burden of proof here is to show that the program configuration returned by \( \text{Accelerate}(\hat{K}_{\text{init}}, s_{\text{init}}) \) is an approximation of \( \hat{K}_{\text{init}} \). Then we can show by the Approximating Transformation lemma that \( \text{Accelerate} \) is conformance-reflecting.

The proof follows the definition of the \( \text{Accelerate} \) algorithm from figure 8.4. We will establish a loop invariant on the while loop that \( \hat{K} \) is well-formed and that \( \hat{K}_{\text{init}} \preceq_{id,id} \hat{K} \).

At the beginning of the loop, \( \hat{K} = \hat{K}_{\text{init}} \). Therefore, \( \hat{K} \) is well-formed and by the Approximation Reflexivity lemma, \( \hat{K}_{\text{init}} \preceq_{id,id} \hat{K} \).

During the loop, if \( \text{IsAccelerateCandidate} \) or \( \text{TryTrans} \) returns false, then the loop does not re-assign the variable \( \hat{K} \), so the loop invariant is maintained. Otherwise, \( \text{TryTrans} \) returns some configuration \( \hat{K}'' \), and by the definition of \( \text{TryTrans} \) (figure 8.2), \( \hat{K} \preceq_{id,id} \hat{K}'' \). Therefore, by the Approximation Transitivity lemma (appendix I), \( \hat{K}_{\text{init}} \preceq_{id,id} \hat{K}'' \).

Furthermore, because \( \text{IsAccelerateCandidate} \) returns true, we know that \( \hat{K} \overset{L}{\rightarrow} \hat{K}' \), where \( \hat{K}' \) is the second configuration given to \( \text{TryTrans} \). By the definition of summary transitions and event steps, there exist some \( \hat{i}_1, \ldots, \hat{i}_n \) such that
\(\hat{K} \xrightarrow{i_1 \text{RA}} \cdots \xrightarrow{i_n \text{RA}} \hat{K}'\). Then by the Abstract Well-Formed Preservation lemma (appendix I) and induction on \(n\), \(\hat{K}'\) is well-formed. Therefore by the TryTrans Well-Formed Preservation lemma, \(\hat{K}''\) is well-formed. The algorithm sets \(\hat{K}''\) as the new value of \(\hat{K}\), so the invariant is maintained.

Thus by the definition of Accelerate, if \(\text{Accelerate}(\hat{K}_{\text{init}}, s_{\text{init}})\) terminates, it returns \(\langle \hat{K}', s_{\text{init}}, id, id \rangle\) for some \(\hat{K}'\). By the loop invariant, \(\hat{K}'\) is well-formed and \(\hat{K}_{\text{init}} \subseteq id, id \hat{K}'\). Therefore, Accelerate is an approximating, well-formed-preserving transformation.

Separately, we must also show that Accelerate is externals-only-preserving. Let there be some \(\hat{K}_{\text{init}}\) and \(s_{\text{init}}\) such that \(\hat{K}_{\text{init}}\) is an externals-only configuration. Again, the proof follows the definition of the Accelerate algorithm. We will establish a loop invariant on the while loop that \(\hat{K}\) is an externals-only configuration.

At the beginning of the loop, \(\hat{K} = \hat{K}_{\text{init}}\), so the invariant holds. During an execution of the loop, if TryTrans returns false, then \(\hat{K}\) does not change, so the invariant holds. If TryTrans returns some configuration \(\hat{K}''\), then by the definition of TryTrans, there exist \(\hat{K}', \hat{K}'', \hat{K}''', L, s, A, \text{ and } M\) such that

- \(\hat{K} \xrightarrow{L} \hat{K}'\),
- \(\langle \hat{K}'', s, A, M \rangle \in T_{\text{NoAcc}}(\hat{K}', s)\),
- \(\hat{K}'' \xrightarrow{L} \hat{K}'''\), and
- \(\langle \hat{K}''', s, A, M \rangle \in T_{\text{NoAcc}}(\hat{K}'''', s)\).

By corollary I.42.2, \(\hat{K}'\) is an externals-only configuration. By the various conformance-reflection theorems in this appendix, \(T_{\text{NoAcc}}\) is conformance-reflecting. Therefore by the Externals-Only Preservation conformance-reflection property, \(\hat{K}''\) is an externals-only configuration. Again by corollary I.42.2, \(\hat{K}'''\) is an externals-only configuration. Then by the Externals-Only Preservation conformance-reflection property again, \(\hat{K}''''\) is an externals-only configuration. The Accelerate algorithm then sets \(\hat{K}\) to \(\hat{K}''''\), so the invariant holds.

When the loop ends, by the loop invariant, \(\hat{K}\) is an externals-only configuration. Therefore, Accelerate is an externals-only-preserving transformation.

Finally, we must show that Accelerate is single-message-reflecting. To do so, we establish a loop invariant on the while loop that every message with quantity single in \(\hat{K}\) is also present with quantity single in \(\hat{K}_{\text{init}}\). Before the first iteration of the loop, this is trivially true. In an iteration of the loop, \(\hat{K}\) is only modified if TryTrans returns a configuration. By the definition of TryTrans, that in turn happens only if every message with quantity single in the new configuration is also present in the previous one. Thus, the loop body maintains the invariant, so Accelerate is single-message-reflecting.

We have shown that Accelerate is an approximating, well-formed-preserving, externals-only-preserving, single-message-reflecting transformation. Therefore, by the Approximating Transformation lemma, Accelerate is conformance-reflecting.
L.35Definitions for Evict Proofs

The various proofs used to show that *Evict* is conformance-reflecting rely on some definitions specific to that transformation. First, the symbol $D$ stands for a set of abstract addresses. Second, the below property expresses the idea that a program configuration $\hat{K}'$ is like another configuration $\hat{K}$, except that certain actors were evicted from $\hat{K}'$, and other actors might have different addresses as a result. The partial function $A$ below is the same kind of address-to-address correspondence function used for the abstract interpretation.

**Definition.** A configuration $\hat{K}' = \langle \hat{\beta}' \mid \hat{\mu}' \mid H' \rangle \leftarrow \hat{\rho}''$ is an eviction of $D$ from $\hat{K} = \langle \hat{\beta} \mid \hat{\mu} \mid H \rangle \leftarrow \hat{\rho}$ corresponding via $A$ if and only if

- for all $\hat{a} \in D$,
  - for all $\hat{a}'$ such that either $\hat{a}' = \hat{a}$ or $\hat{a}'$ appears in $\hat{\beta}(\hat{a})$, there is no $\tau$ such that $(\text{Addr } \tau)$ appears in $\text{ActorType}(\hat{a}')$, and
    - there is no spawn expression in $\hat{\beta}(\hat{a})$,
  - $\text{dom}(\hat{\beta}) = A(\text{dom}(\hat{\beta}) - D)$,
  - for all $\hat{a} \in \text{dom}(\hat{\beta}) - D$, $\text{Remap}(\hat{\beta}(\hat{a}), A, id) \sqsubseteq_{\text{id}, \text{id}} \hat{\beta}(A(\hat{a}))$,
  - $\text{Remap}(\hat{\mu}, A, id) \sqsubseteq_{\text{id}, \text{id}} \hat{\mu}'$,
  - for all $(\hat{a}@H, \hat{v}) \in \text{dom}(\hat{\mu})$, $\hat{a} \in D$,
  - $H = H'$,
  - $\text{Remap}(\hat{\rho}, A, id) = \hat{\rho}''$,
  - for all $(\hat{a}@H, \tau) \in \hat{\rho}'$, $\hat{a} \in D$,
  - for all $\hat{a} \in D$, there exists some $\hat{\rho}''' = \hat{\rho}'' \cap \{ (\hat{a}@H'', \tau) \mid \hat{a}' \neq \hat{a} \}$ (where $\hat{\rho}'''$ is a valid typing for the known internal addresses of the actor at $\hat{a}$ in $\hat{K}$) such that for all $(\hat{a}@H'', \tau) \in \hat{\rho}'''$ and all $\hat{v} \in \text{MaxVals}(\tau)$, $\hat{\mu}'(\hat{a}@H'', \hat{v}) = \text{many}$, and
  - for all $\hat{a} \in \text{dom}(A)$,
    - if $\hat{a} \in D$, then $A(\hat{a})$ is external and $\text{ActorType}(\hat{a}) = \text{ActorType}(A(\hat{a}))$, and
    - if $\hat{a} \notin D$, then $\text{AddrLoc}(A(\hat{a})) = \text{AddrLoc}(\hat{a})$. 
Next, the necessary TransExecEvict property is defined partially in terms of a function EvictExec. The idea is that if there is some execution \( \hat{K}_1 \xrightarrow{\hat{l}_1,\ldots,\hat{l}_n} \ldots \), and some configuration \( \hat{K}_1' \) is an eviction of a set \( D \) from \( \hat{K}_1 \) corresponding via \( A \), then EvictExec\((\hat{K}_1 \xrightarrow{\hat{l}_1,\ldots,\hat{l}_n} \ldots,\hat{K}_1',D,A)\) returns an execution starting from \( \hat{K}_1' \) that is “similar” to \( \hat{K}_1 \xrightarrow{\hat{l}_1,\ldots,\hat{l}_n} \ldots \) in that it removes the steps in which one of the evicted actors takes a step. Formally, EvictExec\((\hat{K}_1 \xrightarrow{\hat{l}_1,\ldots,\hat{l}_n} \ldots,\hat{K}_1',D,A)\) is defined only when

- \( \hat{K}_1' \) is an eviction of a set \( D \) from \( \hat{K}_1 \) corresponding via \( A \) and
- \( \text{dom}(A) = \text{ExtAbsAddr} \cup (\text{dom}(\beta) - D) \) (where \( \hat{\beta} \) is the actor-behavior-map component of \( \hat{K}_1 \))

and is defined by the below cases.

1. If there does not exist a transition \( \hat{K}_1 \xrightarrow{\hat{l}_1,\ldots,\hat{l}_n} \hat{K}_{i+1} \) in which some \( \hat{a} \) identifies the active actor for \( \hat{l}_i,1 \) and \( \hat{a} \notin D \), then EvictExec\((\hat{K}_1 \xrightarrow{\hat{l}_1,\ldots,\hat{l}_n} \ldots,\hat{K}_1',D,A)\) returns an execution starting from \( \hat{K}_1' \) (i.e., the zero-step execution from \( \hat{K}_1' \)).

2. Otherwise, if some address \( \hat{a} \) identifies the active actor for \( \hat{l}_i,1 \) and \( \hat{a} \notin D \), then by corollary L.36.1 to the Eviction Skip lemma (below), \( \hat{K}_1' \) is an eviction of \( D \) from \( \hat{K}_2 \) corresponding via \( A \), and \( \text{dom}(A) = \text{ExtAbsAddr} \cup (\text{dom}(\hat{\beta}_2) - D) \), where \( \hat{\beta}_2 \) is the actor-behavior-map component of \( \hat{K}_2 \). Then EvictExec\((\hat{K}_1 \xrightarrow{\hat{l}_1,\ldots,\hat{l}_n} \ldots,\hat{K}_1',D,A) = \text{EvictExec}(\hat{K}_2 \xrightarrow{\hat{l}_1,\ldots,\hat{l}_n} \ldots,\hat{K}_1',D,A) \).

3. Otherwise, it must be the case that some address \( \hat{a} \) identifies the active actor for \( \hat{l}_i,1 \) and \( \hat{a} \notin D \). Then by corollary L.37.1 to the Eviction Simulation lemma (below), there exist \( \hat{l}'_1,\ldots,\hat{l}'_{n} \), \( \hat{K}'_2 \), and \( A' \) such that for all \( i \in 1\ldots n \) either Remap\((\hat{l}_i, A \uplus A', id) \) \( \subseteq id,\hat{\beta} \), or \( |\hat{l}_i| = \cdot \) and \( |\hat{l}'_i| = \hat{a}'@\circ \hat{\tau} \) for some \( \hat{a}' \in \text{rng}(A \uplus A') \) and \( \hat{\beta} \).\( \hat{K}_2' \xrightarrow{\hat{l}'_1,\ldots,\hat{l}'_{n}} \hat{K}_2', \hat{K}'_2 \) is an eviction of \( D \) from \( \hat{K}_2 \) corresponding via \( A \uplus A' \), and \( \text{dom}(A \uplus A') = \text{ExtAbsAddr} \cup (\text{dom}(\hat{\beta}_2) - D) \), where \( \hat{\beta}_2 \) is the actor-behavior-map component of \( \hat{K}_2 \). Then EvictExec\((\hat{K}_1 \xrightarrow{\hat{l}_1,\ldots,\hat{l}_n} \ldots,\hat{K}_1',D,A) = \text{EvictExec}(\hat{K}_2 \xrightarrow{\hat{l}_1,\ldots,\hat{l}_n} \ldots,\hat{K}_2',D,A \uplus A') \).

Finally, the necessary function UntransExecEvict is partially defined in terms of a function SimulateUnevicted that performs a dual role to EvictExec: given a specification execution \( S_1 \xrightarrow{\Lambda_1} \ldots \) that simulates the execution of some post-Evict program configuration, SimulateUnevicted adds to that specification execution the necessary transitions to simulate the corresponding execution of the
pre-Evict program configuration. Formally \( \text{SimulateUnevicted}(\tilde{a}\tilde{e}, S', S_1' \rightarrow^{N_1} \ldots D, A) \) is defined only when there exists \( \tilde{K}_1 \) such that

- \( \text{EvictExec}(\tilde{a}\tilde{e}, \tilde{K}_1', D, A) = \tilde{a}\tilde{e}' \),
- \( \text{Simulates}(S_1' \rightarrow^{N_1} \ldots D, A) \), and
- all addresses in \( D \) are evictable from \( \langle \tilde{K}_1, s \rangle \) for all \( s \in S_1' \) (where \( \tilde{K}_1 \) is the initial configuration of \( \tilde{a}\tilde{e} \)).

Then similarly to \( \text{EvictExec} \), \( \text{SimulateUnevicted}(\tilde{a}\tilde{e}, \tilde{a}\tilde{e}', S_1' \rightarrow^{N_1} \ldots D, A) \) is defined by the following cases. In each of these cases, let \( \tilde{K}_1 \rightarrow^{\tilde{l}_1, \ldots \tilde{l}_1, \ldots \tilde{l}_1, \ldots \tilde{l}_1} = \tilde{a}\tilde{e} \), and let \( \tilde{K}_1 \rightarrow^{\tilde{l}_1, \ldots \tilde{l}_1, \ldots \tilde{l}_1, \ldots \tilde{l}_1} = \tilde{a}\tilde{e}' \).

1. If \( \text{len}(S_1' \rightarrow^{N_1} \ldots) = 0 \), then by the definition of \( \text{Simulates} \), \( \text{len}(\tilde{a}\tilde{e}') = 0 \), and by the definition of \( \text{EvictExec} \), there is no transition in \( \tilde{a}\tilde{e} \) in which some address \( \tilde{a} \) identifies the active actor for that transition and \( \tilde{a} \not\in D \). Then by corollary L.38.2 to the Eviction Specification Skip lemma, there exist \( L_1, \ldots \) such that \( \text{Simulates}(S \rightarrow^{(L_1, \phi)} S \rightarrow^{(L_2, \phi)} \ldots, \tilde{a}\tilde{e}) \). Thus, define \( \text{SimulateUnevicted}(\tilde{a}\tilde{e}, \tilde{a}\tilde{e}', S_1' \rightarrow^{N_1} \ldots D, A) = S \rightarrow^{(L_1, \phi)} S \rightarrow^{(L_2, \phi)} \ldots \).

2. Otherwise, if some address \( \tilde{a} \) identifies the active actor for \( \tilde{l}_1,1 \) and \( \tilde{a} \in D \), then by corollary L.38.1 to the Eviction Specification Skip lemma, there exists \( L_1 \) such that \( L_1 \) summarizes \( \tilde{l}_1,1, \ldots, \tilde{l}_1,1 \), \( S_1' \rightarrow^{(L_1, \phi)} S_1' \), and all addresses in \( D \) are evictable from \( \langle \tilde{K}_2, s \rangle \) for all \( s \in S_1' \). By the definition of \( \text{EvictExec} \) in this case, \( \text{EvictExec}(\tilde{K}_2 \rightarrow^{\tilde{l}_2, \ldots \tilde{l}_2, \ldots \tilde{l}_2, \ldots \tilde{l}_2} \sim \tilde{K}_1', D, A) = \tilde{a}\tilde{e}' \). Then let \( \tilde{S}_2 \rightarrow^{(L_2, O_2)} \ldots = \text{SimulateUnevicted}(\tilde{K}_2 \rightarrow^{\tilde{l}_2, \ldots \tilde{l}_2, \ldots \tilde{l}_2, \ldots \tilde{l}_2} \sim \tilde{a}\tilde{e}', S_1' \rightarrow^{N_1} \ldots D, A) \), and let \( \tilde{S}_1 = \tilde{S}_1' \). Define \( \text{SimulateUnevicted}(\tilde{a}\tilde{e}, \tilde{a}\tilde{e}', S_1' \rightarrow^{N_1} \ldots D, A) = \tilde{S}_1 \rightarrow^{(L_1, \phi)} \tilde{S}_2 \rightarrow^{(L_2, O_2)} \ldots \).

3. Otherwise, some address \( \tilde{a} \) identifies the active actor for \( \tilde{l}_1,1 \) and \( \tilde{a} \notin D \). Let \( \tilde{a}\tilde{e}'' = \tilde{K}_2 \rightarrow^{\tilde{l}_2, \ldots \tilde{l}_2, \ldots \tilde{l}_2, \ldots \tilde{l}_2} \sim \ldots \). By the definition of \( \text{EvictExec} \) in this case, there exist \( \tilde{a}\tilde{e}'' \) and \( A' \) such that

- \( \tilde{a}\tilde{e}'' = \tilde{K}_2 \rightarrow^{\tilde{l}_2, \ldots \tilde{l}_2, \ldots \tilde{l}_2, \ldots \tilde{l}_2} \sim \ldots \),
- \( \text{EvictExec}(\tilde{a}\tilde{e}'', \tilde{K}_2, D, A \uplus A') = \tilde{a}\tilde{e}'' \),
- \( \text{Remap}(\tilde{l}_1, i, A \uplus A', \text{id}) \subseteq_{id, \text{id}} \tilde{l}_1, i \) for all \( i \in 1 \ldots n \), and
• $\hat{K}'_2$ is an eviction of $D$ from $\hat{K}_2$ corresponding via $A \uplus A'$.

By the definition of that last property, we know that for all $\tilde{a} \in \text{dom}(A \uplus A')$, if $(A \uplus A')(\tilde{a})$ is internal, then so is $\tilde{a}$. Because $\text{Simulates}(S'_1, A \uplus A')$, there exist $\langle L'_1, O_1 \rangle = \tilde{\Lambda}'_1$ such that $L'_1$ summarizes $\hat{l}_{1,1}, \ldots, \hat{l}_{1,n}$. Then by corollary L.39.1 to the Eviction Specification Simulation lemma, there exists $L_1$ such that $L_1$ summarizes $\hat{l}_{1,1}, \ldots, \hat{l}_{1,n}$ and $S'_1 \xrightarrow{(L_1, O_1)} S'_2$. By the definition of $\text{Simulates}$, we also have $\text{Simulates}(S'_2, A \uplus A')$. By corollary L.40.1 to the Evictability Preservation lemma, all addresses in $D$ are evictable from $\langle \hat{K}_2, s' \rangle$ for all $s' \in S'_2$. Therefore, let $S_2 \xrightarrow{(L_2, O_2)} \ldots = \text{SimulateUnevicted}(\hat{a} \hat{e} \hat{x}', \hat{a} \hat{e} \hat{x}'', S'_2, A \uplus A')$, and let $S_1 = S'_1$. Define $\text{SimulateUnevicted}(\hat{a} \hat{e} \hat{x}, \hat{a} \hat{e} \hat{x}', S'_1, A \uplus A')$ whenever those are both defined.

### L.36 Eviction Skip Lemma

**Lemma** (Eviction Skip). For all $\hat{K}_1 = \langle \langle \hat{\beta}_1 \mid H_1 \rangle \rangle \hat{\beta}_1, \hat{K}_2 = \langle \langle \hat{\beta}_2 \mid H_2 \rangle \rangle \hat{\beta}_2$, $\hat{K}'_1, \hat{I}, \tilde{a}, D,$ and $A$, if

- $\hat{K}_1 \xrightarrow{RA} \hat{K}_2$,
- $\tilde{a}$ identifies the active actor for $\hat{I}$ and $\tilde{a} \in D$,
- $\hat{K}'_1$ is an eviction of $D$ from $\hat{K}_1$ corresponding via $A$, and
- $\text{dom}(A) = \text{ExtAbsAddr} \cup (\text{dom}(\hat{\beta}_1) - D)$,

then

- $\hat{K}'_1$ is an eviction of $D$ from $\hat{K}_2$ corresponding via $A$ and
- $\text{dom}(A) = \text{ExtAbsAddr} \cup (\text{dom}(\hat{\beta}_2) - D)$.

**Proof.** By the definition of an “eviction of”, the behavior of the actor at $\tilde{a}$ in $\hat{K}_1$ may not contain a $\text{spawn}$ expression. Therefore, the transition $\hat{K}_1 \xrightarrow{\hat{I}} \hat{K}_2$ cannot be a use of the $A$-$\text{SPWAN}$ rule, so $\text{dom}(\hat{\beta}_2) = \text{dom}(\hat{\beta}_1)$, and therefore $\text{dom}(A) = \text{ExtAbsAddr} \cup (\text{dom}(\hat{\beta}_2) - D)$. 

Let there be some partitioning $\hat{\mu}'_2 \sqcup \hat{\mu}''_2$ of $\hat{\mu}_2$ such that $\hat{\mu}''_2$ contains all messages sent to an address in $D$, and $\hat{\mu}''_2$ contains all others. Similarly, let there be some partitioning $\hat{\mu}'_2 \sqcup \hat{\mu}''_2$ such that $\hat{\mu}''_2$ contains the receptionists with addresses in $D$, and $\hat{\mu}''_2$ contains all others. We show the properties necessary for $K'_1$ to an eviction of $D$ from $K_2$ corresponding via $A$ one at a time.

- Let $\hat{a}'$ be a member of $D$. If $\hat{a}' \neq \hat{a}$, then the $\hat{\beta}_2(\hat{a}') = \hat{\beta}_1(\hat{a}')$, so $\hat{\beta}_2(\hat{a}')$ contains neither a spawn expression nor an address $\hat{a}''$ such that $(\text{Addr } \tau)$ appears in $\text{ActorType}(\hat{a}''')$ for some $\tau$. Otherwise, any spawn expression in $\hat{\beta}_2(\hat{a})$ would have had to come from $\hat{\beta}_1(\hat{a})$, so $\hat{\beta}_2(\hat{a})$ contains no spawn expression. Once spawned, an actor can only obtain new addresses by either spawning an actor (which we have shown is impossible for the actor at $\hat{a}$) or receiving one in a message. Because there is no $\tau$ such that $(\text{Addr } \tau)$ appears in $\text{ActorType}(\hat{a})$, the actor at $\hat{a}$ can receive no new addresses, so there is no address $\hat{a}''$ appearing in $\hat{\beta}_2(\hat{a})$ such that $(\text{Addr } \tau)$ appears in $\text{ActorType}(\hat{a}''')$ for some $\tau$.

- We already know that $\text{dom}(\hat{\beta}_1) = A(\text{dom}(\hat{\beta}_1) - D)$, and we have shown that $\text{dom}(\hat{\beta}_1) = \text{dom}(\hat{\beta}_2)$. Therefore, $\text{dom}(\hat{\beta}_1) = A(\text{dom}(\hat{\beta}_2) - D)$.

- Again, we have shown that $\text{dom}(\hat{\beta}_1) = \text{dom}(\hat{\beta}_2)$. We also showed that $\hat{\beta}_2(\hat{a}') = \hat{\beta}_1(\hat{a}')$ for all $\hat{a}' \in \text{dom}(\hat{\beta}_2) - D$, so $\text{Remap}(\hat{\beta}_2(\hat{a}'), A, \text{id}) \subseteq_{\text{id}, \text{id}} \hat{\beta}_1'(A(\hat{a}''))$.

- Any new messages sent in this transition that end up in $\hat{\mu}''_2$ must be sent to one of the known internal addresses in an evicted actor. The message must be of some type $\tau$ at which the evicted actor has access to that address. By the definition of an "eviction of", every member of $\text{MaxVals}(\tau)$ is in $\hat{\mu}''_2$ (the message-map component of $K'_1$) with quantity many, so $\hat{\mu}''_2$ contains a re-mapped approximation for every such message.

- The property for $\hat{\mu}''_2$ holds by definition.

- Because $\hat{a} \in D$, that actor can neither send nor receive addresses, so the set of used markers does not change, and therefore $\text{Used}(K_2) = \text{Used}(K'_1)$.

- Again the actor at $\hat{a}$ can neither send nor receive addresses, so no new receptionists are created, and therefore $\text{Remap}(\hat{\beta}_2, A, \text{id}) = \hat{\beta}''_2$, where $\hat{\beta}''_2$ is the set of receptionists on $K'_1$.

- As argued above, there are no new known internal addresses for any evicted actor, so the valid typings do not change, and the necessary messages exist.

- Finally, the correspondence function $A$ and the set of evicted addresses $D$ do not change, so we know that those properties on $A$ hold.

\[ \square \]
Corollary L.36.1. For all \( K_1 = \langle (\hat{\beta}_1 | \hat{\mu}_1 | H_1) \rangle \hat{\rho}_1 \), \( K_{n+1} = \langle (\hat{\beta}_{n+1} | \hat{\mu}_{n+1} | H_{n+1}) \rangle \hat{\rho}_{n+1} \), \( K_1', \hat{\iota}_1, \ldots, \hat{\iota}_n, \hat{\iota}, D, \) and \( A \), if

- \( \hat{K}_{1} \xrightarrow{\hat{\iota}_1, \ldots, \hat{\iota}_n} \hat{K}_{n+1} \),
- \( \hat{\alpha} \) identifies the active actor for \( \hat{\iota}_1 \) and \( \hat{\alpha} \in D \),
- \( \hat{K}'_1 \) is an eviction of \( D \) from \( \hat{K}_1 \) corresponding via \( A \), and
- \( \text{dom}(A) = \text{ExtAbsAddr} \cup (\text{dom}(\hat{\beta}_1) - D) \),

then

- \( \hat{K}'_1 \) is an eviction of \( D \) from \( \hat{K}_{n+1} \) corresponding via \( A \) and
- \( \text{dom}(A) = \text{ExtAbsAddr} \cup (\text{dom}(\hat{\beta}_2) - D) \).

Proof: By the definition of the event-step transition relation, there exist \( \hat{K}_2, \ldots, \hat{K}_n \) such that \( \hat{K}_1 \xrightarrow{\hat{\iota}_1} \hat{K}_2 \xrightarrow{\hat{\iota}_2} \ldots \xrightarrow{\hat{\iota}_n} \hat{K}_{n+1} \). Because the actor at \( \hat{\alpha} \) cannot contain a spawn expression, \( \hat{\alpha} \) must identify the active actor for \( \hat{\iota}_i \) for all \( i \in 1 \ldots n \). Then the proof is by the Eviction Skip lemma and a straightforward induction on the number of transitions in that sequence. \( \square \)

L.37 Eviction Simulation Lemma

Lemma (Eviction Simulation). For all \( \hat{K}_1 = \langle (\hat{\beta}_1 | \hat{\mu}_1 | H_1) \rangle \hat{\rho}_1 \), \( \hat{K}_2 = \langle (\hat{\beta}_2 | \hat{\mu}_2 | H_2) \rangle \hat{\rho}_2 \), \( \hat{K}_1', \hat{\iota}, \hat{\alpha}, D, \) and \( A \), if

- \( \hat{K}_1 \xrightarrow{\hat{\iota}} \hat{K}_2 \),
- \( \hat{\alpha} \) identifies the active actor for \( \hat{\iota} \) and \( \hat{\alpha} \in D \),
- \( \hat{K}'_1 \) is an eviction of \( D \) from \( \hat{K}_1 \) corresponding via \( A \), and
- \( \text{dom}(A) = \text{ExtAbsAddr} \cup (\text{dom}(\hat{\beta}_1) - D) \),

then there exist \( \hat{\iota}', \hat{K}'_2, \) and \( A' \) such that

- either Remap(\( \hat{\iota}, A \ominus A', id \)) \( \sqsubseteq_{id, id} \hat{\iota}' \), or \( [\hat{\iota}] = \ast \) and \( [\hat{\iota}'] = \hat{\alpha}' \oplus !\hat{\nu} \) for some \( \hat{\alpha}' \in \text{rng}(A \ominus A') \) and \( \hat{\nu} \).
- \( \hat{K}'_1 \xrightarrow{\hat{\iota}'} \hat{K}'_2 \),
- \( \hat{K}'_2 \) is an eviction of \( D \) from \( \hat{K}_2 \) corresponding via \( A \ominus A' \), and
- \( \text{dom}(A \ominus A') = \text{ExtAbsAddr} \cup (\text{dom}(\hat{\beta}_2) - D) \).
Proof. By the definition of an “eviction of”, Remap($\tilde{\beta}_1(\hat{a}), A, id$) $\subseteq_{id,id} \tilde{\beta}_1'(A(\hat{a}))$, where $\tilde{\beta}_1'$ is the actor-behavior-map component of $\tilde{K}_1'$. Therefore, $\tilde{K}_1'$ can take a similar step with the actor at $A(\hat{a})$ to some $\tilde{K}_2'$, with the following differences:

- If an actor is spawned in these two transitions, the address may differ (because $\tilde{K}_1$ and $\tilde{K}_1'$ have different sets of actors, and therefore of used internal addresses). Let $A'$ be the function that gives this mapping for the new addresses: then we have $dom(A \cup A') = ExtAbsAddr \cup (dom(\tilde{\beta}_2) - D)$. Otherwise, let $A' = \emptyset$, and because no actors were added, we have $dom(A \cup A') = ExtAbsAddr \cup (dom(\tilde{\beta}_2) - D)$.

- If this transition sends a message to an actor with an address in $D$, that address corresponds to an external address in $\tilde{K}_1'$, so that transition performs an external send step rather than an internal one. In that case, by the definition of an “eviction of”, we know that the destination address has no markers on it, so no extra markers are created. In that case, we have that $\{\hat{l} = \bullet \}$ and $\{\hat{l}' = \hat{a}'@\emptyset\}$ for some $\hat{a}'$ in $rng(A \cup A')$ and $\hat{c}$. Otherwise, Remap($\hat{l}, A \cup A', id$) $\subseteq_{id,id} \hat{l}'$.

Finally, let there be some partitioning $\bar{\mu}_2' \cup \bar{\mu}_2''$ of $\tilde{\mu}_2$ such that $\bar{\mu}_2''$ contains all messages sent to an address in $D$, and $\bar{\mu}_2'$ contains all others. Similarly, let there be some partitioning $\bar{\beta}_2' \cup \bar{\beta}_2''$ such that $\bar{\beta}_2''$ contains the receptions with addresses in $D$, and $\bar{\beta}_2'$ contains all others. We show that $\tilde{K}_2'$ is an eviction of $D$ from $\tilde{K}_2$ corresponding via $A \cup A'$ one property at a time.

- Let $\hat{a}$ be a member of $D$. Because $\hat{a}$ does not identify the active actor for $\tilde{l}$, its behaviors do not change, so $\tilde{\beta}_2(\hat{a})$ contains neither a spawn expression nor an address $\hat{a}''$ such that (Addr $\tau$) appears in ActorType($\hat{a}''$) for some $\tau$.

- The new correspondence function $A'$ provides the mapping for all new spawned actors (which are not in $D$), so $dom(\tilde{\beta}_2') = A(dom(\tilde{\beta}_2) - D)$.

- Let $\hat{a}'$ be a member of $dom(\tilde{\beta}_2) - D$. Because the actors that transitioned took similar transitions, we have Remap($\tilde{\beta}(\hat{a}''), A \cup A', id$) $\subseteq_{id,id} \tilde{\beta}(A \cup A'(\hat{a}''))$.

- If the transition from $\tilde{K}_1$ sent an internal message, then either $\tilde{K}_2$ sent a corresponding one, or it sent a message to the environment instead. In the latter case, by the definition of an “eviction of” every member of MaxVals($\tau$) (where $\tau$ is the type at which the actor in $\tilde{K}_1$ has access to the destination of the message) must be in the message-map in $\tilde{K}_1'$ (and therefore in $\tilde{K}_2'$) with a quantity of many. Therefore, in either case there is an approximating message in the message map in $\tilde{K}_2'$. If $\tilde{K}_1$ receives an internal message, then $\tilde{K}_2$ receives a similar one (the message cannot be targeted to an evicted actor, because the active actor is not evicted). Therefore, Remap($\bar{\mu}_2'', A, id$) $\subseteq_{id,id} \bar{\mu}_2''$, where $\bar{\mu}_2''$ is the message-map component for $\tilde{K}_2'$.

- By the definition of $\bar{\mu}_2''$, every message in $\bar{\mu}_2'$ has an evicted actor as its destination.
• If the transition labels correspond via Remap, then the sets of used markers in each configuration must be the same because both configurations took corresponding transitions. Otherwise, we already know that $\lfloor \hat{l}_1 \rfloor = \hat{a} \oplus \phi \hat{v}$, and so because there are no markers on that address, no new markers are created, so the sets of used markers in each configuration must be the same.

• Any new receptionist for a non-evicted actor must have some $A$-correspondent in the message sent in the transition from $\hat{K}_1'$. Therefore, a corresponding receptionist (via Remap) exists.

• By the definition of $\hat{\rho}'_2''$, the address of every receptionist in $\hat{\rho}'_2''$ is a member of $D$.

• The behaviors of the evicted actors did not change, so their valid typings for the known internal addresses did not change, and therefore $\hat{K}_2$ still has the necessary messages from MaxVals with quantity many.

• Let $\hat{a}'$ be a member of $\text{dom}(A \cup A')$. If $\hat{a}' \in \text{dom}(A)$, then we already know that the necessary property holds. Otherwise, $\hat{a}'$ is an internal address from a spawn expression, and $A'(\hat{a}')$ is the corresponding address from evaluating the corresponding spawn. By the definition of Remap and $\sqsubseteq_{\text{id}, \text{id}}$, those two expressions must have the same location, so $\text{AddrLoc}(\hat{a}') = \text{AddrLoc}(A'(\hat{a}'))$.

\[ \square \]

**Corollary L.37.1.** For all $\hat{K}_1 = \langle \hat{\mu}_1, \hat{H}_1 \rangle$, $\hat{K}_{n+1} = \langle \hat{\mu}_{n+1}, \hat{H}_{n+1} \rangle$, $\hat{K}_1'$, $\hat{K}_{n+1}'$, $\hat{\beta}_1, \ldots, \hat{\beta}_n$, $\hat{a}$, $D$, and $A$, if

• $\hat{K}_1'$ is an eviction of $D$ from $\hat{K}_1$ corresponding via $A$, and

• $\text{dom}(A) = \text{ExtAbsAddr} \cup (\text{dom}(\hat{\beta}_1) - D)$,

then there exist $\hat{l}_1', \ldots, \hat{l}_n', \hat{K}_{n+1}'$, and $A'$ such that

• for all $i \in 1 \ldots n$, either Remap($\hat{l}_i, A \cup A', \text{id}$) $\sqsubseteq_{\text{id}, \text{id}} \hat{l}_i$ or $[\hat{l}_i] = \cdot$ and $[\hat{l}_i] = \hat{a}' \oplus \phi \hat{v}$ for some $\hat{a}' \in \text{rng}(A \cup A')$ and $\hat{v}$.

• $\hat{K}_1' \xrightarrow{\hat{l}_1', \ldots, \hat{l}_n'} \hat{K}_{n+1}'$.

• $\hat{K}_{n+1}'$ is an eviction of $D$ from $\hat{K}_{n+1}$ corresponding via $A \cup A'$, and

• $\text{dom}(A \cup A') = \text{ExtAbsAddr} \cup (\text{dom}(\hat{\beta}_2) - D)$. 


Proof. By the definition of the event-step transition relation, there exist $K_2, \ldots, K_n$ such that $K_1 \xrightarrow{I_1} RA \cdots \xrightarrow{I_n} RA K_{n+1}$. Most of the proof is then by the Eviction Simulation lemma and a straightforward induction on the number of transitions in that sequence. Let $A'$ be the disjoint union of all the various $A_i'$ produced by each use of the Eviction Simulation lemma. We also know that the set of actors in those configurations can only grown over time (no actor is ever removed from a configuration by a transition), so $\text{dom}(A \cup A') = \text{ExtAbsAddr} \cup (\text{dom}(\beta_2) - D)$.

L.38. EVICTION SPECIFICATION SKIP LEMMA

**Lemma (Eviction Specification Skip).** For all $K_1, K_2, I, a, D$, and $S$, if

- $K_1 \xrightarrow{I} RA K_2$,
- $a$ identifies the active actor for $I$ and $a \in D$,
- all addresses in $D$ are evictable from $(K_1,s)$ for all $s \in S$,

then $S \xrightarrow{[I],\varnothing} S$ and all addresses in $D$ are evictable from $(K_2,s)$ for all $s \in S$.

Proof. If $[I] = \varnothing$, then we have $S \xrightarrow{[I],\varnothing} S$ by the definition of the $\rightarrow$ relation. Otherwise, if $[I] = a'@H?\nu$ for some $a'$, $H$, and $\nu$, then it must be the case that $\check{a}' = \check{a}$. Because $\check{a}$ is evictable from $(K_1,s)$ for all $s \in S$, we know that $\text{InMon}(S) \cap H = \varnothing$, and so a transition $S \xrightarrow{[I],\varnothing} S$ is possible by the P-UNMONITOREDRECEIVE rule for each PSM, and therefore $S \xrightarrow{[I],\varnothing} S$. Finally, it might be the case that $[I] = a'@H?\nu$ for some $a'$, $H$, and $\nu$. The marked address $a'@H$ must appear in a behavior of the actor at $\check{a}$ in $K_1$, so because $\check{a}$ is evictable from $(K_1,s)$ for all $s \in S$, we know that $\text{OutMon}(S) \cap H = \varnothing$. Therefore a transition $S \xrightarrow{[I],\varnothing} S$ is possible by the P-SEND rule for each PSM, and therefore $S \xrightarrow{[I],\varnothing} S$.

Next, let $a'$ be a member of $D$, and let $s$ be a member of $S$. We know that $a'$ is evictable from $(K_1,s)$; we must show that $a'$ is evictable from $(K_2,s)$. We already have that $\text{holds}$. By the property, both $a'$ and any address appearing in a behavior of $a'$ in $K_1$ have restricted types that cannot receive an address, and therefore no new addresses appear in the behavior of $a'$ in $K_2$. Therefore, both and $\text{hold}$ holds by the Abstract Type Preservation lemma (appendix J). Finally, any spawn expression appearing in a behavior of $a'$ in $K_2$ would have to come from some spawn expression appearing in the behavior of $a'$ in $K_1$. Since there are none in $K_1$ by , we know that $\text{holds}$ for $K_2$.

Corollary L.38.1. For all $K_1$, $K_{n+1}$, $I_1, \ldots, I_n$, $a$, $D$, and $S$, if

- $K_1 \xrightarrow{I_1 \ldots I_n} RA K_{n+1}$,
• \(\hat{a}\) identifies the active actor for \(\hat{\bar{t}}_1\) and \(\hat{a} \in D\).

• all addresses in \(D\) are evictable from \(\langle \tilde{K}_1, s \rangle\) for all \(s \in S\),

then there exists \(L\) such that

• \(L\) summarizes \(\hat{\bar{t}}_1, \ldots, \hat{\bar{t}}_n\),

• \(S \xrightarrow{(L, B)} S\), and

• all addresses in \(D\) are evictable from \(\langle \tilde{K}_{n+1}, s \rangle\) for all \(s \in S\).

**Proof.** Let \(\hat{a}_1': \text{send-ext}(\bar{a}_1''@H_1, \hat{v}_1), \ldots, \hat{a}_m': \text{send-ext}(\bar{a}_m''@H_m, \hat{v}_m)\) be the \text{send-ext} labels in \(\hat{\bar{t}}_1, \ldots, \hat{\bar{t}}_n\), and let \(\hat{\mu} = \emptyset \oplus (\bar{a}_1''@H_1, \hat{v}_1) \ldots \oplus (\bar{a}_m''@H_m, \hat{v}_m)\). Then let \(L = \langle \hat{\bar{t}}_1, \hat{\mu} \rangle\); we have that \(L\) summarizes \(\hat{\bar{t}}_1, \ldots, \hat{\bar{t}}_n\).

Next, by the definition of the event-step transition relation, we know that there exist \(\tilde{K}_2, \ldots, \tilde{K}_n\) such that \(\tilde{K}_1 \xrightarrow{\hat{\bar{t}}_1} \tilde{K}_2 \xrightarrow{\hat{\bar{t}}_2} \ldots \xrightarrow{\hat{\bar{t}}_n} \tilde{K}_{n+1}\). We will show by induction on \(i\) that for all \(i \in 0 \ldots n\), \(S \xrightarrow{\langle \hat{\bar{t}}_i; \rangle} \ldots \xrightarrow{\langle \hat{\bar{t}}_i; \rangle} S\) and all addresses in \(D\) are evictable from \(\langle \tilde{K}_{i+1}, s \rangle\) for all \(s \in S\). In the base case, there is no label to transition, so the first condition holds. By the precondition to this lemma, all addresses in \(D\) are evictable from \(\langle \tilde{K}_1, s \rangle\). In the inductive case, we know that \(S \xrightarrow{\langle \hat{\bar{t}}_i; \rangle} \ldots \xrightarrow{\langle \hat{\bar{t}}_{i+1}; \rangle} S\) and all addresses in \(D\) are evictable from \(\langle \tilde{K}_{i+1}, s \rangle\). An evictable actor cannot have a \text{spawn} expression in any of its behaviors, so \(\hat{a}\) must still be the active actor for \(\hat{\bar{t}}_i\). Therefore, by the Eviction Specification Skip lemma, \(S \xrightarrow{\langle \hat{\bar{t}}_i; \rangle} \ldots \xrightarrow{\langle \hat{\bar{t}}_{i+1}; \rangle} S\) and all addresses in \(D\) are evictable from \(\langle \tilde{K}_{i+1}, s \rangle\).

It remains to show that \(S \xrightarrow{(L, B)} S\). Let \(\hat{\bar{t}}_1, \ldots, \hat{\bar{t}}_p\) be a sequence summarized by \(L\), and let there be some \(i \in 1 \ldots p\). If \(\langle \hat{\bar{t}}_i; \rangle = \bullet\), then we have \(S \xrightarrow{\langle \hat{\bar{t}}_i; \rangle} S\) by the definition of the \(\xrightarrow{}\) relation. If \(\langle \hat{\bar{t}}_i; \rangle = \hat{a}'@H!\bar{v}\) for some \(\hat{a}'\), \(H\), and \(\bar{v}\), then \(\hat{\bar{t}}_i\) must be \(\hat{\bar{t}}_1\) by the definition of a summary transition label, and we have already shown that \(S \xrightarrow{\langle \hat{\bar{t}}_i; \rangle} S\). Finally, if \(\langle \hat{\bar{t}}_i; \rangle = \hat{a}'@H!\bar{v}'\) for some \(\hat{a}'\), \(H\), and \(\bar{v}'\), then there must be some \(j \in 1 \ldots n\) and some \(\bar{v}''\) such that \(\bar{v}''[d_{id}] \subseteq \bar{v}'\) and \(\langle \hat{\bar{t}}_j; \rangle = \hat{a}@H!\bar{v}''\) (We have already shown that \(S \xrightarrow{\langle \hat{\bar{t}}_j; \rangle} S\), and the approximation \(\subseteq\) does not affect this transition (the matchable markers in \(\hat{v}\) must come from positions that also appear in \(\bar{v}'\)), so \(S \xrightarrow{\langle \hat{\bar{t}}_j; \rangle} S\). Therefore, \(S \xrightarrow{\langle \hat{\bar{t}}_i; \rangle} \ldots \xrightarrow{\langle \hat{\bar{t}}_j; \rangle} S\), and therefore \(S \xrightarrow{(L, B)} S\). \(\square\)

**Corollary L.38.2.** For all \(\bar{a} \bar{e} \bar{e}, \tilde{K}_1, \ldots, \hat{\bar{t}}_{i_1}, \ldots, \hat{\bar{t}}_{i_n}, \hat{a}, D,\) and \(S\), if

• \(\bar{a} \bar{e} \bar{e} = \tilde{K}_1 \xrightarrow{\hat{\bar{t}}_{i_1}} \ldots \xrightarrow{\hat{\bar{t}}_{i_n}} \tilde{K}_{n+1}\)

• \(\hat{a}\) identifies the active actor for \(\hat{\bar{t}}_{i_1}\) for all transitions \(\tilde{K}_i \xrightarrow{\hat{\bar{t}}_{i_1}} \ldots \xrightarrow{\hat{\bar{t}}_{i_n}} \tilde{K}_{i+1}\) in the execution and \(\hat{a} \in D\),
• all addresses in D are evictable from \( \langle \hat{R}_1, s \rangle \) for all \( s \in S \),

then there exist \( L_1, \ldots \) such that Simulates(\( S \xrightarrow{(L_1, s)} S \xrightarrow{(L_2, s)} \cdots \xrightarrow{(L_n, s)} \alpha \delta \)).

Proof. By a straightforward coinductive argument with corollary L.38.1, using each individual event-step transition to show progress. \( \square \)

L.39 Eviction Specification Simulation Lemma

Lemma (Eviction Specification Simulation). For all \( S_1, S_2, \hat{I}, \hat{I}, O, \) and \( A \), if

\( S_1 \xrightarrow{[\hat{I}], O} S_2 \),

• either Remap(\( \hat{I}, A, id \)) \( \subseteq_{id, id} \hat{I} \) or \( \hat{I} = \bullet \) and \( \hat{I} = \hat{a}' \! \circ \! \phi \! \circ \! \delta \) for some \( \hat{a}' \in \text{rng}(A) \) and \( \delta \), and

• for all \( \hat{a} \in \text{dom}(A) \), if \( A(\hat{a}) \) is internal then \( \hat{a} \) is internal,

then \( S_1 \xrightarrow{[\hat{I}], O} S_2 \).

Proof. If \( \hat{I} = \bullet \), then by the definition of Remap and \( \{_{-}\} \), \( \hat{I} = \bullet \). Therefore, \( S_1 \xrightarrow{[\hat{I}], O} S_2 \).

If \( \hat{I} = \hat{a}' \! \circ \! H \! \circ \! \delta' \) for some \( \hat{a}' \), \( H \), and \( \delta' \), then by the definition of Remap and \( \{_{-}\} \) there exist \( \hat{a} \) and \( \delta \) such that \( A(\hat{a}) = \hat{a}' \) and \( \text{Remap}(\delta, A, id) = \delta' \). Therefore, by the input-pattern-matching rules, any input pattern that matches \( \delta' \) can match \( \delta \) in the same way (producing the same matched markers), so we have \( S_1 \xrightarrow{\hat{I}, O} S_2 \).

Finally, if \( \hat{I} = \hat{a}' \! \circ \! H \! \circ \! \delta' \) for some \( \hat{a}' \), \( H \), and \( \delta' \), then either there exist \( \hat{a} \) and \( \delta \) such that \( A(\hat{a}) = \hat{a}' \) and \( \text{Remap}(\delta, A, id) = \delta' \), or \( \hat{I} = \bullet \). In the former case, much like the case for \( \hat{I} = \hat{a}' \! \circ \! H \! \circ \! \delta' \) above, the output-pattern-matching rules can match \( \delta \) in the same way as \( \delta' \), specifically because any internal address in \( \delta' \) (the only addresses that can be matched by a self or delayed-fork-addr pattern) must correspond to an internal address in \( \delta \). Therefore \( S_1 \xrightarrow{[\hat{I}], O} S_2 \). Otherwise, when \( \hat{I} = \bullet \), we also know that \( H = \emptyset \). By the definition of \( \rightarrow \), there exist \( S_3 \) and \( S_4 \) such that \( S_1 \rightarrow^{*} S_3 \xrightarrow{[\hat{I}], O} S_4 \rightarrow^{*} S_4 \). Then because \( H = \emptyset \), the transition \( S_3 \xrightarrow{[\hat{I}], O} S_4 \) fulfills no obligations, so \( O = \emptyset \) and \( S_4 = S_3 \). Therefore we have \( S_3 \rightarrow^{*} S_4 \), and therefore \( S_1 \xrightarrow{[\hat{I}], O} S_2 \).

Corollary L.39.1. For all \( L', \hat{I}_1, \ldots, \hat{I}_n, \hat{I}_1', \ldots, \hat{I}_n', S_1, S_2, O, \text{ and } A \) if

• \( L' \) summarizes \( \hat{I}_1, \ldots, \hat{I}_n, \)

\( S_1 \xrightarrow{(L', O)} S_2, \)

• for all \( i \in \{1, \ldots, n\} \), either Remap(\( \hat{I}_i, A, id \)) \( \subseteq_{id, id} \hat{I}_i \) or \( \hat{I}_i = \bullet \) and \( \hat{I}_i = \hat{a}' \! \circ \! \phi \! \circ \! \delta \) for some \( \hat{a}' \in \text{rng}(A) \) and \( \delta \), and
• for all $\hat{a} \in \text{dom}(A)$, if $A(\hat{a})$ is internal then $\hat{a}$ is internal,

then there exists $L$ such that $L$ summarizes $\hat{I}_1, \ldots, \hat{I}_n$ and $S_1 \xrightarrow{(L,O)} S_2$.

Proof. Let $\hat{a}_1' \colon \text{send-ext}(\hat{a}_1@H_1', \hat{v}_1), \ldots, \hat{a}_m' \colon \text{send-ext}(\hat{a}_1@H_1, \hat{v}_m)$ be the $\text{send-ext}$ labels in $\hat{I}_1, \ldots, \hat{I}_n$, and let $\hat{\mu} = \emptyset \oplus \langle \hat{a}_1@H_1, \hat{v}_1 \rangle \ldots \oplus \langle \hat{a}_m@H_m, \hat{v}_m \rangle$. Then let $L = \langle \hat{I}_1, \hat{\mu} \rangle$; we have that $L$ summarizes $\hat{I}_1, \ldots, \hat{I}_n$.

Next, let $\hat{I}_1’’, \ldots, \hat{I}_p’’$ be a sequence summarized by $L$. By the construction of $L$ and because for all $i \in 1 \ldots n$, either $\text{Remap}(\hat{I}_i, A, id) \subseteq_{id, id} \hat{I}_i$ or $|\hat{I}_i| = \bullet$ and $|\hat{I}_i| = \hat{a}’@\emptyset!\hat{v}$ for some $\hat{a}’ \in \text{rng}(A)$ and $\hat{v}$, there exist some $\hat{I}_1’’’, \ldots, \hat{I}_p’’$ summarized by $L$ such that for all $j \in 1 \ldots p$, either $\text{Remap}(\hat{I}_j’’, A, id) \subseteq_{id, id} \hat{I}_j’’$ or $|\hat{I}_j’’| = \bullet$ and $|\hat{I}_j’’| = \hat{a}’@\emptyset!\hat{v}$ for some $\hat{a}’ \in \text{rng}(A)$ and $\hat{v}$. Then because $S_1 \xrightarrow{(L,O)} S_2$ there exist $O_1, \ldots, O_p$ such that $S_1 \langle \hat{I}_1’’’, \ldots, \hat{I}_p’’ \rangle S_2$ and $O \subseteq \bigcup_{i \in 1 \ldots p} O_i$. By induction on the number of transitions in that transition and the Eviction Specification Simulation lemma, we have that $S_1 \xrightarrow{(\hat{I}_1’’’, \ldots, \hat{I}_p’’)} S_2$. Therefore, $S_1 \xrightarrow{(L,O)} S_2$. $\square$

L.40 Evictability Preservation Lemma

Lemma (Evictability Preservation). For all $D, \hat{K}, \hat{K}', S, S'$, and $\hat{I}$, if

• all addresses in $D$ are evictable from $\langle \hat{K}, s \rangle$ for all $s \in S$,

• $\hat{K} \xrightarrow{\hat{I}} \hat{K}'$,

• $S \xrightarrow{\hat{I}} S'$, and

• if $|\hat{I}| = \hat{a}@H!\hat{v}$ for some $\hat{a}$, $H$, and $\hat{v}$, then there exist $\hat{a}', \hat{v}'$, and $A$ such that $S \xrightarrow{\hat{a}@H!\hat{v}} S'$, $\text{Remap}(\hat{v}, A, id) = \hat{v}'$ and $A(\hat{a}')$ is external for all $\hat{a}' \in D$,

then all addresses in $D$ are evictable from $\langle \hat{K}', s' \rangle$ for all $s' \in S'$.

Proof. Let there be some $\hat{a} \in D$. We already know that $\hat{a}$ is evictable from $\langle \hat{K}, s \rangle$.

We will show each of the properties necessary for $\hat{a}$ to be evictable from $\langle \hat{K}', s' \rangle$ one at a time.

• The location of $\hat{a}$ is still the same, so holds.

• The actor at $\hat{a}$ in $\hat{K}$ is not able to receive new addresses because of the restriction, and it is not able to spawn new actors because of the restriction. Therefore, every marked addresses $\hat{a}@H''$ appearing in a behavior for the actor at $\hat{a}$ in $\hat{K}'$ also appeared in a behavior for that actor in $\hat{K}$, so $H'' \cap \text{OutMon}(S) = \emptyset$. Every member of $\text{OutMon}(S') - \text{OutMon}(S)$ must be a fresh marker, so $H'' \cap \text{OutMon}(S') = \emptyset$, and so evprop:NoMonAddr holds.
L.41. EVICTION SYNCHRONIZATION LEMMA

For all $\eta \in \text{InMon}(S') - \text{InMon}(S)$, $\eta$ must be the maximal marker on some internal address $\hat{a}'$ sent to the environment in the transition $\hat{K} \xrightarrow{\eta} \hat{K}'$. In that case, $\hat{[l]}$ must be of the form $\hat{a}''@H!\hat{v}$. Then by the precondition to this lemma, there exist $\hat{a}''$, $\hat{v}'$, and $A$ such that $S \xrightarrow{\hat{a}''@H!\hat{v}'} S'$, $\text{Remap}(\hat{v}, A, id) = \hat{v}'$ and $A(\hat{a}'')$ is external for all $\hat{a}'' \in D$. If $\hat{a}'$ were a member of $D$, then $A(\hat{a}')$ would be external, and therefore markers on it would not be matchable in the transition $S \xrightarrow{\hat{a}''@H!\hat{v}'} S'$. Therefore, $\hat{a}' \neq \hat{a}$, so there are no new monitored receptionists for $\hat{a}$. Therefore, holds.

As argued above, every marked addresses $\hat{a}''@H'''$ appearing in a behavior for the actor at $\hat{a}$ in $\hat{K}'$ also appeared in a behavior for that actor in $\hat{K}$. We know that holds for $\hat{K}$, so therefore it must also hold for $\hat{K}'$.

The evprop:WellTyped property holds by the Abstract Type Preservation lemma (appendix J).

Any spawn expression in a behavior for the actor at $\hat{a}$ in $\hat{K}'$ must come from a spawn expression in a behavior for the actor at $\hat{a}$ in $\hat{K}$. We know that holds for $\hat{K}$, so therefore it also holds for $\hat{K}'$.

Corollary L.40.1. For all $D$, $\hat{K}_1$, $\hat{K}_{n+1}$, $S_1$, $S_{n+1}$, and $\hat{l}_1, \ldots, \hat{l}_n$, if

- all addresses in $D$ are evictable from $\langle \hat{K}_1, s \rangle$ for all $s \in S_1$,
- $\hat{K}_1 \xrightarrow{\hat{l}_1, \ldots, \hat{l}_n} \hat{K}_{n+1}$,
- $L$ summarizes $\hat{l}_1, \ldots, \hat{l}_n$,
- $S_1 \xrightarrow{\hat{l}_1, \ldots, \hat{l}_n} S_{n+1}$, and
- for all $i \in 1, \ldots, n$, if $\hat{[l]}_i = \hat{a}@H!\hat{v}$ for some $\hat{a}$, $H$, and $\hat{v}$, then there exist $\hat{a}'$, $\hat{v}'$, and $A$ such that $S_i \xrightarrow{\hat{a}''@H!\hat{v}'} S_{i+1}$, $\text{Remap}(\hat{v}, A, id) = \hat{v}'$ and $A(\hat{a}'')$ is external for all $\hat{a}'' \in D$,

then all addresses in $D$ are evictable from $\langle \hat{K}_{n+1}, s' \rangle$ for all $s' \in S_{n+1}$.

Proof. By the Evictability Preservation lemma and a straightforward induction on $n$. 

L.41 Eviction Synchronization Lemma

Lemma (Eviction Synchronization). For all $\bar{a}e\bar{x} = \hat{K}_1 \xrightarrow{\hat{l}_{1,1}, \ldots, \hat{l}_{1,n}} \ldots \xrightarrow{\hat{l}_{n,1}, \ldots, \hat{l}_{n,n}} \hat{K}_n$, $\hat{K}'$, $D$, and $A$, if $\text{EvictExec}(\bar{a}e\bar{x}, D, A) = \bar{a}e\bar{x}'$ and $\hat{K}_i'$ is a configuration in $\bar{a}e\bar{x}'$, then there exists a configuration $\hat{K}_j'$ in $\bar{a}e\bar{x}$ and some $A'$ such that $\hat{K}'_j$ is an eviction of $D$ from $\hat{K}_j'$ corresponding via $A \cup A'$. 
Proof. The proof is by induction on $i$, and it proceeds by the cases in the definition of EvictExec.

- If there does not exist a transition $\hat{K}_k \xrightarrow{\hat{1}, \ldots, \hat{j}, n} \hat{K}_{k+1}$ in which some $\hat{a}$ identifies the active actor for $\hat{1}, \ldots, \hat{j}$ and $\hat{a} \notin D$, then $\hat{a} \hat{e} \hat{x} = \hat{K}'$. Then it must be the case that $\hat{K}' = \hat{K}''$, and by the definition of EvictExec, $\hat{K}''$ is an eviction of $D$ from $\hat{K}_1$ corresponding via $A$. Let $j = 1$ and $A' = \emptyset$ to complete the proof.

- Otherwise, if some address $\hat{a}$ identifies the active actor for $\hat{1}, \ldots, \hat{j}$ and $\hat{a} \in D$, we know there must be some transition in $\hat{a} \hat{e} \hat{x}$ where an address not in $D$ identifies the active actor for that transition. Let $\hat{K}_k \xrightarrow{\hat{1}, \ldots, \hat{j}, n} \hat{K}_{k+1}$ be the first such transition. By induction on $k$ and by the definition of EvictExec, $\hat{a} \hat{e} \hat{x}' = \text{EvictExec}(\hat{K}_k \xrightarrow{\hat{1}, \ldots, \hat{j}, n} \ldots, \hat{K}''', D, A)$. Then the next case applies.

- In the final case, there must be some address $\hat{a}$ that identifies the active actor for $\hat{1}, \ldots, \hat{j}$ and $\hat{a} \notin D$. Let $\hat{a} \hat{e} \hat{x}'' = \text{EvictExec}(\hat{K}''', D, A)$. Then the next case applies.

$\square$

L.42 Eviction Disabled Actor Lemma

Lemma (Eviction Disabled Actor). For all $\hat{a} \hat{e} \hat{x} = \hat{K}_1 \xrightarrow{\hat{1}, \ldots, \hat{j}, n} \ldots, \hat{K}''$, $D$, $A$, and $\hat{a}$, if

- $\text{EvictExec}(\hat{a} \hat{e} \hat{x}, \hat{K}'', D, A) = \hat{a} \hat{e} \hat{x}', k$
- $\hat{K}_k$ is a transition in $\hat{a} \hat{e} \hat{x}$ and there exists no $k \geq i$ such that $\hat{K}_k$ is a transition in $\hat{a} \hat{e} \hat{x}$ in which the actor at $\hat{a}$ is necessarily enabled, and
- $\hat{a} \notin D$,

then there exists $j \leq i$ such that $\hat{K}'_j$ is a configuration in $\hat{a} \hat{e} \hat{x}'$ in which the actor at $A(\hat{a})$ is not necessarily enabled.

Proof. The proof is by induction on $i$ and proceeds by case analysis on the cases in the definition of EvictExec.
L.43. **Eviction Running Actor Lemma**

**Lemma** (Eviction Running Actor). For all $\overline{a\text{ex}} = \overline{K}_1 \overset{\overline{\ell}_{1,1}, \ldots, \overline{\ell}_{1,n}}{\rightarrow} \overline{K}_2 \overset{\overline{\ell}_{2,1}, \ldots, \overline{\ell}_{2,n}}{\rightarrow} \overline{K}_3 \overset{\overline{\ell}_{3,1}, \ldots, \overline{\ell}_{3,n}}{\rightarrow} \cdots, \overline{K}_n$, $D$, $A$, and $\overline{a}$, if

- $\text{EvictExec}(\overline{a\text{ex}}, \overline{K}_n, D, A) = \overline{a\text{ex}}'$
- $\overline{K}_i \overset{\overline{\ell}_{i,1}, \ldots, \overline{\ell}_{i,n}}{\rightarrow} \overline{K}_{i+1}$ is a transition in $\overline{a\text{ex}}$ in which $\overline{a}$ identifies the active actor for $\overline{\ell}_{i,1}$, and
- $\overline{a} \notin D$,

then there exists $j \leq i$ such that $\overline{K}_j \overset{\overline{\ell}_{j,1}, \ldots, \overline{\ell}_{j,n}}{\rightarrow} \overline{K}_{j+1}$ is a transition in $\overline{a\text{ex}}'$ in which $A(\overline{a})$ identifies the active actor for $\overline{\ell}_{j,1}$.
Proof. Similar to the proof of the Eviction Disabled Actor lemma, using an induction on \( i \) and showing that in the case where some \( \hat{a} \) identifies the active actor for \( \hat{I}_{1,1} \), \( \hat{a} \notin \text{D} \), the construction of \( \hat{I}_{1,1} \) in \text{EvictExec} shows that \( A(\hat{a}) \) identifies the active actor for \( \hat{I}_{1,1} \). \( \square \)

**L.44 Eviction Message Receive Lemma**

**Lemma** (Eviction Message Receive). For all \( \overrightarrow{aex} = \hat{R}_1 \overset{\hat{I}_{1,1} \ldots \hat{I}_{1,n}}{\longrightarrow} \hat{R}^\prime \), \( \overrightarrow{aex'} = \hat{R}_1' \overset{\hat{I}_{1,1}' \ldots \hat{I}_{1,n}'}{\longrightarrow} \hat{R}^{\prime \prime} \), \( D, A, \hat{a}, \hat{a}', \hat{H}, \hat{v}, \) and \( \hat{v}' \), if

- \text{EvictExec}(\overrightarrow{aex}, \hat{R}_1, D, A) = \overrightarrow{aex'},
- \hat{K}_k = \langle \hat{\beta}_k | \hat{\mu}_k | H_k \rangle \forall i \in 1 \ldots \text{len}(\overrightarrow{aex}) + 1,
- \hat{K}_k' = \langle \hat{\beta}_k' \mid \hat{\mu}_k' \mid H_k' \rangle \forall j \in 1 \ldots \text{len}(\overrightarrow{aex'}) + 1,
- \hat{\mu}_1(\hat{a}@H, \hat{v}) = \text{single},
- either
  - \( \hat{K}_i \) is a configuration in \( \overrightarrow{aex} \) such that \( \langle \hat{a}@H, \hat{v} \rangle \notin \text{dom}(\hat{\mu}_i) \) or \( \hat{\mu}_i(\hat{a}@H, \hat{v}) = \text{many}, \) or
  - \( \hat{K}_i \overset{\hat{I}_{1,1} \ldots \hat{I}_{1,n}}{\longrightarrow} \hat{K}_{i+1} \) is a transition in \( \overrightarrow{aex} \) such that \( \hat{I}_{1,1} = \hat{\alpha}: \text{rcv-int}(H, \hat{v}) \),
- \( A(\hat{a}) = \hat{a}' \),
- \( \text{Remap}(\hat{v}, A, \text{id}) \subseteq \text{id}, \text{id} \hat{v}' \), and
- for all \( \hat{K}_k \) in \( \overrightarrow{aex} \), \( \hat{\mu}'_k(\hat{a}'@H, \hat{v}') = \text{single}, \)

then there exists \( j \leq i \) such that \( \hat{K}_j \overset{\hat{I}_{1,1}' \ldots \hat{I}_{j,n}'}{\longrightarrow} \hat{K}_{j+1}' \) is a transition in \( \overrightarrow{aex'} \) such that \( \hat{I}_{j,1}' = \hat{a}' : \text{rcv-int}(H, \hat{v}'). \)

**Proof.** The proof is by induction on \( i \) and proceeds by case analysis on the cases in the definition of \text{EvictExec}. Note that by the definition of an “eviction of”, it must be the case that \( \hat{a} \notin \text{D} \).

- If there does not exist a transition \( \hat{K}_k \overset{\hat{I}_{k,1} \ldots \hat{I}_{k,n}}{\longrightarrow} \hat{K}_{k+1} \) in which some \( \hat{a}'' \) identifies the active actor for \( \hat{I}_{k,1} \) and \( \hat{a}'' \notin \text{D} \), then there must be no transition in that execution that receives \( \langle \hat{a}@H, \hat{v} \rangle \), because \( \hat{a} \notin \text{D} \). That implies there is some future configuration in \( \overrightarrow{aex} \) in which either \( \langle \hat{a}@H, \hat{v} \rangle \) is not a message in the configuration or it has quantity \text{many}. That could only happen if one of the evicted actors sent a message to \( \hat{a} \), though, and we know by the definition of an “eviction of” that for every message an evicted actor can
SimulateUnevicted Simulation Lemma

Lemma (SimulateUnevicted Simulation). For all \( \tilde{a}_\varepsilon = \hat{K}_1 \xrightarrow{\tilde{I}_{1,1},...,\tilde{I}_{1,\delta}} \cdots \), \( \tilde{a}_\varepsilon' = \hat{K}_1' \xrightarrow{\tilde{I}_1',...,\tilde{I}_\delta'} \cdots, S_1 \xrightarrow{\Lambda_1} \cdots, S_1' \xrightarrow{\Lambda_1'} \cdots, D, \) and \( A \), if \( \text{SimulateUnevicted}(\tilde{a}_\varepsilon, \tilde{a}_\varepsilon', S_1' \xrightarrow{\Lambda_1'} \cdots, D, A) = S_1 \xrightarrow{\Lambda_1} \cdots \), then \( S_1 = S_1' \) and \( \text{Simulates}(S_1 \xrightarrow{\Lambda_1} \cdots, \tilde{a}_\varepsilon) \).

Proof: The proof is by coinduction on the transitions in \( \tilde{a}_\varepsilon \). The proof proceeds by case analysis on the cases in the definition of SimulateUnevicted. Note that by the preconditions for SimulateUnevicted, we know that \( \text{EvictExec}(\tilde{a}_\varepsilon, \hat{K}_1', D, A) = \tilde{a}_\varepsilon' \), \( \text{Simulates}(S_1' \xrightarrow{\Lambda_1'} \cdots, \tilde{a}_\varepsilon') \), and all addresses in \( D \) are evictable from \( \langle \hat{K}_1, s \rangle \) for all \( s \in S_1' \).

- If \( \text{len}(S_1' \xrightarrow{\Lambda_1'} \cdots) = 0 \), then \( S_1 \xrightarrow{\Lambda_1} \cdots = S_1' \xrightarrow{\langle L_1, \varepsilon \rangle} S_1' \xrightarrow{\langle L_2, \varepsilon \rangle} \cdots \), where each \( L_i \) summarizes each sequence \( \tilde{I}_{i,1},...,\tilde{I}_{i,\delta_i} \). Thus, \( S_1 = S_1' \) and

Eventually, there is a corresponding value from MaxVals in \( \hat{K}_1' \) with quantity many. That contradicts the condition that \( \tilde{\mu} \hat{H} = \text{single} \), however, so this case cannot happen.

- Otherwise, if some address \( \hat{a}_\varepsilon'' \) identifies the active for \( \tilde{I}_{1,1} \) and \( \hat{a}_\varepsilon'' \in D \), then let \( \tilde{a}_\varepsilon'' = \hat{K}_2 \xrightarrow{\tilde{I}_{2,3},...,\tilde{I}_{2,n}} \cdots \). Because \( \hat{a}_\varepsilon'' \in D \), the transition \( \hat{K}_1 \xrightarrow{\tilde{I}_{1,1},...,\tilde{I}_{1,\delta}} \hat{K}_2 \) does not receive the message \( \hat{a}_\varepsilon = \hat{a}_\varepsilon \). By the definition of EvictExec, \( \tilde{a}_\varepsilon'' = \text{EvictExec}(\tilde{a}_\varepsilon'', D, A) \). By reasoning similar to the previous case, it must be the case that \( \tilde{\mu} \hat{H} = \text{single} \). Then by the induction hypothesis, there exists \( j \leq i - 1 \) such that \( \tilde{K}_j \xrightarrow{\tilde{I}_{j+1}} \tilde{K}_j' \xrightarrow{\tilde{I}_{j+2}} \tilde{K}_j'' \) is a transition in \( \tilde{a}_\varepsilon' \) such that \( \tilde{I}_{j+1} = \hat{a}_\varepsilon'' : \text{rcv-int}(H, \varepsilon) \) for some \( \hat{a}_\varepsilon'' \).

- In the final case, there must be some address \( \hat{a}_\varepsilon'' \) that identifies the active for \( \tilde{I}_{1,1} \) and \( \hat{a}_\varepsilon'' \notin D \). If \( i = 1 \), then by the construction of \( \tilde{I}_{1,1} \) in this case in the definition of EvictExec, \( \tilde{I}_{1,1} = \hat{a}_\varepsilon'' : \text{rcv-int}(H, \varepsilon) \) for some \( \hat{a}_\varepsilon'' \). Otherwise, by reasoning similar to the previous two cases, it must be the case that \( \tilde{\mu} \hat{H} = \text{single} \). Let \( \tilde{a}_\varepsilon'' = \hat{K}_2 \xrightarrow{\tilde{I}_{2,3},...,\tilde{I}_{2,n}} \cdots \), and let \( \tilde{a}_\varepsilon'' = \hat{K}_2 \xrightarrow{\tilde{I}_{2,3},...,\tilde{I}_{2,n}} \cdots \). By the definition of EvictExec for this case, \( \tilde{a}_\varepsilon'' = \text{EvictExec}(\tilde{a}_\varepsilon'', D, A \cup A'') \) for some \( A'' \). Then by the induction hypothesis, there exists \( j \leq i - 1 \) such that \( \tilde{K}_j \xrightarrow{\tilde{I}_{j+1}} \tilde{K}_j' \xrightarrow{\tilde{I}_{j+2}} \tilde{K}_j'' \) is a transition in \( \tilde{a}_\varepsilon' \) such that \( \tilde{I}_{j+1} = \hat{a}_\varepsilon'' : \text{rcv-int}(H, \varepsilon) \).

\[ \square \]
Simulates($S_1 \xrightarrow{\Lambda_1} \ldots \xrightarrow{\bar{a} \bar{e} x}$).

- Otherwise, if some address $\bar{a}$ identifies the active actor for $\hat{l}_{1,1}$ and $\bar{a} \in D$, then let $\bar{a}ex'' = \hat{K}_2 \xrightarrow{\hat{l}_{2,1}, \ldots, \hat{l}_{2,n}} \ldots$. In this case, there exists some $L_1$ such that
  
  - $L_1$ summarizes $\hat{l}_{1,1}, \ldots, \hat{l}_{1,n}$,
  - $S_1 = S_1'$,
  - $S_1 \xrightarrow{(L_1, O_1)} S_2$, and
  - SimulateUnevicted($\bar{a}ex''$, $\bar{a}ex'$, $S_1' \xrightarrow{\Lambda_1} \ldots$, $D, A$) = $S_2 \xrightarrow{\Lambda_2} \ldots$.

By coinduction, we have that Simulates($S_2 \xrightarrow{\Lambda_2} \ldots, \bar{a}ex''$). Therefore, we have Simulates($S_1 \xrightarrow{\Lambda_1} \ldots, \bar{a}ex$), which completes the proof for this case.

- Otherwise, some address $\bar{a}$ identifies the active actor for $\hat{l}_{1,1}$ and $\bar{a} \notin D$.

  Let $\bar{a}ex'' = \hat{K}_2 \xrightarrow{\hat{l}_{2,1}, \ldots, \hat{l}_{2,n}} \ldots$ and $\bar{a}ex''' = \hat{K}_2' \xrightarrow{\hat{l}_{2,1}, \ldots, \hat{l}_{2,n}} \ldots$. Then by the definition of SimulateUnevicted in this case, there exist $L_1$, $O_1$, and $A'$ such that
  
  - $L_1$ summarizes $\hat{l}_{1,1}, \ldots, \hat{l}_{1,n}$,
  - $S_1 = S_1'$,
  - $S_1 \xrightarrow{(L_1, O_1)} S_2$,
  - SimulateUnevicted($\bar{a}ex''$, $\bar{a}ex'''$, $S_1' \xrightarrow{\Lambda_1} \ldots$, $D, A$) = $S_2 \xrightarrow{\Lambda_2} \ldots$.

By coinduction, we have that Simulates($S_2 \xrightarrow{\Lambda_2} \ldots, \bar{a}ex'''$). Therefore, we have Simulates($S_1 \xrightarrow{\Lambda_1} \ldots, \bar{a}ex$), which completes the proof.

\[\square\]

**L.46 Eviction Fulfillment Lemma**

**Lemma** (Eviction Fulfillment). For all $\bar{a}ex = \hat{K}_1 \xrightarrow{\hat{l}_{1,1}, \ldots, \hat{l}_{1,n}} \ldots$, $\bar{a}ex' = \hat{K}'_1 \xrightarrow{\hat{l}'_{1,1}, \ldots, \hat{l}'_{1,n}} \ldots$, $S_1', \ldots, S_1', \ldots, S_1$, \ldots, $\Lambda_1$, \ldots, $D$, $A$, $\eta$, and $O$, if

- SimulateUnevicted($\bar{a}ex$, $\bar{a}ex'$, $S_1' \xrightarrow{\Lambda_1} \ldots$, $D, A$) = $S_1 \xrightarrow{\Lambda_1} \ldots$,
- $\langle \eta, po \rangle \in \text{Obls}(S_1')$,
- there is a step $S_j' \xrightarrow{(L, O)} S_{j+1}'$ in $S_1' \xrightarrow{\Lambda'_{j+1}} \ldots$ such that $\langle \eta, po \rangle \in O$,
then there is a step \( S_i \xrightarrow{(L',O')} S_{i+1} \) in \( S_1 \xrightarrow{\Lambda_1} \ldots \) such that \( \langle \eta,po \rangle \in O' \).

**Proof.** The proof is by induction on \( i \). The proof proceeds by case analysis on the cases in the definition of SimulateUnevicted.

- If \( \text{len} \left( S_1 \xrightarrow{\Lambda_1} \ldots \right) = 0 \), then \( S_1 \xrightarrow{\Lambda_1} \ldots = S_i' \), so there is no transition in that execution. This contradicts the precondition to the lemma that there is some transition that fulfills the given obligation, so this case does not apply.

- Otherwise, if some address \( \hat{a} \) identifies the active actor for \( \hat{I}_{1,1} \) and \( \hat{a} \in D \), then we know there must be some transition in \( \bar{a}\bar{e}\bar{x} \) where an address not in \( D \) identifies the active actor for that transition. Let \( K_k \xrightarrow{\hat{I}_{k,1,\ldots,k,n}} K_{k+1} \) be the first such transition, and let \( \bar{a}\bar{e}\bar{x}'' \) be the suffix of \( \bar{a}\bar{e}\bar{x} \) starting from \( K_k \). By induction on \( k \) and by the definition of SimulateUnevicted, \( \text{SimulateUnevicted}(\bar{a}\bar{e}\bar{x}'',\bar{a}\bar{e}\bar{x}',S_1 \xrightarrow{\Lambda_1} \ldots,D,A) = S_k \xrightarrow{\Lambda_k} \ldots \). Then the next case applies.

- In the final case, there must be some address \( \hat{a} \) that identifies the active actor for \( \hat{I}_{1,1} \) and \( \hat{a} \notin D \). By the definition of SimulateUnevicted, there exist \( L_1, L_1', O_1 \) such that \( \Lambda_1 = \langle L_1,O_1 \rangle \) and \( \Lambda_1' = \langle L_1',O_1 \rangle \). If \( \langle \eta,po \rangle \in O_1 \), then let \( j = 1 \) to complete the proof. Otherwise, by the induction hypothesis, there is a step \( S_1 \xrightarrow{(L',O')} S_{i+1} \) in \( S_2 \xrightarrow{\Lambda_2} \ldots \) such that \( \langle \eta,po \rangle \in O' \).

\[ \square \]

### L.47 SimulateUnevicted Synchronization Lemma

**Lemma (SimulateUnevicted Synchronization).** For all \( \bar{a}\bar{e}\bar{x} = \bar{K}_1 \xrightarrow{\hat{I}_{1,1,\ldots,k,n}} \ldots \), \( \bar{a}\bar{e}\bar{x}' = \bar{K}_1' \xrightarrow{\hat{I}_{1,1,\ldots,k,n}} \ldots, S_1' \xrightarrow{\Lambda_1'} \ldots, S_1, \ldots, \Lambda_1', \ldots, D, A, \) and \( i \), if

- \( \text{SimulateUnevicted}(\bar{a}\bar{e}\bar{x}, \bar{a}\bar{e}\bar{x}', S_1' \xrightarrow{\Lambda_1'} \ldots,D,A) = S_1 \xrightarrow{\Lambda_1} \ldots \) and

- \( S_i \) is a configuration in \( S_1 \xrightarrow{\Lambda_1} \ldots \),

then there exist \( j \leq i \), \( \bar{a}\bar{e}\bar{x}'', \bar{a}\bar{e}\bar{x}''' \), and \( A' \) such that

\( \text{SimulateUnevicted}(\bar{a}\bar{e}\bar{x}'', \bar{a}\bar{e}\bar{x}''', S_j' \xrightarrow{\Lambda_j} \ldots,D,A') = S_i \xrightarrow{\Lambda_i} \ldots \)

**Proof.** The proof is by induction on \( i \), proceeding by case analysis on the cases in the definition of SimulateUnevicted.

- If \( \text{len} \left( S_1' \xrightarrow{\Lambda_1'} \ldots \right) = 0 \), then by the definition of SimulateUnevicted, \( S_i = S_i' \) for all \( S_i \) in \( S_1 \xrightarrow{\Lambda_1} \ldots \). Let \( j = 1 \), \( \bar{a}\bar{e}\bar{x}'' = \bar{a}\bar{e}\bar{x} \), \( \bar{a}\bar{e}\bar{x}''' = \bar{a}\bar{e}\bar{x}' \), and \( A' = \emptyset \) to complete the proof.
• Otherwise, if some address \( \hat{a} \) identifies the active actor for \( \hat{t}_{1,1} \) and \( \hat{a} \in D \), then let \( \hat{aex}'' = K_2 \xrightarrow{\hat{t}_{2,1}, \ldots, \hat{t}_{2,n}} \ldots \). By the definition of \( \text{SimulateUnevicted} \), \( S_2 = S_1 \), so \( S_1 \) is also a configuration in \( S_2 \xrightarrow{\Lambda_2} \ldots \). We also have in this case that \( \text{SimulateUnevicted}(aex'', aex', S_1' \xrightarrow{\Lambda_1} \ldots, D, A) = S_2 \xrightarrow{\Lambda_2} \ldots \). By the induction hypothesis, there exist some \( j \leq i - 1, aex''', aex''' \), and \( A' \) such that \( \text{SimulateUnevicted}(aex'', aex''', S_j' \xrightarrow{\Lambda_j} \ldots, D, A') = S_i \xrightarrow{\Lambda_i} \ldots \).

• Otherwise, some address \( \hat{a} \) identifies the active actor for \( \hat{t}_{1,1} \) and \( \hat{a} \notin D \). Let \( \hat{aex}'' = K_2 \xrightarrow{\hat{t}_{2,1}, \ldots, \hat{t}_{2,n}} \ldots \) and \( \hat{aex}''' = K_2' \xrightarrow{\hat{t}_{2,1}, \ldots, \hat{t}_{2,n}} \ldots \). Then by the definition of \( \text{SimulateUnevicted} \) in this case, there exists \( A'' \) such that \( \text{SimulateUnevicted}(aex'', aex''' \xrightarrow{\Lambda_2} \ldots, D, A \cup A'') = S_2 \xrightarrow{\Lambda_2} \ldots \). If \( i = 1 \), then let \( j = 1, aex'' = aex, aex''' = aex' \), and \( A' = \emptyset \) to complete the proof.

Otherwise, by the induction hypothesis, there exist some \( j \leq i - 1, aex'', aex''' \), and \( A' \) such that \( \text{SimulateUnevicted}(aex'', aex''', S_j' \xrightarrow{\Lambda_j} \ldots, D, A') = S_i \xrightarrow{\Lambda_i} \ldots \).

\[ \square \]

### L.48 Evict Conformance Reflection Theorem

**Theorem (Evict Conformance Reflection).** Evict is a conformance-reflecting transformation.

**Proof.** First, we define the necessary functions \( \text{TransExec}_{\text{Evict}} \) and \( \text{UntransExec}_{\text{Evict}} \), then we show the conformance-reflection properties for Evict, \( \text{TransExec}_{\text{Evict}} \), and \( \text{UntransExec}_{\text{Evict}} \).

**Definitions of TransExec\text{Evict} and UntransExec\text{Evict}**

First, we must define \( \text{TransExec}_{\text{Evict}} \) and \( \text{UntransExec}_{\text{Evict}} \). For all \( aex = \hat{K}_1 \xrightarrow{\hat{t}_{1,1}, \ldots, \hat{t}_{1,n}} \ldots \), and \( s \), there exists some \( \hat{K}' \) and \( A \) such that \( \text{Evict}(\hat{K}_1, s) = \{ \langle \hat{K}', s, A, id \rangle \} \). If \( \hat{K}' = \hat{K}_1 \) and \( A = id \), then define \( \text{TransExec}_{\text{Evict}}(aex, s) = \{ \langle aex, s, id, id \rangle \} \). Otherwise, by the definition of Evict, there exists \( D \) such that \( \text{Evict}(\hat{K}_1, s) = \{ \langle \hat{K}_1, s, A, id \rangle \} \) and \( \hat{K}' \) is an evic- tion of \( D \) from \( \hat{K}_1 \) corresponding via \( A|_{\text{ExtAbsAddr}(\text{dom}(\hat{\beta}) - D)} \). Then let \( aex' = \text{EvictExec}(aex, \hat{K}', D, A|_{\text{ExtAbsAddr}(\text{dom}(\hat{\beta}) - D)}) \) where \( \hat{\beta} \) is the actor-behavior-map component of \( \hat{K}_1 \), and define \( \text{TransExec}_{\text{Evict}}(aex, s) = \{ \langle aex', s, A, id \rangle \} \).

To define \( \text{UntransExec}_{\text{Evict}} \), for all \( aex = \hat{K}_1 \xrightarrow{\hat{t}_{1,1}, \ldots, \hat{t}_{1,n}} \ldots \), \( s \), and \( X \), we similarly know that there exists some \( \hat{K}' \) and \( A \) such that \( \text{Evict}(\hat{K}_1, s) = \{ \langle \hat{K}', s, A, id \rangle \} \). If \( \hat{K}' = \hat{K}_1 \), \( A = id \), and \( \langle aex, s, id, id \rangle \in \text{dom}(X) \),
then define \( \text{UntransExec}_{\text{Evict}}(\bar{a}ex, s, X) = X(\bar{a}ex, s, id, id) \). Otherwise, if there exist \( D \) and \( \bar{a}ex' \) such that \( \text{EvictExec}(\bar{a}ex, \bar{K}', D, A|_{\text{ExtAbsAddr} \cup \text{dom}(\beta) - D}) = \bar{a}ex' \), \( \text{Simulates}(X(\bar{a}ex', s, A, id), \bar{a}ex') \), \( \{s'\} \) is the first configuration in \( X(\bar{a}ex', s, A, id) \), and all addresses in \( D \) are evictable from \( (\bar{K}_1, s') \), then define \( \text{UntransExec}_{\text{Evict}}(\bar{a}ex, s, X) = \text{SimulateUnevicted}(\bar{a}ex, \bar{a}ex', X(\bar{a}ex', s, A, id), D, A|_{\text{ExtAbsAddr} \cup \text{dom}(\beta) - D}) \). Otherwise, \( \text{UntransExec}_{\text{Evict}}(\bar{a}ex, s, X) \) is undefined.

**Evict Properties**

Let there be \( \bar{K} \) and \( s \) such that \( \bar{K} \) and \( s \) are well-formed, \( \bar{K} \) is an externals-only configuration, \( \text{Mon}(s) \subseteq \text{Used}(\bar{K}) \), and no actor in \( \bar{K} \) is handling an event. By the definition of \( \text{Evict} \), there exist \( \bar{K}' \) and \( A \) such that \( \text{Evict}(\bar{K}, s) = (\bar{K}', s, A, id) \).

**Well-Formed Preservation** If \( \bar{K}' = \bar{K} \), then this holds by default. Otherwise, we can see by the definition of \( \text{Evict} \) in figure 8.5 that no new markers or internal addresses are introduced and the new message-map \( \hat{\mu}'' \) created in every iteration of the main loop is fully merged. Therefore, \( \bar{K}' \) is well-formed.

**Externals-Only Preservation** If \( \bar{K}' = \bar{K} \), then this holds by default. Otherwise, by the definition of \( \text{Evict} \) in figure 8.5 for all \( \bar{a}@H' \) appearing in either the \( \hat{\rho}' \) or \( \bar{\hat{\rho}}' \) component of \( \bar{K}' \), either \( \bar{a}@H' \) appears in either the \( \hat{\rho}' \) or \( \bar{\hat{\rho}}' \) component of \( \bar{K}' \) (in which case \( H' = \phi \) if \( a \) is internal and \( |H'| \leq 1 \) otherwise), or there exists \( \bar{a}'@H' \) in the \( \hat{\rho} \) component of \( \bar{K} \) such that \( \bar{a}' \) is internal, and therefore \( H' = \phi \). Similarly, we also have that for all \( \bar{\langle}a@H', \tau \rangle \) in the \( \bar{\hat{\rho}}' \) component of \( \bar{K}' \), either \( \bar{\langle}a@H', \tau \rangle \) is in the \( \hat{\rho} \) component of \( \bar{K} \) (in which case \( |H'| \leq 1 \) ), or there exists \( \bar{a}@H' \) in the \( \hat{\rho} \) component of \( \bar{K} \) such that \( \bar{a} \) is internal, and therefore \( H' = \phi \).

**All-Awaiting Preservation** By a straightforward examination of the definition of \( \text{Evict} \).

**Used/Monitored Preservation** \( \text{Evict} \) changes neither the used markers of \( \bar{K} \) nor the monitored markers of \( s \), so \( \text{Mon}(s') \subseteq \text{Used}(\bar{K}') \).

**No New Enabled Actors** Let \( \bar{a} \) identify a necessarily enabled actor in \( \bar{K}' \). By the definition of \( \text{Evict} \), there must be an actor at \( \bar{a} \) in \( \bar{K} \) whose only difference from the one in \( \bar{K}' \) is that the addresses for evicted actors have been renamed, so the actor at \( \bar{a} \) in \( \bar{K} \) is necessarily enabled. Finally, by the definition of \( \text{Evict} \), it must be the case that \( A(\bar{a}) = \bar{a} \).

**Atomic Address Reflection** The function \( A \) is like the identity function, except that there are some internal atomic addresses it maps to external collective addresses. Thus, this property is satisfied.
**No New Single Messages** Let \( \langle \hat{a}@H, \hat{v} \rangle \) be a message in \( \hat{K}' \) with quantity **single**. By the definition of \( \text{Evict} \), there must exist some \( \hat{v}' \) such that \( \langle \hat{a}@H, \hat{v}' \rangle \) is a message in \( \hat{K} \) with quantity **single** and \( \text{Remap}(\hat{v}', A, id) = \hat{v} \).

**Internal Address Reflection** The function \( A \) is like the identity function, except that there are some internal atomic addresses it maps to external collective addresses. Thus, this property is satisfied.

**Unique Approximating Messages** Let \( \hat{\mu} \) and \( \hat{\mu}' \) be the message-map components of \( \hat{K} \) and \( \hat{K}' \), respectively, and let there be some \( \langle \hat{a}@H, \hat{v} \rangle \in \text{dom}(\hat{\mu}) \) such that \( A(\hat{a}) \) is internal. Then by the definition of \( \hat{\mu}' \) in the definition of \( \text{Evict} \), there exist \( \hat{a}' \), \( H' \), and \( \hat{v}' \) such that \( \hat{\mu}(\hat{a}@H, \hat{v}) \subseteq \hat{\mu}'(\hat{a}'@H', \hat{v}') \) and \( \langle \hat{a}@H, \hat{v} \rangle \) is the only member of \( \text{dom}(\hat{\mu}') \) such that \( A(\hat{a}) = \hat{a}' \), \( M(H) = H' \), and \( \text{Remap}(\hat{v}, A, M) \subseteq id, id \hat{v}' \).

**Unique Approximated Messages** Let \( \hat{\mu} \) and \( \hat{\mu}' \) be the message-map components of \( \hat{K} \) and \( \hat{K}' \), respectively, and let there be some \( \hat{a}', \hat{\eta}' \), and \( \hat{v}' \) such that \( \hat{\mu}'(\hat{a}@H', \hat{v}') = \text{single} \). Then by the definition of \( \hat{\mu}' \) in the definition of \( \text{Evict} \), there exist \( \hat{a} \), \( H \), and \( \hat{v} \) such that \( \hat{\mu}(\hat{a}@H, \hat{v}) = \text{single} \) and \( \langle \hat{a}@H, \hat{v} \rangle \) is the only member of \( \text{dom}(\hat{\mu}) \) such that \( A(\hat{a}) = \hat{a}' \), \( M(H) = H' \), and \( \text{Remap}(\hat{v}, A, M) \subseteq id, id \hat{v}' \).

**\( \text{TransExec}_{\text{Evict}} \) Properties**

Let there be \( \hat{a}\hat{e} = \hat{K}_1 \xrightarrow{\hat{f}_{1,1}, \ldots, \hat{f}_{1,m}} \ldots \text{ and } s \text{ such that} \)

- \( \hat{K}_1 \) and \( s \) are well-formed,
- \( \hat{K}_1 \) is an externals-only configuration,
- \( \text{Mon}(s) \subseteq \text{Used}(\hat{K}_1) \), and
- no actor in \( \hat{K}_1 \) is handling an event.

By the definition of \( \text{TransExec}_{\text{Evict}} \), there exist \( \hat{a}\hat{e}' = \hat{K}'_1 \xrightarrow{\hat{f}'_{1,1}, \ldots, \hat{f}'_{1,m}} \ldots \) and \( A \) such that \( \text{TransExec}_{\text{Evict}}(\hat{a}\hat{e}, s) = \{ (\hat{a}\hat{e}', s, A, id) \} \). Let \( A' = A|_{\hat{\beta} = \text{AbsAddr}_d, \hat{\beta}(\hat{\beta} \cdot D)} \) where \( \hat{\beta} \) is the actor-behavior-map component of \( \hat{K}_1 \). As a general note, if \( \hat{a}\hat{e} = \hat{a}\hat{e}' \), then by the definition of \( \text{TransExec}_{\text{Evict}} \), there exists \( D \) such that \( \hat{K}'_1 \) is an eviction of \( D \) from \( \hat{K}_1 \) corresponding via \( A' \).

**Initial Pair Correctness** By the definition of \( \text{TransExec}_{\text{Evict}} \).

**Fairness Preservation 1** If \( \hat{a}\hat{e} = \hat{a}\hat{e}' \), then this property holds by default.

Otherwise, let \( \hat{K}'_1 \) be a configuration in \( \hat{a}\hat{e}' \). By the Eviction Synchronization lemma, there exists a configuration \( \hat{K}_j \) in \( \hat{a}\hat{e} \) and some \( A'' \) such that \( \hat{K}'_1 \) is an eviction of \( D \) from \( \hat{K}_j \) corresponding via \( A' \cup A'' \).

For that configuration, first let \( \hat{a}' \) identify a necessarily enabled actor in \( \hat{K}'_1 \). By the definition of an “eviction of”, there exists some \( \hat{a} \) such that \( (A' \cup A'')(\hat{a}) = \hat{a}' \).
and $\hat{a}$ identifies a necessarily enabled actor in $\hat{K}_j$. Because $\hat{a}\hat{e}\hat{x}$ is fair, either there exists some future configuration in that execution after $\hat{K}_j$ in which the actor at $\hat{a}$ is not necessarily enabled, or there exists some future transition in that execution after $\hat{K}_j$ in which $\hat{a}$ identifies the active actor. Then by the Eviction Running Actor and Eviction Disabled Actor lemmas, either there exists some future transition in that execution after $\hat{K}_j$ in which $\hat{a}'$ identifies the active actor, or there exists some future configuration in $\hat{a}\hat{e}\hat{x}'$ after $\hat{K}_j$ in which the actor at $\hat{a}'$ is not necessarily enabled.

Second, let $\langle \hat{a}'@H, \hat{v}' \rangle$ be a message in $\hat{K}_j$ with quantity \textbf{single}. If there exists some future configuration in $\hat{a}\hat{e}\hat{x}'$ execution after $\hat{K}_j$ such that either $\langle \hat{a}'@H, \hat{v}' \rangle$ is not a message in that configuration or it has quantity \textbf{many}, then we’re done. Otherwise, By the definition of an “eviction of”, $\text{Remap}$, and $\sqsubset_{\text{id}, \text{id}}$, there exists a unique message $\langle \hat{a}@H, \hat{v} \rangle$ in $\hat{K}_j$ with a quantity of \textbf{single} such that $A' \cup A'' (\hat{a}) = \hat{a}'$ and $\text{Remap}(\hat{v}, A' \cup A'') \sqsubset_{\text{id}, \text{id}} \hat{v}'$. Because $\hat{a}\hat{e}\hat{x}$ is fair, either there exists a future configuration in that execution after $\hat{K}_j$ such that either $\langle \hat{a}@H, \hat{v} \rangle$ is not a message in that configuration or it has quantity \textbf{many}, or there exists a future transition in that execution after $\hat{K}_j$ that receives the message. Then by the Eviction Message Receive lemma, there exists a future transition in $\hat{a}\hat{e}\hat{x}'$ after $\hat{K}_j$ that receives the message. Therefore, $\hat{a}\hat{e}\hat{x}'$ is fair.

**Fairness Preservation 2** If $\hat{a}\hat{e}\hat{x} = \hat{a}\hat{e}\hat{x}'$, then this property holds by default.

Otherwise, let $\hat{a}$ identify an actor necessarily enabled in $\hat{K}_1$ such that either the actor at $\hat{a}$ is not necessarily enabled in $\hat{K}_1$, or $\hat{K}_1$ is a transition in $\hat{a}\hat{e}\hat{x}$ and $\hat{a}$ identifies the active actor for $\hat{l}_{1,1}$. If $\hat{a} \in D$, then there is no actor $\hat{a}$ in $\hat{K}_1$, or $\hat{a}$ in $\hat{K}_1$ corresponding via $A'$, and therefore $A(\hat{a})$ identifies a necessarily enabled actor in $\hat{K}_1$. If the actor at $\hat{a}$ is not necessarily enabled in $\hat{K}_1$, then by the Eviction Disabled Actor lemma, there exists $j \leq i$ such that $\hat{K}_j$ is a configuration in $\hat{a}\hat{e}\hat{x}'$ and the actor at $A'(\hat{a})$ is not necessarily enabled in $\hat{K}_j$. In the latter case, by the Eviction Running Actor lemma, there exists $j \leq i$ such that $\hat{K}_j$ is a transition in $\hat{a}\hat{e}\hat{x}'$ in which $A'(\hat{a})$ identifies the active actor for $\hat{l}_{j,1}$. By the definition of $A'$, we know that $A'(\hat{a}) = A(\hat{a})$, so we’re done.

**Fairness Preservation 3** If $\hat{a}\hat{e}\hat{x} = \hat{a}\hat{e}\hat{x}'$, then this property holds by default.

Otherwise, let there be some message $\langle \hat{a}@H, \hat{v} \rangle$ in $\hat{K}_1$ with quantity \textbf{single} and some $i$ such that either there exists a configuration $\hat{K}_1$ in $\hat{a}\hat{e}\hat{x}$ such that either $\langle \hat{a}@H, \hat{v} \rangle$ is not a message in that configuration or it has quantity \textbf{many}, or there exists a transition $\hat{K}_1$ is a transition in $\hat{a}\hat{e}\hat{x}$ that receives the message.

Let $\hat{a}' = A(\hat{a})$, $H' = H$, and $\hat{v}' = A(M(\hat{v}))$. If $\langle \hat{a}'@H', \hat{v}' \rangle$ is not a message in $\hat{K}_1$ or it has quantity \textbf{many} in $\hat{K}_1$, then we’re done. Otherwise, by the definition of an “eviction of”, $\text{Remap}$, and $\sqsubset_{\text{id}, \text{id}}$, $\langle \hat{a}@H, \hat{v} \rangle$ is the only message in $\hat{K}_1$, with a quantity of \textbf{single} such that $\text{Remap}(\hat{a}@H, A' \cup A'') \sqsubset_{\text{id}, \text{id}} \hat{a}@H'$ and $\text{Remap}(\hat{v}, A' \cup A'') \sqsubset_{\text{id}, \text{id}} \hat{v}'$. Because $\hat{a}@H'$ is fair, either there exists a future configuration in that execution after $\hat{K}_j$ such that either $\langle \hat{a}@H, \hat{v} \rangle$ is not a message in that configuration or it has quantity \textbf{many}, or there exists a future transition in that execution after $\hat{K}_j$ that receives the message. Then by the Eviction Message Receive lemma, there exists a future transition in $\hat{a}\hat{e}\hat{x}'$ after $\hat{K}_j$ that receives the message. Therefore, $\hat{a}\hat{e}\hat{x}'$ is fair.
Let there be 

UntransExecSimulateUnevictedUntransExecSimulateUnevicted = . Otherwise, let ... 

By the definition of TransExec, holds by the argument made at the end of section L.35.

If Prefix Consistency by the Simulation lemma, then UntransExecEvicted(aex, s, X) = X(aex, s, id, id), so the property holds by default. Otherwise, the property holds by the SimulateUnevicted Simulation lemma.
Fulfillment Reflection 1 If $\overline{aex} = \overline{aex}'$, then $\text{UntransExec}_{\text{Detect}}(\overline{aex}, s, X) = X(\overline{aex}, s, id, id)$, so the property holds by default.

Otherwise, let there be some $\langle \eta, po \rangle \in \text{Obls}(s)$. Because $S'_i = \{s\}$, we have $\langle \eta, po \rangle \in \text{Obls}(S'_i)$. If there is a step $S'_j \overset{(L,O)}{\rightarrow} S'_{j+1}$ in $S'_1 \overset{Λ_1}{\rightarrow} \ldots$ such that $\langle \eta, po \rangle \in O$, then by the Eviction Fulfillment lemma, there is a step $S_i \overset{(L',O')}{\rightarrow} S_{i+1}$ in $S_1 \overset{Λ_1}{\rightarrow} \ldots$ such that $\langle \eta, po \rangle \in O'$.

Fulfillment Reflection 2 If $\overline{aex} = \overline{aex}'$, then $\text{UntransExec}_{\text{Detect}}(\overline{aex}, s, X) = X(\overline{aex}, s, id, id)$, so the property holds by default.

Otherwise, let there be some $i$ such that for all $j \leq i$ and all $\langle \eta, po \rangle \in \text{Obls}(S'_i)$, the execution $S'_1 \overset{Λ_1}{\rightarrow} \ldots$ fulfills the obligation $\langle \eta, po \rangle$. Now let there be some $j \leq i$ and some $\langle \eta, po \rangle \in \text{Obls}(S_j)$. We must show that the execution $S_1 \overset{Λ_1}{\rightarrow} \ldots$ fulfills the obligation $\langle \eta, po \rangle$.

By the SimulateUnevicted Synchronization, there exists some $k \leq j$, $\overline{aex}''$, $\overline{aex}'''$, $D'$, and $A''$ such that $\text{SimulateUnevicted}(\overline{aex}'', \overline{aex}''', S'_j \overset{Λ'_j}{\rightarrow} \ldots, D', A'') = S_j \overset{Λ_j}{\rightarrow} \ldots$. By the definition of $\text{SimulateUnevicted}$, $S_j = S'_k$, and so $\langle \eta, po \rangle \in \text{Obls}(S'_k)$. Therefore because $k \leq j \leq i$, the execution $S'_k \overset{Λ'_k}{\rightarrow} \ldots$ eventually fulfills the obligation $\langle \eta, po \rangle$, so by the Eviction Fulfillment lemma, the execution $S_j \overset{Λ_j}{\rightarrow} \ldots$ eventually fulfills that obligation, too. Then since that obligation is a suffix of our original one, we know that $S_1 \overset{Λ_1}{\rightarrow} \ldots$ eventually fulfills $\langle \eta, po \rangle$. □

L.49 Detect Conformance Reflection Theorem

Theorem (Detect Conformance Reflection). Detect is a conformance-reflecting transformation.

Proof. First, we define the necessary functions $\text{TransExec}_{\text{Detect}}$ and $\text{UntransExec}_{\text{Detect}}$, then we show the conformance-reflection properties for Detect, $\text{TransExec}_{\text{Detect}}$, and $\text{UntransExec}_{\text{Detect}}$.

Definitions of $\text{TransExec}_{\text{Detect}}$ and $\text{UntransExec}_{\text{Detect}}$

As a general note, by the definition of Detect, we know that for all $\hat{K}$ and $s$, either $\text{Detect}(\hat{K}, s) = \emptyset$, or $\text{Detect}(\hat{K}, s) = \{(\hat{K}, s, id, id)\}$.

For all $\overline{aex} = \hat{K}_1 \overset{f_{1,1}, \ldots, f_{1,s}}{\rightarrow} \ldots$ and all $s$, Define $\text{TransExec}_{\text{Detect}}$ as follows:

$$\text{TransExec}_{\text{Detect}}(\overline{aex}, s) = \begin{cases} \emptyset & \text{if } \text{Detect}(\hat{K}_1, s) = \emptyset \\ \{(\overline{aex}, s, id, id)\} & \text{otherwise} \end{cases}$$
Next, define $\text{UntransExec}_{\text{Detect}}$ as follows for all $a\overline{a}=K_1 \rightarrow i_1,\ldots, i_m$ and all $s$,

- If $\text{Detect}(K_1,s)=\emptyset$, then for all $i$ such that $K_i \rightarrow i_1,\ldots, i_n \rightarrow K_{i+1}$ is a transition in $a\overline{a}$, let $a'_1 : \text{send-ext}(a_1 \oplus H_1, \overline{v}_1), \ldots, a'_m : \text{send-ext}(a_m \oplus H_m, \overline{v}_m)$ be the send-ext labels in $i_1,\ldots, i_n$, and let $\hat{\mu}_i = \emptyset \oplus (a_1 \oplus H_1, \overline{v}_1) \oplus \cdots \oplus (a_m \oplus H_m, \overline{v}_m)$. Then define $L_i = (\hat{i}_1,\hat{\mu}_i)$. Then $L_i$ summarizes $i_1,\ldots, i_n$ by construction, and $K_i \xrightarrow{L_i} K_{i+1}$. Finally, define $\text{UntransExec}_{\text{Detect}}(a\overline{a},s,X) = \{s\xrightarrow{(L_1,\emptyset)} s \xrightarrow{(L_2,\emptyset)} \cdots \}$ if $\{s\xrightarrow{(L_i,\emptyset)} s\}$ is a valid transition for all of the $L_i$ (we will show below that it is always a valid transition for the inputs to $\text{UntransExec}_{\text{Detect}}$ that we care about). If any of those transitions is not possible, then $\text{UntransExec}_{\text{Detect}}(a\overline{a},s,X)$ is undefined.

- Otherwise, $\text{Detect}(K_1,s) = \{[K_1,s,id,id]\}$. In that case, define $\text{UntransExec}_{\text{Detect}}(a\overline{a},s,X) = X(a\overline{a},s,id,id)$. If $X(a\overline{a},s,id,id)$ is undefined, then so is $\text{UntransExec}_{\text{Detect}}(a\overline{a},s,X)$.

**Detect Properties**

Let there be $\hat{K}$ and $s$ such that $\hat{K}$ and $s$ are well-formed, $\hat{K}$ is an externals-only configuration, $\text{Mon}(s) \subseteq \text{Used}(\hat{K})$, and no actor in $\hat{K}$ is handling an event. By the definition of Detect, either $\text{Detect}(K,s) = \emptyset$ or $\text{Detect}(\hat{K},s) = \{[\hat{K},s,id,id]\}$. Then the Well-Formed Preservation, Externals-Only Preservation, All-Awaiting Preservation, Used/Monitored Preservation, Atomic Address Reflection, Unique Actor Correspondences, No New Single Messages, and Internal Address Reflection conditions all trivially hold. For No New Enabled Actors, let $\hat{a}$ identify a necessarily enabled actor in $\hat{K}$. Then $id(\hat{a})$ identifies a necessarily enabled actor in $\hat{K}$. For Unique Approximating Messages and Unique Approximated Messages, the message-map component $\hat{\mu}$ is the same in both the pre- and post-transformed configurations. Because $\hat{K}$ is well-formed, $\hat{\mu}$ is fully merged, so by the Message-Map Reflexivity lemma (appendix I, $\hat{\mu} \equiv id, id \hat{\mu}$). Then the rules for $\equiv$ provide the necessary properties.

**TransExec$_{\text{Detect}}$ Properties**

Let there be $a\overline{a}=K_1 \rightarrow i_1,\ldots, i_m$ and $s$ such that

- $\hat{K}_1$ and $s$ are well-formed,
- $\hat{K}_1$ is an externals-only configuration,
- $\text{Mon}(s) \subseteq \text{Used}(\hat{K}_1)$, and
- no actor in $\hat{K}_1$ is handling an event.
If \( \text{Detect}(\hat{K}_1, s) = \{(\hat{K}_1, s, \text{id}, \text{id})\} \), then \( \text{TransExec}_{\text{Detect}}(\hat{aex}, s) = \{(\hat{aex}, s, \text{id}, \text{id})\} \), so Initial Pair Correctness holds in that case, as well. Then Fairness Preservation 1, Detection Properties

Let there be \( \hat{aex} = \hat{K}_1 \xrightarrow{\hat{I}_1, \ldots, \hat{I}_m} s, \) and \( X \) such that

- \( \hat{K}_1 \) and \( s \) are well-formed,
- \( \hat{K}_1 \) is an externals-only configuration,
- \( \text{Mon}(s) \subseteq \text{Used}(\hat{K}_1) \),
- no actor in \( \hat{K}_1 \) is handling an event, and
- \( X \) contains a simulation for all members of

\[ \text{TransExec}_{\text{T}}(\hat{K}_1 \xrightarrow{\hat{I}_1, \ldots, \hat{I}_m} s) \]

If \( \text{Detect}(\hat{K}_1, s) = \{(\hat{K}_1, s, \text{id}, \text{id})\} \), then \( \text{TransExec}_{\text{Detect}}(\hat{aex}, s) = \{(\hat{aex}, s, \text{id}, \text{id})\} \). Because \( X \) contains a simulation for all members of

\[ \text{TransExec}_{\text{T}}(\hat{K}_1 \xrightarrow{\hat{I}_1, \ldots, \hat{I}_m} s) \]

there exists an execution \( S_1 \xrightarrow{\Lambda_1} \ldots \) such that \( S_1 = \{s\} \) and \( \text{Simulates}(S_1 \xrightarrow{\Lambda_1} \ldots, \hat{aex}) \).

Otherwise, \( \text{Detect}(\hat{K}_1, s) = \emptyset \). Let \( L_1, \ldots \) be the sequence of summary transition labels described in the definition of \( \text{UntransExec}_{\text{Detect}} \) above. We know that \( \text{Mon}(s) \subseteq \text{Used}(\hat{K}_1) \), and by a straightforward induction on \( i \) and

We will show that for all \( i \) such that \( L_i \) is a label in that sequence,

- \( \hat{K}_i \) is an externals-only configuration,
- \( \text{Mon}(s) \subseteq \text{Used}(\hat{K}_i) \), and
- \( \{s\} \xrightarrow{\langle L_i, \emptyset \rangle} \{s\} \).

The proof is by induction on \( i \). If \( i = 1 \), then we know that \( \hat{K}_1 \) is an externals-only configuration and \( \text{Mon}(s) \subseteq \text{Used}(\hat{K}_1) \) by the definition of \( \hat{K}_1 \) above. Otherwise, by the induction hypothesis we know that \( \hat{K}_{i-1} \) is an externals-only configuration and \( \{s\} \xrightarrow{\langle L_{i-1}, \emptyset \rangle} \{s\} \), so by corollary I.42.2 to the Abstract Externals-Only Preservation lemma, \( \hat{K}_i \) is an externals-only configuration, and by corollary K.10.1 to the Used/Monitored Marker lemma, \( \text{Mon}(s) \subseteq \text{Used}(\hat{K}_i) \).

Let \( \hat{I}_1, \ldots, \hat{I}_p \) be a sequence summarized by \( L_i \), and let \( \langle \hat{\beta}, \hat{\mu}, H \rangle^{\hat{\beta}} = \hat{K}_i \). Every marker \( H \) appearing in a \texttt{rcv-ext} or \texttt{send-ext} label in that sequence must appear on a similar label in the transition \( \hat{K}_i \xrightarrow{\hat{I}_1, \ldots, \hat{I}_m} \hat{K}_{i+1} \). Then by the definition of the program transition relation, \( \eta \) must either appear in \( \hat{\beta} \), \( \hat{\mu} \), or \( \hat{\rho} \), or it is not a member of \( H \) (i.e., it is a fresh marker created during that transition sequence). By the definition of \( \text{Detect} \) in this case, no marker in \( \text{Mon}(s) \)
appears in $\hat{\beta}$, $\hat{\mu}$, or $\hat{\rho}$, and we know that $Mon(s) \subseteq Used(\hat{K}_i)$, so $\eta \notin Mon(s)$. Thus for all $j \in 1 \ldots p$, if $\hat{[l]}_j = \hat{a}@H?\hat{v}$ or $\hat{[l]}_j = \hat{a}@H!\hat{v}$, then we have $s \xrightarrow{\hat{[l]}_j, s, \phi} s$ by either the P-UNMONITOREDRECEIVE rule or the P-SEND rule. Therefore, $\{s\} \xrightarrow{\hat{[l]}_j, s, \phi} \{s\}$ by the S-SENDORRECEIVE rule, and therefore $\{s\} \xrightarrow{\hat{[l]}_j, s} \{s\}$. Otherwise, $\hat{[l]}_j = \bullet$, and therefore $\{s\} \xrightarrow{\bullet} \{s\}$ by the definition of the $\rightarrow$ relation. Thus, $\{s\} \xrightarrow{\hat{[l]}_j, s, \phi} \{s\}$. Also by the definition of $Detect$ in this case, $InMon(s) = \phi$ and there is no $s'$ such that $s \xrightarrow{\bullet, s, \phi} s'$ (because $s$ has no transitions listed in any of its state definitions). Therefore, $\{s\} \xrightarrow{\bullet, s} \{s\}$. Therefore, $\{s\} \xrightarrow{\bullet, s} \{s\}$ is a valid transition, and so $UntransExec_{Detect}(\alpha x, s, X) = \{s\} \xrightarrow{\bullet, s} \{s\} \xrightarrow{\bullet, s} \{s\}$.

We show the various properties below.

**Execution Simulation** If $Detect(\hat{K}_1, s) = \{(\hat{K}_1, s, id, id)\}$, then we have already shown that $S_1 = \{s\}$ and $Simulates(S_1 \xrightarrow{(L_1)} \ldots , \alpha x)$. Otherwise, the above argument shows that $Simulates([s] \xrightarrow{(L_1)} \ldots , \alpha x)$.

**Prefix Consistency** Let there be some $\alpha x' = \hat{K}_1' \xrightarrow{\hat{L}_1, \ldots , \hat{L}_m} \ldots , \alpha x', \Sigma_1' \xrightarrow{(L'_1, s)} \ldots$, and $i$ such that

1. $\alpha x$ and $\alpha x'$ share a prefix of length $i$,

2. $UntransExec_{\alpha x', s, X'} = \Sigma_1' \xrightarrow{(L'_1, s)} \ldots$, and

3. for all $\alpha x''$, $\alpha x'''$, $s'$, $A$, $M$, and $j$ such that

   - $\langle \alpha x'', s', A, M \rangle \in TransExec_{\alpha x, s}$,

   - $\langle \alpha x''', s', A, M \rangle \in TransExec_{\alpha x', s}$,

   - $\alpha x''$ and $\alpha x'''$ share a prefix of length $j$, and

   - $j \leq i$,

   $X(\alpha x'', s', A, M)$ and $X'(\alpha x''', s', A, M)$ share a prefix of length $j$.

If $Detect(\hat{K}_1, s) = \{(\hat{K}_1, s, id, id)\}$, then $UntransExec_{Detect}(\alpha x, s, X) = X(\alpha x, s, id, id)$ and $UntransExec_{Detect}(\alpha x', s, X') = X'(\alpha x', s, id, id)$. Then we know that $X(\alpha x, s, id, id)$ and $X'(\alpha x', s, id, id)$ share a prefix of length $i$.

Otherwise, both $UntransExec_{Detect}(\alpha x, s, X)$ and $UntransExec_{Detect}(\alpha x', s, X')$ share a prefix of length $i$, because every configuration in both executions is $\{s\}$, and every label $L_i$ is determined entirely from the sequence $\hat{l}_i, \ldots , \hat{l}_{i,m}$.
Fulfillment Reflection 1 If $\text{Detect}(\tilde{K}_1, s) = \{(\tilde{K}_1, s, id, id)\}$, let $\langle \eta, \rho \rangle$ be a member of $\text{Obls}(s)$. In this case, we know that $\text{UntransExec}_{\text{Detect}}(\tilde{a} \tilde{e} x, s, X) = X(\tilde{a} \tilde{e} x, s, id, id)$, so if there is a step in $X(\tilde{a} \tilde{e} x, s, id, id)$ that fulfills that obligation, then there is a step in $\text{UntransExec}_{\text{Detect}}(\tilde{a} \tilde{e} x, s, X)$ that fulfills the obligation, as well.

Otherwise, by the definition of $\text{Detect}$, $\text{Obls}(s) = \emptyset$, so the condition is vacuously true.

Fulfillment Reflection 2 Similarly to the previous property, if $\text{Detect}(\tilde{K}_1, s) = \{(\tilde{K}_1, s, id, id)\}$, then $\text{UntransExec}_{\text{Detect}}(\tilde{a} \tilde{e} x, s, X) = X(\tilde{a} \tilde{e} x, s, id, id)$. Therefore, if all obligations in the first $i$ configurations of $X(\tilde{a} \tilde{e} x, s, id, id)$ are fulfilled, then all obligations in the first $i$ configurations of $\text{UntransExec}_{\text{Detect}}(\tilde{a} \tilde{e} x, s, X)$ are fulfilled, as well.

Otherwise, every configuration in $\text{UntransExec}_{\text{Detect}}(\tilde{a} \tilde{e} x, s, X)$ is $(\text{psm})$, and by the definition of $\text{Detect}$ in this case, $\text{Obls}(s) = \emptyset$. Therefore, it is trivially the case that all obligations in the first $i$ configurations of $\text{UntransExec}_{\text{Detect}}(\tilde{a} \tilde{e} x, s, X)$ are fulfilled. 

L.50 Remap Approximation Composition Lemma

Lemma (Remap Approximation Composition). For all $v, v', v'', A, A', M, M'$, if $\text{Remap}(v, A, M) \sqsubseteq_{id, id} v'$ and $\text{Remap}(v', A', M') \sqsubseteq_{id, id} v''$, then $\text{Remap}(v, A' \circ A, M \circ M') \sqsubseteq_{id, id} v''$.

Proof. By structural induction on $v''$. We give the case for lists below. The case for dictionaries is similar, and the other cases are straightforward.

Case: $v = (\text{list } \{\tilde{v}_1, \ldots, \tilde{v}_n\})$

By the definitions of $\text{Remap}$ and $\sqsubseteq$, there exist $v'_1, \ldots, v'_m$ and $v''_1, \ldots, v''_p$ such that $v' = (\text{list } \{v'_1, \ldots, v'_m\})$ and $v'' = (\text{list } \{v''_1, \ldots, v''_p\})$.

To show that $\text{Remap}(v, A' \circ A, M \circ M') \sqsubseteq_{id, id} v''$, let $v'''$ be a member of the list $\text{Remap}(v, A' \circ A, M \circ M')$. There must exist some $i \in 1 \ldots n$ such that $v''' = \text{Remap}(v_i, A' \circ A, M \circ M')$. We must show there exists $k \in 1 \ldots p$ such that $\text{Remap}(v_i, A' \circ A, M \circ M') \sqsubseteq_{id, id} v''_k$.

Because $\text{Remap}(v, A, M) \sqsubseteq_{id, id} v'$, there exists $j \in 1 \ldots m$ such that $\text{Remap}(v_j, A, M) \sqsubseteq_{id, id} v'_i$. By the definition of $\text{Remap}$, $\text{Remap}(v'_j, A', M')$ is a member of the list $\text{Remap}(v', A', M')$. Because $\text{Remap}(v', A', M') \sqsubseteq_{id, id} v''$, there exists $k \in 1 \ldots p$ such that $\text{Remap}(v''_j, A', M') \sqsubseteq_{id, id} v''_k$. Then by the induction hypothesis, $\text{Remap}(v_i, A' \circ A, M \circ M') \sqsubseteq_{id, id} v''_k$. 

$\square$
L.51 Composition Conformance Reflection Theorem

**Theorem** (Composition Conformance Reflection). For all $T_1$ and $T_2$, if $T_1$ and $T_2$ are conformance-reflecting, then so is $T_1 \odot T_2$.

**Proof.** First, define $\text{TransExec}_{T_1 \odot T_2}$ and $\text{UntransExec}_{T_1 \odot T_2}$ as follows, where $\text{TransExec}_{T_1}$, $\text{TransExec}_{T_2}$, $\text{UntransExec}_{T_1}$, and $\text{UntransExec}_{T_2}$ are the functions given by the conformance-reflection proofs for those transformations.

$$\text{TransExec}_{1 \odot 2}(\bar{a} \bar{e}, s) =$$

$$\bigcup_{(\bar{a} \bar{e}', s', A_2, M_2) \in \text{TransExec}_2(\bar{a} \bar{e}, s)} \bigcup_{(\bar{a} \bar{e}'', s'', A_1, M_1) \in \text{TransExec}_1(\bar{a} \bar{e}', s')} \langle \bar{a} \bar{e}'', s'', A_1 \circ A_2, M_1 \circ M_2 \rangle$$

$$\text{UntransExec}_{1 \odot 2}(\bar{a} \bar{e}, s, X) = \text{UntransExec}_2(\bar{a} \bar{e}, s, X_2)$$

where $X_2 = \bigcup_{(\bar{a} \bar{e}_2, s_2, A_2, M_2) \in \text{TransExec}_2(\bar{a} \bar{e}, s)} [(\bar{a} \bar{e}_2, s_2, A_2, M_2) \mapsto \text{UntransExec}_1(\bar{a} \bar{e}_2, s_2, X_1)]$

and $X_1 = \bigcup_{(\bar{a} \bar{e}_1, s_1, A_1, M_1) \in \text{TransExec}_1(\bar{a} \bar{e}, s)} [(\bar{a} \bar{e}_1, s_1, A_1, M_1) \mapsto X(\bar{a} \bar{e}_1, s_1, A_1 \circ A_2, M_1 \circ M_2)]$

We will show the necessary properties for $T_1 \odot T_2$, $\text{TransExec}_{T_1 \odot T_2}$, and $\text{UntransExec}_{T_1 \odot T_2}$ below.

**Properties for $T_1 \odot T_2$**

Let there be $\bar{K}$ and $s$ such that $\bar{K}$ is an externals-only configuration, $\text{Mon}(s) \subseteq \text{Used}(\bar{K})$, and no actor in $\bar{K}$ is handling an event.

**Well-Formed Preservation** Because $T_2$ is conformance-reflecting, for every $\langle \bar{K}', s', A, M \rangle \in T_2(\bar{K}, s)$, $\bar{K}'$ and $s'$ are well-formed. Then for every $\langle \bar{K}'', s'', A, M \rangle \in T_1(\bar{K}', s')$, $\bar{K}''$ and $s''$ are well-formed. Therefore, for every $\langle \bar{K}'', s'', A, M \rangle \in T_1 \odot T_2(\bar{K}', s')$, $\bar{K}''$ and $s''$ are well-formed.

**Externals-Only Preservation** Similarly to the argument for Well-Formed Preservation, the composition of two conformance-reflecting transformations preserves the externals-only property.

**All-Awaiting Preservation** Similarly to the argument for Well-Formed Preservation, the composition of two conformance-reflecting transformations preserves the property that no actor in the program configuration is handling an event.

**Used/monitored Preservation** Similarly to the argument for Well-Formed Preservation, the composition of two conformance-reflecting transformations preserves the property that $\text{Mon}(s'') \subseteq \text{Used}(\bar{K}'')$ for all $\langle \bar{K}'', s'', A, M \rangle \in T_1 \odot T_2(\bar{K}', s')$. 
L.51. COMPOSITION CONFORMANCE REFLECTION THEOREM 571

No New Enabled Actors  Let \( \langle \tilde{K}'', s'', A, M \rangle \) be a member of \( T_1 \odot T_2(\tilde{K}, s) \), and let \( \tilde{a}'' \) identify a necessarily enabled actor in \( \tilde{K}'' \). By the definition of the composition, there exist \( \tilde{K}', s', A', M', \) and \( M'' \) such that \( \langle \tilde{K}', s', A', M' \rangle \in T_2(\tilde{K}, s), \langle \tilde{K}'', s'', A'', M'' \rangle \in T_1(\tilde{K}', s') \), \( A = A'' \circ A' \), and \( M = M'' \circ M' \). Because \( T_1 \) is conformance-reflecting, there exists some \( \tilde{a}' \) such that \( A''(\tilde{a}) = \tilde{a}' \) and \( \tilde{a}' \) identifies a necessarily enabled actor in \( \tilde{K}' \). Then because \( T_2 \) is conformance-reflecting, there exists \( \tilde{a} \) such that \( A'(\tilde{a}) = \tilde{a}' \) and \( \tilde{a} \) identifies a necessarily enabled actor in \( \tilde{K} \).

Atomic Address Reflection  Let \( \langle \tilde{K}'', s'', A, M \rangle \) be a member of \( T_1 \odot T_2(\tilde{K}, s) \), and let there be \( \tilde{a}'' \) and \( \tilde{a}'' \) such that \( A(\tilde{a}) = \tilde{a}'' \) and \( \tilde{a}'' \) is atomic. By the definition of the composition, there exist \( \tilde{K}', s', A', A'', M', \) and \( M'' \) such that \( \langle \tilde{K}', s', A', M' \rangle \in T_2(\tilde{K}, s), \langle \tilde{K}'', s'', A'', M'' \rangle \in T_1(\tilde{K}', s') \), and \( A = A'' \circ A' \). Therefore, there exists some \( \tilde{a}' \) such that \( A'(\tilde{a}) = \tilde{a}' \) and \( A''(\tilde{a}) = \tilde{a}'' \). By the Atomic Address Reflection property for \( T_1, \tilde{a}' \) is atomic. Then by the Atomic Address Reflection property for \( T_2, \tilde{a} \) is atomic.

Unique Actor Correspondences  Let \( \langle \tilde{K}'', s'', A, M \rangle \) be a member of \( T_1 \odot T_2(\tilde{K}, s) \), and let there be \( \tilde{a}, \tilde{a}' \), and \( \tilde{a}'' \) such that \( A(\tilde{a}) = \tilde{a}' \), \( A(\tilde{a}) = \tilde{a}'' \), and \( \tilde{a}'' \) is atomic. By the definition of the composition, there exist \( \tilde{K}', s', A', A'', M', \) and \( M'' \) such that \( \langle \tilde{K}', s', A', M' \rangle \in T_2(\tilde{K}, s), \langle \tilde{K}'', s'', A'', M'' \rangle \in T_1(\tilde{K}', s') \), \( A = A'' \circ A' \), and \( M = M'' \circ M' \). Then there exist some \( \tilde{a}'' \) and \( \tilde{a}''' \) such that

\[
\begin{align*}
& A'(\tilde{a}) = \tilde{a}' \\
& A'(\tilde{a}') = \tilde{a}'' \\
& A''(\tilde{a}''') = \tilde{a}''' \\
& A''(\tilde{a}'''') = \tilde{a}'''.
\end{align*}
\]

By Unique Actor Correspondences for \( T_1, \tilde{a}''' = \tilde{a}''' \). By Atomic Address Reflection for \( T_1, \tilde{a}'' = \tilde{a}'' \). Then by Unique Actor Correspondences for \( T_2, \tilde{a} = \tilde{a}'' \).

No New Single Messages  Let \( \langle \tilde{K}'', s'', A, M \rangle \) be a member of \( T_1 \odot T_2(\tilde{K}, s) \) let \( \tilde{\mu} \) and \( \tilde{\mu}' \) be the message-map components of \( \tilde{K} \) and \( \tilde{K}'' \) respectively, and let there be \( \tilde{a}'', H'', \) and \( \tilde{v}'' \) such that \( \tilde{\mu}''(\tilde{a}'' \circ H'', \tilde{v}'') = \text{single} \). By the definition of the composition, there exist \( \tilde{K}', s', A', A'', M', \) and \( M'' \) such that \( \langle \tilde{K}', s', A', M' \rangle \in T_2(\tilde{K}, s), \langle \tilde{K}'', s'', A'', M'' \rangle \in T_1(\tilde{K}', s') \), \( A = A'' \circ A' \), and \( M = M'' \circ M' \). Let \( \tilde{\mu}' \) be the message-map component of \( \tilde{K}' \). By the No New Single Messages property for \( T_1 \), there exist \( \tilde{a}', H', \) and \( \tilde{v}' \) such that \( A''(\tilde{a}) = \tilde{a}', M''(\tilde{a}) = \tilde{a}', M''(\tilde{H}) = H'', \text{Remap}(\tilde{v}, A'', M'') = \tilde{v}' \), and \( \tilde{\mu}'(\tilde{a}' \circ H', \tilde{v}') = \text{single} \). By the No New Single Messages property for \( T_2 \), there exist \( \tilde{a}, H, \) and \( \tilde{v} \) such that \( A(\tilde{a}) = \tilde{a}', M'(\tilde{H}) = H', \text{Remap}(\tilde{v}, A', M') = \tilde{v}' \), and \( \tilde{\mu}(\tilde{a} \circ H, \tilde{v}) = \text{single} \). Then we have \( A(\tilde{a}) = \tilde{a}'' \), \( M(\tilde{H}) = H'' \), and \( \text{Remap}(\tilde{v}, A, M) = \tilde{v}'' \), which completes the proof.

Internal Address Reflection  Similar to the proof of the Atomic Address Reflection property.
**Unique Approximating Messages** Let \( \langle \hat{K}'', s'', A, M \rangle \) be a member of \( T_1 \odot T_2(\hat{K}, s) \), let \( \hat{\mu} \) and \( \hat{\mu}' \) be the message-map components of \( \hat{K} \) and \( \hat{K}' \) respectively, and let there be \( \hat{a}, H, \) and \( \hat{v} \) such that \( \langle \hat{a} \hat{\@} H, \hat{v} \rangle \in \text{dom}(\hat{\mu}) \) and \( \Lambda(\hat{a}) \) is internal.

By the definition of the composition, there exist \( \hat{K}', s', A', A'', M', \) and \( M'' \) such that \( \langle \hat{K}', s', A', M' \rangle \in T_2(\hat{K}, s) \), \( \langle \hat{K}'', s'', A'', M'' \rangle \in T_1(\hat{K}', s') \), \( A = A'' \circ A' \), and \( M = M'' \circ M' \). Let \( \hat{\mu}' \) be the message-map component of \( \hat{K}' \).

Let \( \hat{a}' = A'(\hat{a}) \). We then have that \( A''(\hat{a}') = A(\hat{a}) \), which is an internal address, so by Internal Address Reflection for \( T_1 \), \( \hat{a}' \) is internal. Then by Unique Approximating Messages for \( T_2 \), there exist \( \hat{a}'', H'', \) and \( \hat{v}'' \) such that

- \( \hat{\mu}(\hat{a} @ H, \hat{v}) \subset \hat{\mu}'(\hat{a}' @ H'', \hat{v}'') \) and
- \( \langle \hat{a}' @ H', \hat{v}' \rangle \) is the only member of \( \text{dom}(\hat{\mu}') \) such that \( A'(\hat{a}') = \hat{a}'', M'(H) = H'' \), and \( \text{Remap}(\hat{v}', A', M') \supset_{\text{id}, \text{id}} \hat{v}' \).

By Unique Approximating Messages for \( T_1 \), there exist \( \hat{a}''', H''', \) and \( \hat{v}''' \) such that

- \( \hat{\mu}'(\hat{a}' @ H', \hat{v}') \subset \hat{\mu}''(\hat{a}'' @ H''', \hat{v}') \) and
- \( \langle \hat{a}'' @ H'', \hat{v}'' \rangle \) is the only member of \( \text{dom}(\hat{\mu}'') \) such that \( A''(\hat{a}'') = \hat{a}'', M''(H''') = H''' \), and \( \text{Remap}(\hat{v}''', A'', M'') \supset_{\text{id}, \text{id}} \hat{v}''. \)

We then have that \( A(\hat{a}) = \hat{a}'' \) and \( M(\hat{H}) = H''' \), and by the Remap Approximation Composition Lemma, \( \text{Remap}(\hat{v}, A, M) \subset_{\text{id}, \text{id}} \hat{v}''' \).

Finally, let \( \hat{v}''' \) be some value such that \( \text{Remap}(\hat{v}, A, M) \subset_{\text{id}, \text{id}} \hat{v}''' \). We have that \( \text{Merge}(\hat{v}, \hat{v}') \) is defined, \( \text{Remap}(\hat{v}, A, M) \subset_{\text{id}, \text{id}} \hat{v}''' \), and \( \text{Remap}(\hat{v}, A, M) \subset_{\text{id}, \text{id}} \hat{v}''' \). Then by the Approximation Mergeability lemma (appendix I), \( \text{Merge}(\hat{v}'''', \hat{v}'''') \) is defined. By Well-Formed Preservation (proved above), \( \hat{K}''' \) is well-formed, so \( \hat{\mu}''' \) is fully merged. Therefore, it must be the case that \( \hat{v}'''' = \hat{v}''' \), and so \( \langle \hat{a}'' @ H'', \hat{v}'' \rangle \) is the only member of \( \text{dom}(\hat{\mu}'''') \) such that \( A(\hat{a}) = \hat{a}''', M(\hat{H}''') = H''' \), and \( \text{Remap}(\hat{v}, A, M) \subset_{\text{id}, \text{id}} \hat{v}''''. \)

**Unique Approximated Messages** Let \( \langle \hat{K}'', s'', A, M \rangle \) be a member of \( T_1 \odot T_2(\hat{K}, s) \), let \( \hat{\mu} \) and \( \hat{\mu}' \) be the message-map components of \( \hat{K} \) and \( \hat{K}' \) respectively, and let there be \( \hat{a}'', H'', \) and \( \hat{v}'' \) such that \( \hat{\mu}'(\hat{a} @ H, \hat{v}) = \text{single} \). By the definition of the composition, there exist \( \hat{K}', s', A', A'', M', \) and \( M'' \) such that \( \langle \hat{K}', s', A', M' \rangle \in T_2(\hat{K}, s) \), \( \langle \hat{K}'', s'', A'', M'' \rangle \in T_1(\hat{K}', s') \), \( A = A'' \circ A' \), and \( M = M'' \circ M' \). Let \( \hat{\mu}' \) be the message-map component of \( \hat{K}' \).

By Unique Approximated Messages for \( T_1 \), there exist \( \hat{a}', H', \) and \( \hat{v}' \) such that

- \( \hat{\mu}'(\hat{a}' @ H', \hat{v}') = \text{single} \) and
- \( \langle \hat{a}' @ H', \hat{v}' \rangle \) is the only member of \( \text{dom}(\hat{\mu}') \) such that \( A''(\hat{a}') = \hat{a}'', M'(H') = H'' \), and \( \text{Remap}(\hat{v}', A', M') \subset_{\text{id}, \text{id}} \hat{v}''. \)

By Unique Approximated Messages for \( T_2 \), there exist \( \hat{a}, H, \) and \( \hat{v} \) such that

- \( \hat{\mu}(\hat{a} @ H, \hat{v}) = \text{single} \) and
• \( \langle \hat{a} @ H, \hat{v} \rangle \) is the only member of \( \text{dom}(\hat{\mu}) \) such that \( A'(\hat{a}) = \hat{a}', M'(H) = H'' \), and \( \text{Remap}(\hat{v}, A', M') \sqsubseteq_{\text{id}, \text{id}} \hat{v}' \).

We then have that \( A(\hat{a}) = \hat{a}'' \) and \( M(H) = H'' \), and by the \text{Remap} Approximation Composition lemma, \( \text{Remap}(\hat{v}, A, M) \sqsubseteq_{\text{id}, \text{id}} \hat{v}'' \).

It remains to show that \( \hat{v} \) is the only value such that \( \langle \hat{a} @ H, \hat{v} \rangle \in \text{dom}(\hat{\mu}) \) and \( \text{Remap}(\hat{v}, A, M) \sqsubseteq_{\text{id}, \text{id}} \hat{v}'' \). For a contradiction, assume there is some \( \hat{v}'' \neq \hat{v} \) such that \( \langle \hat{a} @ H, \hat{v}'' \rangle \in \text{dom}(\hat{\mu}) \) and \( \text{Remap}(\hat{v}'', A, M) \sqsubseteq_{\text{id}, \text{id}} \hat{v}'' \). By Unique Approximating Messages for \( T_2 \), there exists a unique \( \hat{v}'' \) such that \( \langle \hat{a} @ H', \hat{v}'' \rangle \in \text{dom}(\hat{\mu}') \) and \( \text{Remap}(\hat{v}'', A', M') \sqsubseteq_{\text{id}, \text{id}} \hat{v}'' \). By Unique Approximating Messages for \( T_1 \), there exists a unique \( \hat{v}''' \) such that \( \langle \hat{a} @ H'', \hat{v}''' \rangle \in \text{dom}(\hat{\mu}'') \) and \( \text{Remap}(\hat{v}'', A'', M'') \sqsubseteq_{\text{id}, \text{id}} \hat{v}''' \). Because \( \hat{v} \) is the only value such that \( \langle \hat{a} @ H, \hat{v} \rangle \in \text{dom}(\hat{\mu}) \) and \( \text{Remap}(\hat{v}, A, M) \sqsubseteq_{\text{id}, \text{id}} \hat{v}'' \), \( \hat{v}''' \) is the only value such that \( \langle \hat{a} @ H', \hat{v}' \rangle \in \text{dom}(\hat{\mu}') \) and \( \text{Remap}(\hat{v}'', A', M') \sqsubseteq_{\text{id}, \text{id}} \hat{v}'' \). By the \text{Remap} Approximation Composition lemma, \( \text{Remap}(\hat{v}', A, M) \sqsubseteq_{\text{id}, \text{id}} \hat{v}'' \).

Then we have \( \text{Remap}(\hat{v}'', A, M) \sqsubseteq_{\text{id}, \text{id}} \hat{v}'', \text{Remap}(\hat{v}'', A, M) \sqsubseteq_{\text{id}, \text{id}} \hat{v}''' \), and \( \hat{v}'' \neq \hat{v}'' \). By the Unique Approximating Messages property for \( T_1 \odot T_2 \) (proved above), however, \( \hat{v}'' \) is the only value such that \( \langle \hat{a} @ H'', \hat{v}'' \rangle \in \text{dom}(\hat{\mu}'') \) and \( \text{Remap}(\hat{v}'', A, M) \sqsubseteq_{\text{id}, \text{id}} \hat{v}'' \). Then we have a contradiction, so such a \( \hat{v}'' \) does not exist, and therefore \( \hat{v} \) is unique for this purpose.

**Properties for** \( \text{TransExec}_{T_1 \odot T_2} \)

Let there be \( \hat{a} \text{ex} = \hat{K}_1 \xlongrightarrow{\hat{I}_{1,1}, \ldots, \hat{I}_{1,m}} \ldots \) and \( s \) such that

- \( \hat{K}_1 \) and \( s \) are well-formed,
- \( \hat{K}_1 \) is an externals-only configuration,
- \( \text{Mon}(s) \subseteq \text{Used}(\hat{K}_1) \), and
- no actor in \( \hat{K}_1 \) is handling an event.

We show each of the properties for the transformation of this set of inputs below.

**Initial Pair Correctness** Let \( \langle \hat{a} \text{ex}''', s''', A''', M''' \rangle \) be a member of \( \text{TransExec}_{T_1 \odot T_2}(\hat{a} \text{ex}, s) \). By the definition of that function, there exist \( \hat{a} \text{ex}'', s'', A', M' \) such that \( \langle \hat{a} \text{ex}''', s''', A''', M''' \rangle \in \text{TransExec}_{T_1}(\hat{a} \text{ex}', s') \) and \( \langle \hat{a} \text{ex}', s', A', M' \rangle \in \text{TransExec}_{T_1}(\hat{a} \text{ex}, s) \). Because those two transformations are conformance-reflecting, we have \( \langle \hat{K}'_1, s'' \rangle \in T_2(\hat{K}_1, s) \) where \( \hat{K}'_1 \) is the first configuration in \( \hat{a} \text{ex}'' \), and \( \langle \hat{K}'_1, s''' \rangle \in T_3(\hat{K}_1, s) \) where \( \hat{K}'_1 \) is the first configuration in \( \hat{a} \text{ex}''' \). Therefore, \( \langle \hat{K}_1, s'' \rangle \in T_1 \odot T_2(\hat{K}_1, s) \).

**Fairness Preservation** 1 Because each \( T_1 \) and \( T_2 \) are both conformance-reflecting, both \( \text{TransExec}_{T_1} \) and \( \text{TransExec}_{T_2} \) preserve fairness. Therefore, every program execution in \( \text{TransExec}_{T_1 \odot T_2} \) is fair.
Fairness Preservation 2 Let \( \hat{a} \) identify a necessarily enabled actor in \( \hat{K}_1 \) such that that either the actor is not necessarily enabled in the \( i \)th configuration of \( \hat{a} \) or it runs in the \( i \)th transition of \( \hat{a} \). Let \( \langle \hat{a} \hat{a}''', s'', A, M \rangle \) be a member of \( \text{TransExec}_{1,T_1}\hat{K}_1 \), let \( \hat{K}_i' \) be the first configuration in \( \hat{a} \) and let \( \hat{a}'' = \Lambda(\hat{a}) \). If \( \hat{a}'' \) does not identify a necessarily enabled actor in \( \hat{K}_1' \), then we're done.

Otherwise, by the definition of \( \text{TransExec}_{1,T_2} \), there exist \( \hat{a} \hat{a}'', s', A, M, A' \), and \( M' \) such that \( \langle \hat{a} \hat{a}'', s'', A'', M'' \rangle \in \text{TransExec}_{1,T_2}(\hat{a} \hat{a}'', s') \), \( \langle \hat{a} \hat{a}'', s', A', M' \rangle \in \text{TransExec}_{1,T_2}(\hat{a} \hat{a}'', s') \), \( A = A'' \circ A' \), and \( M = M'' \circ M' \). Let \( \hat{K}_i' \) be the first configuration in \( \hat{a} \hat{a}'', s', A', M' \). By the Initial Pair Correctness property for \( \text{TransExec}_{1,T_2} \), \( \langle \hat{K}_i', s', A', M' \rangle \in T_2(\hat{K}_1, s) \) and \( \langle \hat{K}_i', s'', A'', M'' \rangle \in T_1(\hat{K}_1', s) \).

By the No New Enabled Actors property for \( T_1 \) and \( T_2 \), there exist \( \hat{a}' \) and \( \hat{a}''' \) such that \( A'(\hat{a}') = \hat{a}'' \) \( A''(\hat{a}'') = \hat{a}' \). \( \hat{a}' \) identifies a necessarily enabled actor in \( \hat{K}_1' \), \( \hat{a}''' \) identifies a necessarily enabled actor in \( \hat{K}_1 \). By the All-Awaiting Preservation property for \( T_1 \) and \( T_2 \), no actor in \( \hat{K}_1' \) or \( \hat{K}_1 \) is handling an event, so by the definition of a necessarily enabled actor, both \( \hat{a}''' \) and \( \hat{a}' \) are atomic. We have that \( A''(\hat{a}'') = \hat{a}'' \) and \( A''(A'(\hat{a}')) = \hat{a}' \), so by the Unique Actor Correspondences property for \( T_1 \), \( A'(\hat{a}''') = \hat{a}' \). Then because we also have \( A'(\hat{a}''') = \hat{a}' \), by the Unique Actor Correspondences property for \( T_2 \), \( \hat{a}''' = \hat{a}' \).

By the Fairness Preservation 2 property for \( \text{TransExec}_{T_2} \), there exists some \( k \leq i \) such that either the actor at \( \hat{a}'' \) is not necessarily enabled in the \( 4 \)th configuration of \( \hat{a} \) or it runs in the \( k \)th transition of \( \hat{a} \). Then by the Fairness Preservation 2 property for \( \text{TransExec}_{T_1} \), there exists some \( j \leq k \) such that either the actor at \( \hat{a}''' \) is not necessarily enabled in the \( j \)th configuration of \( \hat{a} \) or it runs in the \( j \)th transition of \( \hat{a} \), which completes the proof.

Fairness Preservation 3 Let \( \hat{\mu} \) be the message-map component of \( \hat{K}_1 \), and let there be \( \hat{a}, H, \hat{v}, \text{ and } i \) such that \( \langle \hat{a} \hat{a}, s''', A, M \rangle \) is a member of \( \text{TransExec}_{1,T_2}(\hat{a} \hat{a}', s) \), \( \hat{K}_i'' \) is the first configuration in \( \hat{a} \hat{a}'', s'', A', M' \) such that \( \langle \hat{a} \hat{a}'', s'', A'', M'' \rangle \in \text{TransExec}_{1,T_2}(\hat{a} \hat{a}'', s') \), \( \langle \hat{a} \hat{a}'', s', A', M' \rangle \in \text{TransExec}_{1,T_2}(\hat{a} \hat{a}'', s') \), \( A = A'' \circ A' \), and \( M = M'' \circ M' \). Let \( \hat{K}_i' \) be the first configuration in \( \hat{a} \hat{a}'', s''', A', M' \). By the Initial Pair Correctness property for \( \text{TransExec}_{1,T_2} \), \( \langle \hat{K}_i', s', A', M' \rangle \in T_2(\hat{K}_1, s) \) and \( \langle \hat{K}_i', s'', A'', M'' \rangle \in T_1(\hat{K}_1', s) \).

By the No New Single Messages property for \( T_1 \) and \( T_2 \), there exist \( \hat{a}', \hat{a}''', H', H'', \hat{v}' \), and \( \hat{v}''' \) such that

- \( A''(\hat{a}') = \hat{a}'' \)
- \( A'(\hat{a}''') = \hat{a}' \)
- \( M'(H') = H'' \)


- $M'(H'') = H'$,
- $\text{Remap}(\tilde{\nu}', A'', M'') = \tilde{\nu}''$,
- $\text{Remap}(\tilde{\nu}'', A', M') = \tilde{\nu}'$,
- $\bar{\mu}(\tilde{\nu}' @ H', \tilde{\nu}') = \text{single}$, and
- $\bar{\mu}(\tilde{\nu}'' @ H'', \tilde{\nu}'') = \text{single}$.

Thus, we have $A(\tilde{\nu}'') = \tilde{a}'', M(H'') = H''$, and $\text{Remap}(\tilde{\nu}'', A, M) = \tilde{\nu}''$. We also already have $\text{Remap}(\tilde{\nu}, A, M) = \tilde{\nu}$, so by the Expression Reflexivity lemma, $\text{Remap}(\tilde{\nu}, A, M) \subseteq \text{id, id}$ $\tilde{\nu}''$ and $\text{Remap}(\tilde{\nu}'', A, M) \subseteq \text{id, id}$ $\tilde{\nu}''$. Then by the Unique Approximated Messages property for $\operatorname{T}_1 \odot T_2$ (proved below), $\tilde{a}'' = \tilde{a}$, $H'' = H$, and $\tilde{\nu}'' = \tilde{\nu}$.

By the Fairness Preservation 3 property for $\operatorname{TransExec}_{T_2}$, there exists some $k \leq i$ such that $\langle \tilde{a}' @ H', \tilde{\nu}' \rangle$ is not a message in $\tilde{K}_i^k$. $\langle \tilde{a}' @ H', \tilde{\nu}' \rangle$ has quantity many in $\tilde{K}_i$, or that message is received in the $k$th transition of $\tilde{a} \tilde{e} x$. Then by the Fairness Preservation 3 property for $\operatorname{TransExec}_{T_1}$, there exists some $j \leq k$ such that $\langle \tilde{a}'' @ H'', \tilde{\nu}'' \rangle$ is not a message in $\tilde{K}_j^j$, $\langle \tilde{a}'' @ H'', \tilde{\nu}'' \rangle$ has quantity many in $\tilde{K}_j^j$, or that message is received in the $j$th transition of $\tilde{a} \tilde{e} x''$, which completes the proof.

**Properties for $\operatorname{UntransExec}_{T_1 \odot T_2}$**

Let there be $\tilde{a} \tilde{e} x = \tilde{K}_1 \overset{\ell_1, \ldots, \ell_m}{\rightarrow} \ldots, s$, and $X$ such that

- $\tilde{K}_1$ and $s$ are well-formed,
- $\tilde{K}_1$ is an externals-only configuration,
- $\text{Mon}(s) \subseteq \text{Used}(\tilde{K}_1)$,
- no actor in $\tilde{K}_1$ is handling an event, and
- $X$ contains a simulation for all members of $\operatorname{TransExec}_{T_1 \odot T_2}(\tilde{a} \tilde{e} x, s)$.

By the definition of $\operatorname{UntransExec}_{T_1 \odot T_2}$, there exist $X_1$ and $X_2$ defined as follows such that $\operatorname{UntransExec}_{T_1 \odot T_2}(\tilde{a} \tilde{e} x, s, X) = \operatorname{UntransExec}_{T_2}(\tilde{a} \tilde{e} x, s, X_2)$.

$$
X_1 = \bigcup_{\tilde{a} \tilde{e} x \in \tilde{K}_1, s_1, A_1, M_1} X_1(\tilde{a} \tilde{e} x, s_1, A_1, M_1)
$$

$$
X_2 = \bigcup_{\tilde{a} \tilde{e} x \in \tilde{K}_2, s_2, A_2, M_2} X_2(\tilde{a} \tilde{e} x, s_2, A_2, M_2)
$$

Furthermore, there exists some $S_1 \overset{\Lambda_1}{\rightarrow} \ldots = \operatorname{UntransExec}_{T_1 \odot T_2}(\tilde{a} \tilde{e} x, s, X)$, and $\operatorname{UntransExec}_{T_1 \odot T_2}(\tilde{a} \tilde{e} x, s, X) = \operatorname{UntransExec}_{T_2}(\tilde{a} \tilde{e} x, s, X_2)$. The proofs for each of the properties follow.
Execution Simulation  By the definition of $UntransExec_{T_1 \circ T_2}$ and the Execution Simulation properties for $UntransExec_{T_1}$ and $UntransExec_{T_2}$.

Prefix Consistency  By the definition of $UntransExec_{T_1 \circ T_2}$ and the Prefix Consistency properties for $UntransExec_{T_1}$ and $UntransExec_{T_2}$.

Fulfillment Reflection 1  Let $\langle \eta, po \rangle$ be a member of $Obls(s)$ such that for all $\langle \tilde{aex}', s'', A, M \rangle \in TransExec_{T_1 \circ T_2}(\tilde{aex}, s)$, either there is no $\eta'$ such that $M(\eta) = \eta'$ and $\langle \eta', po \rangle \in Obls(s'')$, or $X(\tilde{aex}', s'', A, M)$ eventually fulfills that $\langle M(\eta), po \rangle$. We must show that the execution $S_1 \xrightarrow{\Lambda_1} \ldots$ eventually fulfills the obligation $\langle \eta, po \rangle$. Because $UntransExec_{T_1 \circ T_2}(\tilde{aex}, s, X) = TransExec_{T_2}(\tilde{aex}, s, X_2)$, by the Fulfillment Reflection 1 property for $UntransExec_{T_2}$, it is sufficient to show that for all $\langle \tilde{aex}', s', A', M' \rangle \in TransExec_{T_2}(\tilde{aex}, s)$, either there is no $\eta'$ such that $M'(\eta) = \eta'$ and $\langle \eta', po \rangle \in Obls(s')$, or $X_2(\tilde{aex}', s', A', M')$ eventually fulfills $\langle M'(\eta), po \rangle$.

Let $\langle \tilde{aex}', s', A', M' \rangle$ be a member of $TransExec_{T_2}(\tilde{aex}, s)$. If there is no $\eta'$ such that $M'(\eta) = \eta'$ and $\langle \eta', po \rangle \in Obls(s')$, then we're done. Otherwise, by the definition of $X_2$, $X_2(\tilde{aex}', s', A', M') = TransExec_{T_1}(\tilde{aex}', s', X_1)$. By the Fulfillment Reflection 1 property for $TransExec_{T_1}$, it is sufficient to show that for all $\langle \tilde{aex}''', s'', A'', M''' \rangle \in TransExec_{T_1}(\tilde{aex}', s')$, either there is no $\eta''$ such that $M'''(\eta) = \eta''$ and $\langle \eta'', po \rangle \in Obls(s'')$, or $X_1(\tilde{aex}''', s'', A'', M''')$ eventually fulfills $\langle M'''(\eta), po \rangle$.

Let $\langle \tilde{aex}''', s'', A'', M''' \rangle$ be a member of $TransExec_{T_1}(\tilde{aex}', s')$. If there is no $\eta''$ such that $M'''(\eta) = \eta''$ and $\langle \eta'', po \rangle \in Obls(s'')$, then we're done. Otherwise, by the definition of $X_1$, $X_1(\tilde{aex}''', s'', A'', M''') = X(\tilde{aex}''', s'', A'' \circ A', M'' \circ M')$. By the definition of $TransExec_{T_1 \circ T_2}(\tilde{aex}''', s'', A'' \circ A', M'' \circ M') \in TransExec_{T_1 \circ T_2}(\tilde{aex}, s)$. We know that $M(\eta) = \eta''$ and $\langle \eta'', po \rangle \in Obls(s'')$. Therefore, by the above conditions on $X$, $X(\tilde{aex}''', s'', A'' \circ A', M'' \circ M')$ fulfills the obligation $\langle \eta'', po \rangle$, which completes the proof.

Fulfillment Reflection 2  Let there be some $i$ such that for all every execution in $rng(0(X))$ fulfills all obligations appearing in its first $i$ configurations. Then by the definition of $X_1$, every execution in $rng(X_1)$ fulfills all obligations appearing in its first $i$ configurations. Then by the definition of $X_2$ and the Fulfillment Reflection 2 property for $UntransExec_{T_2}$, every execution in $rng(X_2)$ fulfills all obligations appearing in its first $i$ configurations. Finally, by the definition of $X$ and the Fulfillment Reflection 2 property for $UntransExec_{T_2}$, $S_1 \xrightarrow{\Lambda_1} \ldots$ fulfills all obligations appearing in its first $i$ configurations. □
Appendix M

Correctness Proof for ModelCheck

M.1 PsmSimluateOutput Correctness Lemma

Lemma (PsmSimluateOutput Correctness). For all \( s, s', \hat{a}, \hat{v}, m, O, \) and \( S \) if \( \langle O, s', S \rangle \in \text{PsmSimluateOutput}(s, \hat{a}, H, \hat{v}, m) \), then

- if \( m = \text{single} \), then \( \{ s \} \xrightarrow{\hat{a}@H!\hat{v},O} \{ s' \} \cup S \) and \( \langle \eta, po \rangle \in \text{Obls}(s) \) for all \( \langle \eta, po \rangle \in \text{Obls}(s) \), and

- if \( m = \text{many} \), then for all \( n \in \mathbb{N} \) (including 0), there exist \( \hat{\lambda}_1, \ldots, \hat{\lambda}_n \) such that \( \hat{\lambda}_i = \hat{a}@H!\hat{v} \) for all \( i \in 1 \ldots n \), \( O = \emptyset \), \( s' = s \), and \( \{ s \} \xrightarrow{\hat{\lambda}_1,\hat{v}} \ldots \rightarrow \{ s' \} \).

Proof: First, consider the case where \( \text{OutMon}(s) \cap H = \emptyset \). In that case, \( s' = s \), \( O = \emptyset \), and \( S = \emptyset \). If \( m = \text{single} \), then by the P-SEND rule, \( s \xrightarrow{\hat{a}@H!\hat{v},O,S} s \), and \( \text{Obls}(s) = \text{Obls}(s') \), which completes the proof for that case. If \( m = \text{many} \), then the PSM can take that step any number of times, i.e., \( s \xrightarrow{\hat{a}@H!\hat{v},O,\ldots} \ldots \rightarrow s \), and so by repeated uses of the S-SENDORRECEIVE rule, \( s \xrightarrow{\hat{\lambda}_1,\hat{v}} \ldots \rightarrow \{ s' \} \), where \( \hat{\lambda}_i = \hat{a}@H!\hat{v} \) for all \( i \in 1 \ldots n \).

PsmSimluateOutput returns the empty set whenever \( |\text{OutMon}(s) \cap H| > 1 \), so it remains to discuss the results in the case where \( |\text{OutMon}(s) \cap H| = 1 \). Let \( \{ \eta' \} = \text{OutMon}(s) \cap H \). If \( m = \text{many} \), by the definition of PsmSimluateOutput, then there exists some \( s'' \) and \( po \) such that

- \( s'' \) is just like \( s \) except that \( \text{Obls}(s'') = \text{Obls}(s) \cup \langle \eta', po \rangle \),

- \( s \xrightarrow{\cdot,\cdot,\cdot} s'' \), and

- \( \hat{v} \sim po \bowtie \phi, \phi \).
Then we can use the P-SEND rule to get \( s'' \vdash H, \hat{v}, \{ \langle \eta', po \rangle \} \rightarrow s \). Then by the S-FREE-TRANSITION rule, the S-SEND\&RECEIVE rule, and the definition of the \( \rightarrow \) relation, we get \( \{ s \} \rightarrow \{ s \} \). That sequence can be repeated any number of times to get \( \{ s \} \rightarrow \{ s \} \), where \( \hat{\lambda}_i = \hat{a}@H!\hat{v} \) for all \( i \in 1 \ldots n \). Also by the definition of PsmSimulateOutput, \( O = \emptyset \) and \( S = \emptyset \), which completes this case.

Otherwise, \( m = \text{single} \), and by the definition of PsmSimulateOutput, there are three cases to check. In the first case, there exists some \( po \) such that \( \langle \eta', po \rangle \in Obls(s) \). Let \( O = \{ \langle \eta', po \rangle \} \), and let \( s' \) be a PSM just like \( s \) except it is missing the obligation \( \langle \eta', po \rangle \). Then by the P-SEND rule, we have \( s \vdash H, \hat{v}, O \rightarrow s' \), and \( \langle \eta', po \rangle \in Obls(s') \subseteq O \) for all \( \langle \eta', po \rangle \in Obls(s) \).

In the remaining two cases of the algorithm, there exists some \( s' \), \( s'' \), and \( po \) such that \( s \vdash s'' \), \( Obls(s') = Obls(s) \cup \{ \langle \eta', po \rangle \} \), \( \hat{v} \sim po \triangleright H'' \), \( S' \), and \( Obls(s'') = Obls(s) \). Let \( O = \{ \langle \eta', po \rangle \} \). We additionally know by the definition of PsmSimulateOutput that \( s' \) is just like \( s'' \) except for the obligations, so by the P-SEND rule, we have \( s'' \vdash H, \hat{v}, O \rightarrow s' \). Then by the S-FREE-TRANSITION rule, the S-SEND\&RECEIVE rule, and the definition of the \( \rightarrow \) relation, we get \( \{ s \} \rightarrow \{ s' \} \cup S \). Finally, by the above definitions, we have that \( \langle \eta', po \rangle \in Obls(s') \subseteq O \) for all \( \langle \eta', po \rangle \in Obls(s) \), which completes the proof.

\[ \square \]

\section{M.2 SimulateOutput Correctness Lemma}

\textbf{Lemma (SimulateOutput Correctness).} For all \( S, S', \hat{a}, \hat{v}, m, \) and \( O \), if \( \langle O, S' \rangle \in \text{SimulateOutput}(S, \hat{a}, H, \hat{v}, m) \), then

- if \( m = \text{single} \), then \( S \vdash H, \hat{v}, \{ \langle \eta', po \rangle \} \rightarrow S' \) and \( \langle \eta, po \rangle \in Obls(S') \subseteq O \) for all \( \langle \eta, po \rangle \in Obls(S) \), and

- if \( m = \text{many} \), then for all \( n \in \mathbb{N} \) (including 0), there exist \( \hat{\lambda}_1, \ldots, \hat{\lambda}_n \) such that \( \hat{\lambda}_1 = \hat{a}@H!\hat{v} \) for all \( i \in 1 \ldots n \), \( O = \emptyset \), \( S' = S \), and \( S \rightarrow S \).

\textbf{Proof.} The proof is by induction on \( |S| \). In the base case where \( S = \emptyset \), by the definition of SimulateOutput, \( O = \emptyset \) and \( S' = S \). Then by the definition of the \( \rightarrow \) relation and the S-SEND\&RECEIVE rule, it is trivially the case that

- \( S \vdash H, \hat{v}, \{ \langle \eta, po \rangle \} \rightarrow S' \),

- \( \langle \eta, po \rangle \in Obls(S') \subseteq O \) for all \( \langle \eta, po \rangle \in Obls(S) \),

- for all \( n \in \mathbb{N} \) (including 0), there exist \( \hat{\lambda}_1, \ldots, \hat{\lambda}_n \) such that \( \hat{\lambda}_1 = \hat{a}@H!\hat{v} \) for all \( i \in 1 \ldots n \) and \( S \rightarrow S' \).

Otherwise, by the definition of SimulateOutput, there exist \( s, s', S'', S''' \), \( O' \), and \( O'' \) such that
M.3. SimulateOutputs Correctness Lemma

Lemma (SimulateOutputs Correctness). For all $S$, $\hat{\mu}$, $O$, $S''$, and $\bar{l}_1, \ldots, \bar{l}_n$, if

- $\langle S'', O \rangle \in \text{ResolveOutputs}(S, \hat{\mu})$,
- there is no rcv-ext label in $\bar{l}_1, \ldots, \bar{l}_n$,
- $\hat{a}_1 : \text{send-ext}(\hat{a}_1'@H_1, \bar{v}_1), \ldots, \hat{a}_m : \text{send-ext}(\hat{a}_m'@H_m, \bar{v}_m)$ are the send-ext labels in $\bar{l}_1, \ldots, \bar{l}_n$, and
- $\emptyset \oplus \langle \hat{a}_1'@H_1, \bar{v}_1 \rangle \oplus \ldots \oplus \langle \hat{a}_m'@H_m, \bar{v}_m \rangle \subseteq_{\text{id}, \text{id}} \hat{\mu}$,

then there exist $O'_1, \ldots, O'_n$ such that

- $S \langle \bar{l}_1, O'_1 \rangle \ldots \langle \bar{l}_n, O'_n \rangle \rightarrow S''$,
- $O \subseteq O'_1 \cup \ldots \cup O'_n$, and
- for all $\langle \eta, \text{po} \rangle \in \text{Obls}(S)$, $\langle \eta, \text{po} \rangle \in \text{Obls}(S'') \cup O$.
Proof. The proof is by induction on $|\text{dom}(\hat{\mu})|$. In the base case where $\text{dom}(\hat{\mu}) = \emptyset$, the only result is the tuple $\langle \emptyset, S \rangle$. Let $O'_1 = \emptyset$ for all $i \in 1 \ldots n$. There must be no send-ext or rcv-ext label in $\hat{\lambda}_1, \ldots, \hat{\lambda}_n$, so $[\hat{\lambda}_i] = •$ for all $i \in 1 \ldots n$, and therefore $S \langle [\hat{\lambda}_i, O'_i] \rangle \ldots \langle [\hat{\lambda}_n, O'_n] \rangle_S, O \subseteq O'_1 \cup \ldots \cup O'_n$, and for all $\langle \eta, po \rangle \in \text{Obls}(S), \langle \eta, po \rangle \in \text{Obls}(S) \cup O$.

If $\hat{\mu}$ is non-empty, let there be $\langle S'', O \rangle \in \text{ResolveOutputs}(S, \hat{\mu})$. By the definition of SimulateOutputs, there exist $\hat{a}, H, \hat{\nu}, m, \hat{\mu'}, O', O''$, and $S'$ such that

- $\hat{\mu}(\hat{a} @ H, \hat{\nu}) = m$,
- $\hat{\mu'} = \hat{\mu}|_{\text{dom}(\hat{\mu}) - (\hat{a} @ H, \hat{\nu})}$,
- $\langle O', S' \rangle \in \text{SimulateOutput}(\hat{\lambda}, H, \hat{\nu}, m)$,
- $\langle O'', S'' \rangle \in \text{SimulateOutputs}(S', \hat{\mu'})$, and
- $O = O' \cup O''$.

Let $\hat{\lambda}_1, \ldots, \hat{\lambda}_m$ be the sequence of labels from $\hat{\lambda}_1, \ldots, \hat{\lambda}_n$ other than those of the form $\hat{a}': \text{send-ext}(\hat{a}'' @ H', \hat{\nu}'')$ where $\hat{a}'' = \hat{a}, H' = H, \text{Merge}(\hat{\nu}, \hat{\nu}'')$ is defined. Then $\hat{\mu}'$ is the message map created by combining all of the messages from the rcv-ext labels in $\hat{\lambda}_1, \ldots, \hat{\lambda}_m$ as described in the preconditions to this lemma. Therefore, by the induction hypothesis, there exist $O''_1, \ldots, O''_n$ such that

- $S' \langle \hat{\lambda}_1, O''_1 \rangle \ldots \langle \hat{\lambda}_m, O''_m \rangle_S, S''$,
- $O'' \subseteq O''_1 \cup \ldots \cup O''_m$, and
- for all $\langle \eta, po \rangle \in \text{Obls}(S'), \langle \eta, po \rangle \in \text{Obls}(S'') \cup O''$.

Let $\hat{\lambda}_1, \ldots, \hat{\lambda}_p$ be the labels from $\hat{\lambda}_1, \ldots, \hat{\lambda}_n$ not included in the sequence $\hat{\lambda}_1, \ldots, \hat{\lambda}_m$. By the SimulateOutput Correctness lemma, if $m = \text{single}$, then $S \overset{\hat{a} @ H \cup O'}{\longrightarrow} S'$ and $\langle \eta, po \rangle \in \text{Obls}(S') \cup O$ for all $\langle \eta, po \rangle \in \text{Obls}(S)$, and if $m = \text{many}$, then for all $n \in \mathbb{N}$ (including 0), there exist $\hat{\lambda}_1, \ldots, \hat{\lambda}_n$ such that $\hat{\lambda}_i = \hat{a} @ H! \hat{\nu}$ for all $i \in 1 \ldots n$, $O' = \emptyset$, and $S' \overset{\hat{\lambda}_1, \ldots, \hat{\lambda}_n, \emptyset}{\longrightarrow} S'$. Thus, there exist $O''_1, \ldots, O''_p$ such that $S \langle \hat{\lambda}_1, O''_1 \rangle \ldots \langle \hat{\lambda}_p, O''_p \rangle_S' \text{ and } O' \subseteq O''_1 \cup \ldots \cup O''_p$.

If we concatenate the transitions $S \langle \hat{\lambda}_1, O''_1 \rangle \ldots \langle \hat{\lambda}_p, O''_p \rangle S'$ and $S' \langle \hat{\lambda}_1, O''_1 \rangle \ldots \langle \hat{\lambda}_p, O''_p \rangle S''$, then by the Abstract Specification Step Commutativity lemma, we get $S \langle \hat{\lambda}_1, O''_1 \rangle \ldots \langle \hat{\lambda}_p, O''_p \rangle S'' \text{ and } O' \subseteq O''_1 \cup \ldots \cup O''_n$, and for all $\langle \eta, po \rangle \in \text{Obls}(S), \langle \eta, po \rangle \in \text{Obls}(S'') \cup O$.

$\square$
M.4  MatchingSpecSteps Correctness Lemma

Lemma (MatchingSpecSteps Correctness). For all $s$ and $L$ and all $\langle O, S'' \rangle \in$ MatchingSpecSteps$(s, L)$, $\{L, O\}$ $\rightarrow S''$.

Proof. Let $\hat{l}_1, \ldots, \hat{l}_n$ be a label sequence summarized by $L$. It must be the case that $L = \langle \hat{l}_1, \hat{l}_2, \ldots, \hat{l}_n \rangle$ for some $\hat{l}_i$. For each $O$ added to $initialSteps$, either $\langle s, \hat{l}_i \rangle \rightarrow S', \iff [\hat{l}_i] = \ast$, $S' = \{s\}$ and $\{s\} \rightarrow \{s\}$ by the definition of the $\rightarrow$ relation. Therefore in either case, $\langle s, \hat{l}_i \rangle \rightarrow S'$.

Because $\hat{l}_1$ cannot be a send-ext label, by the transition rules for PSMs, it must also be the case that $Obls(s) \subseteq S'$.

Then by the SimulateOutputs Correctness lemma, for each $S' \in initialSteps$ and each $\langle O, S'' \rangle \in$ SimulateOutputs$(S, \hat{l}_i)$, there exist $O_2, \ldots, O_n'$ such that

- $S' \rightarrow \langle [\hat{l}_2], O_2' \rangle \rightarrow \langle [\hat{l}_3], O_3' \rangle \rightarrow \cdots \rightarrow \langle [\hat{l}_n], O_n' \rangle \rightarrow S''$,
- $O \subseteq O_1' \cup \cdots \cup O_n'$, and
- for all $\langle \eta, po \rangle \in Obls(S')$, $\langle \eta, po \rangle \in Obls(S') \cup O$.

The label $\hat{l}_1$ cannot be a send-ext label, so in that case we know that $\langle s, \hat{l}_1 \rangle \rightarrow S'$. Let $O_1' = \emptyset$ and we have $\langle s, \hat{l}_1 \rangle \rightarrow \langle [\hat{l}_2], O_2' \rangle \rightarrow \langle [\hat{l}_3], O_3' \rangle \rightarrow \cdots \rightarrow \langle [\hat{l}_n], O_n' \rangle \rightarrow S''$. Finally, MatchingSpecSteps adds a pair $\langle O, S'' \rangle$ to the result set only if either $Mon(s) = \emptyset$ and there is no $s'$ such that $s \rightarrow \emptyset, s'$, or for all $s'' \in S''$, $Mon(s'') \neq \emptyset$. That satisfies the final condition for a specification summary transition, so we have $\langle s, \hat{l}_1 \rangle \rightarrow S''$. \qed

M.5  Explore Correctness Lemma

Lemma (Explore Correctness). Lemma: For all $T, \hat{K}_{init}, s_{init}, R, Z,$ and $F$, if $T$ is the transformation used in Explore and $Explore(\hat{K}_{init}, s_{init}) = \langle R, Z, F \rangle$, then $R$ is a local-transformation-simulation relation with witness $Z$ and frontier $F$ with respect to $T$.

Proof. We prove this lemma by establishing a loop invariant on the function’s main while loop. The invariant states that $R$ is a local-transformation-simulation relation with witness $Z$ and frontier $W \cup F$.

Just before the loop, $R = \emptyset$ is empty, so the invariant is trivially true.

The body of the loop first takes an arbitrary configuration pair $\langle \hat{K}, s \rangle$ out of the worklist. If $ShouldExplore(s)$ returns false, then Explore adds $\langle \hat{K}, s \rangle$ to the frontier $F$ and returns to the beginning of the loop. This merely moves a member of $W$ into $F$ without modifying $R$ or $Z$, so it is still the case that so $R$ is a local-transformation-simulation relation with witness $Z$ and frontier $W \cup F$.

On the other hand, if $ShouldExplore(s)$ returns true, Explore calls $ProgSteps$ to compute a set of summary transitions that summarize all possible event-step
transitions from $\hat{K}$. Then for each one, it calls $\text{MatchingSpecSteps}$ to find some set of matching specification transitions from $s$ and adds them to $Z$. Specifically, the $\text{MatchingSpecSteps}$ correctness lemma states that, for each transition label $L$ and all $\langle O,S \rangle \in \text{MatchingSpecSteps}(s,L)$, $(s) \xrightarrow{(L,O)} S$. If for any such $L$, $\text{MatchingSpecSteps}(s,L)$ returns the empty set, then $\text{Explore}$ adds $\langle \hat{K},s \rangle$ to the frontier $F$ and returns to the beginning of the loop. Once again, this merely moves a member of $W$ into $F$ without modifying $R$ or $Z$, so it is still the case that $R$ is a local-transformation-simulation relation with witness $Z$ and frontier $W \cup F$.

Otherwise, for each such transition $\hat{K} \xrightarrow{L} \hat{K}', Z(\hat{K} \xrightarrow{L} \hat{K}',s) \neq \emptyset$, and for all $\langle O,S \rangle \in \text{MatchingSpecSteps}(s,L)$, $(s) \xrightarrow{(L,O)} S$. In that case, $\text{Explore}$ adds $\langle \hat{K},s \rangle$ to $R$, and for each such specification transition, all $s' \in S$, and all $\langle \hat{K}'',s'',A,M \rangle \in T(\hat{K}',s')$, $\text{Explore}$ adds $\langle \hat{K}'',s'' \rangle$ to the worklist $W$ if that pair is not already a member of $R$ or $F$. Thus, for all transformation-step transitions from $\langle \hat{K},s \rangle$ using strategy $Z$ and transformation $T$, the reached pair must be in one of $R$, $W$, or $F$. No other changes are made to $R$, $W$, $F$, or $Z$, so $R$ is a local-transformation-simulation relation with witness $Z$ and frontier $W \cup F$.

The loop ends when $W$ is empty. When that happens, $R$ is a local-transformation-simulation relation with witness $Z$ and frontier $F$, which completes the proof.

\[\Box\]

**M.6 Prune Correctness Lemma**

**Lemma** (Prune Correctness). For all $R$, $Z$, $F$, $T$, $R'$, and $Z'$, if $R$ is a local-transformation-simulation relation with witness $Z$ and frontier $F$ with respect to $T$, $T$ is conformance-reflecting and is the transformation used by Prune, and $\text{Prune}(R,Z,F) = \langle R',Z' \rangle$, then $R'$ is a transformation-simulation relation with witness $Z'$ and transformation $T$, $R' \subseteq R$, and for all $\langle \hat{K} \xrightarrow{L} \hat{K}',s \rangle \in \text{dom}(Z')$, $Z'(\hat{K} \xrightarrow{L} \hat{K}',s) \subseteq Z(\hat{K} \xrightarrow{L} \hat{K}',s)$.

**Proof.** We prove this lemma by establishing a loop invariant on the algorithm’s while loop. The invariant states that

1. $R'$ is a local-transformation-simulation relation with witness $Z'$ and frontier $F$ with respect to $T$,
2. $R' \subseteq R$, and
3. for all $\langle \hat{K} \xrightarrow{L} \hat{K}',s \rangle \in \text{dom}(Z')$, $Z'(\hat{K} \xrightarrow{L} \hat{K}',s) \subseteq Z(\hat{K} \xrightarrow{L} \hat{K}',s)$.

At the beginning of the loop, the preconditions of this lemma guarantee that the initial definitions of $R'$ and $Z'$ satisfy the invariants.
M.7. \textsc{FindFulfillingPairs Correctness Lemma}

\textbf{Lemma (FindFulfillingPairs Correctness).} For all $R, Z, T,$ and $R'$, if $R$ is a transformation-simulation relation with witness $Z$ and transformation $T$, $T$ is the transformation used by \textsc{FindFulfillingPairs}, and $\text{FindFulfillingPairs}(R, Z) = R'$, then $R' \subseteq R$ and for all $\langle \hat{K}, s \rangle \in R$, every program-fair transformation-step-execution starting at $\langle \hat{K}, s \rangle$ with witness $Z$ and transformation $T$ is specification-fair.

\textbf{Proof.} Let $\langle \hat{K}, s \rangle$ be a pair in $R'$. Clearly $\langle \hat{K}, s \rangle$ is in $R$, because $R'$ is initialized to the empty set, and the pair $\langle \hat{K}, s \rangle$ can only be added to $R'$ after retrieving it from $R$. It remains to show that every program-fair transformation-step-execution starting at $\langle \hat{K}, s \rangle$ with witness $Z$ and transformation $T$ is specification-fair. We show this by contradiction.

Assume there is some transformation-step-execution starting at $\langle \hat{K}, s \rangle$ with witness $Z$ and transformation $T$ that is program-fair but not specification-fair. By the definition of specification-fairness, there must exist some pair $\langle \hat{K}', s' \rangle$ in that execution and some obligation $\langle \eta, po \rangle \in \text{Obls}(s')$ that is never fulfilled in
the execution from that point. Then the suffix of that execution starting from
\(\langle \hat{K}', s' \rangle\) is an execution in its own right, and that execution is program-fair but
not specification-fair.

As mentioned in chapter 7, because \(R\) is a transformation-simulation relation,
the set \(R\) is closed over transformation-step transitions enabled by the strategy \(Z\)
and transformation \(T\), so \(\langle \hat{K}', s' \rangle \in R\). Thus, there is a path in the non-satisfaction
graph from \(\langle \hat{K}', s' \rangle\) for \(\langle \eta, po \rangle\), \(Z\), and \(T\) representing a program-fair execution
that does not fulfill that obligation. Because \(\langle \hat{K}', s' \rangle \in R\) and \(\langle \eta, po \rangle \in \text{Obls}(s')\),
FindFulfillingPairs checks that graph.

If the non-fulfilling path is finite, then program-fairness dictates that the
implementation configuration \(\hat{K}''\) in the final vertex of that path must be quiescent.
However, if this were the case, then FindFulfillingPairs would find it at line 7
of the algorithm, report that a program-fair path was found, and therefore omit
that vertex from \(R'\). Therefore, the non-fulfilling path cannot be finite.

If the path is infinite, then the finite-ness of \(R\) implies that the path must be
lasso-shaped (i.e., end in a cycle). The lasso-shaped part is a suffix of the exe-
cution from \(\langle \hat{K}', s' \rangle\), and therefore it represents a program-fair execution. Also,
every cycle in a graph is part of a strongly-connected component of that graph,
so all of the vertices in the loop portion of that path must be contained by some
strongly connected component (SCC) of the graph. If that execution is fair, then
by the definition of a program-fair SCC, the SCC must be program-fair. If that
were the case, though, then FindFulfillingPairs would find it at line 11 of the
algorithm report that a program-fair path was found, and therefore omit that
vertex from \(R'\). Therefore, the non-fulfilling path cannot be infinite.

The path must be finite or infinite, so such a path cannot exist, and so we
have a contradiction. Therefore, there is no program-fair transformation-step-
execution starting at \(\langle \hat{K}, s \rangle\) with witness \(Z\) and transformation \(T\) that is not
specification-fair, which completes the proof.

\[\square\]

M.8 ModelCheck Transformation Conformance

Theorem (ModelCheck Transformation Conformance). For all \(P\) and \(\Sigma\),
if the transformation \(T\) used in ModelCheck is conformance-reflecting and
ModelCheck\((P, \Sigma)\) returns true, then \(P \models_T \Sigma\).

Proof. By the definition of ModelCheck, when ModelCheck\((P, \Sigma)\) returns true,
there exist \(K_{\text{init}}, s_{\text{init}}, \hat{K}_{\text{init}}, R_{\text{loc}}, Z_{\text{loc}}, F_{\text{loc}}, R_{\text{sim}}, Z_{\text{sim}}, R_{\text{fulfil}}, R_{\text{conf}},\) and \(Z_{\text{conf}}\)
such that

- \(\langle K_{\text{init}}, s_{\text{init}} \rangle = \text{MaxInstantiate}(P, \Sigma)\),
- \(K_{\text{init}} = |\hat{K}_{\text{init}}|\),
- \(\langle R_{\text{loc}}, Z_{\text{loc}}, F_{\text{loc}} \rangle = \text{Explore}(\hat{K}_{\text{init}}, s_{\text{init}})\),
- \(\langle R_{\text{sim}}, Z_{\text{sim}} \rangle = \text{Prune}(R_{\text{loc}}, Z_{\text{loc}}, F_{\text{loc}})\).
• $R_{fullfill} = \text{FindFulfillingPairs}(R_{sim}, Z_{sim})$,
• $\langle R_{conf}, Z_{conf} \rangle = \text{Prune}(R_{fullfill}, Z_{sim}, R_{sim} \setminus R_{fullfill})$, and
• $\langle \hat{K}_{init}, s_{init} \rangle \in R_{conf}$.

By the definition of $\text{MaxInstantiate}$, $\langle \hat{K}_{init}, s_{init} \rangle$ is a maximal instantiation of $P$ and $\Sigma$. We have that $\hat{K}_{init} \models \hat{K}_{init}$ and $\langle \hat{K}_{init}, s_{init} \rangle \in R_{conf}$, so it remains to show that $R_{conf}$ is a transformation-conformance-dense relation.

By the $\text{Explore Correctness}$ lemma, $R_{loc}$ is a local-transformation-simulation relation with witness $Z_{loc}$ and frontier $F_{loc}$. Then by the $\text{Prune Correctness}$ lemma, $R_{sim}$ is a transformation-simulation relation with witness $Z_{sim}$. Next, by the $\text{FindFulfillingPairs Correctness}$ lemma, $R_{fullfill} \subseteq R_{sim}$ and for all $\langle \hat{K}, s \rangle \in R_{fullfill}$, every program-fair transformation-step-execution starting at $\langle \hat{K}, s \rangle$ is specification-fair. Finally, by the $\text{Prune Correctness}$ lemma again, $R_{conf}$ is a transformation-simulation relation with witness $Z_{conf}$, $R_{conf} \subseteq R_{fullfill}$, and for all $\langle \hat{K}_{L}, \hat{K}\prime, s \rangle \in \text{dom}(Z')$, $\hat{Z}'(\hat{K}_{L} \rightarrow \hat{K}\prime, s) \subseteq \hat{Z}(\hat{K}_{L} \rightarrow \hat{K}\prime, s)$.

By the definition of a transformation-simulation relation, for all $\langle \hat{K}_{1}, s_{1} \rangle \in R_{conf}$ and all $\hat{\imath}_{1}, \ldots, \hat{\imath}_{n}$ and $\hat{K}'$ such that $\hat{K}_{1} \xrightarrow{\hat{\imath}_{1}, \ldots, \hat{\imath}_{n}} \hat{K}'$ and $\hat{K}'$ is not stuck, there exists $L$ such that

- $\hat{K}_{1} \xrightarrow{L} \hat{K}'$ summarizes $\hat{K}_{1} \xrightarrow{\hat{\imath}_{1}, \ldots, \hat{\imath}_{n}} \hat{K}'$, 
- $\hat{Z}(\hat{K}_{1} \xrightarrow{L} \hat{K}', s_{1}) \neq \varnothing$, and
- for all transformation steps $\langle \hat{K}_{1}, s_{1} \rangle \xrightarrow{L, \hat{K}', Z, O, S, \hat{\imath}, A_{1}, M_{1}} \langle \hat{K}_{2}, s_{2} \rangle, \langle \hat{K}_{2}, s_{2} \rangle \in R_{conf}$.

Therefore, $R_{conf}$ satisfies the Simulation condition for a transformation-conformance-dense relation.

For the Fulfillment condition, we know that and for all $\langle \hat{K}, s \rangle \in R_{fullfill}$, every program-fair transformation-step-execution starting at $\langle \hat{K}, s \rangle$ is specification-fair. We also know that $R_{conf} \subseteq R_{fullfill}$ and for all $\langle \hat{K} \xrightarrow{L} \hat{K}', s \rangle \in \text{dom}(Z')$, $\hat{Z}'(\hat{K} \xrightarrow{L} \hat{K}', s) \subseteq \hat{Z}(\hat{K} \xrightarrow{L} \hat{K}', s)$. Therefore, every program-fair transformation-step-execution starting at $\langle \hat{K}, s \rangle$ is specification-fair. As a result, $R_{conf}$ satisfies the Fulfillment condition for a transformation-conformance-dense relation, so $R_{conf}$ is transformation-conformance-dense. 

\qed
Appendix N

Termination Proof for
ModelCheck

This appendix is dedicated to proving that the ModelCheck algorithm always terminates for all inputs, as long as the transformation $T$ it uses is termination-guaranteeing (defined below). The next section introduces several definitions needed for the proof. Various lemmas needed for the proof follow, and the final section proves the top-level termination theorem itself.

In addition to the bounds effected by Unmark, Assimilate, and Canonicalize (namely, that the addresses and markers appearing in every pair come from some bounded set), the proof also relies on a notion of depth to show that the set of explored program configurations is bounded. This roughly corresponds to the lexical depth of the syntactic representation of each configuration, although it also accounts for semantic features that can increase the lexical depth, such as substitution or the unfolding of recursive types. Note that even values with recursive types have a bounded depth in CSA (e.g., it is impossible to write the standard Cons/Nil definition of a list), because all recursive types are of the form $(\text{rec } X (\text{Addr } t))$. Appendix H defines the depth of a program, abstract configuration, or component thereof as part of the type-checking rules, and appendix J proves that a transition never increases the depth of a program configuration.

N.1 Definitions for Termination Proofs

The symbol $D$ stands for a set of abstract addresses.

Definition. Let $c$ be a CSA program, abstract program configuration, or any component thereof. Then $Names(c)$ is defined to be the set of variables, type variables, variant tags, record-field names, state names, and primitive operations appearing in $c$. Additionally, if an address $\tilde{a}$ appears in $c$, $Names(c)$ includes $Names(ActorType(\tilde{a}))$.

Definition. Let $c$ be a CSA program, abstract program configuration, or any
APPENDIX N. TERMINATION PROOF FOR MODELCHECK

component thereof. Then $c$ is width-bounded by $w$ (where $w$ is some natural number) if and only if for all expressions $\hat{e}$, types $\tau$, and state definitions $\hat{Q}$ appearing in $c$ (including every type $\text{ActorType}(\hat{a})$ of every address $\hat{a}$ in $c$), and all S-expressions in the syntactic representation of $\hat{e}$, $\tau$, or $\hat{Q}$, that S-expression is either a list with at most $w$ subexpressions or an atom. Note that a set $\{\hat{v}_1, \ldots, \hat{v}_n\}$ used in the representation of an abstract list or dictionary is not considered an S-expression for this definition, but each element $\hat{v}_i$ in the set is.

**Definition.** An abstract program configuration $\hat{K}$ is a configuration of $P$ with initial externals $\{\hat{a}_1, \ldots, \hat{a}_n\}$ if and only if

- $\hat{K}$ is well-formed (appendix I),
- $\text{Names}(\hat{K}) \subseteq \text{Names}(P)$,
- there exists some width $w$ such that $\hat{K}$ is width-bounded by $w$ and $w$ is the minimal width such that $P$ is width-bounded by $w$,
- for every spawn expression in $\hat{K}$ associated with location $\ell$, there exists a spawn expression in $P$ at the same location $\ell$,
- for all behaviors $\hat{b}$ and spawn expressions with state definitions $\hat{Q}_1, \ldots, \hat{Q}_n$, there exists a spawn expression in $P$ with at least $n$ state definitions,
- for all internal addresses $\hat{a}$ appearing in $\hat{K}$, there exist $\ell$ and $n$ such that there exists a spawn expression in $P$ at location $\ell$ and either $\hat{a} = (\text{addr} \ell n)$ or $\hat{a} = (\text{collective-addr} \ell)$, and
- for all external addresses $\hat{a}$ appearing in $\hat{K}$, either $\hat{a} \in \{\hat{a}_1, \ldots, \hat{a}_n\}$, or there exists $\tau$ such that $\hat{a} \in \text{MaxVals}((\text{Addr} \tau))$.

We simply write $\hat{K}$ is a configuration of $P$ when the initial externals are irrelevant.

**Definition.** A PSM $s = \langle H, H', \phi : \eta_1, \ldots, \eta_n, \Phi, O \rangle$ is a PSM of $\Sigma$ if and only if

- $s$ is well-formed (appendix K),
- either $\phi = \text{Dummy}$ or $\phi$ appears in $\Sigma$,  
- either $n = 2$ or there exists a goto expression in $\Sigma$ with $n$ arguments,
- either $\Phi = (\text{define-state} \ (\text{Dummy}))$, or all of the state definitions $\Phi$ appear in $\Sigma$, and
- for every obligation $\langle \eta, po \rangle \in O$, there exists some $po'$ appearing in $\Sigma$ such that $po$ is $po'$ with all fork-addr patterns replaced with self.

**Definition.** An abstract program configuration $\hat{K}$ is Explore-bounded with maximal marker $\eta$ if and only if
• for every internal address $\hat{a}$ appearing in $\hat{K}$, there exists $\ell$ such that either $\hat{a} = (\text{addr } \ell 0)$ or $\hat{a} = (\text{collective-addr } \ell)$, and

• $\text{max(Used}(\hat{K})) \leq \eta$.

**Definition.** A PSM $s = \langle H, H', \varphi : \eta_1, \ldots, \eta_n, \overline{\alpha}, O \rangle$ is Explore-bounded if and only if $|\eta'| \leq \max(H \cup H')$ (i.e., the monitored markers come from the $n + 2$ smallest markers), and $O$ contains at most one copy of every obligation $\langle \eta, p_o \rangle$.

**Definition.** A transformation $T$ is termination-guaranteeing if and only if for all $\hat{K}, s, P, \Sigma, \hat{a}_1, \ldots, \hat{a}_n, \alpha, d$ if

• $\hat{K}$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$,

• $s$ is a PSM of $\Sigma$,

• there is no actor handling an event in $\hat{K}$,

• $\vdash\text{cfg} \hat{K} : d$, and

• there are at most two atomic actors in $\hat{K}$ for every spawn location $\ell$ (i.e., at most two actors with an address of the form $(\text{addr } \ell n)$ for each $\ell$),

then $T(\hat{K}, s)$ terminates and returns a finite set such that for all $\langle \hat{K}', s', A, M \rangle \in T(\hat{K}, s)$,

• $\hat{K}'$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$,

• $s'$ is a PSM of $\Sigma$,

• there is no actor handling an event in $\hat{K}'$,

• $\vdash\text{cfg} \hat{K}' : d$,

• $\hat{K}'$ is Explore-bounded with maximal marker $\text{max(Mon}(s'))$, and

• either $s'$ is Explore-bounded or $\text{ShouldExplore}(s') = \text{false}$.

**Definition.** A strategy $Z$ is a finite strategy if and only if

• there are finitely many $\langle \hat{K} \xrightarrow{L} \hat{K}', s \rangle \in \text{dom}(Z)$ and

• there exist $P, \Sigma, \hat{a}_1, \ldots, \hat{a}_n, \alpha, d$ such that for all $\langle \hat{K} \xrightarrow{L} \hat{K}', s \rangle \in \text{dom}(Z)$,

  – $\hat{K}'$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$,

  – there is no actor handling an event in $\hat{K}'$,

  – $\vdash\text{cfg} \hat{K}' : d$,

  – there are at most two atomic actors in $\hat{K}'$ for every spawn location $\ell$ (i.e., at most two actors with an address of the form $(\text{addr } \ell n)$ for each $\ell$),
\[ Z(\hat{K} \xrightarrow{L} \hat{K}', s) \text{ is finite, and} \]
\[ \text{for all } (O, S) \in Z(\hat{K} \xrightarrow{L} \hat{K}', s), S \text{ is finite and every } s'' \in S \text{ is a PSM of } \Sigma. \]

### 1.2 Remap Origin Preservation Lemma

**Lemma (Remap Origin Preservation).** For all \( \hat{K}, P \hat{a}_1, \ldots, \hat{a}_n, A, \) and \( M \), if

- \( \hat{K} \) is a configuration of \( P \) with initial externals \( \hat{a}_1, \ldots, \hat{a}_n \),
- every address appearing in \( \hat{K} \) is a member of \( \text{dom}(A) \), and
- for all \( \hat{a} \in \text{dom}(A) \),
  - if \( \hat{a} \) is internal, then \( A(\hat{a}) \) is either an internal address \( \hat{a}' \) such that \( \hat{a} \) and \( \hat{a}' \) have the same location \( \ell \) or an external address \( \hat{a}'' \) such that \( \hat{a}'' \in \text{MaxVals}(\text{Addr}(\tau)) \) for some \( \tau \), and
  - if \( \hat{a} \) is external, then \( A(\hat{a}) = \hat{a} \),

then \( \text{Remap}(\hat{K}, A, M) \) is a configuration of \( P \) with initial externals \( \hat{a}_1, \ldots, \hat{a}_n \).

**Proof.** Because every address appearing in \( \hat{K} \) is a member of \( \text{dom}(A) \), \( \text{Remap}(\hat{K}, A, M) \) is defined. Let \( \hat{K}' = \text{Remap}(\hat{K}, A, M) \).

Most of the conditions necessary to be a configuration of \( P \) with initial externals \( \hat{a}_1, \ldots, \hat{a}_n \) are trivially preserved by the definition of \( \text{Remap} \). It remains to show that

- \( \hat{K}' \) is well-formed (appendix I),
- for all internal addresses \( \hat{a} \) appearing in \( \hat{K}' \), there exist \( \ell \) and \( n \) such that there exists a \text{spawn} expression in \( P \) at location \( \ell \) and either \( \hat{a} = (\text{addr } \ell \ n) \) or \( \hat{a} = (\text{collective-addr } \ell) \), and
- for all external addresses \( \hat{a} \) appearing in \( \hat{K}' \), either \( \hat{a} \in \{\hat{a}_1, \ldots, \hat{a}_n\} \) or there exists \( \tau \) such that \( \hat{a} \in \text{MaxVals}(\text{Addr}(\tau)) \).

By the preconditions to this lemma, for every external address \( \hat{a} \in \text{dom}(A) \), \( A(\hat{a}) \) is external. Therefore by the \( \text{Remap} \) Well-Formed Preservation lemma, \( \hat{K}' \) is well-formed.

Next, let \( \hat{a} \) be an internal address appearing in \( \hat{K}' \). By the definition of \( \text{Remap} \), there exists some \( \hat{a}' \) appearing in \( \hat{K} \) such that \( A(\hat{a}') = \hat{a} \). By the third precondition to this lemma, \( \hat{a}' \) must be internal. Let \( \ell \) be the location associated with \( \hat{a}' \) (i.e., either \( \hat{a}' = (\text{addr } \ell \ n) \) for some \( n \) or \( \hat{a} = (\text{collective-addr } \ell) \)). Because \( \hat{K} \) is a configuration of \( P \) with initial externals \( \hat{a}_1, \ldots, \hat{a}_n \), there exists a \text{spawn} expression in \( P \) at location \( \ell \). By the third precondition to this lemma, \( \hat{a} \) has the same location.

Finally, let \( \hat{a} \) be an external address appearing in \( \hat{K}' \). By the definition of \( \text{Remap} \), there exists some \( \hat{a}' \) appearing in \( \hat{K} \) such that \( A(\hat{a}') = \hat{a} \). If \( \hat{a}' \) is internal,
then by the third precondition to this lemma, $\hat{a}' \in \text{MaxVals}((\text{Addr} \ \tau))$ for some $\tau$. Otherwise, $\hat{a}'$ is external, and by the third precondition to this lemma, $\hat{a} = \hat{a}'$. Then because $\hat{K}$ is a configuration of $P$ with initial externals $\hat{a}_1,\ldots,\hat{a}_n$, either $\hat{a} \in (\hat{a}_1,\ldots,\hat{a}_n)$ or there exists $\tau$ such that $\hat{a} \in \text{MaxVals}((\text{Addr} \ \tau))$. \hfill \Box

\section{N.3 Remap Message Type Preservation Lemma}

**Lemma** (Remap Message Type Preservation). For all $\hat{\mu}, A,$ and $M$, if

- every address appearing in $\hat{\mu}$ is a member of $\text{dom}(A)$,
- for all $\hat{a} \in \text{dom}(A)$, $\text{ActorType}(\hat{a}) = \text{ActorType}(A(\hat{a}))$, and
- for all $(\hat{a}@H,\hat{v}) \in \text{dom}(\hat{\mu})$, there exists $\tau$ such that $\varnothing, \varnothing \vdash \hat{a}@H : (\text{Addr} \ \tau), d$ and $\varnothing, \varnothing \vdash \hat{v} : \tau, d$.

then for all $\langle \hat{a}@H',\hat{v}' \rangle \in \text{dom}(\text{Remap}(\hat{\mu}, A, M))$, there exists $\tau'$ such that $\varnothing, \varnothing \vdash \hat{a}'@H' : (\text{Addr} \ \tau'), d$ and $\varnothing, \varnothing \vdash \hat{v}' : \tau', d$.

**Proof.**  Because every address appearing in $\hat{\mu}$ is a member of $\text{dom}(A)$, $\text{Remap}(\hat{\mu}, A, M)$ is defined. By the definition of Remap, there exist $\hat{a}_1,\ldots,\hat{a}_n$, $H_1,\ldots,H_n$, and $\hat{v}_1,\ldots,\hat{v}_n$ such that $\hat{\mu}' = \varnothing \odot \text{Remap}((\hat{a}_1@H'_1,\hat{v}_1), A, M) \odot \cdots \odot \text{Remap}((\hat{a}_n@H'_n,\hat{v}_n), A, M)$ and for all $i \in 1\ldots n$, $(\hat{a}_i,H_i,\hat{v}_i) \in \text{dom}(\hat{\mu})$.

By the preconditions to this lemma, there exist $\tau_1,\ldots,\tau_n$ such that for all $i \in 1\ldots n$, $\varnothing, \varnothing \vdash \hat{a}_i@H'_i : (\text{Addr} \ \tau_i), d$ and $\varnothing, \varnothing \vdash \hat{v}_i : \tau_i, d$. We also have that $\text{ActorType}(\hat{a}_i) = \text{ActorType}(A(\hat{a}_i))$, so because $\text{Remap}(\hat{a}_i@H_i,A,M) = A(\hat{a}_i)@H'$ for some $H'$, $\varnothing, \varnothing \vdash \text{Remap}(\hat{a}_i@H_i) : (\text{Addr} \ \tau_i), d$. Then for all $i \in 1\ldots n$, $\text{Remap}(\hat{v}_i,A,M)$ is just like $\hat{v}_i$, except that it replaces each address in $\hat{v}_i$ with its $A$-correspondent, and the sets of markers are changed. Because $\text{ActorType}(\hat{a}) = \text{ActorType}(A(\hat{a}))$ for all $\hat{a} \in \text{dom}(A)$, we have $\varnothing, \varnothing \vdash \text{Remap}(\hat{v}_i,A,M) : \tau_i, d$.

Then by induction on $n$ and the Message Addition Type Preservation lemma, for all $\langle \hat{a}'@H',\hat{v}' \rangle \in \text{dom}(\text{Remap}(\hat{\mu}, A, M))$, there exists $\tau'$ such that $\varnothing, \varnothing \vdash \hat{a}'@H' : (\text{Addr} \ \tau'), d$ and $\varnothing, \varnothing \vdash \hat{v}' : \tau', d$. \hfill \Box

\section{N.4 Remap Type Preservation Lemma}

**Lemma** (Remap Type Preservation). For all $\hat{K}, d, A,$ and $M$, if

- $\vdash_{\text{cfg}} \hat{K} : d$,
- every address appearing in $\hat{K}$ is a member of $\text{dom}(A)$, and
- for all $\hat{a} \in \text{dom}(A)$, $\text{ActorType}(\hat{a}) = \text{ActorType}(A(\hat{a}))$ and $\hat{a}$ is internal if and only if $A(\hat{a})$ is,

then $\vdash_{\text{cfg}} \text{Remap}(\hat{K}, A, M) : d$. 
Proof. Because every address appearing in $\hat{K}$ is a member of $\text{dom}(A)$, $\text{Remap}(\hat{K}, A, M)$ is defined. Let $\langle \hat{b} \mid \hat{H} \rangle^{\hat{\rho}} = \hat{K}$, and let $\hat{K}' = \langle \hat{b}' \mid H' \rangle^{\hat{\rho}} = \text{Remap}(\hat{K}, A, M)$. Because $\vdash_{\text{cfg}} \hat{K} : d$, we know that

- for all $\langle \hat{a} @ H', \tau \rangle \in \hat{\rho}$,
  - $\hat{a} \in \text{dom}(\hat{\rho})$,
  - $\varnothing, \varnothing \vdash \hat{a} @ H' : (\text{Addr} \ \tau), d$, and
  - $\text{Depth}(\tau) \leq d$,

- for all $\hat{a}$ appearing in $\hat{b}$ or $\hat{\mu}$, $\text{Depth}(\text{ActorType}(\hat{a})) \leq d$,

- for all $\hat{a} \in \text{dom}(\hat{b})$, $\hat{a}$ is internal, and there exists $\tau$ such that $\text{ActorType}(\hat{a}) = \tau$ and $\varnothing, \varnothing \vdash_{\text{beh}} \hat{b} : d$ for all $\hat{b} \in \hat{b}(\hat{a})$,

- for all $\hat{a}$ appearing in $\hat{K}'$, $\hat{a} \in \text{dom}(\hat{b})$ if $\hat{a}$ is internal, and

- for all $\langle \hat{a}' @ H'', \tau' \rangle \in \text{dom}(\hat{\mu})$, there exists $\tau$ such that $\varnothing, \varnothing \vdash \hat{a}' @ H' : (\text{Addr} \ \tau), d$ and $\varnothing, \varnothing \vdash \hat{\mu} : \tau, d$.

We must show that a similar set of properties holds for $\hat{K}'$.

First, let $\langle \hat{a}' @ H'', \tau' \rangle$ be a member of $\hat{\rho}'$. By the definition of $\text{Remap}$, there exists some $\langle \hat{a} @ H', \tau \rangle \in \hat{\rho}$ such that $\text{Remap}(\langle \hat{a} @ H', \tau \rangle, A, M) = \langle \hat{a}' @ H'', \tau' \rangle$. We know from the above properties that $\hat{a} \in \text{dom}(\hat{\rho})$, $\varnothing, \varnothing \vdash \hat{a} @ H' : (\text{Addr} \ \tau), d$, and $\text{Depth}(\tau) \leq d$. Then by the definition of $\text{Remap}$, $\hat{a}' \in \text{dom}(\hat{\rho}')$ and $\tau' = \tau$. Therefore, $\varnothing, \varnothing \vdash \hat{a}' @ H'' : (\text{Addr} \ \tau'), d$ and $\text{Depth}(\tau') \leq d$.

Next, let $\hat{b}'$ be an address appearing in $\hat{b}$ or $\hat{\mu}'$. By the definition of $\text{Remap}$, there exists some $\hat{a}$ appearing in $\hat{b}$ or $\hat{\mu}$ such that $A(\hat{a}) = \hat{a}'$. By the above properties, we know that $\text{Depth}(\text{ActorType}(\hat{a})) \leq d$. By the third precondition, $\text{ActorType}(\hat{a}) = \text{ActorType}(\hat{a}')$, so $\text{Depth}(\text{ActorType}(\hat{a})) \leq d$.

Next, let $\hat{a}'$ be a member of $\text{dom}(\hat{b}')$, and let $\hat{b}'$ be a member of $\hat{b}(\hat{a}')$. By the definition of $\text{Remap}$, there exists some $\hat{a} \in \text{dom}(\hat{b})$ and some $\hat{b} \in \hat{b}(\hat{a})$ such that $A(\hat{a}) = \hat{a}'$ and $\text{Remap}(\hat{a}, A, M) = \hat{b}'$. By the above properties, we know that $\hat{a}$ is internal and there exists $\tau$ such that $\text{ActorType}(\hat{a}) = \tau$ and $\varnothing, \varnothing \vdash_{\text{beh}} \hat{b} : d$. By the third precondition, $\hat{a}'$ must also be internal and $\text{ActorType}(\hat{a}) = \text{ActorType}(\hat{a}')$.

By the definition of $\text{Remap}$ and the third precondition, every address in $\hat{b}'$ has a corresponding address at that position in $\hat{b}$ with the same $\text{ActorType}$, so $\tau, \varnothing \vdash_{\text{beh}} \hat{b}' : d$.

Let $\hat{a}'$ be an internal address appearing in $\hat{K}'$. By the definition of $\text{Remap}$, there exists some $\hat{a}$ appearing in $\hat{K}$ such that $A(\hat{a}) = \hat{a}'$. By the third precondition, $\hat{a}$ is internal, and by the above properties, $\hat{a} \in \text{dom}(\hat{b})$. Then by the definition of $\text{Remap}$, $\hat{a}' \in \text{dom}(\hat{b}')$.

Finally, by the $\text{Remap}$ Message Type Preservation lemma, for all $\langle \hat{a}' @ H'', \tau' \rangle \in \text{dom}(\text{Remap}(\hat{\mu}, A, M))$, there exists $\tau'$ such that $\varnothing, \varnothing \vdash \hat{a}' @ H' : (\text{Addr} \ \tau'), d$ and $\varnothing, \varnothing \vdash \hat{\mu} : \tau', d$. Therefore, $\vdash_{\text{cfg}} \hat{K}' : d$, which completes the proof. $\square$
N.5 Core Transformation Termination Theorem

**Theorem** (Core Transformation Termination). Let the transformation \( T = \text{Canonicalize} \circ \text{Assimilate} \circ \text{Unmark} \circ \text{Split} \). Then \( T \) is termination-guaranteeing.

**Proof.** Let \( T = \text{Canonicalize} \circ \text{Assimilate} \circ \text{Unmark} \circ \text{Split} \), and let there be \( \hat{K}, s, P, \Sigma, \hat{a}_1, \ldots, \hat{a}_n, \) and \( d \) such that

- \( \hat{K} \) is a configuration of \( P \) with initial externals \( \hat{a}_1, \ldots, \hat{a}_n, \)
- \( s \) is a PSM of \( \Sigma, \)
- there is no actor handling an event in \( \hat{K}, \)
- \( \vdash_{\text{cfg}} \hat{K} : d, \) and
- there are at most two atomic actors in \( \hat{K} \) for every spawn location \( \ell. \)

We must show that \( T(\hat{K}, s) \) terminates and returns a finite set such that for all \( \langle \hat{K}', s', A, M \rangle \in T(\hat{K}, s), \)

- \( \hat{K}' \) is a configuration of \( P \) with initial externals \( \hat{a}_1, \ldots, \hat{a}_n, \)
- \( s' \) is a PSM of \( \Sigma, \)
- \( \vdash_{\text{cfg}} \hat{K}' : d, \)
- there is no actor handling an event in \( \hat{K}', \)
- \( \hat{K}' \) is Explore-bounded with maximal marker \( \max(\text{Mon}(s')) \), and
- either \( s' \) is Explore-bounded or \( \text{ShouldExplore}(s') = \text{false}. \)

By examining the definition of each of the individual transformations, it is easy to see that each one terminates, and therefore their composition terminates. It remains to show that the resulting pairs have the expected properties.

First, consider the effects of \( \text{Split} \). For every \( \langle \hat{K}'', s'' \rangle \in \text{Split}(\hat{K}, s) \), \( \hat{K}'' = \hat{K}, \) so \( \hat{K}'' \) is a configuration of \( P \) with initial externals \( \hat{a}_1, \ldots, \hat{a}_n, \) there is no actor handling an event in \( \hat{K} '', \) \( \vdash_{\text{cfg}} \hat{K}'' : d, \) and there are at most two atomic actors in \( \hat{K}'' \) for every spawn location \( \ell. \) Then for each PSM \( s'' \), either the PSM is one of the dummy PSMs of the form \( \langle \varnothing, \{ \eta \}, \text{Dummy} : \eta, (\text{define-state} (\text{Dummy})), O \rangle, \) with each of the obligations in \( O \) coming from the obligations in \( \Sigma, \) or it is just like the original PSM \( s \) except it has let output-monitored markers and less obligations. Specifically, that latter PSM monitors all of the markers that \( s \) monitors except the independent markers of \( s \) (defined in section 6.4). Each of these new PSMs satisfy the conditions to be a PSM of \( \Sigma, \) and none of them have independent markers.

Next, consider \( \text{Unmark} \). Let \( \langle \hat{K}, s'' \rangle \) be one of the results from \( \text{Split} \) (recall that the program configuration \( \hat{K} \) is the same as the original one in each pair). Then \( \text{Unmark}(\hat{K}, s'') = \{(\hat{K}'', s'', A, M)\}, \) where \( \hat{K}'' = \text{Remap}(\hat{K}, A, M), A = \text{id}, M = \)}
id|_H, and H is the set of monitored markers in s”. By the Remap Origin Preservation and Remap Type Preservation lemmas, \( \tilde{K}'' \) is a configuration of P with initial externals \( \tilde{a}_1, \ldots, \tilde{a}_n \) and \( \vdash_{\text{cfg}} \tilde{K}'' : d \). By the definition of Remap (section 6.5), Remap does not change whether there exists an actor handling a behavior, so there is no actor handling an event in \( \tilde{K}'' \). Finally, because \( A = \text{id} \), the set of actors does not change, so there are at most two atomic actors in \( \tilde{K}'' \) for every spawn location \( \ell \).

Third, we consider the effects of Assimilate. This transformation first computes \( \text{ChooseAssimilationSet}(\tilde{K}'', s') \), the set of addresses to assimilate. By the definition of \( \text{ChooseAssimilationSet} \), for every spawn location such that \( \hat{K} \) has more than one atomic actor for that location, the set contains an address for one definition of \( \text{ChooseAssimilationSet} \). Assimilate then uses Remap to assimilate each of those selected actors into a collective actor, returning \( \langle \tilde{K}'', s'', A', M' \rangle \), for some \( A' \) and \( M' \) where \( \tilde{K}''' = \text{Remap}(\tilde{K}'', A', M') \).

The function \( A' \) is the identity function, except that it maps some internal addresses to their collective correspondent. Therefore by the Remap Origin Preservation lemma, \( \tilde{K}''' \) is a configuration of P with initial externals \( \tilde{a}_1, \ldots, \tilde{a}_n \), and by the Remap Type Preservation lemma, \( \vdash_{\text{cfg}} \tilde{K}''' : d \). The behaviors do not change other than having addresses renamed, so there is no actor handling an event in \( \tilde{K}''' \). Finally, because there are at most two atomic actors in \( \tilde{K}''' \) for every spawn location \( \ell \), and because Assimilate assimilates one actor whenever there are multiple atomic actors for a given location, there is at most one atomic actor in \( \tilde{K}''' \) for every spawn location \( \ell \). That is, every atomic address \( \hat{a} \) in \( \tilde{K}''' \) is location-unique in \( \tilde{K}'' \).

Finally, we consider the effects of Canonicalize. Because every atomic address in \( \tilde{K}''' \) is location-unique, its address-correspondence function \( A'' \) renames every address of the form \( (\text{addr} \ \ell \ n) \) in \( \tilde{K}''' \) to \( (\text{addr} \ \ell \ 0) \). For the markers, we know that \( s'' \) contains only non-independent markers, and every marker appearing in \( \tilde{K}''' \) also appears in \( s' \) (by the definition of Unmark). Let \( n \) be the number of distinct markers appearing in \( s'' \); then marker-correspondence function \( M'' \) used by Canonicalize renames the markers appearing in \( \tilde{K}''' \) and \( s'' \) in a one-to-one fashion to the lowest \( n \) markers. Then \( \text{Canonicalize}(\tilde{K}'''', s'') = \langle \tilde{K}', s', A'', M'' \rangle \), where \( \tilde{K}' = \text{Remap}(\tilde{K}'''', A'', M'') \) and \( s' = M''(s'') \).

For all \( \hat{a} \in \text{dom}(A'') \), \( \hat{a} \) and \( A''(\hat{a}) \) have the same location, so \( \text{ActorType}(\hat{a}) = \text{ActorType}(A(\hat{a})) \). Furthermore, \( A'' \) maps every external address to itself. Therefore, by the Remap Origin Preservation lemma, \( \tilde{K}' \) is a configuration of P with initial externals \( \tilde{a}_1, \ldots, \tilde{a}_n \), and by the Remap Type Preservation lemma, \( \vdash_{\text{cfg}} \tilde{K}' : d \). Similarly, the renaming to create \( s' \) does not affect the “PSM of” rules, so \( s' \) is a PSM of \( \Sigma \). Also, because there is no actor handling an event in \( \tilde{K}'' \), there is also no actor handling an event in \( \tilde{K}' \).

For the program configuration, let \( \eta \) be the maximal marker monitored by \( s' \). Then we have that

- for every internal address \( \hat{a} \) appearing in \( \tilde{K}' \), there exists \( \ell \) such that either \( \hat{a} = (\text{addr} \ \ell \ 0) \) or \( \hat{a} = (\text{collective-addr} \ \ell) \), and
- \( \max(\text{Used}(\tilde{K}')) \leq \eta \).
Therefore, $\hat{K}'$ is Explore-bounded with maximal marker $\eta$.

To show the last property, if $\text{ShouldExplore}(s') = \text{false}$, then we're done. Otherwise, $\text{ShouldExplore}(s') = \text{true}$, which implies that $s'$ does not contain multiple copies of any obligation, and that it output-monitors at most $n + 1$ markers, where $n$ is the number of state arguments in $s'$. Because $s'$ is a PSM of $\Sigma$, $s'$ must be well-formed, and therefore $s'$ input-monitors at most $1$ marker. Thus in total, $s'$ monitors at most $n + 2$ markers, and therefore $s'$ is Explore-bounded. \qed

### N.6 Distinct Spawns Lemma

**Lemma** (Distinct Spawns). For all $\hat{K}_1$, $D$, $H$, and $\hat{a}$, if

- each actor in $\hat{K}_1$ has at most one behavior handling an event, and
- $\hat{a}$ is the address of the highest-priority actor (by the $\prec$ ordering) in $\hat{K}_1$ that is handling an event,

then for all executions $\hat{K}_1 \xrightarrow{\ell_1} RA \ldots \xrightarrow{\ell_n} RA \hat{K}_{n+1}$ in which the actor at $\hat{a}$ in $\hat{K}_1$ is handling an event for all $i \in \{1, 2\}$ but not in $\hat{K}_{n+1}$, if $D$ is the set of addresses for actors spawned during that transition (i.e., actors appearing in $\hat{K}_{n+1}$ but not in $\hat{K}_1$), then there are no two atomic address in $D$ with the same location $\ell$ (i.e., if $\hat{a}_i \in D$ and $\hat{a}_i = (\text{addr} \ell, n_i)$ for $i \in \{1, 2\}$, then either $\hat{a}_1 = \hat{a}_2$ or $\ell_1 \neq \ell_2$).

**Proof.** By induction on $m$, proceeding by case analysis on the transition rule enabling the transition $\hat{K}_1 \xrightarrow{\ell_1} RA \hat{K}_2$. The case for A-SPAWN is given below; the others are straightforward.

**Case: A-SPAWN**

By the induction hypothesis, there exists some $D'$ containing the addresses of the actors spawned in the remainder of the execution, $\hat{K}_2 \xrightarrow{\ell_2} RA \ldots \xrightarrow{\ell_n} RA \hat{K}_{n+1}$. Let $\ell$ be the address of the relevant spawn expression in this step, let $\hat{a}'$ be the address of the actor spawned in this step, and let $D = D' \cup \{\hat{a}'\}$. If $\hat{a}'$ is collective, then we're done.

Otherwise, by the definition of this transition rule, the spawn expression must occur outside the context of a for/fold loop. When an actor handles an event, its handler expression initially contains at most one spawn expression per syntactic location (corresponding to the event handler from the original program $P$), and the only part of the semantics that can make more copies of a spawn expression is the rule for for/fold. Therefore, the handler expressions for both the actor at $\hat{a}$ and the actor at $\hat{a}'$ in $\hat{K}_2$ do not contain a spawn expression with location $\ell$.

Because $\hat{a}$ is the address of the highest-priority actor in $\hat{K}_1$, the active actor for each of the remaining steps in the execution is either the actor at $\hat{a}$, or an actor it (transitively) spawns during the execution (including the actor at $\hat{a}'$. Because
the actors at $\hat{a}$ and $\hat{a}'$ in $\hat{K}_2$ do not contain a spawn expression with location $\ell$, it is impossible for any such actor to spawn an actor whose address has the location $\ell$. Therefore, $D'$ contains no two atomic addresses with the same location $\ell$. □

Corollary. For all $\hat{K}$, $\hat{K}'$, and $L$, if there are no actors in $\hat{K}$ handling an event and there is at most one atomic actor in $\hat{K}$ with an address of the form $(addr \ell n)$ for each $\ell$, then there are at most two atomic actors in $\hat{K}'$ with an address of the form $(addr \ell n)$ for each $\ell$.

Proof. By the definition of a summary transition, there exists some execution $\hat{K}_1 \xrightarrow{\ell_1} \hat{K}_2 \xrightarrow{\ell_2} \hat{K}_3 \xrightarrow{\ell_n} \hat{K}_{n+1}$ such that $\hat{K}_1 = \hat{K}, \hat{K}_{n+1} = \hat{K}'$, and the active actor for $\hat{\ell}_1$ is handling an event in $\hat{K}_i$ for all $i \in 2\ldots n$ but not in $\hat{K}_{n+1}$. Let $D$ be the set of addresses for actors appearing in $\hat{K}_{n+1}$ but not in $\hat{K}_1$. Then by the Distinct Spawns lemma, there are no two atomic addresses in $D$ with the same location $\ell$. Therefore, because there is at most one atomic actor in $\hat{K}$ for each location $\ell$, there are at most two atomic address in $\hat{K}'$ for each location $\ell$. □

N.7 TryTrans Termination Lemma

Lemma (TryTrans Termination). For all $\hat{K}$, $s$, $L$, $\hat{K}'$, $P$, $\Sigma$, $\hat{a}_1, \ldots, \hat{a}_n$, and $d$ if

- $\hat{K}$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$,
- $s$ is a PSM of $\Sigma$,
- there is no actor handling an event in $\hat{K}$,
- $\vdash_{cfg} \hat{K} : d$,
- $\hat{K}$ is Explore-bounded with maximal marker $\max(\text{Mon}(s))$,
- either $s$ is Explore-bounded or $\text{ShouldExplore}(s) = false$, and
- $\hat{K} \xrightarrow{L} \hat{K}'$,

then TryTrans($\hat{K}, s, L, \hat{K}'$) terminates and returns either false or some $\hat{K}''$ such that

- $\hat{K}''$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$,
- there is no actor handling an event in $\hat{K}''$,
- $\vdash_{cfg} \hat{K}'' : d$, and
- $\hat{K}''$ is Explore-bounded with maximal marker $\max(\text{Mon}(s))$, and
- $\hat{K} \xrightarrow{id, id} \hat{K}''$. 

Proof. This proof walks through the steps of the TryTrans algorithm one at a time. By the MatchingSpecSteps Termination lemma, the initial call to MatchingSpecSteps\((s, L)\) terminates and returns a finite set, and for all \((O, S) \in \text{MatchingSpecSteps}(s, L)\), \(S\) is finite. Therefore the search for a matching result terminates.

If the algorithm does not return false at the first check, then it looks for a result \(\langle \hat{K}'', s', A, M \rangle \in T_{\text{NoAcc}}(\hat{K}', s)\) such that \(s' = s\). By the Core Transformation Termination, Detect Termination, and Evict Termination theorems, \(T_{\text{NoAcc}}\) is termination-guaranteeing. We have that

- \(\hat{K}'\) is a configuration of \(P\) with initial externals \(\hat{a}_1, \ldots, \hat{a}_n\) (by the Program Origin Preservation lemma),
- \(s\) is a PSM of \(\Sigma\),
- there is no actor handling an event in \(\hat{K}'\) (by the definition of a summary transition),
- \(\vdash_{\text{cfg}} \hat{K}' : d\) (by the Abstract Type Preservation lemma), and
- there are at most two atomic actors in \(\hat{K}'\) for every location \(\ell\) (by the corollary to the Distinct Spawns lemma).

Therefore, \(T_{\text{NoAcc}}(\hat{K}', s)\) terminates and returns a finite set such that for each \(\langle \hat{K}'', s', A, M \rangle \in T_{\text{NoAcc}}(\hat{K}', s)\),

- \(\hat{K}'\) is a configuration of \(P\) with initial externals \(\hat{a}_1, \ldots, \hat{a}_n\),
- \(s'\) is a PSM of \(\Sigma\),
- there is no actor handling an event in \(\hat{K}''\),
- \(\vdash_{\text{cfg}} \hat{K}'' : d\),
- \(\hat{K}''\) is \(\text{Explore}\)-bounded with maximal marker \(\max(\text{Mon}(s'))\), and
- either \(s'\) is \(\text{Explore}\)-bounded or \(\text{ShouldExplore}(s') = \text{false}\).

If no tuple such that \(s' = s\) is found in the results, then TryTrans returns false and we’re done. Otherwise, it selects an arbitrary \(\hat{K}''\) from those results where \(s' = s\) and moves on to the next step.

Next, the algorithm tries to construct a transition from \(\hat{K}''\) that repeats \(\hat{K} \xrightarrow{L} \hat{K}'\). As described in section 8.1, the actual implementation has a straightforward way of doing this, so that check terminates. If none exists, then TryTrans returns false and we’re done; otherwise TryTrans selects an arbitrary reached configuration \(\hat{K}'''\) from one of those transitions and moves on.

Once again, we have that

- \(\hat{K}'''\) is a configuration of \(P\) with initial externals \(\hat{a}_1, \ldots, \hat{a}_n\) (by the Program Origin Preservation lemma),
• $s$ is a PSM of $\Sigma$,
• there is no actor handling an event in $\hat{K}''$,
• $\vdash_{\text{cfg}} \hat{K}'' : d$ (by the Abstract Type Preservation lemma), and
• there are at most two atomic actors in $\hat{K}''$ for every location $\ell$ (by the corollary to the Distinct Spawns lemma).

Therefore, $T_{\text{NoAcc}}(\hat{K}'', s)$ terminates and returns a finite set such that for each $\langle \hat{K}''', s'', A', M' \rangle \in T_{\text{NoAcc}}(\hat{K}'', s)$,

• $\hat{K}'''$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$,
• $s''$ is a PSM of $\Sigma$,
• there is no actor handling an event in $\hat{K}'''$,
• $\vdash_{\text{cfg}} \hat{K}''' : d$,
• $\hat{K}'''$ is Explore-bounded with maximal marker $\text{max}(\text{Mon}(s''))$, and
• either $s''$ is Explore-bounded or $\text{ShouldExplore}(s'') = \text{false}$.

If no tuple such that $s'' = s$ is found in the results, then $\text{TryTrans}$ returns false and we're done. Otherwise, it selects an arbitrary $\hat{K}'''$ from one of those results where $s'' = s$. If $\hat{K} \not\subseteq_{\text{id, id}} \hat{K}'''$, then $\text{TryTrans}$ returns false and we're done. Otherwise, $\text{TryTrans}$ returns $\hat{K}'''$, which by the above arguments satisfies the properties for this lemma.

\section{N.8 Accelerate Termination Theorem}

\textbf{Theorem (Accelerate Termination).} For all $T$, if $T$ is termination-guaranteeing, then so is $\text{Accelerate } \otimes T$.

\textbf{Proof.} Let there be $\hat{K}, s, P, \Sigma, \hat{a}_1, \ldots, \hat{a}_n, \text{ and } d$ such that

• $\hat{K}$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$,
• $s$ is a PSM of $\Sigma$,
• there is no actor handling an event in $\hat{K}$,
• $\vdash_{\text{cfg}} \hat{K} : d$, and
• there are at most two atomic actors in $\hat{K}$ for every spawn location $\ell$ (i.e., at most two actors with an address of the form $(\text{addr } \ell n)$).

Because $T$ is termination-guaranteeing, $T(\hat{K}, s)$ terminates and returns a finite set such that for all $\langle \hat{K}', s', A, M \rangle \in T(\hat{K}, s)$,

• $\hat{K}'$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$, and

\end{proof}
N.8. ACCELERATE TERMINATION THEOREM

- \( s' \) is a PSM of \( \Sigma \),
- there is no actor handling an event in \( \hat{K}' \),
- \( \vdash_{\text{cfg}} \hat{K}' : d \), and
- \( \hat{K}' \) is Explore-bounded with maximal marker \( \max(\text{Mon}(s')) \), and
- either \( s' \) is Explore-bounded or \( \text{ShouldExplore}(s') = \text{false} \).

We must show that \( \text{Accelerate}(\hat{K}', s') \) terminates and returns a finite set such that for all \( \langle \hat{K}'', s'', A', M' \rangle \in \text{Accelerate}(\hat{K}', s') \), \( \hat{K}'' \) and \( s'' \) have the same properties as those listed above for \( \hat{K}' \) and \( s' \).

Let \( \eta \) be the maximal monitored marker in \( s \), and let \( \hat{K}_{\text{curr}} \) be the “current” configuration in the algorithm (indicated by the variable \( \hat{K} \) in figure 8.4). We start by establishing a loop invariant at the top of the \textbf{while} loop in \textit{Accelerate}, made up of the following properties:

- \( \hat{K}_{\text{curr}} \) is a configuration of \( P \) with initial externals \( \hat{a}_1, \ldots, \hat{a}_n \),
- there is no actor handling an event in \( \hat{K}_{\text{curr}} \),
- \( \vdash_{\text{cfg}} \hat{K}_{\text{curr}} : d \),
- \( \hat{K}_{\text{curr}} \) is Explore-bounded with maximal marker \( \eta \), and
- for all \( \langle L, \hat{K}_{\text{next}} \rangle \) in the worklist \( W \), \( \hat{K}_{\text{curr}} \xrightarrow{L} \hat{K}_{\text{next}} \).

Before the first iteration of the loop, \( \hat{K}_{\text{curr}} = \hat{K}' \), so the first four conditions of the loop invariant are satisfied by the above conditions on \( \hat{K}' \). The last condition is satisfied by the definition of \textit{ProgStepsFor}. Additionally, the \textit{ProgStepsTermination} lemma tells us that there are finitely many transitions of the form \( \hat{K}_{\text{curr}} \xrightarrow{L} \hat{K}_{\text{next}} \), so the call to \textit{ProgStepsFor} terminates.

During an iteration of the loop, there are three possible paths: either \textit{IsAccelerateCandidate} returns false, \textit{TryTrans} returns false, or \textit{TryTrans} returns a new configuration \( \hat{K}'' \). In the first two cases, the configuration \( \hat{K}_{\text{curr}} \) does not change and no items are added to the worklist. In the final case, by the \textit{TryTrans} Termination lemma,

- \( \hat{K}'' \) is a configuration of \( P \) with initial externals \( \hat{a}_1, \ldots, \hat{a}_n \),
- there is no actor handling an event in \( \hat{K}'' \),
- \( \vdash_{\text{cfg}} \hat{K}'' : d \), and
- \( \hat{K}'' \) is Explore-bounded with maximal marker \( \eta \).

Then \( \hat{K}_{\text{curr}} \) is set to be \( \hat{K}'' \), and \( W \) is set to a list of transitions from \( \hat{K}'' \) as given from \textit{ProgStepsFor}. Thus, the loop invariant is satisfied. We have also argued already that \textit{ProgStepsFor} terminates, and it is easy to see by inspection that the rest of the loop body terminates.
To show that the loop iterates only finitely many times, let \( m \) be the maximal number of arguments appearing in any \texttt{goto} expression in \( \Sigma \), let \( H \) be the first \( n + 2 \) markers according to the \(<\) ordering on markers, and let \( D \) be the set of addresses such that for all \( \hat{a} \in D \),

- if \( \hat{a} \) is internal, then there exists a \texttt{spawn} expression in \( P \) at location \( \ell \) such that either \( \hat{a} = (\texttt{addr} \ \ell \ 0) \) or \( \hat{a} = (\texttt{collective-addr} \ \ell) \), and

- if \( \hat{a} \) is external, then either \( \hat{a} \in \{\hat{a}_1, \ldots, \hat{a}_n\} \), or there exists \( \tau \) such that \( \text{Depth}(\tau) \leq d, \text{Names}(\tau) \subseteq \text{Names}(P), \) and \( \hat{a} \in \text{MaxVals}((\texttt{Addr} \ \tau)) \).

By an argument similar to the one in the proof of the Bounded Pairs lemma, \( D \) and \( H \) are finite sets that contain all addresses and markers, respectively, appearing in all configurations \( K_{curr} \) that satisfy the conditions for the loop invariant. Then by the Bounded Program Configurations lemma, there are finitely many such configurations.

The algorithm guarantees that, at the top of the loop, \( \text{seen} \) contains every \( K_{curr} \) seen so far. Therefore, because the set of possible \( K_{curr} \) configurations is finite, the loop eventually reaches a point at which there is no \( \langle L, K_{next} \rangle \) in the worklist such that \( \text{TryTrans}(K_{curr}, s', L, K_{next}) \) returns a configuration not in \( \text{seen} \). At that point, every iteration of the loop removes one item from the worklist, and no iteration adds anything to the worklist. Therefore, the loop eventually terminates, and the algorithm returns \( \{\langle K_{curr}, s', id, id \rangle\} \) as its final result.

By the loop invariant,

- \( K_{curr} \) is a configuration of \( P \) with initial externals \( \hat{a}_1, \ldots, \hat{a}_n \),

- there is no actor handling an event in \( K_{curr} \),

- \( \vdash \text{cfg} K_{curr} : d \), and

- \( K_{curr} \) is \texttt{Explore}-bounded with maximal marker \( \eta \).

We already know that \( s' \) is a PSM of \( \Sigma \), because \( \eta \) is the largest marker that \( s' \) monitors, so we have that \( K' \) is \texttt{Explore}-bounded with maximal marker \( \text{max}(\text{Mon}(s')) \) and either \( s' \) is \texttt{Explore}-bounded or \( \text{ShouldExplore}(s') = \text{false} \). This satisfies the requirements for a termination-guaranteeing transformation, so \( \text{Accelerate} \circ T \) is termination-guaranteeing.

\[ \Box \]

### N.9 Evict Termination Theorem

**Theorem (Evict Termination).** For all \( T \), if \( T \) is termination-guaranteeing, then so is \( T \circ \text{Evict} \).

**Proof.** Let there be \( K, s, P, \Sigma, \hat{a}_1, \ldots, \hat{a}_n, \) and \( d \) such that

- \( K \) is a configuration of \( P \) with initial externals \( \hat{a}_1, \ldots, \hat{a}_n \),
N.9. EVICT TERMINATION THEOREM

- $s$ is a PSM of $\Sigma$,
- there is no actor handling an event in $\hat{K}$,
- $\vdash_{\text{cfg}} \hat{K} : d$, and
- there are at most two atomic actors in $\hat{K}$ for every spawn location $\ell$.

It is easy to see from its definition that $\text{Evict}(\hat{K}, s)$ terminates and returns the singleton set $\{\langle \hat{K}', s, A, M \rangle \}$ for some $\hat{K}'$, $A$, and $M$. We must show that $\hat{K}'$ has the same properties as listed for $\hat{K}$ above. Then because $T$ is termination-guaranteeing, we know that the result of $T(\hat{K}', s)$ will satisfy the properties necessary to show that $T \circ \text{Evict}$ is termination-guaranteeing.

By the definition of $\text{Evict}$, $\hat{K}'$ is like $\hat{K}$, except that some actors and messages from $\hat{K}$ have been removed, other messages have been added to account for messages the evicted actor might send, and the addresses of all removed actors have been renamed to external collective actors with the same $\text{ActorType}$ by $\text{Remap}$. Each new address $\hat{a}'$ for an evicted address $\hat{a}$ is chosen specifically so that $\hat{a}' \in \text{MaxVals}(\langle \text{Addr} \; \tau \rangle)$, where $\tau = \text{ActorType}(\hat{a})$. The new messages contain no addresses because an actor is evictable only if the addresses it knows about cannot receive an address in a message (according to their type). Furthermore, the correspondence functions $A$ and $M$ are both the identity function, except that $A$ maps evicted addresses to externals as described. Therefore by the $\text{Remap}$ Origin Preservation lemma, $\hat{K}'$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$.

Other than renaming the evicted addresses, the behaviors for the remaining actors in $\hat{K}'$ are unchanged, so there is no actor handling an event in $\hat{K}'$. The addresses of the remaining actors are unchanged, so there are still at most two atomic actors in $\hat{K}'$ for every spawn location $\ell$.

To show that $\hat{K}'$ is well-typed, we establish a loop invariant on the for loop in the definition of $\text{Evict}$ that $\vdash_{\text{cfg}} \hat{K}' : d$ for the current configuration $\hat{K}'$ at that point in the algorithm. Before the loop starts iterating, $\hat{K}' = \hat{K}$, so we have $\vdash_{\text{cfg}} \hat{K}' : d$. During an iteration of the loop, let $\langle \langle \hat{\beta} \; \mid \; \hat{\mu} \; \mid \; H \rangle \rangle^\rho = \hat{K}'$, and let $\langle \langle \hat{\beta} \; \mid \; \hat{\mu} \; \mid \; H' \rangle \rangle^\rho'$ be the result of the $\text{Remap}$ call in that loop. Because $\vdash_{\text{cfg}} \hat{K} : d$, we know that

1. for all $\langle \hat{\alpha} \; @ H', \tau \rangle \in \hat{\beta}$, $\hat{\alpha} \in \text{dom}(\hat{\beta})$, $\emptyset, \emptyset \vdash \hat{\alpha} \; @ H' : (\text{Addr} \; \tau), d$, and $\text{Depth}(\tau) \leq d$,
2. for all $\hat{\alpha}$ appearing in $\hat{\beta}$ and $\hat{\mu}$, $\text{Depth}(\text{ActorType}(\hat{\alpha})) \leq d$,
3. for all $\hat{\alpha} \in \text{dom}(\hat{\beta})$, $\hat{\alpha}$ is internal, and there exists $\tau$ such that $\text{ActorType}(\hat{\alpha}) = \tau$ and $\emptyset, \emptyset \vdash_{\text{beh}} \hat{b} : d$ for all $\hat{b} \in \hat{\beta}(\hat{\alpha})$,
4. for all internal addresses $\hat{\alpha}$ appearing in $\hat{K}'$, $\hat{\alpha} \in \text{dom}(\hat{\beta})$, and
5. for all $\langle \hat{\alpha}' \; @ H', \hat{\nu} \rangle \in \text{dom}(\hat{\mu})$, there exists $\tau$ such that $\emptyset, \emptyset \vdash \hat{\alpha}' \; @ H' : (\text{Addr} \; \tau), d$ and $\emptyset, \emptyset \vdash \hat{\nu} : \tau, d$. 
Let \( A' \) be the address-correspondence function given to \( \text{Remap} \) in the loop. We must show that \( \vdash_{\text{cfg}} \langle \langle \hat{b}' \mid \hat{\mu}' \mid H' \rangle \rangle_{\hat{b}'} : d \), which we show by the following arguments.

1. Let \( \langle \hat{a}'@H'', \tau' \rangle \) be a member of \( \hat{\rho}' \). If \( \langle \hat{a}'@H'', \tau' \rangle = \text{Remap}(\langle \hat{a}@H', \tau \rangle, A', id) \), then it must be the case that \( \langle \hat{a}@H'', \tau' \rangle = \langle \hat{a}@H', \tau \rangle \), because actors with corresponding receptionists are not evictable. In that case, we know that the necessary type-check properties for that receptionist hold. Otherwise, \( \langle \hat{a}@H'', \tau' \rangle \) is some valid typing for a known internal address in the evicted actor. Because \( \hat{a}' \) is internal, it must be in \( \text{dom}(\hat{\beta}) \) (and therefore in \( \text{dom}(\hat{\beta}') \)). The type \( \tau' \) is chosen to have the smallest types that still allow a valid typing of that addresses in the evicted actor's behavior, so \( \varnothing, \varnothing \vdash \hat{a}@H' : (\text{Addr} \ \tau), d \), and \( \text{Depth}(\tau') \) cannot be any larger than \( \text{Depth}(\text{ActorType}(\hat{a}')) \), so \( \text{Depth}(\tau') \leq d \).

2. Let \( \hat{a}' \) be an address appearing in \( \hat{\beta}' \) and \( \hat{\mu}' \). If \( \hat{a}' \) appears in \( \hat{\beta} \) or \( \hat{\mu} \), then we know that \( \text{Depth}(\text{ActorType}(\hat{a}')) \leq d \). Otherwise, \( \hat{a}' \) is the replacement for the evicted actor's original address \( \hat{a} \), which appears in \( \hat{\beta} \), and \( \text{ActorType}(\hat{a}') = \text{ActorType}(\hat{a}) \). Therefore, \( \text{Depth}(\text{ActorType}(\hat{a}')) \leq d \).

3. Let \( \hat{a}' \) be a member of \( \text{dom}(\hat{\beta}') \). By the definition of \( \text{Remap} \) and \( A \), \( \hat{a}' \in \text{dom}(\hat{\beta}) \), and therefore \( \hat{a}' \) is internal, and there exists \( \tau \) such that \( \text{ActorType}(\hat{a}') = \tau \) and \( \tau, \varnothing \vdash_{\text{beh}} \hat{b} : d \) for all \( \hat{b} \in \hat{\beta}(\hat{a}') \). For all \( \hat{b}' \in \hat{\beta}'(\hat{a}') \), there exists some \( \hat{b} \in \hat{\beta}(\hat{a}') \) such that \( \hat{b}' = \text{Remap}(\hat{b}, A, M) \). \( \text{Remap} \) here only replaces the evicted address with an address with the same \( \text{ActorType} \), so \( \tau, \varnothing \vdash_{\text{beh}} \hat{b}' : d \).

4. Let \( \hat{a}' \) be an internal address appearing in \( \langle \langle \hat{\beta}' \mid \hat{\mu}' \mid H' \rangle \rangle_{\hat{b}'} \). By the definition of \( A \), \( \hat{a}' \) is not the evicted actor and appears in \( \hat{K}' \) as well, so \( \hat{a} \in \text{dom}(\hat{\beta}) \).

5. Because \( \vdash_{\text{cfg}} \hat{K} : d \), we know that for all \( \langle \hat{a}@H', \hat{v} \rangle \in \text{dom}(\hat{\mu}) \), there exists \( \tau \) such that \( \varnothing, \varnothing \vdash \hat{a}@H' : (\text{Addr} \ \tau), d \) and \( \varnothing, \varnothing \vdash \hat{v} : \tau, d \). We also know that for all known internal addresses \( \hat{a}' \) of the evicted actor, \( \text{Depth}(\text{ActorType}(\hat{a}')) \leq d \), and therefore the type of the destination address for each message added by \( \text{Evict} \) is bounded by \( d \), and those messages type-check at the proper depth and type by the definition of \( \text{MaxVals} \). Therefore by the \( \text{Remap} \) Message Type Preservation lemma, for all \( \langle \hat{a}@H'', \hat{v}' \rangle \in \text{dom}(\hat{\mu}) \), there exists \( \tau' \) such that \( \varnothing, \varnothing \vdash \hat{a}@H'' : (\text{Addr} \ \tau'), d \) and \( \varnothing, \varnothing \vdash \hat{v}' : \tau', d \).

Thus, \( \vdash_{\text{cfg}} \langle \langle \hat{b}' \mid \hat{\mu}' \mid H' \rangle \rangle_{\hat{b}'} : d \). The body of the loop then sets \( \hat{K}' = \langle \langle \hat{b}' \mid \hat{\mu}' \mid H' \rangle \rangle_{\hat{b}'} \), so the body of the loop maintains the loop invariant.

Thus, the final configuration \( \hat{K}' \) returned by \( \text{Evict} \) is well-typed, i.e., \( \vdash_{\text{cfg}} \hat{K}' : d \). Therefore all of the above properties are satisfied, so \( T \odot \text{Evict} \) is termination-guaranteeing.

\( \square \)
N.10 Detect Termination Theorem

**Theorem (Detect Termination).** For all $T$, if $T$ is termination-guaranteeing, then so is $\text{Detect} \circ T$.

**Proof.** Let there be $\hat{K}, s, P, \Sigma, \hat{a}_1, \ldots, \hat{a}_n$, and $d$ such that

- $\hat{K}$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$,
- $s$ is a PSM of $\Sigma$,
- there is no actor handling an event in $\hat{K}$,
- $\vdash_{\text{cfg}} \hat{K} : d$, and
- there are at most two atomic actors in $\hat{K}$ for every spawn location $\ell$ (i.e., at most two actors with an address of the form $(\text{addr } \ell n)$).

Because $T$ is termination-guaranteeing, $T(\hat{K}, s)$ terminates and returns a finite set such that for all $\langle \hat{K}', s', A, M \rangle \in T(\hat{K}, s)$,

- $\hat{K}'$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$,
- $s'$ is a PSM of $\Sigma$,
- there is no actor handling an event in $\hat{K}'$,
- $\vdash_{\text{cfg}} \hat{K} : d$, and
- $\hat{K}'$ is Explore-bounded with maximal marker $\max(\text{Mon}(s'))$, and
- either $s'$ is Explore-bounded or $\text{ShouldExplore}(s') = \text{false}$.

We must show that Detect$(\hat{K}', s')$ terminates and returns a finite set such that for all $\langle \hat{K}'' , s'', A', M' \rangle \in \text{Detect}(\hat{K}', s')$, $\hat{K}''$ and $s''$ have the same properties as those listed above for $\hat{K}'$ and $s'$.

By inspection of the definition of Detect, it is easy to see that Detect$(\hat{K}', s')$ terminates for any $\hat{K}'$ and $s'$. Then Detect$(\hat{K}', s')$ returns either $\{\langle \hat{K}', s', id, id \rangle\}$ or $\emptyset$. In either case, the result trivially satisfies the above properties.

N.11 Label Depth Lemma

**Lemma (Label Depth).** For all $\hat{K}, \hat{K}', d$, and $\hat{l}$, if $\vdash_{\text{cfg}} \hat{K} : d$ and $\hat{K} \xrightarrow{\hat{l}} \hat{K}'$, then $\vdash \hat{l} : d$.

**Proof.** The proof is by case analysis on the rule enabling the transition $\hat{K} \xrightarrow{\hat{l}} \hat{K}'$. 

\[ \top \]
Case: A-ReceiveInternal

In this case, there exist $\tilde{a}$, $H$, and $\tilde{v}$ such that $\tilde{t} = \tilde{a} : \text{rcv-int}(H, \tilde{v})$. By the definition of this rule, there exist some $\tilde{v}, H'$, and $H''$ such that $\langle a @ H, \tilde{v}' \rangle$ is an in-flight message in $\tilde{K}$ and either $\langle \tilde{v}, H' \rangle \in \text{Markings}(\tilde{v}', H'')$ or $\tilde{v} = \tilde{v}'$. Because $\vdash_{\text{cfg}} \tilde{K} : d$, by the type-checking rules, there exists $\tau$ such that $\phi, \phi \vdash \tilde{a} @ H : (\text{Addr} \tau), d$ and $\phi, \phi \vdash \tilde{v} : \tau, d$. Additional markers do not affect type-checking, so $\phi, \phi \vdash \tilde{v} : \tau, d$. Therefore, $\vdash \tilde{t} : d$.

Case: A-ReceiveExternal

In this case, there exist $\tilde{a}$, $H$, and $\tilde{v}$ such that $\tilde{t} = \tilde{a} : \text{rcv-ext}(H, \tilde{v}, r)$. By the definition of this rule, $\langle a @ H, \tau \rangle$ is a receptionist on $\tilde{K}$ and there exist $\tilde{v}', H'$, and $H''$ such that $\phi, \phi \vdash \tilde{v}' : \tau$ and $\langle \tilde{v}, H' \rangle \in \text{Markings}(\tilde{v}', H'')$. Because $\vdash_{\text{cfg}} \tilde{K} : d$, by the type-checking rules, $\text{Depth}(\tau) \leq d$. Then by the Typed Value Depth lemma, $\phi, \phi \vdash \tilde{v}' : \tau, d$. Additional markers do not affect type-checking, so $\phi, \phi \vdash \tilde{v} : \tau, d$. Therefore, $\vdash \tilde{t} : d$.

Case: A-SendInternal

In this case, there exist $\tilde{a}$, $\tilde{a}'$, $H$, and $\tilde{v}$ such that $\tilde{t} = \tilde{a} : \text{send-int}(\tilde{a}' @ H, \tilde{v})$. By the definition of this rule, there exist $\tilde{v}', H'$, and $H''$ such that $\langle \text{send} \tilde{a}' @ H, \tilde{v}' \rangle$ is a sub-expression of a behavior of the actor at $\tilde{a}$ in $\tilde{K}$, and either $\langle \tilde{v}, H' \rangle \in \text{Markings}(\tilde{v}', H'')$ or $\tilde{v} = \tilde{v}'$. Then because $\vdash_{\text{cfg}} \tilde{K} : d$, by the type-checking rules, there exist $\Gamma, \Theta, \tau$ such that $\Gamma, \Theta \vdash \tilde{a}' @ H : (\text{Addr} \tau), d$ and $\Gamma, \Theta \vdash \tilde{v}' : \tau, d$. Furthermore, because $\tilde{a}' @ H$ and $\tilde{v}'$ are both closed values, it must be the case that $\phi, \phi \vdash \tilde{a}' @ H : (\text{Addr} \tau), d$ and $\phi, \phi \vdash \tilde{v}' : \tau, d$. Additional markers do not affect type-checking, so $\phi, \phi \vdash \tilde{v} : \tau, d$. Therefore, $\vdash \tilde{t} : d$.

Case: A-SendExternal

Similar to the previous case.

Remaining Cases

In all remaining cases, $\vdash \tilde{t} : d$ by the type-checking rule for that particular $\tilde{t}$, without further preconditions. \hfill $\square$

N.12 Label Names Lemma

Lemma (Label Names). For all $\tilde{K}, \tilde{K}'$, and $\tilde{t}$ such that $\tilde{K} \xrightarrow{\text{RA}} \tilde{K}'$, $\text{Names}(\tilde{t}) \subseteq \text{Names}(\tilde{K})$.

Proof. The proof is by case analysis on the rule enabling the transition $\tilde{K} \xrightarrow{\text{RA}} \tilde{K}'$. 
Case: A-RECEIVEINTERNAL

In this case, there exist \( \hat{a}, H, \) and \( \hat{v} \) such that \( \hat{I} = \hat{a} : \text{rcv-int}(H, \hat{v}) \). By the definition of this rule, there exist some \( \hat{v}', H', \) and \( H'' \) such that \( \langle \hat{a} @ H, \hat{v}' \rangle \) is an in-flight message in \( \hat{K} \) and either \( \langle \hat{v}, H' \rangle \in \text{Markings}(\hat{v}', H'') \) or \( \hat{v} = \hat{v}' \). Then \( \text{Names}(\hat{a}) \cup \text{Names}(\hat{v}') \subseteq \text{Names}(\hat{K}) \), and \( \text{Names}(\hat{v}) = \text{Names}(\hat{v}') \). We have \( \text{Names}(\hat{I}) = \text{Names}(\hat{a}) \cup \text{Names}(\hat{v}) \), so \( \text{Names}(\hat{I}) \subseteq \text{Names}(\hat{K}) \).

Case: A-RECEIVEEXTERNAL

In this case, there exist \( \hat{a}, H, \) and \( \hat{v} \) such that \( \hat{I} = \hat{a} : \text{rcv-ext}(H, \hat{v}, \tau) \). By the definition of this rule, there exist some \( \hat{v}, H', \) and \( H'' \) such that \( \langle \hat{a} @ H, \hat{v} \rangle \) is an in-flight message in \( \hat{K} \) and \( \langle \hat{v}, H' \rangle \in \text{Markings}(\hat{v}', H'') \). Then \( \text{Names}(\hat{a}) \cup \text{Names}(\hat{v}') \cup \text{Names}(\tau) \subseteq \text{Names}(\hat{K}) \), and \( \text{Names}(\hat{v}) = \text{Names}(\hat{v}') \). We have \( \text{Names}(\hat{I}) = \text{Names}(\hat{a}) \cup \text{Names}(\hat{v}) \cup \text{Names}(\tau) \), so \( \text{Names}(\hat{I}) \subseteq \text{Names}(\hat{K}) \).

Case: A-SENDINTERNAL

In this case, there exist \( \hat{a}, \hat{a}', H, \) and \( \hat{v} \) such that \( \hat{I} = \hat{a} : \text{send-int}(\hat{a}@H, \hat{v}) \). By the definition of this rule, there exist \( \hat{v}', H', \) and \( H'' \) such that \( \langle \text{send} \; \hat{a}@H, \hat{v}' \rangle \) is a sub-expression of a behavior of the actor at \( \hat{a} \) in \( \hat{K} \), and either \( \langle \hat{v}, H' \rangle \in \text{Markings}(\hat{v}', H'') \) or \( \hat{v} = \hat{v}' \). Then \( \text{Names}(\hat{a}) \cup \text{Names}(\hat{a}') \cup \text{Names}(\hat{v}') \subseteq \text{Names}(\hat{K}) \), and \( \text{Names}(\hat{v}) = \text{Names}(\hat{v}') \). We have \( \text{Names}(\hat{I}) = \text{Names}(\hat{a}) \cup \text{Names}(\hat{a}') \cup \text{Names}(\hat{v}) \cup \text{Names}(\tau) \), so \( \text{Names}(\hat{I}) \subseteq \text{Names}(\hat{K}) \).

Case: A-SENDEXTERNAL

Similar to the previous case.

Case: A-SPAWN

In this case, there exist \( \hat{a} \) and \( \hat{a}' \) such that \( \hat{I} = \hat{a} : \text{spawn}(\hat{a}') \). By the definition of this rule, there exist \( \ell, \tau, \hat{e}, \hat{Q} \), and \( n \) such that \( \langle \text{spawn} \; \hat{a}@H, \hat{e}, \hat{Q} \rangle \) is a sub-expression of a behavior of the actor at \( \hat{a} \) in \( \hat{K} \) and \( \text{ActorType}(\hat{a}) = \tau \). Then \( \text{Names}(\hat{a}) \cup \text{Names}(\tau) \subseteq \text{Names}(\hat{K}) \), and \( \text{Names}(\hat{a}') = \text{Names}(\tau) \). We have \( \text{Names}(\hat{I}) = \text{Names}(\hat{a}) \cup \text{Names}(\hat{a}') \), so \( \text{Names}(\hat{I}) \subseteq \text{Names}(\hat{K}) \).

Remaining Cases

In each of the remaining cases, \( \text{Names}(\hat{I}) = \text{Names}(\hat{a}) \), where \( \hat{a} \) identifies the active actor for \( \hat{I} \). By the definition of the reduction rules for these cases, there must be an actor with that address in \( \hat{K} \), so \( \text{Names}(\text{addr}) \subseteq \text{Names}(\hat{K}) \), and therefore \( \text{Names}(\hat{I}) \subseteq \text{Names}(\hat{K}) \). □
N.13 Bounded Merge Lemma

**Lemma** (Bounded Merge). For all \(\hat{\mu}, \hat{a}, H, H', \hat{v}, D, N, \) and \(d\), if for all \(\langle \hat{a}'@H'', \hat{v}' \rangle \in \text{dom}(\hat{\mu}) \cup \langle \hat{a}@H, \hat{v} \rangle\),

- \(\hat{a}' \in D\),
- \(H'' \subseteq H'\),
- \(\text{Names}(\hat{v}') \subseteq N\),
- all addresses and markers appearing in \(\hat{v}'\) are members of \(D\) and \(H'\), respectively, and
- there exists \(\tau\) such that \(\varnothing, \varnothing \vdash \hat{v}' : \tau, d\),

then for all \(\langle \hat{a}'@H'', \hat{v}' \rangle \in \text{dom}(\hat{\mu} \oplus \langle \hat{a}@H, \hat{v} \rangle)\),

- \(\hat{a}' \in D\),
- \(H'' \subseteq H'\),
- \(\text{Names}(\hat{v}') \subseteq N\),
- all addresses and markers appearing in \(\hat{v}'\) are members of \(D\) and \(H'\), respectively, and
- there exists \(\tau\) such that \(\varnothing, \varnothing \vdash \hat{v}' : \tau, d\).

**Proof.** Let \(\langle \hat{a}'@H'', \hat{v}' \rangle\) be a member of \(\text{dom}(\hat{\mu} \oplus \langle \hat{a}@H, \hat{v} \rangle)\). If \(\langle \hat{a}'@H'', \hat{v}' \rangle \in \text{dom}(\hat{\mu})\), then we're done. Otherwise, by the definition of \(\oplus\), \(\hat{a}' = \hat{a}, H'' = H\), and there exist \(\hat{v}''_1, \ldots, \hat{v}''_n\) such that \(\hat{v}' = \text{Merge}(\hat{v}, \hat{v}''_1, \ldots, \hat{v}''_n)\). The preconditions of this lemma give the proof for most of the conditions. It remains to show that

- \(\text{Names}(\hat{v}') \subseteq N\),
- all addresses and markers appearing in \(\hat{v}'\) are members of \(D\) and \(H'\), respectively, and
- there exists \(\tau\) such that \(\varnothing, \varnothing \vdash \hat{v}' : \tau, d\).

The proof is a straightforward structural induction on \(\hat{v}'\), using the definition of \(\text{Merge}\) for each case. \(\square\)

N.14 Program Origin Preservation Lemma

**Lemma** (Program Origin Preservation). For all \(\hat{K}, \hat{K}', \hat{i}, P, \) and \(\hat{a}_1, \ldots, \hat{a}_n\), if \(\hat{K}\) is a configuration of \(P\) with initial externals \(\hat{a}_1, \ldots, \hat{a}_n\) and \(\hat{K} \xrightarrow{\text{RA}} \hat{K}'\), then \(\hat{K}'\) is a configuration of \(P\) with initial externals \(\hat{a}_1, \ldots, \hat{a}_n\).
Proof. The proof is a straightforward case analysis on the rule enabling the transition $\hat{K} \xrightarrow{\hat{l}}_{RA} \hat{K}'$, using the following observations.

- $\hat{K}'$ is well-formed by the Abstract Well-Formed Preservation lemma (appendix I).
- The names on values received from the environment come exclusively from the type $\tau$ of the receptionist that receives that value.
- Every new internal address is a result of some spawn expression in $\hat{K}$.
- A-RECEIVEEXTERNAL is the only rule that introduces new external addresses, which all appear in the received value $\hat{v}$. By the definition of $MaxVals$, for every address $\hat{a}$ in $\hat{v}$, $\hat{a} \in MaxVals(\text{Addr } \tau')$ for some $\tau'$.
- Only closed values are substituted for normal variables, and only closed types are substituted for type variables, so no variable-renaming is necessary during substitution.

Corollary. For all $\hat{K}, \hat{K}', L, P, \hat{a}_1, \ldots, \hat{a}_n$, if $\hat{K}$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$ and $\hat{K} \xrightarrow{L} \hat{K}'$, then $\hat{K}'$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$.

Proof. By the definition of the summary transition relation, there exist $\hat{l}_1, \ldots, \hat{l}_n$ such that $\hat{K} \xrightarrow{\hat{l}_1, \ldots, \hat{l}_n} \hat{K}'$. By the definition of the abstract event-step relation, there exist $\hat{K}_{n+1}'$, such that $\hat{K}_{n+1}' \xrightarrow{\hat{l}_1}_{RA} \ldots \xrightarrow{\hat{l}_n}_{RA} \hat{K}'$, $\hat{K}_1' = \hat{K}$, and $\hat{K}_{n+1}' = \hat{K}'$. Then by induction on $\tau$ and the Program Origin Preservation lemma, all configurations $\hat{K}_1', \ldots, \hat{K}_{n+1}'$ are configurations of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$, and therefore so is $\hat{K}'$ by the definition of $\hat{K}_{n+1}'$.

N.15 Specification Origin Preservation Lemma

Lemma (Specification Origin Preservation). For all $s, s', \Sigma, \hat{\lambda}$, $O$, and $S$, if $s$ is a PSM of $\Sigma$ and $s \xrightarrow{\hat{l}, O, S} s'$, then every PSM $s'' \in S \cup \{s\}$ is a PSM of $\Sigma$.

Proof. By cases on the transition rule enabling the transition $s \xrightarrow{\hat{l}, O, S} s'$.

Case: P-UNMONITOREDRECEIVE

By the definition of this rule, $s' = s$ and $S = \emptyset$. We have that $s'$ is a PSM of $\Sigma$, so every PSM $s'' \in S \cup \{s\}$ is a PSM of $\Sigma$. 
Case: P-MONITORED RECEIVE

Because s is a PSM of Σ, we know that s is well-formed. Then by the Specification Well-Formed Preservation lemma (appendix K), every PSM s′ ∈ S \cup \{s′\} is well-formed.

Let \( \langle H, H', \varphi : \eta_1 \ldots \eta_n, \overline{\Phi}, O' \rangle = s \). Then by the definition of this rule, s′ = \( \langle H, H' \cup H'', \varphi' : \eta'_1 \ldots \eta'_m, \overline{\Phi}, O' \cup O'' \rangle \) for some \( H'', \varphi', \eta'_1, \ldots, \eta'_m, \) and \( O'' \). Because s is a PSM of Σ, all of the state definitions in \( \overline{\Phi} \) appear in Σ. This rule executes one of the transitions from \( \overline{\Phi} \), so \( \varphi' \) appears in \( \overline{\Phi} \) (as therefore in Σ), and there exists a goto expression in \( \overline{\Phi} \) (and therefore in Σ) such that \( \varphi \) is \( \varphi' \) with all fork-addr patterns replaced with self. Therefore, s′ is a PSM of Σ.

We must also show that every PSM in S is a PSM of Σ. Let \( s'' = \langle H'', H''', \varphi'' : \eta''_1 \ldots \eta''_n, \overline{\Phi}, O'' \rangle \) be a member of S. We have already shown that s″ is well-formed. By the definition of the PerformAll function invoked by this transition rule, s″ must be the result of evaluating either a fork effect or a fork-addr output-pattern, and then adding obligations to it with the Dist function. In either case, \( \varphi'' \) and \( \overline{\Phi} \) all appear in \( \overline{\Phi} \) (and therefore in Σ), by the definition of PerformAll and/or Extract. The number of state arguments \( \eta''_1, \ldots, \eta''_n \) is based on the goto expression in either fork effect or fork-addr pattern, so that expression appears in Σ. Finally, as with s′ above, Extract guarantees that for every obligation \( \langle \eta, po \rangle \in O'' \), there exists some \( po' \) appearing in \( \overline{\Phi} \) (and therefore in Σ) such that \( po \) is \( po' \) with all fork-addr patterns replaced with self. Therefore, s″ is a PSM of Σ.

Case: P-FREE TRANSMIT

Similar to the previous case.

Case: P-SEND

Because s is a PSM of Σ, we know that s is well-formed. Then by the Specification Well-Formed Preservation lemma (appendix K), every PSM s″ ∈ S \cup \{s′\} is well-formed.

Let \( \langle H, H', \varphi : \eta_1 \ldots \eta_n, \overline{\Phi}, O' \rangle = s \). Then by the definition of this rule, s′ = \( \langle H \cup H'', H', \varphi : \eta'_1 \ldots \eta'_m, \overline{\Phi}, O' \cup O'' \rangle \) for some \( H'', \varphi', \eta'_1, \ldots, \eta'_m, \) and \( O'' \). The current state, state arguments, and state definitions are the same as in s, and the PSM has no new obligations. Therefore, s′ is a PSM of Σ.

It remains to show that each forked PSM from this transition is also a PSM of Σ. Let \( s'' = \langle H'', H''', \varphi'' : \eta''_1 \ldots \eta''_n, \overline{\Phi}, O'' \rangle \) be a member of S. We have already shown that s″ is well-formed. By the definition of this transition rule and the output-pattern-matching rules, this PSM must have come from matching a delayed-fork-addr pattern. That is, there exists some obligation \( \langle \eta''_n, po \rangle \)
in $O'$ such that \( \text{delayed-fork-addr} \ (\text{goto} \ q'') \ (\overline{\Phi'}) \) appears in $po$, $p = 0$, and $O'' = \emptyset$. Because $s$ is a PSM of $\Sigma$, there exists an output pattern $po'$ such that $po$ is $po'$ with all \text{fork-addr} patterns replaced by \text{self}. Therefore, the pattern $(\text{delayed-fork-addr} \ (\text{goto} \ q'')) \ (\overline{\Phi'})$ appears in $\Sigma$. Then both the state $q''$ and state definitions $\Phi$ appear in $\Sigma$, and there exists a \text{goto} expression in $\Sigma$ with 0 arguments. The PSM $s''$ has no obligations, so $s''$ is a PSM of $\Sigma$.

**Corollary N.15.1.** For all $S$, $S'$, $\Sigma$, and $s$, if every PSM $s \in S$ is a PSM of $\Sigma$ and $S \rightarrow S'$, then every PSM $s' \in S'$ is a PSM of $\Sigma$.

**Proof.** By cases on the transition rule enabling the transition $S \rightarrow S'$.

**Case: S-SEND OR RECEIVE**

Let $s'$ be a member of $S'$. By the definition of this rule, there exist $s \in S$, $O$, and $S''$ such that $s \overset{\lambda, O, S''}{\rightarrow} s'$. We know that $s$ is a PSM of $\Sigma$, so $s'$ is a PSM of $\Sigma$ by the Specification Origin Preservation lemma.

**Case: S-FREE TRANSITION**

Let $s'$ be a member of $S'$. By the definition of this rule, there exists some $s \in S$ such that either $s = s'$, or there exist $O$ and $S''$ such that $s \overset{\lambda, O, S''}{\rightarrow} s'$. We know that $s$ is a PSM of $\Sigma$. In the former case, that implies that $s'$ is also a PSM of $\Sigma$. In the latter case, $s'$ is a PSM of $\Sigma$ by the Specification Origin Preservation lemma.

**Corollary N.15.2.** For all $S$, $S'$, and $\Sigma$, if every PSM $s \in S$ is a PSM of $\Sigma$ and $S \rightarrow S'$, then every PSM $s' \in S'$ is a PSM of $\Sigma$.

**Corollary.** By the definition of the $\rightarrow$ relation, $S \rightarrow \cdots \rightarrow S'$. Then by induction on the number of transitions in that sequence, the previous corollary gives us that every PSM $s' \in S'$ is a PSM of $\Sigma$.

**Corollary N.15.3.** For all $S$, $S'$, $\Sigma$, and $\lambda$, if every PSM $s \in S$ is a PSM of $\Sigma$ and $s \overset{\lambda}{\rightarrow} S'$, then every PSM $s' \in S'$ is a PSM of $\Sigma$.

**Proof.** If $\lambda = \cdot$, then the proof is immediate by the previous corollary. Otherwise, by the definition of the $\rightarrow$ relation, there exist $S''$ and $S'''$ such that $S \overset{\lambda}{\rightarrow} S'' \overset{\cdot}{\rightarrow} S''' \overset{\cdot}{\rightarrow} S'$. By the previous corollary, every PSM $s'' \in S''$ is a PSM of $\Sigma$. By corollary N.15.1, every PSM $s''' \in S'''$ is a PSM of $\Sigma$. Then by the previous corollary again, every PSM $s' \in S'$ is a PSM of $\Sigma$.

**Corollary N.15.4.** For all $S$, $S'$, $\Sigma$, $\sim\!, \sim\!'$, $L$, and $O$, if every PSM $s \in S$ is a PSM of $\Sigma$, $\sim\! \overset{L}{\rightarrow} \sim\!'$, and $S \overset{(L, O)}{\rightarrow} S'$, then every PSM $s' \in S'$ is a PSM of $\Sigma$. 


Proof. By the definition of the summary transition relation, there exist \( \hat{i}_1, \ldots, \hat{i}_n \) such that \( R \xrightarrow{\hat{i}_1, \ldots, \hat{i}_n} R' \). Then by the definition of the summary transition relation for specifications, there exist \( O'_1, \ldots, O'_n \) such that \( S \xrightarrow{O'_1} \cdots \xrightarrow{O'_n} S' \). Then by the definition of that transition relation, \( S \xrightarrow{\hat{i}_1, O'_1} \cdots \xrightarrow{\hat{i}_n, O'_n} S' \). Then by induction on the length of that sequence, the previous corollary gives us that every PSM \( s' \in S' \) is a PSM of \( \Sigma \).

N.16 Bounded Types Lemma

Lemma (Bounded Types). For all depths \( d \) and finite sets \( N \), there are finitely many types \( \tau \) such that \( \text{Depth}(\tau) \leq d \) and \( \text{Names}(\tau) \subseteq N \).

Proof. By the definition of \( \text{Depth} \), \( \text{Depth}(\tau) \leq d \) implies \( \text{LexDepth}(\tau) \leq d \). Then a straightforward structural induction on \( d \) shows that there are finitely many \( \tau \) such that \( \text{LexDepth}(\tau) \leq d \) and \( \text{Names}(\tau) \subseteq N \). It is assumed that the tags of each Variant type and the fields of each Record type are distinct.

N.17 Bounded Expressions Lemma

Lemma (Bounded Expressions). For all \( N, \ell, D, H, w, \) and \( d \), if \( N, \ell, D, \) and \( H \) are all finite, then there are finitely many \( \hat{e} \) such that

- \( \text{Names}(\hat{e}) \subseteq N \),
- for every spawn expression in \( \hat{e} \) associated with some location \( \ell \), \( \ell \in W \),
- every address appearing in \( \hat{e} \) is a member of \( D \),
- every marker appearing in \( \hat{e} \) is a member of \( H \),
- \( \hat{e} \) is width-bounded by \( w \), and
- \( \Gamma, \Theta \vdash \hat{e} : \tau, d \) for some \( \Gamma, \Theta, \) and \( \tau \).

Proof. The proof is by induction on \( d \), showing that there are finitely many expressions of each kind (e.g., spawn expressions, case expressions, etc.) for each \( d \). Most cases are straightforward. The case for the case expression uses an argument for the new type environment similar to the one used in, for example, the Bounded State Definitions lemma below. The proofs for more difficult cases are given here.

Spawn Expressions

By the type rule for a spawn expression, \( \Gamma, \Theta \vdash (\text{spawn}^\ell \tau \hat{e} Q_1 \cdots Q_n) : \tau', d \) implies that \( \text{Depth}(\tau) \leq d - 1 \). To satisfy the above criteria, it must also be the
case that $Names(\tau) \subseteq N$, so by the Bounded Types lemma, there are finitely many such $\tau$.

For a spawn expression as described above, the initialization expression $\hat{e}$ must be such that

- $Names(\hat{e}) \subseteq N$,
- for every spawn expression in $\hat{e}$ associated with some location $\ell$, $\ell \in W$,
- every address appearing in $\hat{e}$ is a member of $D$,
- every marker appearing in $\hat{e}$ is a member of $H$,
- $\hat{e}$ is width-bounded by $w$, and
- $\Gamma, \Theta \vdash \hat{e} : \tau, d - 1$ for some $\Gamma, \Theta$, and $\tau$.

By the induction hypothesis, there are only finitely many such $\hat{e}$.

Also, each state definition $\hat{Q}$ in the spawn expression must be such that

- $Names(\hat{Q}) \subseteq N$,
- for every spawn expression in $\hat{Q}$ associated with some location $\ell$, $\ell \in W$,
- every address appearing in $\hat{Q}$ is a member of $D$,
- every marker appearing in $\hat{Q}$ is a member of $H$,
- $\hat{Q}$ is width-bounded by $w$, and
- $\Gamma, \Theta \vdash state \hat{Q} : d - 1$ for some $\Gamma$ and $\Theta$.

By the Bounded State Definitions lemma, there exist finitely many such state definitions. Because each spawn expression must be width-bounded by $w$, each expression has no more than $w$ state definitions.

Finally, the location $\ell$ on the expression must come from the finite set $W$.

We have shown that each of the components of a spawn expression as described above must come from a finite set, so there are only finitely many such spawn expressions.

**List Expressions**

By the induction hypothesis, there are finitely many $\hat{o}v$ such that

- $Names(\hat{o}v) \subseteq N$,
- for every spawn expression in $\hat{o}v$ associated with some location $\ell$, $\ell \in W$,
- every address appearing in $\hat{o}v$ is a member of $D$,
- every marker appearing in $\hat{o}v$ is a member of $H$,
- $\hat{o}v$ is width-bounded by $w$, and
• \(\Gamma, \Theta \vdash \tau : \tau \cdot d - 1\) for some \(\Gamma, \Theta,\) and \(\tau\).

Every list expression satisfying the criteria for this lemma contain only such values. There are finitely many subsets of a finite set, so therefore there are finitely many such list expressions.

The case for dict expressions is similar. \(\square\)

\section*{N.18 Bounded Timeout Clauses Lemma}

\textbf{Lemma} (Bounded Timeout Clauses). For all \(N, \ell, D, H, w,\) and \(d,\) if \(N, \ell, D,\) and \(H\) are all finite, then there are finitely many \(\hat{tc}\) such that

- \(\text{Names}(\hat{tc}) \subseteq N,\)
- for every spawn expression in \(\hat{tc}\) associated with some location \(\ell, \ell \in W,\)
- every address appearing in \(\hat{tc}\) is a member of \(D,\)
- every marker appearing in \(\hat{tc}\) is a member of \(H,\)
- \(\hat{tc}\) is width-bounded by \(w,\) and
- \(\Gamma, \Theta \vdash \hat{tc} : \tau \cdot d\) for some \(\Gamma\) and \(\Theta.\)

\textit{Proof.} For \(\tau \in \{\text{Nat}, \bot\},\) by the Bounded Expressions lemma, there exist finitely many expressions \(\hat{e}\) such that

- \(\text{Names}(\hat{e}) \subseteq N,\)
- for every spawn expression in \(\hat{e}\) associated with some location \(\ell, \ell \in W,\)
- every address appearing in \(\hat{e}\) is a member of \(D,\)
- every marker appearing in \(\hat{e}\) is a member of \(H,\)
- \(\hat{e}\) is width-bounded by \(w,\) and
- \(\Gamma, \Theta \vdash \hat{e} : \tau \cdot d\) for some \(\Gamma\) and \(\Theta.\)

By the type rule for a timeout clause, both the timeout \(\hat{ov}\) itself and the handler expression \(\hat{e}\) must come from one of those sets. Therefore, there are finitely many of each, and therefore there are finitely many timeout clauses of the form \([\text{timeout} \hat{ov} \hat{e}]\) meeting the above criteria. The only other timeout clause is \textbf{no-timeout}, so there are finitely many timeout clauses \(\hat{tc}\) meeting the above criteria. \(\square\)
N.19  Bounded State Definitions Lemma

Lemma (Bounded State Definitions). For all $N, \ell, D, H, w, and d$, if $N, \ell, D, and H$ are all finite, then there are finitely many $\hat{Q}$ such that

- $\text{Name}(\hat{Q}) \subseteq N$,
- for every $\text{spawn}$ expression in $\hat{Q}$ associated with some location $\ell, \ell \in W$,
- every address appearing in $\hat{Q}$ is a member of $D$,
- every marker appearing in $\hat{Q}$ is a member of $H$,
- $\hat{Q}$ is width-bounded by $w$, and
- $\Gamma, \Theta \vdash \text{state } \hat{Q} : d$ for some $\Gamma$ and $\Theta$.

Proof. Every state definition that is width-bounded by $w$ has at most $w$ state arguments, and therefore $w$ declared types for those arguments. By the type rule for state definitions, $\Gamma, \Theta \vdash \text{state } \hat{Q} : d$ for some $\Gamma$ and $\Theta$ implies that $\text{Depth}(\tau) \leq d$ for every state-argument type $\tau$. Then by the Bounded Types lemma, there are finitely many such types such that $\text{Name}(\tau) \subseteq N$. Also, for all $\hat{Q}$ such that $\text{Name}(\hat{Q}) \subseteq N$, the state name $q$, argument variables $x_1, \ldots, x_n$, and message variable $x'$ must be members of $N$.

By the Bounded Timeout Clauses lemma, there are finitely many timeout clauses $\hat{tc}$ such that

- $\text{Name}(\hat{tc}) \subseteq N$,
- for every $\text{spawn}$ expression in $\hat{tc}$ associated with some location $\ell, \ell \in W$,
- every address appearing in $\hat{tc}$ is a member of $D$,
- every marker appearing in $\hat{tc}$ is a member of $H$,
- $\hat{tc}$ is width-bounded by $w$, and
- $\Gamma, \Theta \vdash \text{tc } \hat{tc} : d$ for some $\Gamma$ and $\Theta$.

The set of state definitions as described in the statement of this lemma contains only timeout clauses from this set, so those clauses are selected from a finite set.

Finally, by the Bounded Expressions lemma, there are finitely many $\hat{e}$ such that

- $\text{Name}(\hat{e}) \subseteq N$,
- for every $\text{spawn}$ expression in $\hat{e}$ associated with some location $\ell, \ell \in W$,
- every address appearing in $\hat{e}$ is a member of $D$,
- every marker appearing in $\hat{e}$ is a member of $H$,
- $\hat{e}$ is width-bounded by $w$, and
• $\Gamma, \Theta \vdash e : \bot, d$ for some $\Gamma$ and $\Theta$.

The message-handler expression in a state definition as described in the statement of this lemma must come from this set. Therefore, there are finitely many choices of message-handler expression.

We have shown that all of the components of the state definitions described must come from finite sets. Therefore, there are finitely many state definitions $\hat{Q}$ satisfying the given conditions.

\[ \square \]

### N.20 Bounded Behaviors Lemma

**Lemma** (Bounded Behaviors). For all $N, \ell, n, D, H, w, \text{ and } d$, if $N, \ell, D, \text{ and } H$ are all finite, then there are finitely many $\hat{b}$ such that

- $\text{Name}(\hat{b}) \subseteq N$,
- for every spawn expression in $\hat{b}$ associated with some location $\ell, \ell \in W$,
- $\hat{b}$ has at most $n$ many state definitions,
- every address appearing in $\hat{b}$ is a member of $D$,
- every marker appearing in $\hat{b}$ is a member of $H$,
- $\hat{b}$ is width-bounded by $w$, and
- $\tau, \Gamma \vdash_{beh} \hat{b} : d$ for some $\tau$ and $\Gamma$.

**Proof.** By the Bounded State Definitions lemma, there are only finitely state definitions $\hat{Q}$ meeting the above criteria such that

- $\text{Name}(\hat{Q}) \subseteq N$,
- for every spawn expression in $\hat{Q}$ associated with some location $\ell, \ell \in W$,
- every address appearing in $\hat{Q}$ is a member of $D$,
- every marker appearing in $\hat{Q}$ is a member of $H$,
- $\hat{Q}$ is width-bounded by $w$, and
- $\Gamma, \Theta \vdash_{state} \hat{Q} : d$ for some $\Gamma$ and $\Theta$.

Every behavior $\hat{b}$ as described above contains at most $n$ state definitions from this finite set.

By the Bounded Expressions lemma, there exist finitely many $\hat{e}$ such that

- $\text{Name}(\hat{e}) \subseteq N$,
- for every spawn expression in $\hat{e}$ associated with some location $\ell, \ell \in W$,
- every address appearing in $\hat{e}$ is a member of $D$,
N.21  BOUNDED PROGRAM CONFIGURATIONS LEMMA

Lemma (Bounded Program Configurations). For all \( P, D, H, \) and \( d \), if \( D \) and \( H \) are finite, then there are finitely many \( \bar{K} \) such that

- \( \bar{K} \) is a configuration of \( P \),
- every address appearing in \( \bar{K} \) is a member of \( D \),
- every marker appearing in \( \bar{K} \) is a member of \( H \), and
- \( \vdash \text{cfg} \bar{K} : d \).

Proof. Let \( W \) be the set of locations associated with `spawn` expressions in \( P \), and let \( N = \text{Names}(\text{prog}) \); both sets are finite. Let \( w \) be the width bound on \( P \) as given by the definition of a “configuration of \( P \)”, and let \( n \) be the maximal number of state definitions in any `spawn` expression in \( P \). We will show that there are only finitely many choices for each of the components of \( \bar{K} \).

First, for the actor-behavior map \( \bar{\beta} \), the domain of \( \bar{\beta} \) must be a subset of \( D \), so there are finitely many such domains. Then for every address \( \bar{a} \in D \), by the Bounded Behaviors lemma, there are finitely many behaviors \( \bar{b} \) such that
• Names($\hat{b}$) $\subseteq$ N,
• for every spawn expression in $\hat{b}$ associated with some location $\ell$, $\ell \in W$,
• $\hat{b}$ has at most $n$ many state definitions,
• every address appearing in $\hat{b}$ is a member of $D$,
• every marker appearing in $\hat{b}$ is a member of $H$,
• $\hat{b}$ is width-bounded by $w$, and 
• $\tau, \Gamma \vdash_{beh} \hat{b} : d$ for some $\tau$ and $\Gamma$.

The range of each $\hat{b}$ that can appear in a configuration $\hat{K}$ as described above must be a subset of the powerset of those behaviors, by the type rule for a program configuration. Therefore, there are only finitely many such maps.

Next, for the message maps $\hat{\mu}$, the domain of each one must be a subset of the set of all $\langle \hat{a}@H', \hat{v} \rangle$ pairs such that
• $\hat{a} \in D$,
• $H' \subseteq H$,
• Names($\hat{v}$) $\subseteq$ N,
• every address appearing in $\hat{v}$ is a member of $D$,
• every marker appearing in $\hat{v}$ is a member of $H$,
• $\hat{v}$ is width-bounded by $w$, and 
• $\Gamma, \Theta \vdash \hat{v} : \tau, d$ for some $\Gamma$, $\Theta$, and $\tau$.

No value contains a spawn expression. Therefore, by the Bounded Expressions lemma, there are only finitely many values $\hat{v}$ meeting the above criteria. Therefore, there are finitely many domains for such a map. The range of each map must be a subset of $\{\text{single, many}\}$, so there are also finitely many ranges. Therefore, there are only finitely many such maps.

Each configuration’s used-marker component $H'$ must be a subset of the finite set $H$, so there are only finitely many such sets.

Finally, each receptionist set $\hat{\rho}$ contains only pairs $\langle \hat{a}@H', \tau \rangle$ such that
• $\hat{a} \in D$,
• $H' \subseteq H$,
• Names($\tau$) $\subseteq$ N, and 
• Depth($\tau$) $\leq d$.

By the Bounded Types lemma, there are finitely many such $\tau$. Therefore, there are finitely many such receptionist sets.

Because each configuration $\hat{K}$ must be constructed of components from the above finite sets, there are only finitely many such configurations.
N.22 Bounded PSMs Lemma

Lemma (Bounded PSMs). For all $\Sigma$, there are finitely many $s$ such that $s$ is a PSM of $\Sigma$ and is Explore-bounded.

Proof. Let $n$ be the maximum number of arguments found in any goto expression in $\Sigma$. Any Explore-bounded PSM of $\Sigma$ has at most $n+2$ input-monitored and output-monitored markers, so there are at most finitely many input-monitored and output-monitored sets matching those criteria.

For any PSM of $\Sigma$, its current state must appear in $\Sigma$ or be the special Dummy state, and it must have no more than $\max(n, 2)$ state arguments. Because a PSM of $\Sigma$ must be well-formed, those arguments must come from its input-monitored and output-monitored sets, and if that PSM is Explore-bounded, then those sets are subsets of some finite set of no more than $n+2$ markers.

The state definitions of a PSM of $\Sigma$ must all appear in $\Sigma$ or be the special Dummy state definition, so there are finitely many state definitions to choose from. The state definitions are assumed to be distinct, so the total number of state definitions in the PSM is bounded by $\max(1, m)$, where $m$ is the number of state definitions appearing in $\Sigma$.

For the obligations, again because a PSM of $\Sigma$ must be well-formed, the destination marker $\eta$ of any obligation in a PSM of $\Sigma$ must come from its input-monitored and output-monitored sets. Again, if that PSM is also Explore-bounded, then those sets are subsets of some finite set of markers. For a PSM of $\Sigma$, each obligation pattern $po$ must be such that there exists some $po'$ in $\Sigma$ such that $po$ is $po'$ with all fork-addr patterns replaced with self. Therefore, there are only finitely many such patterns. Finally, an Explore-bounded PSM must not contain any duplicate obligations. Therefore, because the possible destination markers and patterns for each obligation must come from two finite sets, and because there can be no duplicates, there are finitely many obligation multisets $O$ that an Explore-bounded PSM of $\Sigma$ can contain.

We have shown that each of the components of an Explore-bounded PSM of $\Sigma$ must come from a finite set. Therefore, there are only finitely many such PSMs.

\[\square\]

N.23 Bounded Pairs Lemma

Lemma (Bounded Pairs). For all $P$, $\Sigma$, $d$, and $\hat{a}_1, \ldots, \hat{a}_n$, there are finitely many $\langle \hat{K}, s \rangle$ pairs such that

- $\hat{K}$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$,
- $s$ is a PSM of $\Sigma$,
- $\vdash_{ctg} \hat{K} : d$,
- $\hat{K}$ is Explore-bounded with maximal marker $\max(\text{Mon}(s))$, and
- $s$ is Explore-bounded.
Proof. We will show that the Bounded Program Configurations and Bounded PSMs lemmas bound the possible program configurations and PSMs, respectively.

To use the Bounded Program Configurations lemma, we must come up with finite sets containing all addresses and markers in each program configuration. Let $D$ be the set of addresses such that for all $\hat{a} \in D$,

- if $\hat{a}$ is internal, then there exists a spawn expression in $P$ at location $\ell$ such that either $\hat{a} = (\text{addr} \, \ell \, 0)$ or $\hat{a} = (\text{collective-addr} \, \ell)$, and
- if $\hat{a}$ is external, then either $\hat{a} \in \{\hat{a}_1, \ldots, \hat{a}_n\}$, or there exists $\tau$ such that $\text{Depth}(\tau) \leq d$, $\text{Names}(\tau) \subseteq \text{Names}(P)$, and $\hat{a} \in \text{MaxVals}(\text{Addr} \, \tau)$.

For any configuration $\hat{K}$ such that $\vdash_{\text{cfg}} \hat{K} : d$ and every address $\hat{a}$ appearing in $\hat{K}$, the type rules ensure that $\text{Depth}(\text{ActorType}(\hat{a})) \leq d$. By the definition of $\text{MaxVals}$ and $\text{ExtRepAddr}$, $\hat{a} \in \text{MaxVals}(\text{Addr} \, \tau)$ implies $\text{ActorType}(\hat{a}) = \tau$. Thus, it is easy to see that for any Explore-bounded $\hat{K}$ that is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$ and such that $\vdash_{\text{cfg}} \hat{K} : d$, $D$ contains all addresses appearing in $\hat{K}$.

The program $P$ has a finite number of spawn expressions, so $D$ contains finitely many internal addresses. By the Bounded Types lemma, there are finitely many types $\tau$ such that $\text{Names}(\tau) \subseteq \text{Names}(P)$ and $\text{Depth}(\tau) \leq d$. By the Finite Maximal Values lemma, $\text{MaxVals}(\tau)$ is finite for each such $\tau$, so $D$ contains finitely many external addresses. Therefore, $D$ is finite.

To create the marker set, let $m$ be the maximal number of arguments appearing in any goto expression in $\Sigma$, and let $H$ be the first $n + 2$ markers according to the $<$ ordering on markers. For every Explore-bounded PSM $s$ of $\Sigma$, $\text{Mon}(s) \subseteq H$. Furthermore, because $\hat{K}$ is Explore-bounded with maximal marker $\text{max}(\text{Mon}(s))$ and $s$ is a PSM of $\Sigma$, $H$ contains every marker appearing in $\hat{K}$.

Using the above sets $D$ and $H$, by the Bounded Program Configurations lemma, there are finitely many $\hat{K}$ such that

- $\hat{K}$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$,
- $\vdash_{\text{cfg}} \hat{K} : d$,
- every address appearing in $\hat{K}$ is a member of $D$, and
- every marker appearing in $\hat{K}$ is a member of $H$.

Then by the Bounded PSMs lemma, there are finitely many $s$ such that $s$ is a PSM of $\Sigma$ and is Explore-bounded. Therefore, there are finitely many $\langle \hat{K}, s \rangle$ pairs such that

- $\hat{K}$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$,
- $s$ is a PSM of $\Sigma$,
- $\vdash_{\text{cfg}} \hat{K} : d$, and
• $\hat{K}$ is Explore-bounded with maximal marker $\text{max}(\text{Mon}(s))$, and

• $s$ is Explore-bounded.

N.24 Unique Context Lemma

Lemma (Unique Context). For all $\hat{e}$, if $\hat{e}$ is not a value, then there exist unique $\hat{E}$ and $\hat{e}'$ such that $\hat{e} = \hat{E}[\hat{e}']$ and $\hat{e}'$ has one of the following forms.

- $(\text{spawn} \, \tau \, \hat{e} \, \hat{Q})$
- $(\text{goto} \, q \, \hat{v}_1 \ldots \hat{v}_n)\$
- $(\text{send} \, \hat{v}_1 \, \hat{v}_2)\$
- $(\text{begin} \, \hat{v} \, \hat{e}_1 \ldots \hat{e}_n)\$
- $(:: \, \hat{v} \, r)\$
- $(\text{case} \, \hat{v}(\text{t x} \, \hat{e}))\$
- $(\text{unfold} \, \hat{v})\$

Proof. By structural induction on $\hat{e}$. The proof proceeds by cases on the shape of $\hat{e}$. An example case is given below; others are similar.

Case: $\hat{e} = (\text{begin} \, \hat{e}_1' \ldots \hat{e}_n')$

The syntax of abstract CSA guarantees that $n \geq 1$. If $\hat{e}_1''$ is a value, then let $\hat{E} = []$ and $\hat{e}' = \hat{e}$. Because $\hat{e}_1''$ is a value, it cannot contain any of the acceptable forms for $\hat{e}'$ listed above, so $\hat{E}$ is unique.

Otherwise, $\hat{e}_1''$ is not a value. By the induction hypothesis, there exist unique $\hat{E}'$ and $\hat{e}'$ such that $\hat{e}_1' = \hat{E}'[\hat{e}']$ and $\hat{e}'$ has one of the acceptable forms listed above. Let $\hat{E} = (\text{begin} \, \hat{E}' \, \hat{e}_2'' \ldots \hat{e}_n'')$; then $\hat{e} = \hat{E}[\hat{e}']$. The decomposition of $\hat{e}$ into $\hat{E}$ and $\hat{e}'$ is unique because a $\text{begin}$ expression cannot be decomposed into any other context form other than a $\text{begin}$ context or $[]$, $\hat{E} = []$ would not yield an acceptable $\hat{e}'$ of one of the forms above (because $\hat{e}_1''$ is not a value), and the decomposition of $\hat{e}_1''$ into $\hat{E}'$ and $\hat{e}'$ is unique.

N.25 Finite Maximal Values Lemma

Lemma (Finite Maximal Values). For all $\tau$, $\text{MaxVals}(\tau)$ is finite.

Proof. By structural induction on $\tau$. The proof proceeds by cases on the shape of $\tau$. 
Case: \( \tau = (\text{List } \tau') \)
By definition, \(|\text{MaxVals}(\tau)| = 1\).

Case: \( \tau = (\text{Dict } \tau' \tau'') \)
By definition, \(|\text{MaxVals}(\tau)| = 1\).

Case: \( \tau = (\text{Addr } \tau') \)
There are at most one \( \ell \) and one \( n \) such that \( \text{ExtRepAddr}(\tau') = (\text{addr } \ell \ n) \). Therefore, by the definition of \( \text{MaxVals} \), \(|\text{MaxVals}(\tau)| \leq 1\).

Case: \( \tau = (\text{Variant } [t_1 \tau'_{1,1} \ldots \tau'_{1,m}] \ldots [t_n \tau'_{n,1} \ldots \tau'_{n,m}]) \)
For each \( i \in 1 \ldots n \) and each \( j \in 1 \ldots m \), by the induction hypothesis, \( \text{MaxVals}(\tau'_{i,j}) \) is finite. There are finitely many variant tags \( t_1, \ldots, t_n \) and finitely many component types \( \tau'_{i,1}, \ldots, \tau'_{i,m} \) for each tag, so \( \text{MaxVals}(\tau) \) is finite.

Case: \( \tau = (\text{Record } [r_1 \tau'_1] \ldots [r_n \tau'_n]) \)
Similar to the Variant case.

Case: \( \tau = (\text{rec } X \ (\text{Addr } \tau')) \)
Let \( \tau'' = (\text{Addr } \tau')[X \leftarrow \tau] = (\text{Addr } \tau'[X \leftarrow \tau]) \). As in the Addr case above, there are at most one \( \ell \) and one \( n \) such that \( \text{ExtRepAddr}(\tau'') = (\text{addr } \ell \ n) \). Therefore, by the definition of \( \text{MaxVals} \), \(|\text{MaxVals}(\tau)| \leq 1\).

Case: \( \tau = \text{Nat} \)
By definition, \(|\text{MaxVals}(\tau)| = 1\).

Case: \( \tau = \text{String} \)
By definition, \(|\text{MaxVals}(\tau)| = 1\).

N.26 Finite Triggers Lemma

Lemma (Finite Triggers). For all \( \hat{K} \) such that \( \vdash_{\text{cfg}} \hat{K} \) and no actor in \( \hat{K} \) is handling an event, there are only finitely many transition labels \( \hat{l} \) such that there exists some \( \hat{K}' \) such that \( \hat{K} \underset{\hat{l}}{\longrightarrow_{\text{RA}}} \hat{K}' \).
N.27  Finite Loop Allocations Lemma

This lemma states that while an actor is evaluating a for/fold loop, only finitely many new addresses are allocated, and no new markers are allocated.

**Lemma** (Finite Loop Allocations). For all $\hat{K}_1$, $D$, $H$, and $\hat{a}$, if

- each in $\hat{K}_1$ has at most one behavior handling an event,
- $D$ and $H$ contain all addresses and markers, respectively, appearing in $\hat{K}_1$, and
- the actor at $\hat{a}$ is the highest-priority actor in $\hat{K}_1$ and has a behavior $\langle \overline{Q}, C \mid E \mid \{ \text{for/fold} \ [x \hat{\epsilon}] \ [x' \hat{\epsilon}'] \hat{\epsilon} \} \rangle$,

then there exists a finite set $D'$ such that for all executions $\hat{K}_1 \xrightarrow{\ell_1} \hat{K}_2 \xrightarrow{\ell_2} \ldots \xrightarrow{\ell_m} \hat{K}_{m+1}$, if for all $i \in 1\ldots m$, the actor at $\hat{a}$ in $\hat{K}_i$ has a behavior $\langle \overline{Q}, C \mid E \mid \{ \text{for/fold} \ [x \hat{\epsilon}''] \ [x' \hat{\epsilon}'] \hat{\epsilon}'' \} \rangle$ for some $\hat{\epsilon}''$, then $D \cup D'$ and $H$ contain all addresses and markers, respectively, appearing in the configurations and labels in that execution.

**Proof.** Let $\{\ell_1, \ldots, \ell_n\}$ be the set of syntactic locations associated with spawn expressions in $\hat{K}_1$. Then let $\hat{a}'_i = (\text{collective-addr} \ \ell_i)$ for all $i \in 1\ldots n$, and let

Proof. Because no actor in $\hat{K}$ is handling an event, every such label $\hat{I}$ must be of the form $\hat{a} : \text{rcv-ext}(H, \hat{\nu}, \tau)$, $\hat{a} : \text{rcv-int}(H, \hat{\nu})$, or $\hat{a} : \text{timeout}$. We show that there are only finitely many labels for each type such that there is a transition $\hat{K} \xrightarrow{\ell} \hat{K}'$.

For a transition labeled with $\hat{a} : \text{rcv-ext}(H, \hat{\nu}, \tau)$, by the A-RECEIVEEXTERNAL rule there must exist a receptionist $\langle \hat{a} \hat{\tau}, H, \tau \rangle$ on $\hat{K}$. Furthermore, there must exist $H'$, $H''$, and $\hat{\nu}'$ such that $H' = \text{Used}(\hat{K})$, $\hat{\nu}' \in \text{MaxVals}(\tau)$, and $\text{Mark}(\hat{\nu}', H') = \langle \hat{\nu}, H'' \rangle$. There are finitely many receptionists on $\hat{K}$, and by the Finite Maximal Values lemma, there are finitely many $\hat{\nu}'$ for each $\tau$. Therefore, there are finitely many labels of the form $\hat{a} : \text{rcv-ext}(H, \hat{\nu}, \tau)$ such that there is a transition $\hat{K} \xrightarrow{\ell} \hat{K}'$.

For a transition labeled with $\hat{a} : \text{rcv-int}(H, \hat{\nu})$, by the A-RECEIVEINTERNAL rule there exist $H'$, $H''$, and $\hat{\nu}'$ such that $H' = \text{Used}(\hat{K})$, $\langle \hat{a} \hat{\tau}, H, \tau \rangle$ is an in-flight message in $\hat{K}$, and either or $\text{Mark}(\hat{\nu}', H') = \langle \hat{\nu}, H'' \rangle$ or $\hat{\nu}' = \hat{\nu}$. There are only finitely many in-flight messages in $\hat{K}$, so there are finitely many such labels such that there is a transition $\hat{K} \xrightarrow{\ell} \hat{K}'$.

Finally, for a transition labeled with $\hat{a} : \text{timeout}$, by the A-TIMEOUT rule there exists some actor in $\hat{K}$ with address $\hat{a}$. There are finitely many actors in $\hat{K}$, so there are finitely many such labels such that there is a transition $\hat{K} \xrightarrow{\ell} \hat{K}'$. \qed
APPENDIX N. TERMINATION PROOF FOR MODELCHECK

\[ D' = \{ \vec{a}_1', \ldots, \vec{a}_n' \}. \] We will show that no markers are allocated while evaluating the loop, and that the only new addresses allocated come from \( D' \).

Let \( \vec{K}_1 \xrightarrow{\ell_1} \ldots \xrightarrow{\ell_m} \vec{K}_{m+1} \) be an execution such that for all \( i \in 1 \ldots m \), there exists \( \vec{e}'' \) such that the actor at \( \vec{a} \) in \( \vec{K}_i \) has a behavior \( \vec{Q}, C[\vec{E}[\text{(for/fold } x \vec{e}'' \text{)} [x' \vec{v} \text{]} \vec{e}']] \). We will show by induction on \( i \) that

- \( D \cup D' \) and \( H \) contain all addresses and markers, respectively, appearing in \( \vec{K}_i \) and \( \vec{l}_{i-1} \),

- each actor in \( \vec{K}_i \) has at most one behavior handling an event, and

- for all actors with a priority higher than that of \( \vec{a} \) that are handling an event in \( \vec{K}_i \), the actor is collective and there exist \( \vec{Q}' \) and \( \vec{e}''' \) such that the actor has a behavior \( \vec{Q}', (\text{in-loop } \vec{e}''') \).

In the base case, there is no label \( \vec{l}_{i-1} \). We know from the preconditions of this lemma that \( D \) and \( H \) contain the addresses and markers appearing in \( \vec{K}_1 \). There is no actor in \( \vec{K}_1 \) with a priority higher than that of \( \vec{a} \) that are handling an event.

In the inductive case, let \( \vec{l}_i \xrightarrow{\ell} \vec{l}_{i+1} \) be a step of that execution, and let \( \vec{a}'' \) be the address of the active actor in \( \vec{l}_i \). The proof is by cases on the rule enabling the transition.

**A-SendInternal** If \( \vec{a}'' \neq \vec{a} \), then the actor at \( \vec{a}'' \) must have a higher priority than the actor at \( \vec{a} \). Therefore by the induction hypothesis, that actor has a behavior \( \vec{Q}', (\text{in-loop } \vec{e}''') \). We also know from the induction hypothesis that that must be the only behavior for \( \vec{a}'' \) handling an event in \( \vec{K}_i \). Therefore the send expression is inside an in-loop context. Otherwise, if \( \vec{a}'' \neq \vec{a} \), then the send expression must be inside the for/fold loop. In either case, by the definition of this rule, the transition does not mark the sent message, so the transition creates no new addresses or markers.

This rule simply replaces the behavior of the actor at \( \vec{a}'' \) with a new behavior, so each actor in \( \vec{K}_{i+1} \) has at most one behavior handling an event. This rule does not create any new actors, so the condition on actors with a higher priority than \( \vec{a} \) still holds.

**A-SendExternal** Similar to the previous case

**A-Spawn** If \( \vec{a}'' \neq \vec{a} \), then the actor at \( \vec{a}'' \) must have a higher priority than the actor at \( \vec{a} \). Therefore by the induction hypothesis, \( \vec{a}'' \) is a collective actor. Otherwise, we know that the spawn expression must be inside the for/fold loop in the behavior for the actor at \( \vec{a} \). Then in either case, the new actor has an address \( \vec{a}''' = \text{(collective-addr } \ell) \), where the location \( \ell \) is the location associated with
the spawn expression. Therefore, $\hat{a}''' \in D'$. No other addresses or markers are created in this transition.

As in the A-SendINTERNAL case, this rule replaces the behavior of the actor at $\hat{a}''$ with a new behavior, so the actor at $\hat{a}''$ and all other previously existing actors from $\hat{K}_i$ have at most one behavior handling an event in $\hat{K}_{i+1}$. The address $\hat{a}'''$ must have a higher priority than $\hat{a}''$ (all children have a higher priority than their parents, because the spawn expression is lexically nested inside the spawn expression that created the parent), so because $\hat{a}''$ is the active actor for this transition, there must be no behavior for $\hat{a}'''$ in $\hat{K}_i$ that is handling an event. Therefore, the actor at $\hat{a}'''$ in $\hat{K}_{i+1}$ has just one behavior handling an event.

Also by the induction hypothesis, if $\hat{a}''' \neq \hat{a}$, then the spawn expression is inside an in-loop context. Therefore in either case, there exist $\vec{Q}'$ and $\hat{e}'''$ such that the new behavior of that actor is $\langle \vec{Q}', (\text{in-loop } \hat{e}''' \rangle$. The induction hypothesis gives us that the property holds for all other actors with a higher priority than $\hat{a}$ in $\hat{K}_i$, so it holds for all such actors in $\hat{K}_{i+1}$, as well. (Note the transition does not change the parent context for the actor at $\hat{a}''$).

**A-GOTO** This rule never creates new actors or markers. The rule merely replaces the behavior of the active actor with a new behavior, so every actor in $\hat{K}_{i+1}$ has at most one behavior handling an event. The actor at $\hat{a}'''$ is not handling an event in $\hat{K}_{i+1}$, and no other actor is modified, so the condition on actors with a higher priority than $\hat{a}$ still holds.

**A-GOTO and A-FUNC** This rule never creates new actors or markers. The rule merely replaces the behavior of the active actor with a new behavior, so every actor in $\hat{K}_{i+1}$ has at most one behavior handling an event. The rule does not change the parent context of the actor at $\hat{a}'''$ and no other actor is modified, so the condition on actors with a higher priority than $\hat{a}$ still holds.

**A-ReceiveINTERNAL, A-ReceiveEXTERNAL, and A-Timeout** Each of these rules has an actor start handling an event. The $\rightarrow_{RA}$ relation does not allow such a step while at least one other actor is already handling an event, so none of these rules enable the transition and therefore we can ignore them.

**N.28 Finite Allocations Lemma**

This lemma states that while an actor is handling an event, only finitely many new addresses and markers are allocated.

**Lemma** (Finite Allocations). For all $\hat{K}_i$, $D$, $H$, and $\hat{a}$, if

- $\vdash_{cfg} \hat{K}_i$,
- each actor in $\hat{K}_i$ has at most one behavior handling an event,
APPENDIX N. TERMINATION PROOF FOR MODELCHECK

- $D$ and $H$ contain all addresses and markers, respectively, appearing in $K_1$, and
- $\hat{a}$ is the address of the highest-priority actor (by the $<$ ordering) in $K_1$ that is handling an event,

then there exist finite sets $D'$ and $H'$ such that for all executions $\overrightarrow{K_1}$ $\overrightarrow{K_m}$ in which the actor at $\hat{a}$ in $K_i$ is handling an event for all $i \in 1 \ldots m$ but not in $K_{m+1}$, $D \cup D'$ and $H \cup H'$ contain all addresses and markers, respectively, appearing in the configurations and labels in that execution.

Proof. By the preconditions to this lemma, the actor at $\hat{a}$ has exactly one behavior handling an event; let $\hat{b} = \langle \hat{Q}, C[\hat{e}] \rangle$ be that behavior.

Define the control expressions of CSA as the spawn, goto, send, begin, :, case, unfold, and primitive-operation expressions. The proof is by induction on the number of control expressions in $\hat{e}$.

By the type-checking rules, there exist some $\Gamma$ and $\Theta$ such that $\Gamma, \Theta \vdash \hat{e} : \perp$. No value can have type $\perp$, so $\hat{e}$ is not a value. Then by the Unique Context lemma, there exist unique $\hat{E}$ and $\hat{e}'$ such that $\hat{e} = \hat{E}[\hat{e}']$ and $\hat{e}'$ has one of the forms listed in that lemma. The proof proceeds by cases on the shape of $\hat{e}'$.

Case: $\hat{e}' = (\text{goto } \varphi \, \hat{v}_1 \ldots \hat{v}_n)$

Because the $\overrightarrow{RA}$ transition relation transitions only the highest priority actor, the only transition possible from $K_1$ is a transition by the A-GOTO rule. The rule uniquely determines a transition $\hat{K}_1 = \hat{K}_2$, such that the actor at $\hat{a}$ is not handling an event in $\hat{K}_2$. Let $D' = \emptyset$ and $H' = \emptyset$. The only execution that finishes evaluating that actor's handler expression is the one-step sequence $\hat{K}_1 \overrightarrow{RA} \hat{K}_2$.

The transition does not create any new addresses or markers, so we're done.

Case: $\hat{e}' = (\text{send } \hat{v}_1 \hat{v}_2)$

Because $\hat{K}_1$ is well-typed, by the type rules there exist some $\Gamma$, $\Theta$, and $\tau$ such that $\Gamma, \Theta \vdash \hat{v}_1 : (\text{Addr } \tau)$ and $\Gamma, \Theta \vdash \hat{v}_2 : \tau$. Because $\hat{v}_1$ is a value with an Addr type, it must be a marked address $\hat{a}'@H''$. Then a step with either the A-SENDINTERNAL or A-SENDEXTERNAL rule is possible, depending on whether $\hat{a}'$ is internal or external. In either case, if $C = []$ and $\hat{E}$ is not of the form $E'[\text{for/fold } \{x \in \{x' \in \{x'' \in \hat{E}'[\hat{e}']\} \} \} \hat{e}']$, then the Mark function marks every address in $\hat{v}_2$ with a fresh marker. A value can contain only finitely many addresses, so there are finitely many new markers; let $H''$ be that finite set of new markers. Otherwise, if the transition does not mark the value, then let $H'' = \emptyset$.

Let $\hat{K}_1 \overrightarrow{RA} \hat{K}_2$ be the described transition ($\hat{K}_1$ and $\hat{K}_2$ are both uniquely determined by both of those rules and the Mark function used for a $\overrightarrow{RA}$ transition). The behaviors for all actors in $\hat{K}_2$ are the same as those for $\hat{K}_1$, except that the behavior $\hat{b}$ for the actor at $\hat{a}$ is replaced with $\hat{b}' = \langle \hat{Q}, C[\hat{E}((\text{variant Unit})\})]$. 


The set $D$ contains all addresses appearing in $\hat{I}_1$ and $\hat{K}_2$ (the send transition created no new addresses), and $H \cup H''$ contains all markers appearing in $\hat{I}_1$ and $\hat{K}_2$. By the induction hypothesis, there exist some finite sets $D'$ and $H'''$ such that for all executions $\hat{K}_2 \xrightarrow{I_2} \hat{K} \xrightarrow{I_m} \hat{K}_{m+1}$ in which the actor at $\hat{a}$ in $\hat{K}_1$ is handling an event for all $i \in [2, m]$ but not in $\hat{K}_{m+1}$, $D \cup D'$ and $H \cup H'' \cup H'''$ contain all addresses and markers, respectively, appearing in the configurations and labels in that execution.

Let $H' = H'' \cup H'''$, $H'$ is finite because both $H''$ and $H'''$ are. Let $\hat{K}_1 \xrightarrow{I_1} \hat{K}_2 \xrightarrow{I_2} \hat{K} \xrightarrow{I_m} \hat{K}_{m+1}$ be an execution with $\hat{K}_1 = \hat{K}_2$ in which the actor at $\hat{a}$ in $\hat{K}_1$ is handling an event for all $i \in [1, m]$ but not in $\hat{K}_{m+1}$. Because a $\xrightarrow{RA}$ transition can only transition the highest priority running actor (if there is one), the first transition in that execution must be $\hat{K}_1 \xrightarrow{I_1} \hat{K}_2$. Then $D \cup D'$ and $H \cup H'$ contain all addresses and markers, respectively, appearing in the configurations and labels in that execution.

**Case: $\hat{c}' = (\text{spawn}^\ell \tau \hat{c}'' Q)$**

Because the $\xrightarrow{RA}$ transition relation transitions only the highest priority actor, the only transition possible from $\hat{K}_1$ is a transition by the A-SPAWN rule. The $\xrightarrow{RA}$ relation imposes a deterministic scheme for picking new addresses, so the transition $\hat{K}_1 \xrightarrow{I_1} \hat{K}_2$ is uniquely determined. Let $\hat{a}'$ be the address of the newly spawned actor; then $D \cup \{\hat{a}'\}$ contains all addresses appearing in $I_1$ and $\hat{K}_2$, and $H$ contains all markers appearing in $\hat{I}_1$ and $\hat{K}_2$ (this step creates no new markers).

The new actor at $\hat{a}'$ has a single behavior $\hat{b}' = (\overline{Q[\text{self} \leftarrow \hat{a}' @ \emptyset], \hat{c}''[\text{self} \leftarrow \hat{a}' @ \emptyset]})$. The spawn expression that created the actor at $\hat{a}$ must have contained a spawn expression with location $\ell$ (i.e., the spawn expression that eventually led to this one). Therefore $\ell$ has a higher priority than the location of that previous, outer spawn expression, and therefore $\hat{a}'$ has a higher priority than $\hat{a}$ (and therefore identifies the highest priority actor in $\hat{K}_2$). Then by the induction hypothesis, there exist finite sets $D''$ and $H''$ such that for all executions $\hat{K}_2 \xrightarrow{I_2} \hat{K} \xrightarrow{I_p} \hat{K}_{p+1}$ in which the actor at $\hat{a}'$ in $\hat{K}_i$ is handling an event for all $i \in [2, p]$ but not in $\hat{K}_{p+1}$, $D \cup \{\hat{a}'\} \cup D''$ and $H \cup H''$ contain all addresses and markers, respectively, appearing in the configurations and labels in that execution.

To show that there are only finitely many final configurations $\hat{K}_{p+1}$ for such an execution $\hat{K}_2 \xrightarrow{I_2} \hat{K} \xrightarrow{I_p} \hat{K}_{p+1}$, let $N = \text{Names}(\hat{K}_1)$ and let $W$ be the set of all spawn locations appearing in $\hat{K}_1$. Both sets must be finite. No transition rule adds new names or locations, so for each such $\hat{K}_{p+1}$, $\text{Names}(\hat{K}_{p+1}) \subseteq N$ and the locations appearing in $\hat{K}_{p+1}$ are a subset of $W$. Also, by the Abstract Type Preservation lemma, $\vdash \text{cfg} \hat{K}_{p+1} : d$. We have already shown above that $D \cup \{\hat{a}'\} \cup...$
Case: $\bar{e}' = (\text{for/fold } [x \ ar{E}'''] [x' \ ar{o}'''] \ ar{e}')$

By the Finite Loop Allocations lemma, there exists a finite set $D''$ such that for all executions $\hat{K}_1 \xrightarrow{l_1} RA \ldots \xrightarrow{l_p} RA \hat{K}_{p+1}$ representing this actor evaluating the loop, $D \cup D''$ and $H$ contain all addresses and markers, respectively, appearing in the configurations and labels in that execution. Similar to the spawn case above, each such execution introduces no new names or locations, and the Abstract Type Preservation lemma tells us that $\Gamma_{cfg} \hat{K}_{p+1} : d$ for each such $\hat{K}_{p+1}$. Therefore, by the Bounded Program Configurations lemma, there are finitely many such $\hat{K}_{p+1}$.

By the induction hypothesis, for each such $\hat{K}_{p+1}$ there exist $D'''$ and $H'$ such that for all executions $\hat{K}_{p+1} \xrightarrow{l_{p+1}} RA \ldots \xrightarrow{l_{p+q}} RA \hat{K}_{p+q+1}$ in which the actor at $\bar{a}$ in $\hat{K}_i$ is handling an event for all $i \in p + 1 \ldots p + q$ but not in $\hat{K}_{p+q+1}$, $D \cup \{\bar{a}'\} \cup D'' \cup D'''$ and $H \cup H' \cup H'''$ contain all addresses and markers, respectively, appearing in the configurations and labels in that execution.

Let $D'$ be the union of all such $D''$ and $D'''$ along with $\{\bar{a}'\}$, and let $H'$ be the union of all such $H''$ and $H'''$. Because there are only finitely many $\hat{K}_{p+1}$, $D'$ and $H'$ are finite. Let $\hat{K}_1 \xrightarrow{l_1} RA \ldots \xrightarrow{l_m} RA \hat{K}_{m+1}$ be an execution with $\hat{K}_1 = \hat{K}_1$ in which the actor at $\bar{a}$ in $\hat{K}_i$ is handling an event for all $i \in 1 \ldots m$ but not $\hat{K}_{m+1}$. That execution must be constructed by appending executions $\hat{K}_1 \xrightarrow{l_1} RA \hat{K}_2, \hat{K}_2 \xrightarrow{l_2} RA \ldots \xrightarrow{l_p} RA \hat{K}_{p+1},$ and $\hat{K}_{p+1} \xrightarrow{l_{p+1}} RA \ldots \xrightarrow{l_{p+q}} RA \hat{K}_{p+q+1}$ as described above. Then $D \cup D'$ and $H \cup H'$ contain all addresses and markers, respectively, appearing in the configurations and labels in that execution.
N.29 ProgSteps Termination Lemma

This lemma shows that the (possibly infinite) set of event-step transitions from a well-typed abstract program configuration can be summarized by finitely many summary transitions.

Lemma (ProgSteps Termination). For all $\hat{K}$, $P$, $\hat{a}_1, \ldots, \hat{a}_n$, and $d$, such that

- $\hat{K}$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$,
- $\vdash_{\text{cfg}} \hat{K} : d$,
- no actor in $\hat{K}$ is handling an event,

there exists a finite set $\{\langle L_1, \hat{K}_1' \rangle, \ldots, \langle L_m, \hat{K}_m' \rangle\}$ such that for all event-step transitions $\hat{K} \xrightarrow{\hat{l}_1, \ldots, \hat{l}_p} \hat{K}'$ from $\hat{K}$, there exists $i \in 1\ldots m$ such that $\hat{K} \xrightarrow{L_i} \hat{K}_i'$ summarizes $\hat{K} \xrightarrow{\hat{l}_1, \ldots, \hat{l}_p} \hat{K}'$.

Proof. By the Finite Triggers lemma, there are only finitely many transition labels $\hat{l}_1$ such that there exists some $\hat{K}_2$ such that $\hat{K} \xrightarrow{\hat{l}_1} \hat{K}_2$. Furthermore, because no actor in $\hat{K}$ is handling an event, $\hat{l}_1$ must be a handler-start label.

Let $D$ and $H$ be the sets of abstract addresses and markers, respectively, appearing in $\hat{K}$ and $\hat{l}_1$. Both $\hat{K}$ and $\hat{l}_1$ contain finitely many addresses and markers, so both sets are finite. The label $\hat{l}_1$ includes any new addresses or markers created in a transition $\hat{K} \xrightarrow{\hat{l}_1} \hat{K}_2$, so $D$ and $H$ include all addresses and markers appearing in any such $\hat{K}_2$. By the Program Origin Preservation lemma, $\hat{K}_2$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$ for any such $\hat{K}_2$. By the Abstract Type Preservation lemma, $\vdash_{\text{cfg}} \hat{K}_2 : d$ for any such $\hat{K}_2$. Therefore, by the Bounded Program Configurations lemma, there are finitely many next configurations $\hat{K}_2$ for each $\hat{l}_1$.

For each such $\hat{K}_2$, because $\hat{l}$ is a handler label, the rule enabling $\hat{K} \xrightarrow{\hat{l}_1} \hat{K}_2$ must be one of A-RECEIVEINTERNAL, A-RECEIVEEXTERNAL, or A-TIMEOUT. By case analysis on those rules, there is exactly one actor handling an event in $\hat{K}_2$ (with just one of its behaviors handling an event); let $\hat{a}$ be the address of that actor. Then by the Finite Allocations lemma, there exist finite sets $D'$ and $H'$ such that for all executions $\hat{K}_2 \xrightarrow{\hat{l}_2} \hat{K}_3 \ldots \xrightarrow{\hat{l}_p} \hat{K}_{p+1}$ in which the actor at $\hat{a}$ in $\hat{K}_i$ is handling an event for all $i \in 2\ldots p$, the sets $D \cup D'$ and $H \cup H'$ contain all addresses and markers, respectively, appearing in the configurations and labels in that execution.
For each such execution $\cdots \xrightarrow{\hat{l}_p} \cdots \xrightarrow{\hat{l}_1} \hat{\bar{K}}_2$, $\xrightarrow{\text{RA}} \hat{\bar{K}}_{p+1}$, by the Abstract Type Preservation lemma (appendix H), and $\hat{\bar{K}}_i$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$ by the Program Origin Preservation lemma. By the Label Depth lemma, $\vdash_{\text{cfg}} \hat{\bar{K}}_i : d$ for all $i \in 2 \ldots p + 1$. By the Label Names lemma, $\text{Names}(\hat{l}_i) \subseteq \text{Names}(\hat{\bar{K}}_i)$, and therefore $\text{Names}(\hat{l}_i) \subseteq \text{Names}(P)$. Then by the type-checking rules, for all sent or received values $\hat{v}$ appearing in each $\hat{l}_i$, there exists $\tau$ such that $\phi, \phi \vdash \hat{v} : \tau, d$.

We must show that a finite set of abstract message maps can summarize the outputs from all of those executions. For each $\hat{\bar{K}}_2$, let $M$ be the set of abstract message maps $\hat{\mu}$ such that for all $\langle \hat{a}'@H'', \hat{v} \rangle \in \text{dom}(\hat{\mu})$,

- $\hat{a}' \in D \cup D'$,
- $H'' \subseteq H \cup H'$,
- $\text{Names}(\hat{v}) \subseteq \text{Names}(P)$,
- all addresses and markers appearing in $\hat{v}$ are members of $D \cup D'$ and $H \cup H'$, respectively, and
- there exists $\tau$ such that $\phi, \phi \vdash \hat{v} : \tau, d$.

Therefore, by the Bounded Expressions lemma, the set of possible $\hat{v}$'s is finite, so the domain of each $\hat{\mu}$ is a subset of a finite set. The range of each $\hat{\mu}$ is bounded by $\{\text{single, many}\}$, so $M$ is finite.

Let $\cdots \xrightarrow{\hat{l}_2} \cdots \xrightarrow{\hat{l}_1} \hat{\bar{K}}_2$, by one of the above executions from $\hat{\bar{K}}_2$, and let $\hat{a}'_1 : \text{send-ext}(\hat{a}'_{1@H'_{1}}, \hat{v}'_{1}), \ldots, \hat{a}'_m : \text{send-ext}(\hat{a}'_{m@H'_{m}}, \hat{v}'_{m})$ be the send-ext labels in $\hat{l}_2, \ldots, \hat{l}_p$. Then let $\hat{\mu}' = \phi \oplus (\hat{a}'_{1@H'_{1}}, \hat{v}'_{1}) \ldots \oplus (\hat{a}'_{m@H'_{m}}, \hat{v}'_{m})$. By the Bounded Merge lemma, for all $\langle \hat{a}''@H'', \hat{v}'' \rangle \in \text{dom}(\hat{\mu}')$,

- $\hat{a}'' \in D \cup D'$,
- $H'' \subseteq H \cup H'$,
- $\text{Names}(\hat{v}) \subseteq \text{Names}(P)$,
- all addresses and markers appearing in $\hat{v}$ are members of $D \cup D'$ and $H \cup H'$, respectively, and
- there exists $\tau$ such that $\phi, \phi \vdash \hat{v} : \tau, d$.

Therefore, $\hat{\mu}'$ is a member of $M$.

For a given $\hat{\bar{K}}_1$, we have shown that

- there are finitely many transition labels $\hat{l}_1$ that label a transition from $\hat{\bar{K}}_1$.
- for each such $\hat{l}_1$, there are finitely many $\hat{\bar{K}}_2$ such that $\hat{\bar{K}}_2 \xrightarrow{\text{RA}} \hat{\bar{K}}_1$. 

\section*{APPENDIX N. TERMINATION PROOF FOR MODELCHECK}

For each such execution $\cdots \xrightarrow{\hat{l}_2} \cdots \xrightarrow{\hat{l}_1} \hat{\bar{K}}_2$, $\xrightarrow{\text{RA}} \hat{\bar{K}}_{p+1}$, $\vdash_{\text{cfg}} \hat{\bar{K}}_i : d$ for all $i \in 2 \ldots p + 1$.
Proof. Straightforward structural induction on the derivation of \( \bar{K}_2 \xrightarrow{\ell_2} \bar{K}_1 \xrightarrow{\ell_1} \bar{K}_p \xrightarrow{\ell_p} \bar{K}_{p+1} \) in which at least one actor is handling an event in the configurations \( \bar{K}_2, \ldots, \bar{K}_p \).

Therefore, the cross product of all such \( \bar{K}_1 \) and all such \( \bar{\mu} \) is finite. A summary transition label \( L \) is defined as such a pair, so let \( \{L_1, \ldots, L_k\} \) be that set.

To bound the set of final reached configurations, we know that for each such \( \bar{K}_2 \), there are finite sets \( D \cup D' \) and \( H \cup H' \) such that for all executions \( \bar{K}_2 \xrightarrow{\ell_2} \bar{K}_1 \xrightarrow{\ell_1} \bar{K}_p \xrightarrow{\ell_p} \bar{K}_{p+1} \) in which at least one actor is handling an event in the configurations \( \bar{K}_2, \ldots, \bar{K}_p \), \( D \cup D' \) and \( H \cup H' \) contain all addresses and markers appearing in every configuration and label in that execution. Because there are finitely many such \( \bar{K}_2 \), there are finitely many such sets \( D \cup D' \) and \( H \cup H' \).

Let \( D'' \) and \( H'' \) be the union of all of those sets; then \( D'' \) and \( H'' \) are both finite and contain all addresses and markers appearing in all \( \bar{K}_{p+1} \) as described above. By the Program Origin Preservation and Abstract Type Preservation lemmas, each such \( \bar{K}_{p+1} \) is a configuration of \( P \) with initial externals \( \hat{a}_1, \ldots, \hat{a}_n \) and \( \hat{r} \). Then by the Bounded Program Configurations lemma, there are only finitely many such \( \bar{K}_{p+1} \). Let \( \{ \bar{K}_1', \ldots, \bar{K}_j' \} \) be that set.

Let \( \{\langle L_1, \bar{K}_1'\rangle, \ldots, \langle L_m, \bar{K}_m'\rangle\} \) be the cross product of \( \{L_1, \ldots, L_k\} \) and \( \{\bar{K}_1', \ldots, \bar{K}_j'\} \); the set is finite by construction. Then for all event-step transitions \( \bar{K} \xrightarrow{L_i} \bar{K}' \) from \( \bar{K} \), by the above arguments there exists \( i \in 1 \ldots m \) such that \( \bar{K} \xrightarrow{L_i} \bar{K}_i' \) summarizes \( \bar{K} \xrightarrow{\ell_1 \ldots \ell_p} \bar{K}' \).

\( \square \)

### N.30 Finite Output Matches Lemma

**Lemma** (Finite Output Matches). For all \( \bar{v} \) and \( po \), there exist finitely many \( H \) and \( S \) such that \( \bar{v} \sim po \triangleright H, S \), and for all \( H \) and \( S \) such that \( \bar{v} \sim po \triangleright H, S \) are finite.

**Proof.** Straightforward structural induction on the derivation of \( \bar{v} \sim po \triangleright H, S \). \( \square \)

### N.31 PsmSimluateOutput Termination Lemma

**Lemma** (PsmSimluateOutput Termination). For all \( s, \hat{a}, H, \bar{v}, \) and \( m \), \( \text{PsmSimluateOutput}(s, \hat{a}, H, \bar{v}, m) \) terminates and returns a finite set, and for all \( \langle O, s', S \rangle \in \text{PsmSimluateOutput}(s, \hat{a}, H, \bar{v}, m) \), \( S \) is finite.

**Proof.** As the algorithm runs, if any of the first three conditionals are triggered, then the algorithm immediately terminates and the results trivially satisfy the expected condition.

Next, if \( m = \text{many} \), the algorithm finds all single-step transitions of that PSM labeled with \( \bullet \) that generate an obligation that matches the given output. There
APPENDIX N. TERMINATION PROOF FOR MODELCHECK

is at most one such transition per state transition in the PSM’s current state, so there are finitely many such transitions. The algorithm then returns one result per matching transition, with each result containing no forked PSMs.

Otherwise, if $m = \text{single}$, the algorithm tries three different approaches successively (listed in figure 7.12 in chapter 7). First, it tries to match the output against each existing obligation $\langle \eta'', po \rangle$ in the PSM, of which there can be only finitely many. The inner loop of that section loops over all possible $H'''$ and $S$ such that $\hat{v} \sim po \triangleright H'', S$. By the Finite Output Matches lemma, there are finitely many such pairs, so the inner loop terminates. Each iteration of the loop adds a single result with the set of PSMs $S$, which is finite by the Finite Output Matches lemma. If the results are non-empty after the outer loop terminates, then the algorithm returns the accumulated results.

The second approach is similar to the approach for matching many outputs, by trying all transitions labeled with $\bullet$ from $s$ that generate a matching obligation. Again, there are only finitely many such transitions. The inner loop is similar to the inner loop of the previous approach, so it terminates and adds a finite number of results to the result set, with each one containing a finite number of PSMs. Once the outer loop terminates, the algorithm returns the accumulated results if the set is non-empty.

Finally, the third approach is similar to the second one, but it additionally allows transitions of the PSM that change its state or state arguments. It terminates and returns results with the expected conditions for the same reasons as the previous approach, except that this one returns the results at the end even if none were found.

\[ \square \]

**N.32 SimulateOutput Termination Lemma**

**Lemma (SimulateOutput Termination).** For all $S$, $\hat{a}$, $H$, $\hat{v}$, and $m$, if $S$ is a finite set, then $\text{SimulateOutput}(S, \hat{a}, H, \hat{v}, m)$ terminates and returns a finite set, and for all $\langle O, S' \rangle \in \text{SimulateOutput}(S, \hat{a}, H, \hat{v}, m)$, $S'$ is finite.

**Proof.** The proof is by induction on $|S|$. In the base case where $S = \emptyset$, the algorithm terminates immediately with a result of $\langle \emptyset, S \rangle$, which satisfies the expected postcondition.

In the inductive case, the algorithm picks an arbitrary element $s$ of $S$ and calls $\text{PsmSimluateOutput}(s, \hat{a}, H, \hat{v}, m)$. By the $\text{PsmSimluateOutput}$ Termination lemma, the call to $\text{PsmSimluateOutput}$ terminates and returns a finite set of results such that for each result $\langle O, s', S' \rangle$, $S'$ is finite. The algorithm then loops over each result and recurs on $\text{SimulateOutput}$ with $S - \{s\}$. By the induction hypothesis, that call terminates and returns a finite set of results such that for each result $\langle O', S''' \rangle$, $S'''$ is finite. Therefore the inner and outer loops both iterate a finite number of times, and for each result $\langle O \cup O', S'''' \cup \{s'\} \cup S' \rangle$, the set $S'''' \cup \{s'\} \cup S'$ is finite. Thus, the algorithm terminates, returns a finite set of results such that the specification configuration in each result is finite. $\square$
N.33  SimulateOutputs Termination Lemma

Lemma (SimulateOutputs Termination). For all $S$ and $\hat{\mu}$, if $S$ is a finite set, then $\text{SimulateOutputs}(S, \hat{\mu})$ terminates and returns a finite set, and for all $\langle O, S' \rangle \in \text{SimulateOutputs}(S, \hat{\mu})$, $S'$ is finite.

Proof. All message maps $\hat{\mu}$ have a domain of finitely many elements, so the proof is by induction on $|\text{dom}(\hat{\mu})|$. In the base case where $\text{dom}(\hat{\mu}) = \emptyset$, then the algorithm terminates immediately with a result of $\langle \emptyset, S \rangle$. We know that $S$ is finite, so that satisfies the expected postcondition.

In the inductive case, the algorithm picks an arbitrary element $\langle \hat{a}@H, \hat{v} \rangle$ from $\text{dom}(\hat{\mu})$, determines the abstract quantity $m$ of that element in $\hat{\mu}$, and calls $\text{SimulateOutput}(S, \hat{a}, H, \hat{v}, m)$. By the SimulateOutput Termination lemma, $\text{SimulateOutput}(S, \hat{a}, H, \hat{v}, m)$ terminates and returns a finite set, and for all $\langle O, S' \rangle \in \text{SimulateOutput}(S, \hat{a}, H, \hat{v}, m)$, $S'$ is finite. The algorithm then loops over those results from $\text{SimulateOutput}$ and recurs on $\text{SimulateOutputs}$ with an abstract message map with $\langle \hat{a}@H, \hat{v} \rangle$ removed. By the induction hypothesis, that recursive call terminates and returns a finite set of results, and for every result $\langle O', S'' \rangle$, $S''$ is finite. Thus the inner loop executes a finite number of times, and each iteration adds a single result to $\text{results}$ with a finite set $S''$. Therefore the eventual return value $\text{results}$ is a finite set such that for each element $\langle O'', S''' \rangle$, $S'''$ is finite.

N.34  MatchingSpecSteps Termination Lemma

Lemma (MatchingSpecSteps Termination). For all $s$ and $L$, $\text{MatchingSpecSteps}(s, L)$ terminates and returns a finite set, and for all $\langle O, S \rangle \in \text{MatchingSpecSteps}(s, L)$, $S$ is finite.

Proof. Let $\langle \hat{I}, \hat{\mu} \rangle = L$. The initial steps the algorithm attempts to simulate a transition labeled with $\hat{I}$ are bounded by the number of transitions in the PSM’s current state definition (note that the input-pattern-matching rules are deterministic, so there is at most one way for a value $\hat{I}$ to match a pattern $\pi$). There are finitely many transitions in each state definition, so that part of the algorithm terminates. Additionally, each transition forks only a finite number of additional PSMs, so each produced configuration $S'$ is finite. Therefore, by the SimulateOutputs Termination lemma, each $\text{SimulateOutputs}(S', \hat{\mu})$ call terminates, and for all $\langle O, S'' \rangle \in \text{SimulateOutputs}(S', \hat{\mu})$, $S''$ is finite. Those results make up the results of $\text{MatchingSpecSteps}$, so for all $\langle O, S'' \rangle$ in $\text{results}$, $S''$ is finite.

N.35  Explore Termination Lemma

Lemma (Explore Termination). For all $\hat{K}_{\text{init}}$ and $s_{\text{init}}$, if the transformation $T$ is termination-guaranteeing and there exist $P$, $\Sigma$, and $K$ such that
• \( \vdash_{\text{prog}} P \),
• \( \langle \hat{K}, s_{\text{init}} \rangle \) is an instantiation of \( P \) and \( \Sigma \), and
• \( |\hat{K}| = \hat{K}_{\text{init}} \),

then \( \text{Explore}(\hat{K}_{\text{init}}, s_{\text{init}}) \) terminates and returns a result \( \langle R, Z, F \rangle \) such that

• \( R \) is finite,
• \( Z \) is a finite strategy,
• \( F \) is finite, and
• \( R \cap F = \emptyset \).

Proof. The key to this proof is to show that the set of items that \( \text{Explore} \) adds to the worklist \( W \) is bounded. Then because no item is added to the worklist more than once, the loop iterates a finite number of times and therefore eventually terminates.

Loop Invariant

Let there be \( P \) and \( \Sigma \) as described in the statement of the lemma. By the Instantiation Type Preservation lemma (appendix J), \( \vdash_{\text{cfg}} \hat{K} \). By the Abstraction Type Preservation lemma, (appendix H), there exists \( d \) such that \( \vdash_{\text{cfg}} \hat{K}_{\text{init}} : d \). Let \( \hat{a}_1, \ldots, \hat{a}_n \) be the external addresses appearing in \( \hat{K}_{\text{init}} \). We will establish a loop invariant that holds at the head of the main \textbf{while} loop. To do so, we first define a set \( E \) that contains the pairs that the \( \text{Explore} \) algorithm will explore. Formally, let \( E \) be the set of pairs \( \langle \hat{K}, s \rangle \) such that

- \( \hat{K} \) is a configuration of \( P \) with initial externals \( \{\hat{a}_1, \ldots, \hat{a}_n\} \),
- \( s \) is a PSM of \( \Sigma \),
- there is no actor handling an event in \( \hat{K} \),
- \( \vdash_{\text{cfg}} \hat{K} : d \), and
- either \( \langle \hat{K}, s \rangle = \langle \hat{K}_{\text{init}}, s_{\text{init}} \rangle \) or
  - \( \hat{K} \) is \textit{Explore}-bounded with maximal marker \( \text{max}(\text{Mon}(s)) \), and
  - \( s \) is \textit{Explore}-bounded.

We must also include the pairs that get added to the worklist but are not explored (because of \textit{ShouldExplore}). Therefore, the loop invariant states that for every pair \( \langle \hat{K}, s \rangle \) in \( W \), either

- \( \langle \hat{K}, s \rangle \in E \), or
- there exist \( \hat{K}', \hat{K}'', L, O, S, s', s'', A, \) and \( M \) such that
Before the first iteration of the loop, $W$ contains only $⟨\hat{K}_{\text{init}}, s_{\text{init}}⟩$. We know that $\hat{K}$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$ and that $s$ is a PSM of $\Sigma$ by the definition of instantiation, and because we stated above that $\hat{a}_1, \ldots, \hat{a}_n$ are the external addresses appearing in $\hat{K}_{\text{init}}$. We have already established that $\vdash_{\text{cfg}} \hat{K}_{\text{init}} : d$, and there is no actor handling an event in $\hat{K}_{\text{init}}$ by the definition of instantiation. Therefore, the invariant holds before the first iteration.

To show that an iteration of the loop maintains the invariant, let $⟨\hat{K}, s⟩$ be a pair added to $W$ during an iteration. By the definition of Explore, there exist $\hat{K}'$, $\hat{K}''$, $L$, $O$, $S$, $s'$, $s''$, $A$, and $M$ such that

- $⟨\hat{K}', s'⟩ \in W$,
- $\text{ShouldExplore}(s') = \text{true}$,
- $⟨L, \hat{K}''⟩ \in \text{ProgSteps}(\hat{K}')$,
- $⟨O, S⟩ \in \text{MatchingSpecSteps}(s', L)$,
- $s'' \in S$,
- $⟨\hat{K}, s, A, M⟩ \in T(\hat{K}'', s'')$, and
- $\text{ShouldExplore}(s)$ returns false.

By the loop invariant and because $\text{ShouldExplore}(s') = \text{true}$, $⟨\hat{K}', s'⟩ \in E$. By the definition of $\text{ProgSteps}$, $\hat{K}' \xrightarrow{L} \hat{K}''$. By the corollary to the Program Origin Preservation lemma, $\hat{K}''$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$. By the definition of the summary transition relation, there is no actor handling an event in $\hat{K}''$. By the corollary to the Abstract Type Preservation lemma, $\vdash_{\text{cfg}} \hat{K}'' : d$. By the MatchingSpecSteps Correctness lemma, $\{s'\} (L, O) S$. By corollary N.15.4 to the Specification Origin Preservation lemma, $s''$ is a PSM of $\Sigma$. By the corollary to the Distinct Spawns lemma, there are at most two atomic actors in $\hat{K}''$ for every location $\ell$. Then because $T$ is termination-guaranteeing,

- $\hat{K}$ is a configuration of $P$ with initial externals $\hat{a}_1, \ldots, \hat{a}_n$,
- $s$ is a PSM of $\Sigma$,
- there is no actor handling an event in $\hat{K}$,
- $\vdash_{\text{cfg}} \hat{K} : d$, and
- $\hat{K}$ is Explore-bounded with maximal marker $\max(\text{Mon}(s))$, and
either \( s \) is Explore-bounded or \( \text{ShouldExplore}(s) = \text{false} \).

Therefore, \( \langle \hat{K}, s \rangle \) satisfies the conditions for the loop invariant, and therefore every iteration of the loop maintains the invariant.

**Finite Worklist**

Next, we show that the total set of items added to the worklist is finite, and therefore the algorithm eventually terminates. By the Bounded Pairs lemma, there are finitely many \( \langle \hat{K}, s \rangle \) pairs such that

- \( \hat{K} \) is a configuration of \( P \) with initial externals \( \hat{a}_1, \ldots, \hat{a}_n \),
- \( s \) is a PSM of \( \Sigma \),
- \( \vdash_{ctg} \hat{K} : d \), and
- \( \hat{K} \) is Explore-bounded with maximal marker \( \max(\text{Mon}(s)) \), and
- \( s \) is Explore-bounded.

Because \( E \) is defined as those pairs plus the single additional pair \( \langle \hat{K}_{\text{init}}, s_{\text{init}} \rangle \), \( E \) is finite.

To show that the set of pairs added from a given pair \( \langle \hat{K}, s \rangle \) during the loop is finite, we have that

- by the \( \text{ProgSteps} \) Termination lemma, there are finitely many \( \langle L, \hat{K}' \rangle \) pairs in \( \text{ProgSteps}(\hat{K}) \),

- by the \( \text{MatchingSpecSteps} \) Termination lemma, there are finitely many \( \langle O, S \rangle \) pairs in \( \text{MatchingSpecSteps}(s, L) \) for each \( L \) and finitely many \( s' \) in each \( S \), and

- there are finitely many \( \langle \hat{K}'', s'' \rangle \in T(\hat{K}', s') \) for each pair \( \hat{K}' \) and \( s' \) because \( T \) is termination-guaranteeing.

As a result of the above argument, the set of items that can be added to the worklist \( W \) is finite. Furthermore, \( \text{MatchingSpecSteps}(s', L) \) terminates for all \( s' \) and \( L \) by the \( \text{MatchingSpecSteps} \) Termination lemma, so every iteration of the loop terminates. No item is added to the worklist twice, because every item removed from the worklist is added to either the frontier \( F \) or the relation \( R \), and \( \text{Explore} \) checks whether a given pair is in either of those sets before adding it to \( W \). Thus, there can be only finitely many additions to \( W \), and every iteration of the loop removes an item from \( W \). Therefore, \( W \) is eventually empty and the loop terminates. Therefore, \( \text{Explore}(\hat{K}_{\text{init}}, s_{\text{init}}) \) eventually terminates.
Finite Return Values

It remains to show that the result \( (R, Z, F) \) returned by Explore satisfies the desired properties. It suffices to establish a loop invariant on the while loop that states that

1. \( R \) is finite,
2. \( Z \) is a finite strategy,
3. \( F \) is finite,
4. \( R \cap F = \emptyset \), and
5. \( W \cap (R \cup F) = \emptyset \).

Initially, \( R = \emptyset \) and \( F = \emptyset \), and for all \( \langle \hat{K} \xrightarrow{L} \hat{K}'', s \rangle \in \text{dom}(Z) \), \( Z(\hat{K} \xrightarrow{L} \hat{K}'', s) = \emptyset \).

Therefore, the invariant is satisfied before the loop starts iterating.

We show how the body of the loop maintains each invariant individually.

1. Every iteration of the while adds at most finitely many elements to \( R \).
2. We have already shown that \( \text{ProgSteps}(\hat{K}) \) returns finitely many pairs of the form \( \langle L, \hat{K}'' \rangle \). The first for loop in the while loop iterates over those pairs and for each one, sets \( Z(\hat{K} \xrightarrow{L} \hat{K}'', s) = \text{MatchingSpecSteps}(s, L) \). We have already shown that in each case,
   - \( \hat{K}'' \) is a configuration of \( P \) with initial externals \( \hat{a}_1, \ldots, \hat{a}_n \),
   - there is no actor handling an event in \( \hat{K}'' \),
   - \( \vdash_{\text{cfg}} \hat{K}'' : d \),
   - there are at most two atomic actors in \( \hat{K}'' \) for every spawn location \( \ell \),
   - \( \text{MatchingSpecSteps}(s, L) \) is finite, and
   - for all \( \langle O, S \rangle \in Z(\hat{K} \xrightarrow{L} \hat{K}', s) \), \( S \) is finite and every \( s'' \in S \) is a PSM of \( \Sigma \).

Therefore, the loop maintains the invariant that \( Z \) is a finite strategy.

3. Every iteration of the while adds at most finitely many elements to \( F \).

4. Each iteration of the loop adds the element removed from \( W \) to either \( R \) or \( F \), but not both, so because \( W \cap (R \cup F) = \emptyset \), the loop maintains the invariant that \( R \cap F = \emptyset \).

5. Whenever the body of the loop adds an element to \( R \) or \( F \), it is always the element that was removed from \( W \) in that iteration. Furthermore, the algorithm explicitly adds an element to \( W \) only if it is explicitly not in \( R \cup F \), so the loop body maintains the invariant that \( R \cap F = \emptyset \).

\( \square \)
N.36 Finite Transformation Steps Lemma

Lemma (Finite Transformation Steps). For all \( Z \) and \( T \) such that \( Z \) is a finite strategy and \( T \) is termination-guaranteeing, the set of transformation-step transitions \( \langle \hat{K}, s \rangle \xrightarrow{L,K'' Z, O, S.T.A.M} \langle \hat{K}', s' \rangle \) using that strategy \( Z \) and transformation \( T \) is finite and can be computed in a finite amount of time.

Proof. Because \( Z \) is a finite strategy, we know that there are finitely many inputs to \( Z \) that yield a non-empty result. It is assumed that that set can be computed in a finite amount of time. (In the actual implementation, \( Z \) is represented as a dictionary whose keys are the inputs that lead to non-empty outputs.) Each such input is of the form \( \langle \hat{K}, L \hat{K}', s \rangle \). All of the desired transformation-step transitions must start from a pair \( \langle \hat{K}, s \rangle \) from one of those pairs, so it remains to show that we can compute the set of edges from each such input in a finite amount of time, and that that set is finite for each one.

Let \( \langle \hat{K}, L \hat{K}'', s \rangle \) be one of those inputs. Because \( Z \) is a finite strategy, we know that there exist \( P, \Sigma, \hat{a}_1, \ldots, \hat{a}_n \), and \( d \) such that

\( \hat{K}'' \) is a configuration of \( P \) with initial externals \( \hat{a}_1, \ldots, \hat{a}_n \),

there is no actor handling an event in \( \hat{K}'' \),

\( \vdash \text{cfg} \hat{K}'' : d \),

there are at most two atomic actors in \( \hat{K}' \) for every spawn location \( \ell \),

\( Z(\hat{K}, L \hat{K}'', s) \) is finite, and

for all \( \langle O, S \rangle \in Z(\hat{K}, L \hat{K}'', s) \), \( S \) is finite and every \( s'' \in S \) is a PSM of \( \Sigma \).

Because \( Z(\hat{K}, L \hat{K}'', s) \) is finite and each \( S \) is finite, it remains to show that \( T(\hat{K}'', s'') \) terminates and returns a finite set. We get this from the definition of a termination-guaranteeing transformation, so we’re done.

N.37 Prune Termination Lemma

Lemma (Prune Termination). For all \( R, Z, \) and \( F \), if

\( R \) is finite,

\( Z \) is a finite strategy,

\( F \) is finite,

\( R \cap F = \emptyset \), and

the transformation \( T \) used in the algorithm is termination-guaranteeing.
then $\text{Prune}(R, Z, F)$ terminates and returns some $(R', Z')$ such that $R'$ is finite and $Z'$ is a finite strategy.

**Proof.** First, we show an invariant on the while loop that $R' \cap F = \emptyset$. Before the loop starts iterating, $R' = R$, so the invariant holds by the last precondition on this lemma. During the iteration of the loop, any element added to $F$ is removed from $R'$ at the same time, so the loop body maintains the invariant.

Next, we show that the while loop iterates finitely many times. Within the body of that while loop, any pairs added to $F$ must have already been a member of $R'$, and one pair is removed from $F$. Then because $F \cap R' = \emptyset$ by the above loop invariant, $|F \cup R'|$ decreases with every iteration of the loop. The loop ends when $F = \emptyset$, so the loop iterates finitely many times.

To show that the forall loop iterates finitely many times, we need a loop invariant on the outer while loop to show that $Z'$ is a finite strategy. Before the loop starts, $Z' = Z$, and $Z$ is a finite strategy by a precondition of this lemma. Inside the loop, it is easy to see from the definition of a finite strategy that removing one of the possible solutions $(O, S)$ from $Z'$ maintains the property that $Z'$ is a finite strategy, so the body of the loop maintains the invariant. Then by the Finite Transformation Steps lemma, the set of transformation-step transitions $\langle \hat{K}', s' \rangle \xrightarrow{L, \hat{K}'' Z, O, S, T, A, M} \langle \hat{K}, s \rangle$ iterated over in the inner loop is finite and can be computed in a finite amount of time. Therefore, the forall loop iterates a finite number of times. It is easy to see that the body of the loop always terminates.

For the return values, we have already shown that $Z'$ is a finite strategy. The set $R'$ is initially defined as the finite set $R$, and the algorithm only removes items from $R'$, so the final returned $R'$ is finite.

$\square$

**N.38 FindFulfillingPairs Termination Lemma**

**Lemma** (FindFulfillingPairs Termination). For all $R$ and $Z$, if

- $R$ is finite,
- for all $(\hat{K}, s) \in R$, there are finitely many transformation-step transitions $\langle \hat{K}, s \rangle \xrightarrow{L, \hat{K}'' Z, O, S, T, A, M} \langle \hat{K}', s' \rangle$ using the given strategy $Z$ and the transformation $T$ used by the algorithm, and
- for every such transition $\langle \hat{K}, s \rangle \xrightarrow{L, \hat{K}'' Z, O, S, T, A, M} \langle \hat{K}', s' \rangle$ from a member of $R$, $(\hat{R}', s') \in R$,

then $\text{FindFulfillingPairs}(R, Z)$ terminates and returns a set $R'$ such that $R' \subseteq R$.

**Proof.** $\text{FindFulfillingPairs}$ never modifies the given set $R$, so the outer for loop iterates finitely many times. The set of obligations $\text{Obls}(s)$ for any PSM is assumed to be finite, so the inner for loop also iterates finitely many times.

Let $(\hat{K}, s)$ be a member of $R$, let $(\eta, po)$ be a member of $\text{Obls}(s)$, and let $G$ be the non-satisfaction graph from $(\hat{K}, s)$ for $(\eta, po)$, $Z$, and $T$. Let $(\hat{R}', s', \eta')$ be
a vertex in that graph. By the inductive definition of a non-satisfaction graph, that vertex must be reachable from \( \langle \hat{K}, s \rangle \) in finitely many transformation-step transitions. By induction on the number of transitions and the third precondition of this lemma, \( \langle \hat{K}', s' \rangle \in R \). Therefore, every vertex in \( G \) is a member of \( R \), and therefore \( G \) has finitely many vertices.

By the definition of a non-satisfaction graph, every edge must correspond to a transformation-step transition from one of its vertices, and by the second precondition of this lemma, there are finitely many such transitions from each member of \( R \). Therefore, every vertex in \( G \) has finitely many outgoing edges, and therefore \( G \) has finitely many edges overall.

The inner loop of \textit{FindFulfillingPairs} constructs \( G \), checks whether \( G \) contains a program-fair SCC. Because \( G \) has finitely many vertices and edges, it can be constructed in a finite number of steps by following the inductive definition for a non-satisfaction graph. Any given program configuration is assumed to have finitely many actors and in-flight messages, so checking whether a program configuration is quiescent is a terminating process, and therefore the check for such a vertex terminates.

For the SCC's, there are well-known terminating algorithms for computing the SCC's of a graph \( G \) (of which there are finitely many for a graph with finitely many vertices). To determine whether a given SCC is program-fair, for every vertex \( \langle \hat{K}_1, s_1, \eta \rangle \) in the SCC and every address \( \hat{a} \) identifying a necessarily enabled actor in \( \hat{K}_1 \), we must perform two checks corresponding to the definition of a program-fair SCC.

1. Determine whether there is a path in the SCC from \( \langle \hat{K}_1, s_1, \eta \rangle \) to a vertex \( \langle \hat{K}_n, s_n, \eta \rangle \) with edges labeled \( (L_1, A_1, M_1), \ldots, (L_{n-1}, A_{n-1}, M_{n-1}) \) and an address \( \hat{a}' = A_{n-1} \circ \ldots \circ A_1(\hat{a}) \) such that either
   - there is no actor in \( \hat{K}_n \) with address \( \hat{a}' \),
   - the actor at \( \hat{a}' \) is not necessarily enabled in \( \hat{K}_n \), or
   - there is an edge from \( \langle \hat{K}_n, s_n, \eta_n \rangle \) in the SCC with label \( \langle L_n, A_n, M_n \rangle \) such that \( \hat{a}' \) identifies the active actor for \( L_n \).

2. Determine whether, for every message \( \langle \hat{a}@H, \hat{\nu} \rangle \) with abstract quantity \textit{single} in \( \hat{K}_1 \), there is a path in the SCC from \( \langle \hat{K}_1, s_1, \eta \rangle \) to a vertex \( \langle \hat{K}_n, s_n, \eta \rangle \) with edges labeled \( (L_1, A_1, M_1), \ldots, (L_{n-1}, A_{n-1}, M_{n-1}) \) and that there exist \( A' = A_{n-1} \circ \ldots \circ A_1, M' = M_{n-1} \circ \ldots \circ M_1, \hat{a}' = A'(\hat{a}), H'' = M'(H'), \hat{\nu}' = A'(M'(\hat{\nu})), \hat{\mu}', H'', \) and \( \hat{\nu}' \) such that \( \hat{K}_n = \langle \hat{\nu}', \hat{\mu}' \mid H'' \rangle \hat{\nu}' \) and either
   - \( \langle \hat{a}'@H'', \hat{\nu}' \rangle \notin \text{dom}(\hat{\nu}') \),
   - \( \hat{\nu}'(\hat{a}'@H'', \hat{\nu}') = \text{many} \), or
   - there is an edge from \( \langle \hat{K}_n, s_n, \eta_n \rangle \) in \( C \) with label \( \langle L_n, A_n, M_n \rangle \) such that and \( L_n = \langle \hat{a}' : \text{rcv-int}(H'', \hat{\nu}'), \hat{\mu}' \rangle \) for some \( \hat{\mu}' \).
It is straightforward to check each property by a depth-first traversal of the
graph, keeping track of either the current address $\bar{a}'$ (for the first property) or
the current message $\langle \bar{a}'@H''',\vec{v}' \rangle$ (for the second property) as described above,
and stopping the traversal whenever the same vertex is reached with the same
current address/message. Therefore, each check terminates, and because there
are only finitely many vertices to check and finitely many actors and messages
per vertex (i.e., in the program configuration in that vertex), checking whether an
SCC is program-fair takes a finite amount of time.

The rest of the body of the loop is trivially terminating. We have shown that
the body of the loop terminates, and that both the inner and outer loops iterate a
finite number of times, so $\text{FindFulfillingPairs}(R,Z)$ terminates.

The set $R'$ is initially defined as the empty set, and the algorithm only adds
elements of $R$ to $R'$. Therefore, the final $R'$ is a subset of $R$.

\section*{ModelCheck Termination Theorem}

\textbf{Theorem (ModelCheck Termination).} For all $P$ and $\Sigma$, if $\vdash \text{prog } P$ and the transformation $T$ is termination-guaranteeing, then $\text{ModelCheck}(P,\Sigma)$ terminates.

\textbf{Proof.} The initial “instantiable” check in the $\text{ModelCheck}$ algorithm merely checks whether the input-monitored and output-monitored addresses declared on $\Sigma$ have correspondents on $P$, so it is easy to see that that check terminates. Similarly, instantiation of programs and specifications are obviously terminating operations by inspection of their definitions (see sections 3.5.2 and 3.6.2), so the actual instantiation of $P$ and $\Sigma$ into $\tilde{K}_{\text{init}}$ and $s_{\text{init}}$ terminates. By structural induction on the various components in $\tilde{K}_{\text{init}}$, it is also easy to see that the abstraction $\tilde{\|}\tilde{K}_{\text{init}}\|\tilde{\|}$ to define $\tilde{K}_{\text{init}}$ terminates.

By the Explore Termination lemma, $\text{Explore}(\tilde{K}_{\text{init}},s_{\text{init}})$ terminates and returns a tuple $(R_{\text{loc}},Z_{\text{loc}},F_{\text{loc}})$ such that

- $R_{\text{loc}}$ is finite,
- $Z_{\text{loc}}$ is a finite strategy,
- $F_{\text{loc}}$ is finite, and
- $R_{\text{loc}} \cap F_{\text{loc}} = \emptyset$.

By the Prune Termination lemma, $\text{Prune}(R_{\text{loc}},Z_{\text{loc}},F_{\text{loc}})$ terminates and returns some $(R_{\text{sim}},Z_{\text{sim}})$ such that $R_{\text{sim}}$ is finite and $Z_{\text{sim}}$ is a finite strategy. By the $\text{FindFulfillingPairs}$ Termination lemma, $\text{FindFulfillingPairs}(R_{\text{sim}},Z_{\text{sim}})$ terminates and returns some $R_{\text{fulfill}}$ such that $R_{\text{fulfill}} \subseteq R_{\text{sim}}$. Therefore, because $R_{\text{sim}}$ is finite, so are $R_{\text{fulfill}}$ and $R_{\text{sim}} - R_{\text{fulfill}}$. Then by the Prune Termination lemma again, $\text{Prune}(R_{\text{fulfill}},Z_{\text{sim}},R_{\text{sim}} - R_{\text{fulfill}})$ terminates and returns some $(R_{\text{conf}},Z_{\text{conf}})$ such that $R_{\text{conf}}$ is finite. Therefore, it takes a finite amount of time to check whether $\langle \tilde{K}_{\text{init}},s_{\text{init}} \rangle \in R_{\text{conf}}$.

We have shown that each of the individual steps of the ModelCheck algorithm terminate. Therefore, $\text{ModelCheck}(P,\Sigma)$ terminates. \qed