Specifying Finite-State Actors

November 9, 2015

Jonathan Schuster  
*Northeastern University*

Stephen Chang  
*Northeastern University*

Matthias Felleisen  
*Northeastern University*

Report No. NU-CCIS-TR-2015-001

This report is available open access, hosted by Northeastern University.
Specifying Finite-State Actors*

Jonathan Schuster, Stephen Chang, Matthias Felleisen
Northeastern University, Boston, Massachusetts, USA
{schuster,stchang,matthias}@ccs.neu.edu

Abstract. Programmers often code up asynchronous message-passing systems as communicating finite-state actors. An actor in some state listens for messages, responds to those messages, and transitions to another state. Most of these systems allow messages to carry actor addresses.

This paper presents a kernel language for implementing and specifying such systems. A specification consists of finite-state machines, expressing what kind of messages each component should expect and what kind of actions it should take in response. In addition, a specification may prescribe how a component may use an address received from a message as well as how a component handles inputs on addresses that it sends out. Finally, the paper adds a conformance relation that articulates when an implementation meets such a specification.

1 Communicating FSM Actors

When programmers work on concurrent systems, many reach for Hewitt-style actor [1] frameworks such as Akka [2] and Erlang [3]. In addition, programmers often organize actors as state machines that, in any given state, listen for messages, and when messages arrive, process them and transition to another state.

A basic notion of correctness for such actor systems must be formulated in terms of a protocol from the perspective of individual actors. Therefore, a specification framework must focus on descriptions of an actor’s actions in response to messages. For example, in a telephony system, an actor managing a telephone may be responsible for ensuring that

– every new call request receives an accept or reject response,
– during a call, every incoming call is rejected, and
– hanging up the receiver terminates the call.

Smith and Talcott’s specification diagrams (SD) [22] provide a completely general setting for specifying all kinds of actors. SD supports both graphical and textual forms of specification. Its specifications view actor

* This is an extended version of a paper presented at WSFM/BEAT 2015.
systems as non-deterministic compositions, with mathematical expressions serving as constraints. The very expressiveness and generality of SD makes it difficult, however, to prove conformance between specifications and programs. Indeed, Smith and Talcott focus on refinements among specifications alone, though one could extend this notion to programs.

To specify actor protocols that admit tractable conformance proofs, our approach is to focus on the widely used finite-state organization of actors and to exploit this structure. Our contributions are

1. a language of communicating FSM actors, which distills the essence of implementing FSMs in Akka and Erlang into a small model;
2. a specification language for protocols in this setting, and
3. a conformance relation between specifications and programs.

We then add a conformance proof for an actor protocol akin to Akka’s interface to TCP.

The paper is organized as follows. Section 2 motivates the problem with an example from the world of network protocols. Section 3 then introduces the programming language and sketches how to implement the protocol example from section 2 via actors. The heart of the paper—sections 4 through 6—presents the specification language, a conformance relation between specifications and implementations, and a conformance proof sketch (appendix A contains the full proof). Sections 7 and 8 round off the paper with related and future work.

2 The Alternating Bit Transport Protocol

Our running example is an implementation of a hypothetical protocol, called the Alternating Bit Transport Protocol (ABTP). At the application level, ABTP mimics the interface of the Akka TCP implementation as a representative of typical patterns in actor programs. At the network level, ABTP combines TCP and the Alternating Bit Protocol [5].

Using ABTP, a sender program on one network host can reliably transmit a sequence of messages to some existing receiver program on a different host over unreliable network links. As in the Alternating Bit Protocol, ABTP achieves this reliability by repeatedly transmitting one message at a time until it receives a matching acknowledgment. Two-way handshakes establish and close a connection.

The ABTP Session Lifecycle Figure 1 illustrates one possible message sequence of this protocol. The manager is the entry point process on
the side of the initiating party; it creates new sessions and tracks which
port numbers are allocated to sessions. To create an ABTP sender and
start a new session, the application layer sends the manager a connection
request. During its initialization, the sender negotiates a connection with
the receiver. Once the sender receives a SynAck back from the receiver,
it sends the application layer a Connected status message, providing the
application layer with the sender’s address.

To send a message to the receiver, the application layer issues a Write
command to the sender, including an address to send the response to. The
latter responds with Queued, if the message has been queued for trans-
mission, or WriteFailed, if the session is closed (not shown). For each
queued message, the sender attaches a sequence number (Seq0 or Seq1)
and sends it to the receiver. It then waits the corresponding acknowledg-
ment (Ack0 or Ack1, respectively) before sending the next message with
the alternate sequence number. If the receiver does not acknowledge a
message within a certain amount of time, it is resent.

When it wants to end the session, the application layer requests a Close
from the sender, resulting in a Fin message to the receiver. When
the receiver responds with FinAck, the sender sends a Closed status to
confirm that the session is closed. Whenever the connection times out,
the application layer receives an error message (not shown in figure 1).

**Correctness Properties** The details of the protocol at the network
level—the shaded portion of figure 1—are irrelevant to the application
layer. Instead, an application-layer programmer expects the following:
The sender sends a status update when a connection attempt either succeeds or times out.

The sender sends exactly one response for every write request.

The address provided in a connection response refers to an actor with the previous two properties.

All of these properties are communication-based and specify how the process should react to events in its environment. They are also local properties in that each depends only on the behavior of the manager and the sender (not the whole program). Finally, the properties involve dynamic communication topology; for example, Queued is not sent to a statically known address, but to the one provided in the Write request.

A formal description of this high-level protocol would be useful for documentation, modeling, and verification. While the diagram of figure 1 is useful, it is too imprecise and incomplete to act as a proper specification. A suitable specification language must be concrete enough to express important properties yet abstract enough to remain concise.

First, though, we need a language for programming actors.

### 3 CSA: The Programming Language

Protocols are essentially finite-state machines, which is why programmers have long used FSMs to describe protocols [6,8,16,18]. This trend continues today with Erlang’s `gen_fsm` behavior and Akka’s FSM trait. We have distilled this pattern into a small programming language called CSA (Communicating State Agents). As we illustrate in section 6, this model naturally accommodates conformance proofs.

We have prototyped CSA in Racket [14] and thus use S-expressions for its syntax. This section introduces the syntax of CSA, its pragmatics, and a worked example. It concludes with a formal reduction semantics.

**Overview** A running CSA program consists of interacting, concurrent agents. Each agent is a state machine that reacts to events and communicates with other agents via asynchronous messages. While handling an event, an agent can send messages, spawn new agents, and transition to its next state. Every agent has an address that other agents use to send it messages. Messages are delivered out-of-order, but without any losses.

---

1 Available at [https://github.com/schuster/csa](https://github.com/schuster/csa)

2 Ordered delivery requires only small changes to the languages presented here.
Figure 2 presents the syntax of CSA. An agent (created with spawn) includes an initialization expression and a set of named states that define how the agent reacts to events. Each state definition declares the name of the state, parameters $x_s \ldots$ that are in scope for its actions (in general, “$a \ldots$” denotes any number of $a$s), a binding $x_h$ for a received message, a message handler expression $e$, and an optional timeout clause.

$$e ::= (\text{spawn } e \ldots) \mid (\text{goto } s e \ldots) \mid (\text{send } e) \mid \text{self} \mid (\text{begin } e \ldots)$$

$$S ::= (\text{define-state } (s x_s \ldots) (x_h) e)$$

$$p \in \text{Pat} ::= t \mid (\text{list } p \ldots) \mid * \mid x_a \mid x_n \ldots$$

$$s \in \text{StateName}$$

$$x \in \text{Var}$$

$$t \in \text{Symbol}$$

$$n \in \text{N}$$

Fig. 2. CSA syntax

An agent transitions to a state $s$ with (goto $s e \ldots$) and there awaits the arrival of a message. The results of $e \ldots$ are the actual values for the state’s parameters. A match form evaluates the body of the first clause whose pattern matches the given value. Symbol patterns match only that symbol, list patterns match lists whose members match the sub-patterns, * matches anything, and variables match any value of the proper type and bind it in the clause.

The ABTP Implementation Figure 3 shows partial definitions of the ABTP Manager and Sender agents in CSA. Manager has a single state Ready. It handles every connection request by spawning a new Sender; it then transitions back to Ready.

In its initialization, Sender sends the initial 'Syn to the receiver and transitions to Connecting, saving the parameters from the connection request as parameters in that state. In Connecting, if an incoming message from the receiver is a 'SynAck, Sender sends 'Connected to the application layer and transitions to Connected; otherwise, it takes no action. Connecting's timeout clause sends a failure message and transitions to Closed after 3 seconds without a received message.

---

3 CSA includes a simple type system to rule out improper programs, but we omit it to simplify the presentation. As indicated with subscripts, pattern variables come with type annotations.
Evaluation Syntax  Formulating a reduction semantics of CSA programs requires additional syntax; see figure 4 for details. In the figure, \( \mathcal{P}(X) \) denotes the set of subsets of \( X \), and \( \mathcal{M}(X) \) denotes the set of multisets with elements in \( X \). The set of run-time values \( v \) includes natural numbers \( n \), symbols \( t \), lists of values, and agent addresses \( a \). The rcv expressions indicate an agent awaiting a message.

Fig. 4. CSA evaluation syntax

Following Agha et al. [1], an execution state is a configuration:

\[
\left\langle \alpha \mid \mu \right\rangle^\rho_x
\]

where \( \alpha \) maps agents’ addresses \( a \) to their states \((S\ldots)\) and current behavior \( e \), written \( a \mapsto \left\langle (S\ldots), e \right\rangle \). Also \( \mu \) stands for multisets of in-transit message packets \( \langle a \leftarrow v \rangle \), pairs of values \( v \) and destination addresses \( a \).
The configuration includes an interface to the program’s environment. The set \( \rho \) of receptionists represents the set of addresses that may receive messages from the environment. Similarly, the set \( \chi \) of externals collects the set of environmental addresses to which the program may send a message. These sets grow as messages are exchanged with the environment.

A transition label \( a?v \) indicates the reception of a value \( v \) from the environment at \( a \), and \( a!v \) indicates the release of \( v \) to the environment on \( a \). The rest indicate internal actions. Without distinguishing these internal labels, we would not have a fair system.

\[
\begin{align*}
\langle a \mapsto ..., \alpha \mid \mu \rangle^\rho & \quad \langle a \mapsto ..., \alpha \mid \mu \rangle^\chi \\
\end{align*}
\]

**E-Goto**

\[
\langle (S\ldots),E[(\text{goto } s \ldots)] \rangle \quad a: \text{func} \rightarrow \langle (S\ldots), (\text{rcv } (x_h) e[x_s \leftarrow v\ldots]) \rangle
\]

if \( \text{define-state } (s x_s \ldots) (x_h) e \) is in \( S\ldots \)

**E-Receive**

\[
\langle a \mapsto ((S\ldots),(\text{rcv } (x_h) e)) \rangle | \langle (a \leftarrow v), \mu \rangle \quad a: \text{rcv}(v) \rightarrow \langle (S\ldots), e[x_h \leftarrow v] \rangle | \mu
\]

**E-In**

\[
\langle a \mapsto ((S\ldots),(\text{rcv } (x_h) e)), \alpha \mid \mu \rangle^\rho \quad a?v \rightarrow \langle a \mapsto ((S\ldots), e[x_h \leftarrow v]), \alpha \mid \mu \rangle^\chi
\]

if \( a \in \rho \) and \( \text{addresses}(v) \cap (\{a\} \cup \text{dom}(\alpha)) \subseteq \rho \)

where \( \chi' = \chi \cup \text{addresses}(v) - (\{a\} \cup \text{dom}(\alpha)) \)

**E-Send**

\[
\langle (S\ldots),E[(\text{send } a' v)] \rangle | \langle (a \leftarrow v), \mu \rangle \quad a: \text{func} \rightarrow \langle (S\ldots), E[v] \rangle | \langle a' \leftarrow v \rangle, \mu
\]

**E-Out**

\[
\langle a \mid (a \leftarrow v), \mu \rangle^\rho \quad a!v \rightarrow \langle a \mid \mu \rangle^\rho' \\
\text{if } a \in \chi, \text{ where } \rho' = \rho \cup (\text{addresses}(e) \cap \text{dom}(\alpha))
\]

**E-Timeout**

\[
\langle (S\ldots),(\text{rcv } (x_h) e [\text{timeout } n) e'] \rangle \quad a: \text{timeout} \rightarrow \langle (S\ldots), e' \rangle
\]

**E-Spawn**

\[
\langle (S\ldots),E[(\text{spawn } e S'\ldots)] \rangle \quad a: \text{func} \rightarrow \langle (S\ldots), E[e'] \rangle,
\]

\[
a' \mapsto ((S'[\text{self } \leftarrow a']\ldots), e[\text{self } \leftarrow a'])
\]

such that \( a' \notin (\{a\} \cup \text{dom}(\alpha)) \cup \chi \)

**Fig. 5.** Selected CSA configuration reduction rules

**Reduction Semantics** A CSA program consists of an expression \( e \) and an initial substitution \( \{x \mapsto a, \ldots\} \) of variables to external addresses.
Its initial configuration has a single actor that evaluates $e[x \leftarrow a] \ldots$ and transitions to a dead-end state. The addresses from the substitution comprise the initial configuration’s externals $\chi$.

Figure 5 displays the essential reduction rules of CSA’s operational semantics. The rule $\text{E-Goto}$ transitions an agent into its next state (represented by a $\text{rcv}$ expression), in which it waits to receive a message. The message variable and handler expression of the $\text{rcv}$ expression come from the named state. The rule drops the continuation $E$; in practice $\text{goto}$ expressions must be in tail position.

$\text{E-Receive}$ picks an arbitrary message packet sent to agent $a$ and substitutes it into the handler expression.

$\text{E-In}$ is similar to $\text{E-Receive}$, with the message coming from the environment. If an address is in the receptionist set, then the environment can send any message to it, as long as the message does not contain addresses known only to the program. $\text{E-In}$ uses the $\text{addresses}$ function to detect all addresses provided in the received message and adds those that do not belong to an agent in the configuration to the set of externals.

$\text{E-Send}$ pairs the message $v$ with its destination $a'$ and adds it to the set of message packets.

$\text{E-Out}$ outputs to the environment a message packet directed to one of the external addresses. The rule adds any addresses in the message to the receptionists set if it belongs to an agent in the configuration, because the environment gains access to them.

$\text{E-Timeout}$ allows an agent to run its state’s timeout handler. The semantics does not model a clock for timeouts.

$\text{E-Spawn}$ creates an agent with a globally unique address. The initialization expression becomes the agent’s body; the result of the $\text{spawn}$ is the new address $a'$. This is substituted for $\text{self}$ in the new agent.

The full semantics includes additional rules (not shown) labeled $a: \text{func}$ that define conventional reductions for $\text{match}$, $\text{begin}$, and $\text{let}$, as well as $\text{E-Goto}$, $\text{E-Receive}$, and $\text{E-In}$ variants for states with timeouts.

4 APS: Specifications for CSA

As with any programming language, CSA programs contain low-level implementation details, such as the number of times an ABTP sender re-transmits a message. This section presents a formal protocol specification language, called APS (Agent Protocol Specifications), that hides many details to focus on address-passing. Its main novelty is an extension of
the FSM specification model with mechanisms for describing dynamic communication topologies.

A dynamic communication topology poses significant problems. First, a program must use addresses provided by the environment appropriately, i.e., it must send the correct messages to them at the correct times. Second, for an output to be correct, not only must it be structurally equivalent to some expected value, but the program must respond appropriately to messages received at addresses carried in the output. Intuitively, this situation is analogous to specifying 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.

$$\epsilon ::= (\text{goto } s u \ldots) \mid (\text{let-spec } (x \text{ goto } s u \ldots) \hat{S} \ldots) \hat{e} \quad \text{(Specification Expressions)}$$
$$\hat{S} ::= (\text{define-state } (s x \ldots) [\epsilon \rightarrow \hat{e}] \ldots) \quad \text{(Specification States)}$$
$$\epsilon ::= p \mid \text{unobs} \quad \text{(Events)}$$
$$u ::= x \quad \text{(Specification Arguments)}$$
$$p_o \in \text{OutPat ::= } p \mid \text{self} \quad \text{(Output Patterns)}$$

Fig. 6. APS syntax

Syntax and Intuitive Semantics Figure 6 presents the syntax of APS. The rest of this section describes its intuitive meaning.

APS specifications describe the communication behavior of a CSA agent starting from a given point in time, relative to some set of contextual addresses. Thus, a program may have multiple specifications, each taking a different perspective of the same agent by taking a different set of addresses and messages to be observed.

Like a CSA agent, each specification (created with let-spec) defines an initial expression and a set of states. Unlike in CSA, the only parameters allowed in specification states are the observed addresses on which the program can send messages to the environment. This design choice limits APS to an almost purely name-passing calculus and is the key innovation that allows our specifications to express interesting communication patterns without requiring sophisticated proofs of conformance.

Each clause in a state specifies a transition, consisting of a triggering event and an expression representing its effects and next state. A triggering event in APS may have more than one clause in the same state, allowing for indeterminate behavior.

Two kinds of events trigger transitions. An observed event $p$ is an input message received from the environment matching pattern $p$. Unlike in CSA, a name pattern $x$ in APS may only match an address (to maintain
the property that the only parameters are addresses). The next section defines the precise matching semantics.

An unobserved (unobs) event represents either an event that this specification cannot observe, such as a timeout or an internal message exchange, or an event that does not need to be distinguished. The observed/unobserved distinction enables us, for example, to specify the ABTP application-layer interface without specifying network messages.

Transitions cause two kinds of effects: output commitments and specification activations. The with-outputs expression generates output commitments \([u \ p_0]\) which represent outputs that must match \(p_0\) and that the program must eventually send to \(u\).

The let-spec expression activates a new specification instance, meaning that from that time forward, the program should also behave as described by the given specification, relative to the activation’s named address \(x\). The instance starts in the given initial state.

The activated instance’s name \(x\) is in scope for the body of let-spec for output commitment patterns, further activations, and the state transition. When used as an output pattern, the instance name specifies that the program sends that instance’s address to the environment (self does the same for the current instance). The environment then gains access to that instance’s address, so the program must respond appropriately to messages delivered to that address.

A Sample Specification Figure 7 is an excerpt from the application-layer specifications of Manager and Sender. Because it abstracts over parts of the implementation, the specification is much more concise than the program it specifies: 26 LOC versus almost 150, respectively. We expect this ratio to grow for sophisticated protocols such as TCP.

---

\[ \text{Fig. 7. ABTP application-layer specification} \]
ManagerSpec expresses that on receiving 'Connect, the program activates SenderSpec. The SenderSpec instance starts in the Connecting state, with the provided status used as an output address in that state.

SenderSpec specifies the application-layer behavior of an ABTP sender. Analogously to figure 3, figure 7 lists only the Connecting state. The first clause in Connecting indicates that a conforming program is allowed to send a 'Connected message to status and transition to the Connected state. The self pattern in the 'Connected output commitment specifies that whenever the environment sends a message to the provided address, the program should react as specified by SenderSpec. The second indicates that the program is also allowed to instead send 'ConnectFailed to status and transition to Closed.

Figure 8 illustrates SenderSpec in the style of a CFSM [7]. The part to the left of the slash in a transition label is the event that triggers the transition; the right part represents the output commitments and activations. Unobserved events are represented by •. The diagram omits the obvious state parameters in SenderSpec transitions. The Connecting node’s transitions match the transitions in SenderSpec in figure 7. The direct correspondence with the specification is suggestive of how APS exploits the mental model of CFSMs that protocol programmers already use. The diagram also illustrates the properties described at the end of the previous section: every 'Write command outputs to the included
response address, and every transition to Connected and Closed has an output to the status address s.

5 Conformance

Abstractly, an APS specification denotes all programs that implement it. To make this obvious statement concrete for APS/CSA, we introduce the notion of specification configurations and transitions between those, which we then relate to transitions between programs.

Specification Configurations A specification configuration describes the behavior that a running program is expected to have at a particular point in time. Figure 9 presents the formal grammar for specification configurations. In general, a metavariable with a “hat” denotes a feature that corresponds to the CSA feature denoted by the underlying metavariable, e.g., $\hat{K}$ denotes a program configuration, while $\hat{\hat{K}}$ denotes a specification configuration. A configuration $\hat{\hat{K}}$ consists of a map $\hat{\alpha}$ from specification addresses $\hat{a}$ to instances $z$ and a map $O$ from addresses to multisets of patterns, representing the output commitments for those addresses.

A specification instance consists of its state definitions, its current expression, and either an address $a$, indicating the specified agent’s address, or null if its address has not yet been observed.

We extend the language of output patterns to include specification addresses and agent addresses, which are substituted for variables.

Specification Configuration Transitions Figure 10 defines the transition relation $\rightarrow$ on specifications via labeled reduction rules. A config-
uration can either take an instance’s transition or output a message to satisfy a commitment, as indicated by label $l$.

$$
\begin{align*}
\overset{r^*}{\tilde{K}} \xrightarrow{s} \tilde{K}' \quad \#\tilde{K}'' & \quad \overset{s}{\tilde{K}'} \xrightarrow{s} \tilde{K}'' \\
\tilde{K} \xrightarrow{l} \tilde{K}'
\end{align*}
$$
\text{(S-EVENT)}

$$
\begin{align*}
v \subseteq_\alpha p_\alpha, \alpha \triangleright \{\hat{a'} \mapsto a', \ldots\} \\
\langle \tilde{\alpha}, O[a \mapsto \{p_\alpha, p'_\alpha, \ldots\}] \rangle \xleftarrow{a^\alpha} \langle \tilde{\alpha}[a' \leftarrow a'], O[a \mapsto \{p'_\alpha, \ldots\}] \rangle
\end{align*}
$$
\text{(S-OUTPUT)}

**Fig. 10.** Specification transition rules

S-EVENT says that if a specification can choose a transition $l$ from the current configuration and fully evaluate it with a number of internal ($\tau$) expression-level steps, then there is also a configuration-level step that encompasses all the effects of that transition.

S-OUTPUT says that a configuration can step by sending a message matching one of its output commitments on the appropriate address. The output pattern matching relation, written $v \subseteq_\alpha p_\alpha, \alpha \triangleright \{\hat{a'} \mapsto a', \ldots\}$, maps instance addresses to corresponding agent addresses provided in the message. With a slight abuse of notation, S-OUTPUT uses the generated mapping to substitute the addresses for null in the corresponding tuples in $\tilde{\alpha}$. If an instance already has a corresponding address, the matched address must be the same. The rest of the matching rules are standard:

$$
\begin{align*}
v \subseteq_\epsilon \epsilon \triangleright \emptyset \\
t \subseteq_\alpha t, \alpha \triangleright \emptyset \\
a \subseteq_\alpha \hat{a}, \tilde{\alpha}[\hat{a} \mapsto \langle (\hat{S} \ldots), \hat{e}, \text{null} \rangle] \triangleright \{\hat{a} \mapsto a\}
\end{align*}
$$

$$
\begin{align*}
\langle \hat{\alpha}, O[a \mapsto \{p_\alpha, p'_\alpha, \ldots\}] \rangle \xleftarrow{a^\alpha} \langle \hat{\alpha}[a' \leftarrow a'], O[a \mapsto \{p'_\alpha, \ldots\}] \rangle
\end{align*}
$$

Figure 11 defines the expression-level steps for specification configurations. T-UNOBS says that when an instance performs a goto, it can take any of the state’s unobs transitions, substituting the state parameters into the body. T-IN is similar, except that it requires the specification’s concrete address to be known and matches the received message against the transition’s pattern, generating substitutions to use in the body:

$$
\begin{align*}
v \subseteq_i \bullet \triangleright \emptyset \\
t \subseteq_i t \triangleright \emptyset \\
a \subseteq_i x \triangleright \{x \mapsto a\}
\end{align*}
$$

$$
\begin{align*}
\langle \hat{\alpha}, O[a \mapsto \{p_\alpha, p'_\alpha, \ldots\}] \rangle \xleftarrow{a^\alpha} \langle \hat{\alpha}[a' \leftarrow a'], O[a \mapsto \{p'_\alpha, \ldots\}] \rangle
\end{align*}
$$
\[ \langle \hat{a} \mapsto \ldots , O \rangle \]

\[ T\text{-Unobs} \]

\begin{align*}
\langle \hat{S} \ldots \rangle, (\text{goto } s \ a \ldots ) , \sigma \quad & \xrightarrow{\circ} \quad \langle \hat{S} \ldots \rangle, \hat{e}[x \leftarrow a] \ldots , \sigma \\
\text{if } (\text{define-state } (s \ x \ldots ) \ldots [\text{unobs } \rightarrow \hat{e}] \ldots ) \text{ is in } \hat{S} \ldots &
\end{align*}

\[ T\text{-In} \]

\begin{align*}
\langle \hat{S} \ldots \rangle, (\text{goto } s \ a \ldots ) , a'' \quad & \xrightarrow{a'' v} \quad \langle \hat{S} \ldots \rangle, \hat{e}[x \leftarrow a] \ldots [x' \leftarrow a'] \ldots , \hat{a}'' \\
\text{if } (\text{define-state } (s \ x \ldots ) \ldots [p \rightarrow \hat{e}] \ldots ) \text{ is in } \hat{S} \ldots \text{ and } v \subseteq p \triangleright \{x' \mapsto a', \ldots \} &
\end{align*}

\[ T\text{-WithOutputs} \]

\begin{align*}
\langle \hat{S} \ldots \rangle, (\text{with-outputs } (\{a\ p_0\} \ldots ) \hat{e}) , \sigma \quad & \xrightarrow{\tau} \quad \langle \hat{S} \ldots \rangle, \hat{e}, \sigma , \ O[a \rightarrow \{p_0\}] \ldots \\
\end{align*}

\[ T\text{-LetSpec} \]

\begin{align*}
\langle \hat{S} \ldots \rangle, (\text{let-spec } (x \ e' \ \hat{S}' \ldots ) \hat{e}) , \sigma \quad & \xrightarrow{\tau} \quad \langle \hat{S} \ldots \rangle, \hat{e}[x \leftarrow \hat{a'}] , \hat{e}'[\text{self } \leftarrow \hat{a'}] , \null , \left[ \hat{a'} \mapsto z \right] \\
\text{where } z = \langle \hat{S}'[\text{self } \leftarrow \hat{a'}] \ldots , \hat{e}'[\text{self } \leftarrow \hat{a'}] , \null \rangle , \text{ such that } \hat{a'} \notin (\text{dom}(\hat{a}) \cup \hat{a}) &
\end{align*}

Fig. 11. Specification expression reduction rules

T-\text{WithOutputs} adds the specified output commitments to the output commitment set. Finally, T-\text{LetSpec} creates a new instance with a fresh address \( \hat{a'} \), which is substituted for the given name in the original instance and self in the new instance.

**The Conformance Relation** Roughly, conformance means that the program takes only the steps allowed by the specification and eventually produces every output required by the specification. A relation \( K \vdash \hat{K} \) captures this notion, such that a program configuration \( K \) conforms to the specification instance \( z = \langle \hat{S} \ldots \rangle, (\text{goto } s \ a \ldots ) , \sigma \rangle \) if and only if \( K \vdash \langle \{\hat{a} \mapsto z\} \rangle \{a \mapsto \emptyset \ldots \} \rangle \) for some unique \( \hat{a} \). Formally, we define \( \vdash \) as the largest relation between \( K \) and \( \hat{K} \) such that:

1. If \( K \xrightarrow{a?v} K' \), \( \text{obs}(a?v, \hat{K}) \), and \( \text{addresses}(v) \cap \text{mentions}(K) = \emptyset \), then \( \hat{K} \xrightarrow{a?v} \hat{K}' \) and \( K' \vdash \hat{K}' \)
2. If \( K \xrightarrow{a!v} K' \) and \( \text{obs}(a!v, \hat{K}) \), then \( \hat{K} \xrightarrow{a!v} \hat{K}' \) and \( K' \vdash \hat{K}' \)
3. If \( K \xrightarrow{l} K' \), \( \text{obs}(l, \hat{K}) \) does not hold, and \( \text{addresses}(v) \cap \text{mentions}(K) = \emptyset \) if \( l = a?v \), then either \( K' \vdash \hat{K} \) or \( \hat{K} \xrightarrow{\circ} \hat{K}' \) and \( K' \vdash \hat{K}' \)
4. Let $\hat{K} = \langle \cdot, O \rangle$. For every fair run $r$ that starts with $K$, there is some finite prefix $r'$ of $r$ that satisfies all commitments in $O$.

We explain and refine each of the rules in turn. Rule 1 says that if the program can accept a value $v$ at address $a$, and the specification observes the transition, then the specification can take a matching transition.

Observability of a transition $l$ is defined relative to a specification configuration. An input transition is observed if the communication is on a known input address of the specification configuration and has a matching specification transition; inputs to known addresses without a matching pattern are left unspecified. An output transition is observed if the address has an entry in the commitment map $O$:

\[
\text{(define-state (s \ldots) \ldots [p_i \rightarrow \ldots] \ldots) } \in \{S, \ldots\} \quad v \subseteq_i p_i \triangleright - \quad \text{obs}(a?v, (\hat{\alpha}[\hat{a} \mapsto ((S\ldots), (\text{goto s} \ldots), a)], O))
\]

\[
\frac{a \in \text{dom}(O)}{\text{obs}(a!v, ⟨\hat{\alpha}, O⟩)}
\]

Rule 1 (as well as rule 3) applies to only those input messages that do not contain addresses already mentioned in the program configuration, to avoid conflating uses of addresses received at different times. The addresses mentioned by a configuration $K$ are the addresses of all agents in $K$ plus $K$’s set of externals $\chi$.

Rule 2 is the analogue of rule 1 for outputs.

Rule 3 says that if the program takes some step that the specification does not observe, then either the specification is already related or it can take a single internal step to transition to a related state.

The explanation of rule 4 requires three definitions. A run $r$ is a (possibly infinite) reduction sequence of the form $K \xrightarrow{l} K' \xrightarrow{l'} \ldots$. A fair run is a run that includes a transition on $l$ ($\neq a?v$) infinitely often if $l$ is possible at infinitely many configurations in the run. We say that a run $r$ satisfies all commitments in $O$ if there is a finite-length prefix of $r$ such that for each $a$ mapped in $O$ and for each $p_o$ associated with that $a$, $r$ outputs a unique message on $a$ that matches $p_o$. The rule requires all fair runs of a conforming program to eventually satisfy all of the current commitments, but not in any particular order or amount of time.
6 Proving Conformance

The ABTP Manager conforms to ManagerSpec. This section briefly describes the general technique for proving this kind of statement; appendix A contains the full proof.

The strategy is to prove conformance for pairs of agent/specification states, starting with some initial state and initial address mapping. The starting point for ABTP is the Ready state of Manager and the Ready state of ManagerSpec, with ManagerSpec’s address being the same as Manager’s. In this state, we prove conformance rules 1–3 individually by showing that certain possible transitions in the program have a matching transition in the specification. Each match generates an obligation to prove conformance for the pair of states where the transitions end. For rule 4, we show that every run from this point satisfies all outstanding output commitments; this initial state pair has no output commitments.

This process continues for each state-pair obligation generated by the proof. The proof proceeds by coinduction on program configuration transition steps; that is, the proof is complete once we have conformance for every reachable state pair.

As described so far, however, such a proof would involve many tedious proofs for pairs of states that differ by only a single internal step. We alleviate this issue with two Handler metalemmas, which state that if some program configuration $K$ conforms to a specification configuration $\hat{K}$, and if some other program configuration $K'$ can reach $K$ by evaluating only internal and output steps, then $K'$ also conforms to $\hat{K}$, modulo any satisfied output commitments. In this way, the proof need only describe those pairs in which the relevant agent is waiting to receive a message.

Similarly, if a program has more than one agent running at a time, the proof must check every possible interleaving of those agents’ transition steps. We address this with a Disjoint metalemma, which states that if two independent halves of a program do not share addresses and thus cannot communicate with each other, and if each half conforms to some specification, then the whole program conforms to a combination of those two specifications. In ABTP, this allows us to prove the conformance for Sender separately from that of Manager at the point where Manager spawns a new Sender.

7 Related Work

SDL [23] 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** [22] formalize and extend SDL-like diagrams with arbitrary mathematical constraints. Smith and Talcott present a refinement relation between specifications similar to our treatment of conformance, although the arbitrary constraints often lead to complex proofs. In contrast, our finite-state specifications and state-based program structure makes the correspondence between a program and its specification almost obvious.

**Communicating finite-state machines** [7] (CFSMs) are a common model for communication protocols. CSA and APS are inspired by this model, as it matches the way programmers think about protocols and programs. CFSMs cannot spawn new CFSMs, however, restricting them to a static topology.

**Session types** [15] are another well-studied means for specifying communication between processes, including dynamic communication topology. Indeed, some of the work in this area uses CFSMs as specifications [13].

Session types can express properties similar to ours, but they require that every process have a type compatible with that of its communication partners in order to guarantee progress of each session. We consider this approach impractical as it requires all conceivable communication patterns be expressible as session types. This precludes, for instance, common situations involving a dynamic number of communication channels and processes, such as the ABTP Manager that can accept requests from unknown application-layer clients. Our work trades off progress guarantees in favor of these additional communication patterns.

The fixed nature of an actor’s address 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 multiple, possibly simultaneous, sessions.

Three projects have added session types to actor models. Crafa [11] presents a type system that checks the communication actions of actors that halt within a fixed number of steps. Mostrous and Vasconcelos [19] add a session type system with restricted message structure to Erlang that verifies the stated protocols, preventing incomplete sessions and un-
received messages. Neykova and Yoshida [20] propose a hybrid actor/session type system that requires a central message broker to connect actors. The broker does not allow evolving communication topology.

**Contracts for mobile processes** [9] are another behavioral type system for specifying dynamic communication behavior. It is not clear whether contracts can express APS features such as unobservability and output commitments.

**The first-order \( \mu \text{-calculus} \)** [12] can specify protocols with dynamic communication topology in Erlang, at the cost of complex proofs and verbose formulas needed to express every possible ordering of output messages.

**I/O automata** [17] specify distributed algorithms as communicating automata, generalizing APS with infinitely many states and arbitrary transition relations between them, again at the cost of complex proofs. For example, expressing dynamic communication topology requires reasoning about infinitely many possible messages. Also, there is no mechanism for directly specifying that a program *eventually* sends a message.

The *ioco* relation [25] defines conformance for a variant of I/O automata in terms of testing theory. *ioco* cannot directly express dynamic communication topologies, but some of its ideas may improve our work. We plan to explore this connection further.

**Typestate** [24] ensures that clients invoke an object’s methods only in states in which they are valid. An actor, however, may have multiple, concurrent clients, making it difficult to track the actor’s state from the perspective of any given client. APS accommodates concurrent interaction with non-determinism in the specifications and a reduction-semantics-based conformance relation

**Verification and testing tools** are often used to find errors in Erlang programs. Dialyzer [10] detects general communication errors but not violations of a specific protocol. Other tools check a program against a specification, but none of them are based on a specification language that is both as high-level as ours and able to express dynamic communication topologies. For instance, the QuickCheck-inspired PropEr [21] randomly tests for properties expressed as Erlang functions. etomcrl [4] translates programs into a process algebra form so that a model checker can check them against \( \mu \text{-calculus} \) formulas. Future work may explore similar approaches to automatically check CSA programs against APS specifications.
8 Conclusions and Future Work

Abstract specifications of protocols allow developers to separate the expected behavior of a component from its implementation details. Specifications for actor-based languages in particular require a means to describe behavior involving dynamic communication topology.

APS is a specification language that fulfills this need for actors structured as FSMs. APS combines the well-known communicating-finite-state-machine model with facilities for specifying communication to addresses embedded in messages. Similarly, the programming language CSA merges actors with finite-state machines, giving structure to a ubiquitous pattern in real-world frameworks such as Erlang and Akka. As section 6 illustrates, conformance proofs can take advantage of this structure to factor out the proof into separate arguments on pairs of program/specification states, leading to comprehensible, systematic proofs.

Because conformance in APS is defined only in terms of a program configuration’s transition labels and not its internal finite-state structure, we conjecture that the ideas in APS can scale up to full actor languages. Furthermore, we conjecture that the proof technique can apply to any program in such languages that uses the FSM pattern.

The structure and relative simplicity of conformance proofs in our framework suggest that these checks are automatable. Our next step is to develop a conservative static analysis for APS that automatically detects when a program is going to violate its specification.

Acknowledgements This work was supported by a gift from Cisco and a grant from the NSF. The authors kindly acknowledge comments on early drafts by Tony Garnock-Jones, Olin Shivers, and Vincent St-Amour.

References

A Full Conformance Proof

This appendix presents a detailed proof that the ABTP Manager conforms to ManagerSpec. It also formally states and proves the Handler and Disjoint metalemmas.

A.1 Handler Metalemmas

The first two metalemmas allow skipping of intermediate states of an agent while it is evaluating some handler expression. Intuitively, if a program configuration \( K \) conforms to some specification configuration \( \hat{K} \), and if another program configuration \( K' \) can take an intermediate handler step to \( K \), \( K' \) should also conform to \( \hat{K} \) modulo any satisfied output commitment, because handler steps are not observable to the environment.

**Handler metalemma 1.** If \( K \models K' \) and \( K' \xrightarrow{l} K \) where \( l \) is of the form \( a: \text{func} \) or \( a!v \) such that \( \text{obs}(a!v, \hat{K}) \) does not hold, then \( K' \models \hat{K} \).

**Handler metalemma 2.** If \( K \models \hat{K} \), \( K' \xrightarrow{a!v} K \), and \( \hat{K}' \xrightarrow{a!v} \hat{K} \), then \( K' \models \hat{K}' \).

The proof of Handler metalemma 1 proceeds by coinduction on program reduction steps and examines each rule of the conformance judgment separately. In the following, let \( a \) be the address for the agent that transitions with label \( l \).

The first three conformance rules deal with how the specification reacts if the program \( K' \) takes a step. We consider two cases: either the program takes the step \( l \) and becomes \( K \), or it takes some other step. If it takes \( l \), it’s an unobservable step, so the specification takes no transitions and we are left with proving \( K \models \hat{K} \), which we already know. Otherwise, if \( K' \) takes some other step \( l' \) to \( K'' \), then \( K \) must be able to take the same step to \( K''' \), because all agents in the configuration other than \( a \) must be identical, and \( a \) cannot take any step other than \( l \). We have that \( K'' \xrightarrow{l} K''' \), so by coinduction, if \( K''' \models \hat{K}' \) for some \( \hat{K}' \), then \( K'' \models \hat{K}' \).

For rule 4, every fair run of \( K' \) must eventually take the step \( l \). Therefore, the fair runs of \( K' \) are the fair runs of \( K \) with an extra \( l \) step, and so they satisfy the same commitment sets.

The proof for Handler metalemma 2 is similar, except that it removes the output commitment from the specification when it take the step \( a!v \).
A.2 Disjoint Metalemma

The third metalemma allows breaking down specification configurations, enabling conformance proofs for one specification at a time. If two independent halves of a program do not share addresses and thus cannot communicate with each other, and if each half conforms to some specification, the whole program should conform to a combination of those two specifications.

To state this metalemma, we must first define what it means for two program configurations to share no addresses. We say that $K$ and $K'$ are address-disjoint iff $\text{mentions}(K) \cap \text{mentions}(K') = \emptyset$. We also define $K \sqcup K'$, the composition of two address-disjoint configurations, to be the configuration containing the union of the agents, messages, receptionists, and externals in $K$ and $K'$.

Similarly, two specification configurations $\hat{K}$ and $\hat{K}'$ are address-disjoint iff the addresses in the specification instances (i.e. the state definitions, current behavior expressions, and corresponding program addresses) of $\hat{K}$ are disjoint from those of $\hat{K}'$. Thus, $\langle \hat{\alpha}, O \rangle \sqcup \langle \hat{\alpha}', O' \rangle$ is $\langle \hat{\alpha} \cup \hat{\alpha}', O \cup O' \rangle$.

We also say that a configuration $\alpha \mid \mu \chi$ is valid iff every address that appears in a behavior expression or state definition in $\alpha$ or a message in $\mu$ is either the address of an agent in $\alpha$ or a member of $\chi$.

**Disjoint metalemma.** Let $\tilde{K}_i = \langle \hat{\alpha}_i, O_i \rangle$ and $K_i = \alpha_i \mid \mu_i \chi_i$ for $i \in \{1, 2\}$. Assume $K_1$ and $K_2$ as well as $\tilde{K}_1$ and $\tilde{K}_2$ are address-disjoint, and let $K_3 = K_1 \sqcup K_2$ and $\tilde{K}_3 = \tilde{K}_1 \sqcup \tilde{K}_2$. If $K_i$ is valid, $\text{dom}(O_i) \subseteq \chi_i$ and $K_i \vDash \tilde{K}_i$ for $i = 1, 2$, then $K_3 \vDash \tilde{K}_3$.

Similar to the Handler metalemmas proof, the proof is by coinduction on program reduction steps, examining each conformance judgment rule individually. For rules 1–3, consider a transition of the form $(K_1 \sqcup K_2) \xrightarrow{l} (K'_1 \sqcup K_2)$. Because $K_1 \vDash \tilde{K}_1$, we know that the transition $K_1 \xrightarrow{l} K'_1$ has a matching specification transition $\tilde{K}_1 \xrightarrow{i} \tilde{K}_1'$. We also know that $\tilde{K}_1$ and $\tilde{K}_2$ are address-disjoint, so we have $(\tilde{K}_1 \sqcup \tilde{K}_2) \xrightarrow{i} (\tilde{K}_1' \sqcup \tilde{K}_2)$. Because we do not consider transitions in which the environment provides an address owned by the program back to the program (see the definition of rule 1), $K'_1$ and $K_2$ must still be address-disjoint, as are $\tilde{K}_1'$ and $\tilde{K}_2$. Therefore, by coinduction, we have $K'_1 \sqcup K_2 \vDash \tilde{K}_1' \sqcup \tilde{K}_2$. A similar argument holds for transitions on $K_2$.  

22
For rule 4, every fair run of the combined configuration must contain a fair run of $K_1$ and a fair run of $K_2$. Because each of those configurations conform to their corresponding specifications, we know that their fair runs satisfy all output commitments in those specifications. Therefore, a run that combines a fair run from each configuration satisfies all output commitments in $\hat{K}_1 \sqcup \hat{K}_2$.

A.3 Conformance for ABTP

The proof that the Manager agent in figure 3 conforms to ManagerSpec in figure 7 proceeds by coinduction on program configuration transition steps. The key strategy is to prove conformance for pairs of agent/specification states, starting with some initial state and initial address mapping and proving conformance for other state pairs as necessary. We prove conformance for two state pairs to illustrate the idea.

The starting point is the Ready state of Manager, say $K$, and the Ready state of ManagerSpec, with the specification instance’s address assigned to the agent’s address, say $\hat{K}$. For each corresponding pair of states, we establish each rule of the conformance relation in turn.

Manager: Ready/ManagerSpec: Ready

Rule 1 The only observed event in this state is to receive a 'Connect message. This transition in ManagerSpec activates an instance of SenderSpec and transitions back to Ready. SenderSpec starts in the Connecting state, saving the status address from the request. Let $\hat{K}'$ be the resulting specification configuration, and let $K'$ be the program configuration after taking the receive step. We must show that $K' \models \hat{K}'$.

Let $K''$ be the configuration $K'$ reaches after eventually transitioning by E-SPAWN to spawn a new Sender. By repeated applications of the Handler metalemmas, if we can show that $K'' \models \hat{K}'$, we know that $K' \models \hat{K}'$.

To prove $K'' \models \hat{K}'$, we use the Disjoint metalemma to reason about Manager and Sender independently. Let $K''_m$ be the configuration containing just the Manager agent from $K''$, while $K''_s$ is the configuration containing the Sender agent. Similarly, let $\hat{K}'_m$ be the configuration containing just the ManagerSpec specification, while $\hat{K}'_s$ contains the SenderSpec specification. The commitment map $O$ in $\hat{K}'_m$ is empty, but the one in $\hat{K}'_s$ maps the address for status to an initial empty set of commitments. We have that $K'' = K''_m \sqcup K''_s$ and $\hat{K}' = \hat{K}'_m \sqcup \hat{K}'_s$. 

23
so if we show that $K''_m \models \widehat{K}_m'_{'}$ and $K''_s \models \widehat{K}_s'_{'}$, then by the Disjoint metalemma we have $K'' \models \widehat{K}'_m$.

We can see that $K''_m = K$ and $\widehat{K}_m'_{'} = \widehat{K}_m$, which is an instance of this case, so by coinduction we have $K''_m \models \widehat{K}_m'_{'}$.

To prove $K''_s \models \widehat{K}_s'_{'}$, let $K''_s$ be the configuration that $K''_s$ reaches when it finishes the Sender initialization and the agent’s behavior expression is reduced to $(\text{rcv \ (msg)})$...). The initialization has no observable output actions, so by repeated application of Handler metalemma 1, if $K''_s \models \widehat{K}_s'_{'}$, $K''_s \models \widehat{K}_s'_{'}$. For the proof of $K''_s \models \widehat{K}_s'_{'}$, see the section on Sender/SenderSpec below.

Rules 2, 3 $K$ cannot take any output or unobservable steps.

Rule 4 There are no output commitments in $\widehat{K}$.

**Sender: Connecting/SenderSpec: Connecting** Let $K$ be the program configuration with only a Sender agent in the Connecting state. Let $\widehat{K}$ be a specification configuration containing only SenderSpec in the Connecting state, mapping status to the corresponding Sender address, and mapping the address for status to the empty set in the commitment map. We prove that in this context, $K \models \widehat{K}$.

Rule 1 There are no possible observed input events, because Sender has not yet sent the Connected message that includes its address.

Rule 2 $K$ cannot take an output transition step from this state.

Rule 3 $K$ has two possible unobservable steps: it can either receive an unobserved message, or the timeout can fire. If the agent receives a message, the agent pattern-matches on it. If the message is a 'SynAck with the proper format, the agent sends on status a 'Connected message with its own address and transitions to Connected. This corresponds to the first unobs transition in SenderSpec’s Connecting state, which transitions to Connected. By another use of the Handler metalemmas, all that remains is to show that Sender in the Connected state conforms to SenderSpec in Connected, which follows from similar reasoning.

For all other messages, the agent transitions back to Connecting. In that case, the specification does not transition at all, and we must prove $K \models \widehat{K}$. The latter is an instance of the current case, so by coinduction, the judgment holds.

Otherwise, if the timeout fires, the agent sends 'ConnectFailed on status and transitions to Closed, which corresponds to the second unobs transition in SenderSpec’s Connecting state. By the Han-
dler metalemmas, the 'ConnectFailed' message satisfies the generated output commitment, and all that remains is to show that Sender in Closed conforms to SenderSpec in Closed. This again follows from similar reasoning.

**Rule 4** There are no output commitments in $\hat{K}$. Handler metalemma 2 ensures that the handler in Connecting satisfies any commitments generated by transitions from the current state, as described in the argument for rule 3.

The proofs for the other state pairs are similar. □
B Substitution

This appendix defines the capture-avoiding substitution for both CSA and APS. The only non-standard cases are spawn and let-spec, which avoid substituting for self in certain positions to avoid overwriting a different agent/instance’s self.

B.1 Substitution in CSA

\[
\begin{align*}
x[x ← v] & = v \\
x'[x ← v] & = x' \quad \text{if } x \neq x' \\
n[x ← v] & = n \\
t[x ← v] & = t \\
a[x ← v] & = a \\
(spawn e S...)\{self ← v\} & = (spawn e S...) \\
(spawn e S...)\{x ← v\} & = (spawn e[x ← v] S[x ← v]...) \\
& \quad \text{if } x \neq \text{self} \\
(goto s e...\{x ← v\}) & = (goto s e[x ← v]...) \\
(send e e')\{x ← v\} & = (send e[x ← v] e'[x ← v]) \\
(let ([x' e]...) e')\{x ← v\} & = (let ([x' e[x ← v]]...) e') \\
& \quad \text{if } x \in \{x',\ldots\} \\
(let ([x' e]...) e')\{x ← v\} & = (let ([x' e[x ← v]]...) e'[x ← v]) \\
& \quad \text{if } x \notin \{x',\ldots\} \\
(match e [p e']...)\{x ← v\} & = (match e[x ← v] [p e'][x ← v]...) \\
(list e...\{x ← v\}) & = (list e[x ← v]...) \\
(rcv (x) e)\{x ← v\} & = (rcv (x) e) \\
(rcv (x') e)\{x ← v\} & = (rcv (x') e[x ← v]) \\
& \quad \text{if } x \neq x' \\
(rcv (x) e [(timeout n) e']\{x ← v\}) & = (rcv (x) e [(timeout n) e']) \\
(rcv (x') e [(timeout n) e']\{x ← v\}) & = \\
& \quad (rcv (x') e[x ← v] [(timeout n) e'[x ← v]]) \quad \text{if } x \neq x'
\end{align*}
\]

We also extend substitution to match clauses and state definitions:

\[
[p e]\{x ← v\} = \begin{cases} 
[p e] & \text{if } p \text{ binds } x \\
[p e[x ← v]] & \text{if } p \text{ does not bind } x
\end{cases}
\]
(define-state (s x_1 ... x_n) (x_h) e)[x ← v] =
(define-state (s x_1 ... x_n) (x_h) e)
if \( x \in \{x_1, \ldots, x_n\} \)

(define-state (s x_1 ... x_n) (x_h) e)[x ← v] =
(define-state (s x_1 ... x_n) (x_h) e[x ← v])
if \( x \notin \{x_1, \ldots, x_n\} \)

(define-state (s x_1 ... x_n) (x_h) e)[timeout n][x ← v] =
(define-state (s x_1 ... x_n) (x_h) e'[x ← v])
if \( x \in \{x_1, \ldots, x_n\} \)

(define-state (s x_1 ... x_n) (x_h) e)[timeout n][x ← v] =
(define-state (s x_1 ... x_n) (x_h) e'[x ← v])
if \( x \notin \{x_1, \ldots, x_n\} \)

B.2 Substitution in APS

Substitution in APS is defined on output patterns, arguments, and specification expressions. We let \( \hat{v} \) stand for addresses \( a \) and specification addresses \( \hat{a} \), both of which are used in APS substitutions.

\[
x'[x ← \hat{v}] = \hat{v} \\
x'[x ← \hat{v}] = x' \quad \text{if } x \neq x' \\
a[x ← \hat{v}] = a \\
\hat{a}[x ← \hat{v}] = \hat{a} \\
t[x ← \hat{v}] = t \\
\ast[x ← \hat{v}] = \ast \\
(list p ...)[x ← \hat{v}] = (list p[x ← \hat{v}] ...)
\]

(goto s u ...)[x ← \hat{v}] = (goto s u[x ← \hat{v}] ...)

(let-spec (x (goto s u ... ) \( \hat{S} \) ... ) \( \hat{e} \))[x ← \hat{v}] =

(let-spec (x (goto s u[x ← \hat{v}] ... ) \( \hat{S} \) ... ) \( \hat{e} \))

(let-spec (x (goto s u ... ) \( \hat{S} \) ... ) \( \hat{e} \))[self ← \hat{v}] =

(let-spec (x (goto s u ... ) \( \hat{S} \) ... ) \( \hat{e} \))[self ← \hat{v}]

(let-spec (x'[x ← \hat{v}] ... ) \( \hat{S} \) ... ) \( \hat{e} \))[x ← \hat{v}] =

(let-spec (x'[x ← \hat{v}] ... ) \( \hat{S} \) ... ) \( \hat{e} \))[x ← \hat{v}]

if \( x \notin \{x'[x ← \hat{v}] \ldots \} \)

(with-outputs (u p_0 ... ) \( \hat{e} \))[x ← \hat{v}] =

(with-outputs (u[x ← \hat{v}] p_0[x ← \hat{v}] ... ) \( \hat{e} \))[x ← \hat{v}] \)