Third-Party Composition of AOP Mechanisms

Sergei Kojarski

M.Sc., Northeastern University, 2004
B.Sc., Perm State Technical University, 2001

A thesis submitted to the
Faculty of the Graduate School of
Northeastern University in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
College of Computer and Information Science
2008
This thesis entitled:
Third-Party Composition of AOP Mechanisms
written by Sergei Kojarski
has been approved for the College of Computer and Information Science

David H. Lorenz  5/28/08

Gene Cooperman  5/28/08

Erik Ernst  7/04/08

David Kaeli  5/28/08

Karl Lieberherr  5/28/08

Date  8/8/08

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.
Abstract

Domain-specific aspect-oriented language extensions offer unique capabilities to deal with a variety of crosscutting concerns. In principle, one should be able to use several of these extensions together in a single program. In practice, however, each extension implements its own specialized weaver and the different weavers are incompatible. Even if the weavers were compatible, combining them is a difficult problem in general, because each extension defines a new language with its own semantics. In this dissertation work, we introduce, analyze and present a solution to the aspect extension composition problem. We present a scalable, general, and practical framework, named AWESOME, for composing aspect weavers. To be scalable, the AWESOME framework supports third-party composition of aspect weavers. To be general, the framework provides means for customizing the composition behavior. Furthermore, to be practically useful, there is no framework-associated degradation in the performance of compiled aspect programs.
In memory of John Vlissides —

— I was greatly privileged to have had him on my Committee
Acknowledgements

I thank my mother for supporting me in my education pursuits. Despite difficult times that our family faced in 1990s my mother managed to give me the best education our home city has to offer. She then stimulated me to apply for PhD studies in United States, and fully supported me in my preparation for graduate exams. During my PhD studies she has been continuously encouraging me in my education and research efforts.

I thank my advisor Professor David H. Lorenz for his outstanding contribution to my professional establishment. David invested enormous amount of time and effort into development of my research skills. I spent most of my studentship standing next to a whiteboard in David’s office, working with him on technical papers, and discussing research ideas. It is this collaboration that taught me a discipline of thought, methods of analysis, and presentation techniques that are fundamental for a researcher.

David not only developed my skills but also did an exceptional job to ensure that I’m fully supported throughout the duration of my studies. This effort goes far beyond regular administrative activities of a PhD advisor. I’m extremely thankful to David for this support. It gave me peace of mind and time necessary to go deep into the research work.

I also thank the other members of my thesis committee, Professors Gene Cooperman, Eric Ernst, David Kaeli, Karl Lieberherr and Dr. John Vlissides, for their valuable comments. I am particularly grateful to Professor Cooperman whose goal was always to support my own goals throughout the completion of this dissertation.

The generous support of the National Science Foundation is acknowledged.
## Contents

<table>
<thead>
<tr>
<th>Sections</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>vi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiii</td>
</tr>
<tr>
<td>List of Listings</td>
<td>xv</td>
</tr>
<tr>
<td>List of Terms</td>
<td>xviii</td>
</tr>
<tr>
<td>List of Symbols</td>
<td>xxiii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Thesis</td>
<td>2</td>
</tr>
<tr>
<td>1.2 The Aspect Extension Composition Problem</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1 The component abstraction problem</td>
<td>4</td>
</tr>
<tr>
<td>1.2.2 The composition specification problem</td>
<td>4</td>
</tr>
<tr>
<td>1.2.3 The composition implementation problem</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Main Results</td>
<td>6</td>
</tr>
<tr>
<td>1.3.1 Pluggable AOP</td>
<td>6</td>
</tr>
<tr>
<td>1.3.2 The component abstraction problem</td>
<td>7</td>
</tr>
<tr>
<td>1.3.3 The composition specification problem</td>
<td>7</td>
</tr>
<tr>
<td>1.3.4 The composition implementation problem</td>
<td>8</td>
</tr>
</tbody>
</table>
1.4 Structure of the Dissertation ........................................... 8

2 State of the Art ............................................................... 10

2.1 Aspect-Oriented Languages ............................................. 10

2.1.1 AspectJ ................................................................. 10

2.1.2 AspectWerkz ........................................................ 14

2.1.3 COOL ................................................................. 15

2.1.4 Hyper/J ............................................................... 18

2.2 Modeling AOP ............................................................. 22

2.3 Composition of Aspects .................................................. 25

2.4 Composition of Aspect Extensions ..................................... 26

2.4.1 Reflex ................................................................. 26

2.4.2 XAspects ........................................................... 29

2.4.3 AspectJ/5 ............................................................ 30

3 Motivation ........................................................................ 33

3.1 Significance of the Composition Problem ............................ 33

3.1.1 Aspect DSLs ......................................................... 35

3.1.2 Reuse ................................................................. 35

3.2 Lack of Support for Composing Aspect Extensions ............... 36

3.2.1 AspectJ/5 ............................................................ 37

3.2.2 Reflex ............................................................... 39

3.2.3 XAspects ........................................................... 41

3.3 Complexity of the Composition Problem ............................ 43

3.3.1 No Correct Translation ............................................ 44

3.3.2 No Correct Sequential Instrumentation ........................ 45

3.4 Summary ..................................................................... 46
4 The Component Abstraction Problem

4.1 Evolution of AOP Models ............................................. 49

4.2 Analysis ............................................................... 51

4.2.1 The Weaving Problem .............................................. 52

4.2.2 The Weaving Plan .................................................. 52

4.2.3 The Weaving Process .............................................. 53

4.2.4 Notation ............................................................... 55

4.3 Design of a Nonreactive Weaving Process ......................... 56

4.4 Design of a Reactive Weaving Process ............................ 58

4.5 The Duality of Open Classes ........................................ 62

4.5.1 A Nonreactive OC Weaving Process .............................. 62

4.5.2 A Reactive OC Weaving Process ................................. 62

4.6 Composing Aspect Mechanisms ..................................... 65

4.6.1 Nonreactive extensions ............................................ 65

4.6.2 Reactive extensions ................................................ 65

4.7 Beyond Conceptual Models ......................................... 67

4.7.1 Abstract Domains .................................................. 67

4.7.2 The Weaving Process as a Unifying Concept .................. 68

4.7.3 Cyclic Dependency in AspectJ ..................................... 68

4.7.4 Advising Advice Execution ....................................... 68

4.7.5 Untangling the Base and the Aspect Mechanisms ............. 69

4.8 Top-Down Classification ............................................. 69

4.8.1 Symmetric versus Asymmetric Languages ...................... 69

4.8.2 Static versus Dynamic Languages ............................... 70

4.8.3 Static versus Dynamic Semantics ................................ 71

4.9 Top-Down Construction ............................................. 71

4.9.1 Component-Based Design ......................................... 71
4.9.2 New AOP Functionality .............................................. 72
4.10 Summary ............................................................... 73
5 The Composition Specification Problem 75
5.1 Scope ................................................................. 76
5.2 Features .............................................................. 77
  5.2.1 Join Point Features ............................................... 78
  5.2.2 Advice Features .................................................. 79
5.3 Patterns of Interaction ............................................... 81
  5.3.1 Influencing versus Inducing ..................................... 82
  5.3.2 Genuine versus Incidental ....................................... 82
  5.3.3 Foreign Advising versus Co-advising ......................... 83
5.4 Co-advising Patterns ............................................... 84
5.5 Foreign Advising Patterns ......................................... 87
5.6 Building a Composition Specification ......................... 92
  5.6.1 The COOLAJ Specification .................................... 93
  5.6.2 The AJW Specification ......................................... 96
  5.6.3 The COOLAJW Specification .................................. 99
  5.6.4 Scalability and Completeness ................................ 102
5.7 Analyzing Existing Compositions ................................. 102
5.8 Summary .............................................................. 104
6 The Composition Implementation Problem 106
6.1 System Design Requirements ................................. 108
  6.1.1 Granularity ..................................................... 108
  6.1.2 Decoupling ...................................................... 109
  6.1.3 Composability .................................................. 109
  6.1.4 Customizability ............................................... 112
6.2 An Aspect Compiler ................................................. 112
6.3 A Compiler for COOL ............................................. 119
  6.3.1 Front-end Translation ........................................ 119
  6.3.2 Back-end Weaving ........................................... 122
6.4 An Aspect-Oriented Architecture .............................. 124
  6.4.1 Decoupling .................................................. 124
  6.4.2 Composability .............................................. 127
  6.4.3 Customizability .............................................. 130
6.5 Implementation by Refactoring AspectJ ...................... 133
  6.5.1 Implementing a Platform .................................... 134
  6.5.2 An Abstract Aspect Mechanism ............................. 136
  6.5.3 Implementing an AspectJ Mechanism ...................... 137
  6.5.4 Implementing an AspectWerkz Mechanism ................ 138
  6.5.5 Implementing a COOL Mechanism ......................... 140
6.6 Summary .......................................................... 142
7 Evaluation .......................................................... 143
  7.1 Case Study: Composing AspectJ, AspectWerkz, and COOL 145
    7.1.1 Implementing COOLAJ ................................. 145
    7.1.2 Implementing AJW ....................................... 148
    7.1.3 Implementing COOLAJW ............................... 149
  7.2 Third-party Composition ..................................... 149
  7.3 Testing ........................................................ 152
    7.3.1 Testing AspectJ ........................................... 155
    7.3.2 Testing AspectWerkz .................................... 156
    7.3.3 Testing COOL ............................................ 157
    7.3.4 Testing AJW ............................................. 158
7.3.5 Testing COOLAJ ................................................. 160
7.3.6 Testing COOLAJW ............................................... 161
7.4 Performance ....................................................... 163
7.5 Summary ........................................................... 165

8 Conclusion .......................................................... 167
  8.1 Summary of Contribution ....................................... 171
  8.2 Benefits of Approach ........................................... 173
  8.3 Limitations of Approach ....................................... 173
  8.4 Future Directions ................................................ 175

Bibliography ............................................................ 177

A Publications .......................................................... 183
  A.1 Pluggable AOP: Designing aspect mechanisms for third-party composition. . . I
  A.2 Modeling aspect mechanisms: A top-down approach. .......................... II
  A.3 Identifying feature interactions in aspect-oriented frameworks. ............... III
  A.4 AWESOME: An aspect co-weaving system for composing multiple aspect-oriented extensions. .................................................. IV
List of Tables

2.1 Approaches to modeling aspect mechanisms .............................. 22

7.1 Test Plan ............................................................................. 153
7.2 Test Cases ........................................................................... 155
7.3 Comparison of aspect extension composition frameworks. .......... 165
List of Figures

1.1 Mechanisms ....................................................... 5
1.2 Default composition ........................................... 5
1.3 Specified composition ........................................... 5

4.1 The ABB Model ................................................ 50
4.2 The ABX Model ................................................ 50
4.3 The CRX Model ................................................ 51
4.4 The Weaving Problem ......................................... 53
4.5 The CRX Weaving Process .................................... 53
4.6 Hyper/J’s Nonreactive Weaving System ...................... 56
4.7 DFD 3.0: COMPOSE ......................................... 56
4.8 Reactive PA Weaving System ................................. 59
4.9 DFD 4.0: WEAVE ........................................... 59
4.10 DFD 4.1: ADVISE and DFD 4.2: EVALUATE ............... 60
4.11 Nonreactive OC Weaving System ............................. 62
4.12 A simplified representation of a Java program’s AST .... 63
4.13 Reactive OC Weaving System ................................. 63
4.14 Co-application of integration rules .......................... 64
4.15 Composition of Reactive Mechanisms ....................... 66
4.16 Syntactical model versus semantical model ............... 67
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Feature Interactions in a composition of COOL and AspectJ</td>
<td>81</td>
</tr>
<tr>
<td>5.2</td>
<td>Feature Interactions in COOLAJ</td>
<td>94</td>
</tr>
<tr>
<td>6.1</td>
<td>DFD 4.1.1.1: RELATE, DFD 4.1.1.2: SELECT, and DFD 4.1.1.3: ORDER</td>
<td>114</td>
</tr>
<tr>
<td>6.2</td>
<td>A stand-alone weaver for AspectJ</td>
<td>126</td>
</tr>
<tr>
<td>6.3</td>
<td>A default multi-weaver</td>
<td>127</td>
</tr>
<tr>
<td>6.4</td>
<td>A customized multi-weaver</td>
<td>130</td>
</tr>
<tr>
<td>7.1</td>
<td>An AspectJ weaver</td>
<td>151</td>
</tr>
<tr>
<td>7.2</td>
<td>A COOL weaver</td>
<td>151</td>
</tr>
<tr>
<td>7.3</td>
<td>An AJW weaver</td>
<td>152</td>
</tr>
<tr>
<td>7.4</td>
<td>A COOLAJ weaver</td>
<td>152</td>
</tr>
<tr>
<td>7.5</td>
<td>A COOLAJW weaver</td>
<td>152</td>
</tr>
<tr>
<td>7.6</td>
<td>AWESOME performance</td>
<td>164</td>
</tr>
</tbody>
</table>
List of Listings

2.1 A simple logger aspect in AspectJ .................................................. 12
2.2 A simple logger aspect in AspectWerkz .......................................... 16
2.3 A non-synchronized stack ............................................................... 17
2.4 A coordinator in COOL ................................................................. 17
2.5 Personal View .................................................................................. 19
2.6 Tax View ......................................................................................... 19
2.7 Hyperspace Specification ................................................................... 21
2.8 Composition Rules ........................................................................... 21
2.9 Composition Clauses ........................................................................ 21
2.10 A stack logger aspect in AspectJ ....................................................... 28
2.11 A translated metaobject class in Java .............................................. 28
2.12 A class selector in Reflex .................................................................. 28
2.13 An operation selector in Reflex ......................................................... 29
2.14 The main configuration class in Reflex ............................................ 29
2.15 Synchronization aspect in COOL ..................................................... 30
2.16 Synchronization aspect in AspectJ .................................................... 31
3.1 AJAdviceLogger.java ........................................................................ 38
3.2 AWAspect.java ................................................................................ 38
3.3 Main.java ......................................................................................... 39
3.4 An empty aspect in AspectJ .............................................................. 40
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>A metaobject class for AJNothing</td>
<td>40</td>
</tr>
<tr>
<td>3.6</td>
<td>A class selector for AJNothing</td>
<td>40</td>
</tr>
<tr>
<td>3.7</td>
<td>An operation selector for AJNothing</td>
<td>41</td>
</tr>
<tr>
<td>3.8</td>
<td>Call stack logger in AspectJ</td>
<td>42</td>
</tr>
<tr>
<td>5.1</td>
<td>A logger aspect in AspectJ</td>
<td>83</td>
</tr>
<tr>
<td>5.2</td>
<td>AJAccessLogger.java</td>
<td>89</td>
</tr>
<tr>
<td>5.3</td>
<td>AJStackExecScope.java</td>
<td>91</td>
</tr>
<tr>
<td>5.4</td>
<td>AJStackLogger.java</td>
<td>91</td>
</tr>
<tr>
<td>6.1</td>
<td>A weaver</td>
<td>115</td>
</tr>
<tr>
<td>6.2</td>
<td>Identifying Shadows in Classes and Aspects</td>
<td>116</td>
</tr>
<tr>
<td>6.3</td>
<td>Identifying Instruction-level Shadows</td>
<td>116</td>
</tr>
<tr>
<td>6.4</td>
<td>Advice matching in AspectJ</td>
<td>117</td>
</tr>
<tr>
<td>6.5</td>
<td>History Relations in AspectJ</td>
<td>118</td>
</tr>
<tr>
<td>6.6</td>
<td>The AspectJ weaver</td>
<td>119</td>
</tr>
<tr>
<td>6.7</td>
<td>A translated COOL coordinator class</td>
<td>120</td>
</tr>
<tr>
<td>6.8</td>
<td>A synchronized bounded stack</td>
<td>121</td>
</tr>
<tr>
<td>6.9</td>
<td>The COOL weaver</td>
<td>123</td>
</tr>
<tr>
<td>6.10</td>
<td>Platform reify methods</td>
<td>135</td>
</tr>
<tr>
<td>6.11</td>
<td>match and order methods</td>
<td>136</td>
</tr>
<tr>
<td>6.12</td>
<td>relate and mix functions</td>
<td>136</td>
</tr>
<tr>
<td>6.13</td>
<td>Mechanism aspect</td>
<td>137</td>
</tr>
<tr>
<td>6.14</td>
<td>The AspectJ mechanism</td>
<td>138</td>
</tr>
<tr>
<td>6.15</td>
<td>The AspectWerkz mechanism</td>
<td>140</td>
</tr>
<tr>
<td>6.16</td>
<td>Implementation of the COOL mechanism</td>
<td>141</td>
</tr>
<tr>
<td>7.1</td>
<td>Ordering multi-extension advice in COOLAJ</td>
<td>146</td>
</tr>
<tr>
<td>7.2</td>
<td>Normalizing multi-extension join points in COOLAJ</td>
<td>147</td>
</tr>
<tr>
<td>7.3</td>
<td>Restricting advisability of foreign join points in COOLAJ</td>
<td>147</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>7.4 Configuring AJW compiler</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td>7.5 Configuring the COOLAJW weaver</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>7.6 LogAdviceOnStack</td>
<td>154</td>
<td></td>
</tr>
</tbody>
</table>
List of Terms

ABB model .................................................. 49
ABX model .................................................. 49
Abstract extension ........................................ 75
Abstract features on an aspect extensions .......... 75
Advice computation ....................................... 60
Advice execution interaction ............................ 88
Advice execution .......................................... 80
Advice features ............................................ 79
Advice ordering ............................................ 80
Advice type .................................................. 79
Advice ......................................................... 11
Advise process ............................................. 59
Aspect compiler ........................................... 112
Aspect composition problem ............................ 25
Aspect extension composition problem ............... 3
Aspect extension ........................................... 1
Aspect granularity ........................................ 108
Aspect interaction ......................................... 26
Aspect mechanism composition framework .......... 3
Aspect mechanism .......................................... 1
<table>
<thead>
<tr>
<th>term</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aspect sand-box framework</td>
<td>23</td>
</tr>
<tr>
<td>Aspect weaver</td>
<td>49</td>
</tr>
<tr>
<td>Aspect-oriented programming</td>
<td>1</td>
</tr>
<tr>
<td>AspectJ-compatible extensions</td>
<td>108</td>
</tr>
<tr>
<td>AspectJ/5</td>
<td>26</td>
</tr>
<tr>
<td>AspectJ</td>
<td>10</td>
</tr>
<tr>
<td>AspectWerkz</td>
<td>10</td>
</tr>
<tr>
<td>Aspect</td>
<td>1</td>
</tr>
<tr>
<td>Asymmetric AOP languages</td>
<td>69</td>
</tr>
<tr>
<td>Base granularity</td>
<td>108</td>
</tr>
<tr>
<td>Base language</td>
<td>1</td>
</tr>
<tr>
<td>Base mechanism</td>
<td>2</td>
</tr>
<tr>
<td>COOL</td>
<td>10</td>
</tr>
<tr>
<td>CRX model</td>
<td>51</td>
</tr>
<tr>
<td>CRX weaving process</td>
<td>54</td>
</tr>
<tr>
<td>Co-advising interactions</td>
<td>83</td>
</tr>
<tr>
<td>Composability design requirement</td>
<td>109</td>
</tr>
<tr>
<td>Composed program elements</td>
<td>52</td>
</tr>
<tr>
<td>Composed program</td>
<td>52</td>
</tr>
<tr>
<td>Composition clause</td>
<td>20</td>
</tr>
<tr>
<td>Composition of abstract extensions</td>
<td>81</td>
</tr>
<tr>
<td>Composition rules</td>
<td>19</td>
</tr>
<tr>
<td>Computation</td>
<td>113</td>
</tr>
<tr>
<td>Compute process</td>
<td>60</td>
</tr>
<tr>
<td>Concern integration</td>
<td>49</td>
</tr>
<tr>
<td>Concern program elements</td>
<td>52</td>
</tr>
<tr>
<td>Concern program</td>
<td>52</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Control process</td>
<td>60</td>
</tr>
<tr>
<td>Coordinator</td>
<td>16</td>
</tr>
<tr>
<td>Crosscutting concern problem</td>
<td>1</td>
</tr>
<tr>
<td>Customizability design requirement</td>
<td>112</td>
</tr>
<tr>
<td>Decoupling design requirement</td>
<td>109</td>
</tr>
<tr>
<td>Default multi-weaver</td>
<td>109</td>
</tr>
<tr>
<td>Dynamic AOP languages</td>
<td>70</td>
</tr>
<tr>
<td>Dynamic AOP semantics</td>
<td>71</td>
</tr>
<tr>
<td>Dynamic context relation over join points</td>
<td>117</td>
</tr>
<tr>
<td>Emergent advice ordering interaction</td>
<td>86</td>
</tr>
<tr>
<td>Emergent advice type interaction</td>
<td>86</td>
</tr>
<tr>
<td>Emergent join point granularity interaction</td>
<td>84</td>
</tr>
<tr>
<td>Evaluate process</td>
<td>59</td>
</tr>
<tr>
<td>Extension-level interactions</td>
<td>39</td>
</tr>
<tr>
<td>Foreign advising interactions</td>
<td>83</td>
</tr>
<tr>
<td>Genuine versus incidental feature interactions</td>
<td>82</td>
</tr>
<tr>
<td>Granularity design requirement</td>
<td>108</td>
</tr>
<tr>
<td>Granularity of an aspect extension</td>
<td>34</td>
</tr>
<tr>
<td>Hyper/J</td>
<td>10</td>
</tr>
<tr>
<td>Hypermodule</td>
<td>19</td>
</tr>
<tr>
<td>Hyperslice</td>
<td>18</td>
</tr>
<tr>
<td>Hyperspace</td>
<td>19</td>
</tr>
<tr>
<td>Influencing versus inducing feature interactions</td>
<td>82</td>
</tr>
<tr>
<td>Integration rules</td>
<td>54</td>
</tr>
<tr>
<td>Interaction patterns</td>
<td>77</td>
</tr>
<tr>
<td>Join point advisability interaction</td>
<td>91</td>
</tr>
<tr>
<td>Join point advisability</td>
<td>79</td>
</tr>
<tr>
<td>Symbol</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>$M_B$</td>
<td>2</td>
</tr>
<tr>
<td>$C$</td>
<td>52</td>
</tr>
<tr>
<td>$R$</td>
<td>54</td>
</tr>
<tr>
<td>$\mathcal{X}$</td>
<td>52</td>
</tr>
<tr>
<td>$Ctx$</td>
<td>60</td>
</tr>
<tr>
<td>$P$</td>
<td>52</td>
</tr>
<tr>
<td>$ctx_{jp}$</td>
<td>60</td>
</tr>
<tr>
<td>$c$</td>
<td>52</td>
</tr>
<tr>
<td>$instr$</td>
<td>61</td>
</tr>
<tr>
<td>$r$</td>
<td>54</td>
</tr>
<tr>
<td>$x_{jp}$</td>
<td>60</td>
</tr>
<tr>
<td>$x$</td>
<td>52</td>
</tr>
<tr>
<td>$M_1$</td>
<td>2</td>
</tr>
<tr>
<td>$\boxplus$</td>
<td>4</td>
</tr>
<tr>
<td>$\varphi_a$</td>
<td>60</td>
</tr>
<tr>
<td>$Base$</td>
<td>1</td>
</tr>
<tr>
<td>$Ext$</td>
<td>1</td>
</tr>
<tr>
<td>$\varphi$</td>
<td>52</td>
</tr>
<tr>
<td>$\text{adv}$</td>
<td>155</td>
</tr>
<tr>
<td>$match$</td>
<td>53</td>
</tr>
<tr>
<td>Term</td>
<td>Page</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>mix</td>
<td>53</td>
</tr>
<tr>
<td>wp</td>
<td>52</td>
</tr>
<tr>
<td>ABB</td>
<td>49</td>
</tr>
<tr>
<td>ABX</td>
<td>49</td>
</tr>
<tr>
<td>AOP</td>
<td>1</td>
</tr>
<tr>
<td>CMP</td>
<td>23</td>
</tr>
<tr>
<td>CRX</td>
<td>51</td>
</tr>
<tr>
<td>J&amp;A</td>
<td>75</td>
</tr>
<tr>
<td>OC</td>
<td>14</td>
</tr>
<tr>
<td>PA</td>
<td>11</td>
</tr>
<tr>
<td>TRV</td>
<td>23</td>
</tr>
<tr>
<td>WPM</td>
<td>47</td>
</tr>
<tr>
<td>coolx</td>
<td>150</td>
</tr>
<tr>
<td>awx</td>
<td>150</td>
</tr>
<tr>
<td>ajx</td>
<td>150</td>
</tr>
<tr>
<td>ajwx</td>
<td>151</td>
</tr>
<tr>
<td>coolajx</td>
<td>151</td>
</tr>
<tr>
<td>coolajwx</td>
<td>151</td>
</tr>
<tr>
<td>AJW</td>
<td>96</td>
</tr>
<tr>
<td>AWESOME</td>
<td>106</td>
</tr>
<tr>
<td>COOLAJW</td>
<td>99</td>
</tr>
<tr>
<td>COOLAJ</td>
<td>93</td>
</tr>
<tr>
<td>Xcool</td>
<td>154</td>
</tr>
<tr>
<td>match</td>
<td>114</td>
</tr>
<tr>
<td>mix</td>
<td>114</td>
</tr>
<tr>
<td>order</td>
<td>114</td>
</tr>
<tr>
<td>reify</td>
<td>114</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Aspect-oriented programming (AOP) addresses the crosscutting concern problem that is inherent in conventional programming paradigms. Crosscutting is attributed to the inability of conventional programming languages to encapsulate concerns (e.g., logging, security) that cut across program modules. To alleviate crosscutting, AOP extends syntactically and semantically a conventional object-oriented language, which is referred to as the base language. The syntactical extension, called an aspect extension, provides the programmer with lingual means for expressing problematic concerns separately, in dedicated modules called aspects. The semantical extension, called an aspect mechanism, binds the aspects to the rest of the program, by integrating (weaving) them into the other modules. For a choice of a base language (e.g., Java [Arnold and Gosling, 1996]), there are typically multiple aspect extensions available (e.g., AspectJ [Kiczales et al., 2001], AspectWerkz [Bonér, 2004], COOL [Lopes, 1997]).

The value in having many different aspect extensions is well recognized [Tanter and Noyé, 2005, Courbis and Finkelstein, 2005, Wand, 2003, Shonle et al., 2003, Lopes et al., 2003, Czarnecki and Eisenecker, 2000, Hugunin, 2001]. However, the simultaneous use of these multiple aspect extensions has not been thoroughly studied. To date, studies of AOP have typically considered only a single-extension AOP language, which integrates a certain aspect extension, $Ext_1$, with a certain base language, $Base$. For example, $Ext_1$ might be AspectJ [Kiczales et al., 2001] and $Base$ might be Java [Arnold and Gosling, 1996]. The inte-
gration $\text{Base} \times \text{Ext}_1$ is achieved by amending the semantics for the base language. Given a pair of programs $(\text{base, aspect}_1) \in \text{Base} \times \text{Ext}_1$, the amended semantics explain the meaning of $\text{base}$ in the presence of $\text{aspect}_1$. In these studies, in the process of integration the aspect mechanism $\mathcal{M}_1$ for $\text{Ext}_1$ (implementing the semantics for $\text{Ext}_1$) is strongly coupled with the base mechanism $\mathcal{M}_\text{B}$ (the base language semantics). Consequently, it is difficult to reuse or combine aspect extensions. For each newly introduced aspect extension, say $\text{Ext}_2$, the semantics for $\text{Base} \times \text{Ext}_2$ needs to be reworked. Moreover, given the semantics for $\text{Base} \times \text{Ext}_1$ and the semantics for $\text{Base} \times \text{Ext}_2$, the semantics of $\text{Base} \times \text{Ext}_1 \times \text{Ext}_2$ is undefined even though $\text{Ext}_1$ and $\text{Ext}_2$ are both aspect extensions to the same base language. Generally, multi-extension AOP languages of the form $\text{Base} \times \text{Ext}_1 \times \cdots \times \text{Ext}_n$ have not been considered in the literature.

1.1 Thesis

The overall objective of this Ph.D. Thesis is to enable the construction and use of multi-extension AOP languages. The general approach is to encapsulate individual aspect extensions as composable components that can be subjected to third-party composition. The thesis presented is that, with an appropriate design of the base mechanism, $\mathcal{M}_\text{B}$, and the independently developed aspect mechanisms, $\mathcal{M}_1, \ldots, \mathcal{M}_n$, a multi-extension AOP language can be assembled from $\mathcal{M}_\text{B}$ and $\mathcal{M}_1, \ldots, \mathcal{M}_n$. Specifically, for each $i \in \{1, \ldots, n\}$, the $i^{\text{th}}$ component $\mathcal{M}_i$ encapsulates the $i^{\text{th}}$ aspect mechanism, decoupled from the the base mechanism $\mathcal{M}_\text{B}$.

The dissertation raises and answers several new and previously unexplored research questions:

(1) How does one decouple an aspect mechanism from the base mechanism?

(2) How does one specify a multi-extension AOP language?

(3) How does one design the base and the aspect mechanisms for composition?
The main result of the research is a practical and general composition framework. The framework is practical because it enables construction of a multi-extension AOP language by a third-party composition of aspect extensions. The framework is general because it supports a wide range of composable aspect extensions.

1.2 The Aspect Extension Composition Problem

The dissertation introduces and systematically analyzes the aspect extension composition problem, formulated as follows:

\textbf{PROBLEM 1 (ASPECT EXTENSION COMPOSITION)} Given a set of \( n \) AOP languages, \( \langle \text{Base} \times \text{Ext}_1 \rangle, \ldots, \langle \text{Base} \times \text{Ext}_n \rangle \), build a multi-extension AOP language \( \text{Base} \times \text{Ext}_1 \times \ldots \times \text{Ext}_n \).

The thesis resolves the aspect extension composition problem by constructing an aspect mechanism composition framework that supports pluggable, collaborative, and composable aspect mechanisms:

\textbf{Pluggable:} For \( 1 \leq i \leq n \), \( \text{Base} \times \text{Ext}_i \) is constructed by plugging together the aspect mechanism \( M_i \) of \( \text{Ext}_i \) and the base mechanism \( M_B \);

\textbf{Collaborative:} The semantics of the multi-extension AOP language \( \text{Base} \times \text{Ext}_1 \times \ldots \times \text{Ext}_n \) is defined as an interaction between the respective mechanisms \( M_B, M_1, \ldots, M_n \) of the base language and the aspect extensions;

\textbf{Composable:} The composition semantics is realized by plugging all the \( M_1, \ldots, M_n \) mechanisms into \( M_B \) under the specified collaboration behavior.

The dissertation approaches the aspect extension composition problem and builds the composition framework by formulating and addressing the following requirements and sub-problems (Section 1.2.1; Section 1.2.2; Section 1.2.3):
1.2.1 The component abstraction problem

**PROBLEM 1A (ASPECT EXTENSION COMPONENT ABSTRACTION)** Build an abstract, modular, and precise model of an aspect extension.

A model is said to be abstract, modular, and precise if:

**Abstract:** The model generalizes uniformly over existing aspect extensions;

**Modular:** For each \(1 \leq i \leq n\), the model specifies the semantics of the AOP language \(\text{Ext}_i \times \text{Base}\) as a composition of an aspect mechanism \(M_i\) for \(\text{Ext}_i\) and the base mechanism \(M_B\) for \(\text{Base}\); and

**Precise:** The model formally defines the process of aspect weaving in an aspect mechanism.

The model enables a detailed analysis of the aspect extension composition problem. The precision defines terms and abstractions for formulating the composition problem more formally. The abstractiveness generalizes the problem over arbitrary aspect extensions; and the modularity reduces the general problem to the specific problem of composing aspect mechanisms.

1.2.2 The composition specification problem

**PROBLEM 1B (ASPECT EXTENSION COMPOSITION SPECIFICATION)** Provide a methodology for specifying a composition of aspect extensions.

An aspect extension composition specification identifies and resolves all semantical interactions between the composed aspect extensions. More formally, the specification of a multi-extension AOP language \(\text{Base} \times \text{Ext}_1 \times \cdots \times \text{Ext}_n\) defines a combinator function \(\boxplus\), such that:

\[
\mathcal{A} = \boxplus(M_B, M_1, \ldots, M_n)
\]

where \(M_1, \ldots, M_n\) are mechanisms of the \(\text{Ext}_1, \ldots, \text{Ext}_n\) extensions, respectively; \(M_B\) denotes the semantics of \(\text{Base}\); and \(\mathcal{A}\) denotes the AOP language semantics.
1.2.3 The composition implementation problem

**Problem 1C (Aspect Extension Composition Implementation)** Design a composition framework that enables multiple, independent aspect mechanisms to be assembled with the base mechanism, in a manner similar to the assembly of software components.

A software component is a “unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition” [Szyperski, 2002]. In the context of the aspect extension composition implementation problem, the aspect extensions should be:

**Units of independent production:** the mechanisms $M_1, \ldots, M_n$ are independently defined (Figure 1.1);

**Units of composition:** the mechanisms are subject to third-party composition with the base mechanism $M_B$ into a default *multi-mechanism* with a reasonable weaving behavior (Figure 1.2);

**Units of collaboration:** given the composition specification $\Box$, plugging the mechanisms into the framework implements the semantical function $A$ of the multi-extension AOP language (Figure 1.3).
1.3 Main Results

The main results of this research are summarized in four key publications:


The first publication introduces the aspect extension composition problem. The other three publications correspond to the three research problems.

1.3.1 Pluggable AOP

The aspect extension composition problem is stated and analyzed in *Pluggable AOP* [Kojarski and Lorenz, 2005]. The paper establishes the fundamentals of the aspect extension composition methodology by formulating and studying the component abstraction, the composition
specification, and the composition implementation problems. To illustrate the methodology concretely, the paper provides a confined solution to each problem, and builds a framework for third-party construction and composition of dynamic aspect mechanisms. The framework defines an aspect mechanism as a transformer of an expression interpreter. The composition is specified by two principles: the visibility of the aspectual effect and the encapsulation of the weaving process; and the composed mechanisms collaborate by delegating or exposing the evaluation of expressions. Although the framework is capable of composing arbitrary dynamic aspect mechanisms, it is not flexible and efficient enough for building complex real-world compositions of aspect extensions. Construction of a practical solution requires a systematic study of the component abstraction, the composition specification, and the composition implementation problems.

1.3.2 The component abstraction problem

The aspect extension component abstraction problem is resolved in Modeling Aspect Mechanisms [Kojarski and Lorenz, 2006]. The paper analyzes the design space that existing aspect mechanisms inhabit. It presents a precise, modular, and abstract model of their weaving processes. The analysis yields a conceptual framework for understanding and evaluating existing mechanisms. The model provides a formalism for specifying AOP terms in an abstract way, independent of a particular aspect extension.

1.3.3 The composition specification problem

The composition specification problem is resolved in Identifying Feature Interactions [Kojarski and Lorenz, 2007b]. The paper addresses the problem from the feature interaction perspective. Under this perspective, a specification of a multi-extension composition identifies and resolves all semantical interactions between the features of the composed extensions. The paper establishes a methodology for identifying feature interactions in a multi-extension composition, and uses it to specify compositions of concrete real-world aspect extensions.
1.3.4 The composition implementation problem

The composition implementation problem is resolved in AWESOME [Kojarski and Lorenz, 2007a]. The paper presents a practical framework for composing aspect mechanisms into a compile-time weaving system that realizes the semantics of a multi-extension AOP language. The framework is pluggable, customizable, and efficient. Pluggability enables third-party composition of aspect mechanisms into a multi-mechanism with a reasonable weaving behavior. Customizability provides means for configuring the behavior of the constructed multi-mechanism to cater for a concrete composition specification. An efficient compile-time weaving scheme enables practical use of the framework. The paper uses the framework to compose real-world aspect extensions, and compares the constructed multi-mechanisms to standard aspect compilers and weaving algorithms.

1.4 Structure of the Dissertation

Chapter 2 provides background on the thesis research. The chapter overviews state-of-the-art composition tools, analyzes dominant aspect extension models, introduces and explains relevant aspect extensions (including AspectJ, AspectWerkz, COOL, Hyper/J), and discusses other related topics. The analysis shows that the dominant aspect extension models fail to provide a desired level of abstraction, precision, and uniformity. The models are built bottom-up by generalizing over specific aspect mechanisms: their abstractions are geared toward concrete aspect extensions, and they do not generalize over other AOP languages.

Chapter 3 motivates the need for composing aspect extensions. It explains why the aspect extension composition problem is difficult, and shows that existing composition techniques fail to adequately address this problem. The motivation emphasizes the vision of domain-specific aspect languages that enables a simultaneous use of domain-specific and general-purpose aspects in the same program. The difficulty of the composition problem is due to the complex nature of interactions between the extensions in a composition. The chapter shows concrete
examples demonstrating that current composition tools weave multi-extension aspect programs in an ad-hoc manner, and explains why these tools generally produce erroneous woven code.

Chapter 4, Chapter 5, and Chapter 6 respectively address the component abstraction, the specification, and the implementation problems. Chapter 4 builds a model of aspect mechanism, and uses it to formalize the aspect extension composition problem in concrete terms. Chapter 5 introduces the methodology for specifying a multi-extension AOP language, and uses it to specify compositions of real-world aspect extensions. Chapter 6 introduces a practical framework for composing aspect mechanisms, and uses it to build compile-time weaving systems for multi-extension AOP languages that are specified in Chapter 5.

In Chapter 7 the framework is evaluated and compared to other aspect extension composition frameworks. The basis for the comparison is the composability, customizability, and efficiency properties of the framework. Chapter 8 concludes the work by summarizing the research results and discussing directions for the future work. Copies of the main publications can be found in the Appendix.
Chapter 2

State of the Art

Related research fall into four general categories, namely AOP languages, aspect mechanism models, aspect composition, and aspect extension composition. Section 2.1 reviews four aspect-oriented languages that realize different approaches to AOP: AspectJ (Section 2.1.1), AspectWerkz (Section 2.1.2), COOL (Section 2.1.3), and Hyper/J (Section 2.1.4). Section 2.2 presents an overview of the current aspect mechanism models in the context of the aspect extension component abstraction problem. Section 2.3 discusses the aspect composition problem. Lastly, Section 2.4 reviews state-of-the-art tools for composing aspect extensions.

2.1 Aspect-Oriented Languages

This section covers four AOP languages that represent dominant approaches to AOP. AspectJ and AspectWerkz are today’s most popular general-purpose AOP languages. COOL represents a domain-specific approach to AOP. Hyper/J is a multi-dimensional concern separation tool. These languages are used in the dissertation for illustration and evaluation purposes.

2.1.1 AspectJ

AspectJ is a general-purpose aspect extension to Java. The extension provides an aspect module construct for encapsulating crosscutting concerns. An AspectJ aspect can non-intrusively change the behavior and structure of the other program modules. The behavioral
changes are supported by join point, pointcut and advice abstractions; and the structural changes are specified by inter-type declarations.

2.1.1 Join point, Pointcut, and Advice

A join point represents and describes a computation in an execution of a program. A pointcut is an AspectJ construct that allows the programmer to specify a set of join points. An advice construct provides a placeholder for crosscutting concern code. An aspect binds an advice to the program by associating it with a pointcut.

AspectJ observes the execution of a program as a sequence of join points. The AspectJ join points include calls and executions of methods and constructors, read and write accesses to instance variables, object and class initializations, and executions of advice. A join point is described using lexical, static, and dynamic data. The lexical data describes the source code that the join point computation executes (e.g., file name, line numbers). The static description includes the signature of a class member that is associated with the join point. The dynamic description includes objects in the execution environment of the join point computation, that is, the callee (this) and the caller (target) objects, and the argument values (the actual parameters of a method call).

The dynamic AspectJ mechanism, called pointcut and advice (PA), weaves aspects into the program by (1) matching the description of each join point against the aspects’ pointcuts; (2) selecting pieces of advice that are associated with the matching pointcuts; and (3) ordering and executing the selected advice at the join point. Intuitively, the join points define an interface between the program and the aspects.

For illustration consider an AJSimpleLogger aspect that logs every method execution in a program. The aspect, shown in Listing 2.1, selects join points using the pointcuts targetJps (line 3), scope (line 4), and toLog (line 5). The aspect affects the program’s behavior at toLog join points with before (line 7), around (lines 9 – 15), and after (line 17) advice.
public aspect AJSimpleLogger {

    pointcut targetJps(): execution(* *.*(..));
    pointcut scope(): !cflow(within(AJSimpleLogger));
    pointcut toLog(): targetJps() && scope();

    before(): toLog() {log("Before", thisJoinPoint);} 

    Object around(): toLog() {
        Object result;
        log("Around (before proceed)", thisJoinPoint);
        result = proceed();
        log("Around (after proceed)", thisJoinPoint);
        return result;
    }

    after(): toLog() {log("After", thisJoinPoint);} 

    void log(String message, JoinPoint jp) {
        System.out.println(message+":"+jp);
    }
}

Listing 2.1: A simple logger aspect in AspectJ

AJSimpleLogger pointcuts are defined using four kinds of pointcut expressions, namely primitive pointcuts, within and cflow constructs, and composite pointcuts:

- Primitive pointcuts select join points of a specific type and signature. For example, the `execution(* *.*(..))` pointcut expression (Listing 2.1, line 3) matches all method execution join points in a program: the `execution` keyword specifies the join point type, and the signature pattern `* *.*(..)` matches any method signature.

  The signature pattern may also be defined to match only specific join points. For example, `call(String Main.foo(int))` matches join points representing calls to a method `foo` of type `Main`, which accepts a single argument of type `int`, and returns an object of type `String`.

- `within` pointcuts select join points that are lexically defined within certain types. For
example, \texttt{within}(AJSimpleLogger) (Listing 2.1, line 4) selects all join points within the AJSimpleLogger aspect definition.

- \texttt{cflow} pointcuts select join point computations that execute within a dynamic control flow of other join points. For example, \texttt{cflow(within}(AJSimpleLogger)) (Listing 2.1, line 4) selects all join points that are constructed during executions of the AJSimpleLogger aspect.

- A composite pointcut is constructed from other pointcuts using conjunction, disjunction, and negation operations, denoted by the \\
\texttt{&&}, \\
\texttt{||}, \texttt{and} \texttt{!} symbols, respectively. For example, the \texttt{!cflow(within}(AJSimpleLogger)) pointcut (Listing 2.1, line 4) selects all join points \texttt{except} those within the dynamic scope of an AJSimpleLogger execution. Definition of the \texttt{toLog} pointcut in Listing 2.1, line 5 is another example. \texttt{toLog} selects only those targetJps join points that are also in the scope set.

AspectJ supports \texttt{before}, \texttt{after}, and \texttt{around} advice that respectively execute before, after, or instead of the original join point computation. While the \texttt{before} and the \texttt{after} advice execute along with the original join point computation, the \texttt{around} advice “wrap” around it. The \texttt{around} advice may execute the wrapped computation by executing a \texttt{proceed} expression. For example, at each \texttt{toLog} join point AJSimpleLogger first executes \texttt{before} advice, then \texttt{around} advice, and finally the \texttt{after} advice. The \texttt{around} advice calls \texttt{proceed} (Listing 2.1, line 12) to execute the original join point computation.

Advice types, however, do not fully specify a partial order of \texttt{before}, \texttt{after}, and \texttt{around} advice at a join point. The lexical order of advice supersedes advice type ordering. An \texttt{around} advice takes precedence over \texttt{after} advice that are defined before it and over \texttt{before} advice that are defined after it. For example, if one swaps the definitions of the \texttt{before} and \texttt{after} advice in AJSimpleLogger, then the two advice would execute only when the \texttt{around} advice calls \texttt{proceed}.

AspectJ reifies the description of a join point to its aspects as an object. The object is an
instance of the org.aspectj.lang.JoinPoint interface. The object is accessible from within advice body via the thisJoinPoint variable. For example, the AJSimpleLogger advice accesses the thisJoinPoint object at lines 7, 11, 13, and 17.

### 2.1.1.2 Inter-type Declarations

There are two kinds of inter-type declarations: member and super type. A **member declaration** introduces a new member into the target Java type. For example, the following inter-type declaration introduces an observers field into the Main class:

```java
public Vector Main.observers;
```

A **super type declaration** effect transforms the inheritance relationship of the target type. It is specified by a **declare parents** construct. For example, the following declaration resets Main’s superclass to be Observable:

```java
declare parents: Main extends Observable;
```

Although inter-type declarations are normally defined in the same aspect modules as pointcuts and advice, they apply at a different time. The AspectJ mechanism of inter-type declarations, called Open Classes (OC), is applied separately and before the application of the dynamic PA mechanism. The collaboration between two mechanisms is very limited, and they can be studied separately.

### 2.1.2 AspectWerkz

Semantically, AspectWerkz is almost equivalent to AspectJ. AspectWerkz provides the same abstractions (e.g., join points, pointcuts, inter-type declarations) and mechanisms as AspectJ. The main difference between AspectJ and AspectWerkz is in the way they define aspects. In AspectWerkz, aspects are specified as annotated Java classes: advice is defined by a regular Java method, annotated with advice type and a pointcut expression. A named pointcut can be defined as a field of type Pointcut, or as a method.
For illustration, consider the AspectWerkz aspect AWSimpleLogger shown in Listing 2.2. AWSimpleLogger is functionally equivalent to AspectJ’s AJSimpleLogger (Listing 2.1). Its code is written in plain Java. The annotation @Aspect("perJVM") specifies that the AWSimpleLogger class is actually a singleton aspect. The annotated instance variables scope, targetJps, and toLog of type Pointcut are named pointcuts. Their pointcut expressions are specified by @Expression annotations. The three pointcuts match the same join points as the corresponding pointcuts in AJAspectLogger (Listing 2.1).

An AspectWerkz aspect declares before, around, and after advice by annotating methods with @Before, @Around, and @After annotations, respectively. For example, the @Before(toLog) annotation in Listing 2.2, line 27 declares the beforeAdvice method (lines 28 – 30) as before advice. The method receives a join point description object as an argument. The aspect declares aroundAdvice and afterAdvice methods as around and after advice in the similar fashion.

Another difference between AspectJ and AspectWerkz is advice ordering rules. Unlike AspectJ, AspectWerkz orders advice only according to their types. In AspectWerkz, before advice always execute first, and the after advice always execute the last. E.g., although the AWSimpleLogger advice are defined in different lexical order than the AJSimpleLogger advice, the two aspects exhibit the same behavior when they (independently) apply to the same Java program.

In comparison to AspectJ, AspectWerkz supports fewer join point types. For example, AspectWerkz does not support advice-execution join points.

2.1.3 COOL

COOL extends Java with a method synchronization mechanism. COOL aspects, called coordinators, allow a programmer to impose synchronization policy on program methods in

---

1 A singleton aspect is instantiated only once during the execution of a program.
@Aspect("perJVM")
public class AWSimpleLogger {

    @Expression("lcflow(within(AWSimpleLogger))")
    Pointcut scope;

    @Expression("execution(* *.*(..))")
    Pointcut targetJps;

    @Expression("targetJps && scope")
    Pointcut toLog;

    @After("toLog")
    public void afterAdvice(JoinPoint jp) throws Throwable {
        log("After", jp);
    }

    @Around("toLog")
    public Object aroundAdvice(JoinPoint jp) throws Throwable {
        Object result;
        log("Around (before proceed)", jp);
        result = joinPoint.proceed();
        log("Around (after proceed)", jp);
        return result;
    }

    @Before("toLog")
    public void beforeAdvice(JoinPoint jp) throws Throwable {
        log("Before", jp);
    }

    void log(String message, JoinPoint jp) {
        System.out.println(message+":"+jp);
    }
}

Listing 2.2: A simple logger aspect in AspectWerkz

an aspect-oriented manner. For example, Stack (Listing 2.4) is a coordinator in COOL\(^2\) that advises a Stack Java class (Listing 2.3). The Stack class implements a bounded stack. It defines two public methods: push and pop. An attempt to pop objects off an empty stack or

\(^2\) A coordinator in COOL is the equivalent of an aspect in AspectJ.
public class Stack {

    public Stack(int capacity) {
        buf = new Object[capacity];
    }

    public void push(Object obj){
        buf[ind] = obj;
        ind++;
    }

    public Object pop() {
        Object top = buf[ind-1];
        buf[--ind] = null;
        return top;
    }

    private Object[] buf;
    private int ind = 0;
}

Listing 2.3: A non-synchronized stack

coordinator Stack {
    selfex {push, pop};
    mutex {push, pop};
    int len=0;
    condition full=false,empty=true;
    push: requires !full;
    on_exit {
        empty=false;
        len++;
        if(len==buf.length) full=true;
    }
    pop: requires !empty;
    on_entry {len--;}
    on_exit {
        full=false;
        if(len==0) empty=true;
    }
}

Listing 2.4: A coordinator in COOL

push objects onto a full stack throws an exception.³

³java.lang.ArrayIndexOutOfBoundsException
COOL relieves the implementor of Stack from dealing with multi-threading. A separate Stack coordinator imposes the synchronization logic over push and pop in an aspect-oriented manner. The Stack methods are not synchronized. But in the presence of the Stack coordinator, the stack object operates correctly even when multiple client threads execute methods simultaneously.

The synchronization policy is expressed in COOL using declarations (mutex, selfex, condition), expressions (requires), and statements (on_exit, on_entry). The selfex declaration (line 2) specifies that neither push nor pop may be executed by more than one thread at a time. The mutex declaration (line 3) prohibits push and pop from being executed concurrently. The requires expressions (lines 6 and 12) further guard push and pop executions. If the guard is false, a thread suspends, even if the mutex and selfex conditions are satisfied. The execution resumes when the guard becomes true. full and empty are condition boolean variables (line 5).

The on_entry and on_exit blocks update the aspect state immediately before and immediately after the execution of an advised method body, respectively. They are used in this example to track the number of elements in the stack (lines 9 and 13) and to keep the condition variables full and empty current.

Java expressions within COOL statements have read and write access to the coordinator’s fields. In addition, expressions may inspect instance variables of the coordinatee, e.g., access the buf field of the Stack object (line 10). Note that the coordinator’s expressions may access not only public but also package-protected and even private fields of the coordinatee object.

2.1.4 Hyper/J

Hyper/J allows a programmer to modularize a concern as a partial Java program that is called a hyperslice. The hyperslice implements only those classes and methods that realize the concern; and it references the other concerns (if needed) only through abstract fields, methods,
and classes.

Hyperslices are generally not executable. Hyper/J provides composition rules that allow a programmer to define an executable program as a composition of several hyperslices. In Hyper/J’s terms, the set of hyperslices that defines an input program is called a hyperspace, and the executable composed program is called a hypermodule. Intuitively, the composition process “glues” hyperslices together by resolving abstract references of one hyperslice with implementation of the others; and by merging methods from different hyperslices into the same hypermodule class or method.

We demonstrate the composition process through a case study. Consider two classes, one describing a personal view (Listing 2.5), and the other describing a tax view of a person (Listing 2.6). We use Hyper/J to produce a combination of the views. We start by specifying a hyperspace and mapping the classes to hyperspace concerns (Listing 2.7). The Person hyperspace defines two hyperslices, namely, PersonalView and TaxView. The composition
rules (Listing 2.8) define the hypermodule Result as a composition of the hyperslices under
the mergeByName composition strategy.

The composition rules are specified in terms of unit trees. A unit tree is an abstraction of a program’s abstract syntax tree (AST), where units represent nodes. In our example, the composition rules (Listing 2.8) define the Result hypermodule as a composition of the PersonalView and TaxView hyperslices under the mergeByName composition strategy. The strategy specifies that same-name-same-type hyperspace units (e.g., the class units PersonalView.Person and TaxView.Person; or the method units PersonalView.Person.getName and TaxView.Person.getName) should be merged in the hypermodule.

A composition rule specifies one or more hypermodule units. The meaning of a composition rule (the units), however, depends on a hyperspace unit tree structure. For example, the mergeByName composition strategy can be applied to virtually any input hyperspace. Given different hyperspace, the rule would specify different composition result.

The semantics of Hyper/J’s composition rules [Ossher et al., 1995] is defined through a translation to composition clauses. The composition clauses are low-level instructions that allow full and flexible composition specifications. They provide a precise specification of a composed hypermodule as a combination of specific hyperspace units. The composition clauses produced by the Hyper/J compiler for our example are shown in Listing 2.9.5

In Listing 2.9, the OPERATIONS section specifies how to compose the hypermodule operations (method signatures) from signatures of the hyperspace methods. The CLASSES section defines a hypermodule’s class-graph (classes and their instance variables) as a composition of hyperspace class-graph nodes. The MAPPING section introduces methods into the hypermodule classes by associating a hypermodule operation with a set of hyperspace realizations (method-body expressions) and a class. Thus, a composition clause defines a hypermodule unit as a

5 For readability, the actual composition clauses produced by the Hyper/J compiler were amended to resemble the notation in [Ossher et al., 1995].
### Listing 2.7: Hyperspace Specification

```plaintext
-hyperspace
  hyperspace Person
    composable class personal.*;
    composable class taxes.*;
  -concerns
    package personal: PersonalView
    package taxes: TaxView
```

### Listing 2.8: Composition Rules

```plaintext
hypermodule Result
  hyperslices: PersonalView, TaxView;
  relationships: mergeByName;
end hypermodule;
```

### Listing 2.9: Composition Clauses

```plaintext
hypermodule Result

OPERATIONS
  getName: EQUIVALENT(SIGNATURES(<PersonalView.getName, TaxView. getName>))
  getDoB: IDENTITY(SIGNATURES(<PersonalView.getDoB>))
  getSSN: IDENTITY(SIGNATURES(<TaxView.getSSN>))
  //the remainder of the operations code is omitted

CLASSES
  class Person
    INSTANCE VARIABLES:
      name: EQUIVALENT(TYPES(<PersonalView.Person.name, TaxView. Person.name>))
      dob: IDENTITY(TYPES(<PersonalView.Person.dob>))
      ssn: IDENTITY(TYPES(<TaxView.Person.ssn>))

MAPPING
  getName: class Person
    CALL ACTION: SEQUENCE
      PersonalView.getName.Person
      TaxView.getName.Person
  getDoB: class Person
    CALL ACTION: SIMPLE
      PersonalView.getDoB.Person
    getSSN: class Person
    CALL ACTION: SIMPLE
      TaxView.getSSN.Person
  //the remainder of the mapping code is omitted
```
Table 2.1: Approaches to modeling aspect mechanisms

<table>
<thead>
<tr>
<th>Approach</th>
<th>Semantical</th>
<th>Conceptual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design level</td>
<td>Implementation</td>
<td>Analysis</td>
</tr>
<tr>
<td>Generalization method</td>
<td>-</td>
<td>Bottom-Up</td>
</tr>
<tr>
<td>Mechanisms modeled</td>
<td>PA</td>
<td>PA\textsubscript{AJ}, OC, CMP, TRV</td>
</tr>
<tr>
<td>Unifying concept</td>
<td>-</td>
<td>Join points, JPM</td>
</tr>
<tr>
<td>Dichotomy</td>
<td>Semantical</td>
<td>Syntactical</td>
</tr>
<tr>
<td>Resolution</td>
<td>Fine-grained</td>
<td>Coarse-grained</td>
</tr>
<tr>
<td>Abstract</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Precise</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Uniform</td>
<td>-</td>
<td>No</td>
</tr>
</tbody>
</table>

combination of specific hyperspace units. The clauses and hyperspace units are passed to the Hyper/J mechanism that composes a result hypermodule.

2.2 Modeling AOP

Models of aspect mechanisms have focused either on a selected mechanism, or on a bottom-up generalization of several mechanisms. In a bottom-up abstraction, a model is built by identifying and generalizing common patterns. Since each aspect mechanism is useful precisely because it does some crosscutting thing very different, abstracting over several aspect mechanisms is difficult and often results in a fine-grained model which would likely need to be further extended to fit future aspect mechanisms.\footnote{This is a criticism of the term aspect-oriented being used today to describe a broad array of diverse programming mechanisms, rather than a criticism of bottom-up generalization per se.}

We distinguish between two general approaches to modeling aspect mechanisms, namely, semantical and conceptual (Table 2.1). Existing semantical models either explain a specific AspectJ-like Pointcut and Advice (PA) mechanism or generalize a PA model. Lämmel [Lämmel, 2002] explains a PA mechanism named Method-Call Interception (MCI). Wand et al. [Wand et al., 2004] explain an aspect mechanism for AspectJ. Walker et al. [Walker et al., 2003] define aspects through explicitly labeled program points and first-class dynamic advice. Jagadeesan et
al. [Jagadeesan et al., 2003] use PA to define AOP functionality. Orleans [Orleans, 2005, 2002] introduces a programming language named Socrates that generalizes over PA and OO abstractions using a predicate dispatch mechanism. Although very precise, all of these semantical models do not generalize over non-PA aspect mechanisms.

An example of a conceptual model is presented by the work of Masuhara and Kiczales [Masuhara and Kiczales, 2003] on modeling crosscutting in aspect-oriented mechanisms. Their aspect sand-box (ASB) [Masuhara and Kiczales, 2003, Ubayashi et al., 2005] framework generalizes over four mechanisms: Pointcut and Advice, Open Classes (OC), Traversal (TRV), and Compositor (CMP). PA and OC are implemented by AspectJ; TRV is found in Demeter; and CMP is realized by Hyper/J.

ASB represents each aspect mechanism as a weaver that combines an aspect program and a base program into a result computation (or program). In ASB, an aspect mechanism realizes a function with the signature:

$$A \times B \times META \rightarrow X$$

where $A$ and $B$ denote domains of aspect and base programs, respectively; $X$ is a domain of composed computations (programs); and $META$ stands for composition rules existing only in CMP.

The aspect mechanism is said to model a weaving process, which is defined informally as [Masuhara and Kiczales, 2003]:

“taking two programs and coordinating their coming together into a single combined computation.”

The weaver’s semantics is presented by an 11-tuple structure:

$$\langle X, X_{JP}, A, A_{ID}, A_{EFF}, A_{MOD}, B, B_{ID}, B_{EFF}, B_{MOD}, META \rangle$$

where $A$ and $B$ denote the languages of the input programs, $X$ is the result domain of the weaving process, and $X_{JP}$ are join points in $X$. The elements $A_{ID}, A_{EFF}$ of $A$ and $B_{ID}, B_{EFF}$

---

7 For all of the other mechanisms, $META$ is left out of the model.
of $B$ provide a common frame of reference. Programs in $A$ and $B$ refer to $X_{JP}$ using $A_{ID}$ and $B_{ID}$ and contribute their $A_{EFF}$ and $B_{EFF}$ effects, respectively, to the semantics of the corresponding $X$ computations (program declarations). The remaining two elements, $A_{MOD}$ and $B_{MOD}$, refer to structures of modularity in the input languages.

The introduction of a result domain $X$ is the main contribution of ASB. In ASB, join points exist only in $X$. An aspect mechanism combines semantics contributed by $A$ and $B$ at join points in $X$ using the common frame of reference. $A$ and $B$ do not crosscut each other directly, but only with respect to $X$.

The essential shortcomings of ASB are:

- **Syntactical dichotomy.** ASB defines aspect mechanisms over an AOP language of a fixed syntactical form. Specifically, an AOP program is required to consist of an aspect program $p_A \in A$ and base program $p_B \in B$, both specifying concerns. When the syntax of the AOP language is different, the model includes it as a special case. Most noticeable, the $META$ component was added so that ASB can also model Hyper/J. The definition of an aspect mechanism in ASB does not generalize neatly over languages with similar semantics but a different syntactical structure. For example, Classpects [Rajan and Sullivan, 2005] has no aspects, just classes. AspectWerkz [Bonér, 2004] has no advice, just methods. In AspectWerkz, a method annotated as advice can be invoked explicitly as a method or implicitly as advice. This duality is difficult to explain in an ASB model due to its firm syntactical distinction between aspect ($p_A \in A$) and base ($p_B \in B$) programs. Furthermore, the annotations in AspectWerkz can be extracted and placed in a separate XML file. These XML integration rules are not in $A$, $B$, or $X$.

- **AspectJ-based abstraction.** ASB selects $X_{JP}$ as the central abstraction found in all aspect mechanisms. In ASB terms, the common frame of reference is $X_{JP}$ and the programs connect at the join points. While the concept of a join point is essential in
PA, it is found in only a subset of existing aspect mechanisms. Specifically, OC may be explained with or without join points (Section 4.5); it is unclear that join points are the right abstraction for TRV [Lieberherr and Lorenz, 2005]; and CMP does not involve join points in X at all [Masuhara and Kiczales, 2003, Ubayashi et al., 2005].

- **Coarse-grained process.** In ASB, a join point in \( X_{JP} \) describes a computation in \( X \), and the computation in \( X \) is constructed using the join point in \( X_{JP} \). Obviously, an aspect mechanism must construct the join point prior to the construction of the corresponding computation it describes. ASB does not explain how this cyclic dependency is resolved, and generally lacks an explicit weaving process model.

- **Over generalization.** ASB takes upon itself to abstract over four mechanisms that do not necessarily share properties that can be reasonably generalized. PA and CMP are oblivious [Filman and Friedman, 2000] and provide integration mechanisms. TRV, on the other hand, is not oblivious and provides an adaptation mechanism [Lieberherr and Lorenz, 2005]. Yet, ASB generalizes over all of them [Ubayashi et al., 2005]. To reconcile the difference between these four mechanisms, ASB provides a fine-enough grained structure. ASB’s bottom-up generalization underscores the similarities between various aspect mechanisms. What is lost in the generalization, however, is the ability to also understand their differences.

As Table 2.1 shows, existing models of aspect mechanisms do not exhibit a desired level of abstraction, precision, and uniformity. Semantical models are precise, but neither abstract nor uniform. Conceptual models are abstract, but lack uniformity and precision.

### 2.3 Composition of Aspects

The aspect extension composition problem [Kojarski and Lorenz, 2005] is fundamentally different from the aspect composition problem [Lopez-Herrejon et al., 2006]. The former
concerns **extensions**, while the latter concerns **aspects**. The problem of composing extensions is the problem of defining the semantics of a new multi-extension language. In contrast, the problem of composing aspects is the problem of specifying the behavior of an AOP program under the pre-defined semantics of the AOP language.

Lopez-Herrejon et al. [Lopez-Herrejon et al., 2006] study the problem of undefined semantics for aspect compositions in AspectJ. Although the problem of undefined semantics in a single-extension AOP language is orthogonal to the topic of this research, their algebraic model can be used to specify multi-extension aspect composition more flexibly.

### 2.4 Composition of Aspect Extensions

The idea of domain specific aspect-oriented extensions is not new [De Volder et al., 2001, D’Hondt and D’Hondt, 1999], but very few of the related work deal with making such extensions available concurrently. This section reviews the most significant works in this area, namely AspectJ/5, Reflex and XAspects:

- The latest version and compiler for AspectJ/5 [Colyer, 2005a], which is a composition of AspectJ 1.2 [Kiczales et al., 2001] and AspectWerkz [Bonér, 2004].

- Reflex [Tanter and Noyé, 2005], which is a multi-extension AOP kernel based on an intermediate reflective representation for implementing AOP extensions and resolving aspect interactions. It includes a plugin for a subset of AspectJ.

- XAspects [Shonle et al., 2003], which is a framework for composing aspects written in different AOP languages by translating them to AspectJ. It includes support for AspectJ and COOL.

#### 2.4.1 Reflex

Tanter et al. [Tanter and Noyé, 2005] present a framework for resolving **aspect interactions**. Concretely, the designer can resolve “aspect interactions [that] occur when several
aspects affect the same program point", namely ordering and nesting of aspects, mutual exclusion (an aspect should not apply whenever another aspect applies), and implicit cut (an aspect should apply whenever another aspect applies).

The Reflex framework is also used to address the aspect extension composition problem, by reducing the problem of composing multi-extension aspects to the problem of composing multiple aspects in Reflex. The framework applies a translation approach: it takes as input an aspect program written in a source aspect extension and outputs an aspect program in Reflex [Tanter and Noyé, 2005]. A designer of a multi-extension composition then uses Reflex to resolve interactions between the translated aspects.

An aspect program in Reflex consists of two parts, namely **metaobjects** and a **configuration**. Intuitively, a metaobject encapsulates behavior of a crosscutting concern, and the configuration specifies concern integration logic. Given the configuration, the framework’s weaver binds metaobjects to the other program modules.

For example, consider a StackLogger aspect in AspectJ (Listing 2.10) that logs executions of the Stack.push method (Listing 2.3). Reflex translates the aspect to a set of Java classes. The metaobject class StackLogger (Listing 2.11) implements the body of the aspect’s advice as the adv_0 method. The configuration classes ClSelector (Listing 2.12) and OpSelector (Listing 2.13) implement matching logic of the aspect’s pointcut. The ClSelector.accept method matches on an enclosing class of a join point, and selects join points that are defined within the Stack class. The OpSelector.accept method matches on a signature of a join point, and selects executions of a push method.

The top-level configuration class Config (Listing 2.14) binds the metaobject to the program classes in three steps. The first step creates a description of an aspect (line 10), and builds the pointcut selector (the hookSet variable, line 13). The second step links the pointcut to advice (lines 17 - 21): the link variable associates the pointcut selector (hookset) with the metaobject; the call to the setMOCall method associates the hookSet pointcut with the advice method and advice type (Control.BEFORE); and the call to the aspAspect.addAd-
public aspect StackLogger {
  before(): execution(* Stack.push(..)) {
    System.out.println("Stack.push execution");
  }
}

Listing 2.10: A stack logger aspect in AspectJ

package metaobjects;

public class StackLogger implements BMetaobject {
  public void adv_0() {
    System.out.println("Stack.push execution");
  }

  public static StackLogger aspectOf()
      return ASPECT_INSTANCE;

  private static StackLogger ASPECT_INSTANCE = new StackLogger();
}

Listing 2.11: A translated metaobject class in Java

public class ClSelector implements ClassSelector{
  public boolean accept(RClass aClass) {
    if (!PATTERN0.matcher(aClass.getName()).matches()) return false;
    return true;
  }

  static final Pattern PATTERN0 = Pattern.compile("Stack \[[\[\]\]]*\)\);}

Listing 2.12: A class selector in Reflex

vice method saves the relation in the aspect description. The third step weaves the aspect by passing its description to the Reflex weaver (line 24).

Overall, Reflex translates aspects to metaobjects and configuration classes in a straightforward way. An advice is translated to a metaobject’s method with the same body; and a pointcut is translated to a set of selector classes. The selector classes realize the same join point selection logic as the pointcut.
public class OpSelector implements OperationSelector {
    public boolean accept(Operation aOp, RClass aClass) {
        MsgReceive theOp = (MsgReceive) aOp;
        if (!theOp.getName().equals("push")) return false;
        return true;
    }
}

Listing 2.13: An operation selector in Reflex

public class Config {

    public static void initReflex(){
        // Plugin API access
        AJPAPI theAPI = (AJPAPI) API.plugins().getPluginAPI("AJP");
        PluginLinksAPI theLinkAPI = theAPI.links();
        AspectsAPI theAspectAPI = theAPI.aspects();

        // 1. Instantiating an aspect descriptor object
        Aspect aspAspect = theAspectAPI.createAspect(
            "StackLogger", "metaobjects");
        // and a pointcut selector object
        PrimitiveHookset hookSet = new PrimitiveHookset(
            MsgReceive.class, new ClSelector(), new OpSelector());

        // 2. Linking the pointcut selector to the metaobject
        AJPBLink link = (AJPBLink)theLinkAPI.createBLink(
            hookSet, "metaobjects.StackLogger");
        hookSet.setMOCall(Control.BEFORE,
            new CallDescriptor("metaobjects.StackLogger", "adv_0");
        aspAspect.addAdvice(Kind.BEFORE, link);

        // 3. Weaving the aspect
        theAspectAPI.addAspect(aspAspect);
    }
}

Listing 2.14: The main configuration class in Reflex

2.4.2 XAspects

Shonle et al. [Shonle et al., 2003] present a framework for compiling aspects written in
multiple domain specific aspect extensions. The XAspects framework uses a translation-based
approach. The framework’s composition semantics is to reduce all extensions to a single general
For example, consider a Stack coordinator shown in Listing 2.15. The \texttt{mutex} exclusion set \{push, pop\} specifies that push may not be executed by a thread while pop is being executed by a different thread, and vice versa. In addition, the \texttt{selfex} exclusion set prohibits different threads from simultaneously executing either push or pop.\(^8\)

The XAspects framework translates the coordinator to the \texttt{StackCoord} aspect in AspectJ, shown in Listing 2.16.\(^9\) The aspect implements a monitor using two condition variables \texttt{pop\_thread} and \texttt{push\_thread}. Using two pieces of \texttt{around} advice, the aspect obtains locks (\texttt{pop\_thread} and \texttt{push\_thread}) for the duration of executing \texttt{proceed} (execution of \texttt{pop} and \texttt{push}, respectively). This guarantees that no more than one thread operates on the stack at a time. If \texttt{pop\_thread} or \texttt{push\_thread} are locked by some other thread, the advice waits. When the thread has a lock, it runs \texttt{proceed} and afterwards releases the lock by signaling \texttt{notifyAll()}, which in turn wakes up other waiting threads.

2.4.3 AspectJ/5

To a large extent, AspectJ/5 can be viewed as the product of integrating features of AspectJ 1.2 with features of AspectWerkz [Bonér, 2005]. AspectJ/5 supports both AspectJ- and AspectWerkz-style aspects. For example, AspectJ/5 can apply AspectJ’s \texttt{AJSimpleLogger} (Listing 2.1) together with and AspectWerkz’s \texttt{AWSimpleLogger} (Listing 2.2).

\(^8\) However, the same thread is not prohibited from entering both push and pop.

\(^9\) We slightly edited the aspect for readability.
The AspectJ/5 compiler extends the AspectJ 1.2 compiler with AspectWerkz features. The AspectJ/5 compiler ensures that both AspectJ- and AspectWerkz-style aspects are executed under semantics of AspectJ. Particularly, AspectJ/5 orders advice of AspectWerkz-style aspects using advice ordering rules of AspectJ. For example, AspectJ/5 orders the advice of the AWSimpleLogger aspect (Listing 2.2) differently than AspectWerkz. At every method execution join point, AspectWerkz runs the before advice (the beforeAdvice method, lines 28...
32

- 30) first, then the **around** advice (the `aroundAdvice` method, lines 19 – 25), and finally the **after** advice (the `afterAdvice` method, lines 14 – 16). In AspectJ/5, the **around** advice dominates (i.e., “wraps”) over the **after** and **before** advice.
Chapter 3

Motivation

Composition of aspect extensions is an important and difficult problem that is not properly addressed by the current composition tools. Section 3.1 emphasizes importance of the problem by discussing advantages of using multi-extension AOP languages over single-extension AOP languages. Section 3.2 analyzes existing tools and demonstrates that they fail to compose extensions correctly. Section 3.3 analyzes current composition techniques and explains why it is difficult to use them for composing aspect extensions.

3.1 Significance of the Composition Problem

The need for the simultaneous use of multiple aspect extensions stems mainly from the favorable trade-offs that a domain-specific aspect extension can offer over a general-purpose extension:

Abstraction. A general-purpose aspect extension offers low-level abstractions for covering a wide range of crosscutting concerns. Domain-specific aspect extensions, in contrast, can offer abstractions more appropriate for the crosscutting cases in the domain at hand, letting the programmer concentrate on the problem, rather than on low-level details.

Experimentation. AOP is still at an active stage of research and development. Many new features are being proposed and evaluated in different domain-specific aspect exten-
sions. Composition of these extensions with each other and with main stream ones is important for making them accessible for experimentation and use in realistic settings, and can broaden their impact. The ability to program in multiple extensions can also help compare the features.

**Granularity.** The granularity of an aspect extension dictates all possible concern effect points within an application. Combining domain-specific aspect extensions allows to overcome the fixed granularity limitation of general-purpose AOP languages [Lopes et al., 2003].

**Expressiveness vs. Complexity.** The granularity of a general-purpose AOP language exposes a non-linear relationship between the language expressiveness and complexity. An increase in the language granularity significantly increases the language complexity while achieving a relatively small increase in expressiveness. Domain-specific aspect extensions, in contrast, can offer independent diverse ontologies [Wand, 2003].

**Reuse.** The need also arises from the sheer abundance of available aspect extensions (and their evolving aspect libraries). For the Java programming language alone there are numerous aspect extensions that are being used in a variety of commercial and research projects. These include: AspectJ (ajc [Colyer, 2005b] and abc [Avgustinov et al., 2005]), AspectWerkz [Bonér, 2004], COOL [Lopes, 1997], JBoss-AOP [JBoss], JAsCo [Vanderperren et al., 2005], Object Teams [Herrmann, 2002], ComposeJ [Wichman, 1999], to name just a few.\(^1\) Ability to use these aspect extensions together will allow to reuse existing (and future) aspect libraries written for the different aspect extensions.

As illustration, consider the bounded stack example implemented in Java (Listing 2.3).

Suppose you have three aspect extensions to Java at your disposal:

---

\(^1\) For a complete list of commercial and research aspect extensions see [http://www.aosd.net/technology/](http://www.aosd.net/technology/)
• COOL [Lopes, 1997]—a domain-specific aspect extension for expressing coordination of threads;

• AspectWerkz [Bonér, 2004]—a general-purpose lightweight AOP framework for Java;

• AspectJ—a general-purpose aspect extension for Java;

and two concerns to address, namely, a synchronization concern and a tracing concern. A comparison of COOL and AspectJ motivates the vision of domain-specific aspect languages (aspect DSLs). A composition of AspectJ and AspectWerkz illustrates the benefits of reuse.

3.1.1 Aspect DSLs

The synchronization concern can be expressed as a coordinator aspect in COOL (e.g., Listing 2.15) or alternatively as an aspect in AspectJ (e.g., Listing 2.16). The coordinator provides an elegant declarative description of the desired synchronization. The COOL code is expressive, concise, readable, and easy to understand. It provides the right abstractions. Studies [Murphy et al., 2001a, Walker et al., 1999, Murphy et al., 1999, Walker et al., 1995] have shown that “participants could look at COOL code and understand its effect without having to analyze vast parts of the rest of the code”, and that “COOL as a synchronization aspect language eased the debugging of multi-threaded programs, compared to the ability to debug the same program written in Java” [Walker et al., 1998].

While it is possible to express the same concern in AspectJ, the code will be much longer. In comparison to the COOL code, the AspectJ implementation (Listing 2.16) requires 10 times more lines of code. Moreover, the imperative code of the aspect is much harder to explain and understand than the declarative code of the coordinator.

3.1.2 Reuse

Syntactical differences between AspectJ and AspectWerkz present programmers with a desired choice of alternatives. AspectJ distinguishes between aspects and classes at the syntac-
tic level, and supports aspect-specific abstractions and mechanisms (e.g., aspect inheritance). In AspectWerkz, aspects are defined as annotated Java classes. AspectWerkz enables straightforward adoption of AOP features by Java programmers, but supports a smaller set of AOP abstractions (e.g., lack of advice-execution join points).

AspectJ and AspectWerkz were created and evolved separately. Despite similarities, their aspects were not readily compatible: an AspectJ 1.2 (AspectWerkz) programmer cannot use AspectWerkz (AspectJ) aspects in her programs. AspectJ and AspectWerkz teams recognized the problem, and manually merged the two extensions in the AspectJ/5 platform [Bonér, 2005]. The merger allows aspects like those in Listing 2.2 and Listing 2.1 to run side by side.

Although AspectJ/5 composess AspectJ and AspectWerkz successfully, manual composition is not an adequate approach to the aspect extension composition problem. The manual approach is expensive, non-extensible, and error-prone. The AspectJ/5 compiler is built by changing the implementation of the AspectJ 1.2 compiler in an intrusive, sophisticated way. Further extensions of the AspectJ/5 compiler (e.g., to support COOL) would require similar (if not larger) effort. Moreover, for some aspect programs the AspectJ/5 compiler exhibits unexpected weaving behavior (Section 2.4.3).

Arguably, if AspectWerkz and AspectJ were designed to be composable third-party aspect mechanisms, building AspectJ/5 would have been much easier. Moreover, third-party composition of aspect mechanisms would have made other domain-specific combinations possible, like combining COOL with AspectWerkz and AspectJ. A disciplined composition methodology would also help to avoid composition-related weaving errors and underspecifications.

3.2 Lack of Support for Composing Aspect Extensions

A current trend in language and tool support for aspect-oriented software development (AOSD) [Filman et al., 2005] is to provide “multi-language” aspect-oriented frameworks which
seamlessly integrate several AOP languages. Some development environments, such as IBM’s CME, even strive to provide developers with “a common platform in which different AOSD tools can interoperate and integrate” [IBM].

Unfortunately, there is no methodology to facilitate the construction of multi-extension aspect-oriented frameworks. Concrete integrations are generally built ad-hoc. We examined the level of integration support and tested the quality of the integrated product in the three state-of-the-art bodies of code, namely AspectJ/5, Reflex, and XAspects. We found that, from a user perspective, the available compositions exhibit obscure, unexpected, and even arguably incorrect behavior. Reflex handles advice weaving erroneously; AspectJ/5 exhibits unexpected behavior [Lorenz and Kojarski, 2006b]; and programs pre-processed in XAspects may behave incorrectly [Kojarski and Lorenz, 2005]. From a tool developer perspective, implementing such integrations in the respective frameworks is difficult, sometimes impossible.

3.2.1 AspectJ/5

We analyzed the AspectJ/5 compiler, and found that it is exhibits unexpected behavior. Consider advice definitions in AspectJ and AspectWerkz. AspectJ differentiates syntactically between a method and an advice; whereas AspectWerkz does not. In AspectWerkz an advice is an annotated method with a dual purpose. The method may be executed implicitly when playing the role of advice. It may also be invoked explicitly as a regular Java method [Arnold and Gosling, 1996].

This deceivingly small difference between AspectJ and AspectWerkz is of larger significance in their composition. Specifically, the method–advice duality poses a question:

Should executions of AspectWerkz advice methods in AspectJ/5 generate advice-execution join points?

This is not an issue in AspectJ 1.2, which possesses no such duality. It is also not an issue in

---

\(^2\) The term multi-language is a misnomer since it really refers to multiple aspect extensions to the same base language.
public aspect AJAdviceLogger {
  before(): adviceexecution()
    && !cflow(within(AJAdviceLogger)) {
      System.out.println("AJAdviceLogger:
        + thisJoinPoint);
    }
}

Listing 3.1: AJAdviceLogger.java

AspectWerkz, which lacks advice execution join points. Rather, the question *emerges* in the composition.

Interestingly, in AspectJ/5 advice-execution join points are generated for both explicit and implicit advice method invocations. Consider the AspectJ aspect, AJAdviceLogger (Listing 3.1), which logs advice executions, and the AspectWerkz aspect, AWAspect (Listing 3.2), which logs calls to `foo` using a `beforeAdvice` method annotated as advice. AJAdviceLogger logs not only implicit `beforeAdvice` executions, but also executions that follow explicit method calls. E.g., running the program in Listing 3.3 would result in the trace:

AJAdviceLogger:execution(ADVICE: void aj5.AWAspect.beforeAdvice(Object))

AWAspect: before foo method call on base.Main

Main: foo execution

AJAdviceLogger:execution(ADVICE: void aj5.AWAspect.beforeAdvice(Object))

AWAspect: before foo method call on null

@Aspect
public class AWAspect {
  @Before("call(\*.foo(..)) && target(o)")
  public void beforeAdvice(Object o) {
    System.out.println("AWAspect:"
        + " before foo method call on "
        + o);
  }
}

Listing 3.2: AWAspect.java
In AspectJ/5 the call

new Main().foo();

invokes beforeAdvice as advice; whereas the call

new AWAspect().beforeAdvice(null);

invokes beforeAdvice as a method. The trace illustrates that AJAdviceLogger logs both executions of beforeAdvice as advice executions.

In our view, AspectJ/5 generates unexpected advice-execution join points without any actual advice execution. A more intuitive composition of AspectJ and AspectWerkz would establish advice-execution join points only for implicit advice method invocations.

### 3.2.2 Reflex

While the Reflex framework provides adequate support for resolving aspect-level interactions in a specific Reflex program, it does not provide any means for resolving extension-level interactions in the multi-extension AOP language. Most notably, the framework lacks customizability necessary for preventing aspects from “misadvising” foreign aspects. This deficiency renders the Reflex framework inapplicable for composing aspect extensions in general.

---

3 Extension-level interactions are interactions between features of the composed extensions.
For example, the empty advice in the `AJNothing` aspect (Listing 3.4) is supposed to do nothing but results in an infinite loop when weaved by Reflex. The source of the problem is that Reflex translates the `before` advice to the empty method `adv_0` in the target metaobject class `AJNothing` (Listing 3.5), while preserving the pointcut’s logic in the translated selector classes `ClSelector` (Listing 3.6) and `OpSelector` (Listing 3.7). The translated Reflex configuration then erroneously observes and advises executions of the advice method as `method-execution` join points.\(^4\) This simple failure illustrates an inability of Reflex to compose aspect extensions correctly.

\(^4\) The configuration class `Config` binds the `AJNothing` metaobject to the selector classes in the same manner as `Config` shown in Listing 2.11 binds the `StackLogger` metaobject.
public class OpSelector implements OperationSelector {
    public boolean accept(Operation aOp, RClass aClass) {
        MsgReceive theOp = (MsgReceive) aOp;
        return true;
    }
}

Listing 3.7: An operation selector for AJNothing

3.2.3 XAspects

The XAspects framework exhibits the same problem as Reflex. For example, consider the Stack coordinator in Listing 2.15. XAspects translates the coordinator to the AspectJ aspect in Listing 2.16. While the translated aspect may seem to be a correct substitution for the COOL coordination aspect, in the presence of the AJStackLogger aspect (Listing 3.8) it is actually not. AJStackLogger is a call stack logger aspect in AspectJ. The toLog() pointcut specifies that every method call (not in the control flow of a AJCallStackLogger join point) should be logged. The aspect prints a current call stack at entry and exit points of every call. The aspect inspects only the first MAX_DEPTH calls in the stack, while the other calls are ignored.

A property of the COOL synchronization concern is transparency with respect to the AspectJ logging concerns. There should not be any interference between the two. The COOL aspect does not contain any join points that should be visible to the AspectJ mechanism. This property is not preserved by XAspects. Calls to wait (Listing 2.16, lines 11 and 26) and notifyAll (Listing 2.16, lines 16 and 32), which do not exist in the COOL code, will nonetheless be unexpectedly reflected by the stack logger.\textsuperscript{5}

Worse yet, the unexpected join points in the target program may break existing invariants, resulting in our case in a deadlock. An implicit invariant of the COOL aspect is that if both push and pop are not currently executing by some other thread, then the thread can enter and execute them. The translated AspectJ synchronization aspect, however, violates this invariant. Assume

\textsuperscript{5} Note that calls to wait and notifyAll cannot be avoided.
public privileged aspect AJCallStackLogger {

  pointcut toLog():
    call(* *.*(..)) && !cflow(within(AJCallStackLogger));

  before(): toLog() {
    try{
      stack.push(thisJoinPoint);
      printCallStack("ENTER");
    } catch (Exception e) {} 
  }

  after() returning: toLog() {
    printCallStack("EXIT");
    stack.pop();
  }

  after() throwing: toLog() {
    printCallStack("THROW");
    stack.pop();
  }

  protected synchronized void printCallStack(String message) {
    System.out.println(message+":");
    Object[] jps = stack.buf;
    for (int i=jps.length-1;i>=0;i--)
      if (jps[i]!=null) System.out.println(jps[i]);
  }

  private Stack stack = new Stack(MAX_DEPTH);
  static final int MAX_DEPTH = 10;
}

Listing 3.8: Call stack logger in AspectJ

that two threads concurrently access the buffer. The first thread acquires the lock, while the second invokes wait on the StackCoord object. However, before wait is invoked, the AJCallStackLogger aspect calls Stack.push (Listing 3.8, line 8). The latter call causes the second thread to enter the guarded code again and trigger a second call to wait. Since the second wait call is in the cflow of the logger, it is not advised, and the thread finally

6 Assuming that the first thread still owns the lock.
suspends. When the first thread releases the lock, the second thread wakes up after the second
wait. It acquires the lock, completes the advice execution, releases the lock, and proceeds to
the first wait invocation. At this point, the buffer is not locked; the second thread waits on the
StackCoord object monitor; and if no other thread ever accesses the buffer, the second thread
waits for ever—deadlock!

3.3 Complexity of the Composition Problem

The difficulty and complexity of the aspect extension composition problem is due to adverse interactions between features of the composed extensions. The interactions generally involve all the extensions, they are specific to the interacting parties, and can be resolved in multiple reasonable ways. Failure to identify and resolve these interactions propagates to unexpected and incorrect behavior in a multi-extension AOP language.

The source of the feature interaction problem lies in the invasive nature of aspects. A program in a multi-extension AOP language (a multi-extension program) contains aspects written in various aspect extensions. Aspects written in one extension generally advise not only code written in that extension and in the base language, but also code written in the foreign extensions. Moreover, aspects written in different extensions may collaboratively affect the same join point. These behaviors are not defined by any of the composed extensions, but rather emerge in their composition.

For example, consider a composition of COOL and AspectJ. COOL extends Java with a method synchronization mechanism; AspectJ extends Java with an advice binding mechanism. The two extensions differ syntactically and semantically: COOL does not have pointcuts; join points in COOL are implicit and are not reflected in the syntax; and advice in COOL is defined using multiple separate terms and expressions. In the composition, features of COOL and AspectJ interact, raising many questions including, e.g.:

- What are the join points in COOL, and how do they correspond to join points in As-
Should the execution of an advice in COOL generate an advice execution join point in AspectJ?

- Should aspects in AspectJ be permitted to advise the expressions requires, on_exit, and on_entry, and if so, what precisely are the rules of engagement?

- Should the execution of mutex and selfex pre-conditions be advisable by aspects in AspectJ, and if so, what join points do such executions generate?

- Should COOL synchronize advice in AspectJ that apply to the same method, or just synchronize the body of the method?

A well-defined composition of COOL and AspectJ must identify and answer all such questions. However, there is no simple way to do so. Current techniques for composing aspect extensions, i.e., translation and sequential instrumentation are of no help.

3.3.1 No Correct Translation

In the translation approach, aspect programs in different aspect extensions are translated to a common target aspect extension. This approach is used in Reflex and XAspects. Unfortunately, a translation from one aspect language to another does not generally preserve the behavior of the source aspect program in the presence of other aspects. The source of the problem is that translation synthesizes and introduces implementation-specific operations into the target aspects. The implementation-specific operations are not explicit in the code of the source aspect; they realize behavior of terms, expressions, and concepts that are specific to the source extension. For example, means of synchronizing threads that are implicit in a COOL coordinator might be translated into explicit wait and notifyAll method calls in a target aspect.

The implementation-specific operations thus introduce synthetic join points into a translated aspect program. The program aspects cannot distinguish the synthetic (unexpected) from
the genuine (expected) join points, and can erroneously advise the synthetic points. Advice executions at synthetic join points are not only unexpected to an aspect programmer, but also render the program incorrect.

### 3.3.2 No Correct Sequential Instrumentation

Aspect mechanisms can be implemented by means of program instrumentation. Such multiple independent aspect mechanisms can be trivially composed by passing the output of one aspect mechanism as the input to another aspect mechanism. Unfortunately, a multi-extension program that is sequentially instrumented by multiple aspect mechanisms generally exhibits unexpected behavior.

One would expect the two aspects written in AspectWerkz (Listing 2.2) and AspectJ (Listing 2.1) to interact as if they were two aspects written in a single aspect extension. On the one hand, the AspectJ logger should log executions of the `log` method within the `AWSimpleLogger` aspect. On the other hand, the AspectWerkz logger should log executions of the `log` method within `AJSimpleLogger`. (And both should log all method executions in the base program as well.)

However, applying the AspectJ and AspectWerkz instrumentation mechanisms sequentially, in any order, produces an unexpected result. The mechanism that is run first may not be able to interpret the second extension’s aspect program. Specifically, the AspectWerkz mechanism does not understand AspectJ’s syntax. It can be applied to the base program’s code but not to the `AJSimpleLogger` aspect. Thus, when AspectWerkz is run first, some expected log messages will be missing.

The mechanism that is run last logs executions of synthetic methods that are not supposed to be logged. For example, when AspectWerkz is run second, the following unexpected log message is generated by the `AWSimpleLogger` aspect:

```java
After:public void AJSimpleLogger.ajc$afterReturning(JoinPoint)
```
3.4 Summary

The available extensions vary in their potential effectiveness [Walker et al., 1999]. General purpose aspect extensions, e.g., AspectJ [Kiczales et al., 2001] or AspectWerkz [Bonér, 2004], express imperatively a wide range of crosscutting concerns, but do so at the price of complex aspect descriptions. In comparison, domain specific aspect extensions, e.g., COOL [Lopes, 1997], express domain concerns declaratively, but lack the expressiveness to tackle all cases of crosscutting.

The ability to concurrently use several aspect extensions, whether general purpose or domain specific, improves the overall effectiveness of AOP. It allows an aspect programmer to use both domain specific and general purpose abstractions in her programs; and reuse aspect libraries written in different aspect extensions. An ability to easily extend main stream aspect extensions with new ones facilitates the testing of the latter, and fosters their adoption.

Unfortunately, the aspect extension composition problem is a difficult one to solve. Complexity of the problem is due to interactions between the composed extensions. Simple approaches, such as translation and sequential instrumentation, do not resolve all the interactions. The current composition tools use these approaches and fail to compose aspect extensions correctly.
Chapter 4

The Component Abstraction Problem

The chapter introduces an abstract and precise model of AOP that outlines aspect and base mechanisms as separate components. The model reifies weaving processes in AOP language semantics. At an abstract level, we model weaving as concern integration process, and define it formally as a mapping between the concern program and the composed program domains. We then derive weaving process models (WPMs) of concrete AOP languages top-down, differentiating between reactive and nonreactive weaving processes. A reactive WPM comprises aspect mechanism and base mechanism components, and a nonreactive WPM models the aspect mechanism of an AOP language. In the sequel we refer to implementation of a weaving process as a weaving system.

The model is build through analysis and design. The analysis phase characterizes the external behavior of a weaving process; and design phase models its internal behavior. The novelty of the model lies in its semantical rather than syntactical generalization. We define domains of the weaving process over a semantical representation of an AOP program, which can be constructed from various syntactical forms. The model explains the essentials of existing weaving systems, and categorizes them according to semantical operations.

The model emphasizes integration. The integration logic is defined by integration rules, separate from concern code defined by aspect or base programs. The model provides a natural abstraction for integration-based weaving processes. It captures the similarities as well as
the differences between weaving processes on various levels of abstraction. The abstract WPM captures similarities between all the integration-based weaving processes, and its top-down specializations respect individual features of concrete weaving systems. For example, the top level WPMs of the Hyper/J and AspectJ weaving systems are similar; and a deeper level reveals how different they are.

The model introduces terms and abstractions that enable thorough analysis of the aspect extension composition problem. We formulated and analyzed the problem of composing aspect extensions as the problem of composing their weaving processes. The analysis captures the essentials of composing reactive weaving processes, and argues that nonreactive weaving processes are trivially composable.

The model provides a common framework for understanding and evaluating existing weaving systems. It gives an in-depth explanation of their key subprocesses. The model can also guide the implementor of new weaving systems. It can guide the designer when systems implementing new kinds of weaving are needed. It can also help teach aspect-oriented programming.

In Section 4.1 we overview the evolution of AOP models. In Section 4.2 we analyze the weaving problem and present a general CRX weaving process model (WPM). The problem and the model are defined in abstract terms. They are constructed top-down, independent from any particular weaving system. The top-down construction introduces different categories of a weaving process. In Sections 4.3 and 4.4 we design a nonreactive and a reactive specialization of CRX, respectively. We use concrete weaving systems to exemplify each category. In Section 4.5 we explain the duality of the AspectJ’s Open Classes (OC) weaving system with respect to these categories. Section 4.6 uses the model to analyze the aspect extension composition problem. The analysis narrows the scope of the problem to reactive aspect extensions. In Section 4.7 we compare the CRX model to the ASB model (Section 2.2), in terms of providing a deeper explanation of AOP. Section 4.8 discusses how the classification of weaving processes is different from existing categorizations of AOP languages. Finally, Section 4.9 discusses how
4.1 Evolution of AOP Models

Concern integration is an essential process in aspect-oriented programming. Kiczales et al. [Kiczales et al., 1997] explicitly express the goal of AOP in terms of combining separate crosscutting concerns:

“The goal of Aspect-Oriented Programming is to make it possible to deal with cross-cutting aspects of a system’s behavior as separately as possible. We want to allow programmers to first express each of a system’s aspects of concern in a separate and natural form, and then automatically combine those separate descriptions into a final executable form using a tool called an Aspect Weaver.”

The process of concern integration is called weaving, and the implementation of an “aspect weaver”—a weaving system. This chapter presents a top-down model of a weaving system and its internal weaving process.

A model primarily reflects understanding. We identify the following evolutionary stages in the common understanding of AOP:

1. \((A \times B \rightarrow B)\) In early AOP models, a weaving system was explained in terms of compilation semantics. This led to thinking about AOP in terms of code instrumentation, where an aspect program \((p_A \in A)\) is woven into a base program \((p_B, p'_B \in B)\). We refer to this initial understanding of AOP as the ABB model (Figure 4.1).

2. \((A \times B \rightarrow X)\) Later, code instrumentation was found to be misleading and confusing in explaining the essence of AOP. When AOP was better understood, the compilation semantics was replaced with dynamic weaving semantics. The main insight was that crosscutting is an emerging property found in the composed program \((p_X \in X)\) rather than a pre-existing property in either \(p_A \in A\) or \(p_B \in B\). We refer to this observation as the ABX model (Figure 4.2), named to correspond with Masuhara and Kiczales’
ASB model of crosscutting [Masuhara and Kiczales, 2003].

(3) \((C \times R \rightarrow X)\) More recently, the syntactical distinction between aspect (A) and base (B) has begun to diminish. In modern AOP languages, such as AspectWerkz [Bonér, 2004] and Classpects [Rajan and Sullivan, 2005], there is no essential distinction between aspects and classes. The traditional, syntactical dichotomy into \(p_A \in A\) and \(p_B \in B\) is being slowly replaced with a semantical distinction between concerns \(p_C \in C\).

---

1 For consistency with the other models and to avoid confusion we use A to denote the aspect language and B to denote the base language, whereas Masuhara and Kiczales originally used A for the base language and B for the aspect language.
Figure 4.3: The CRX Model

where

\[ C = A \cup B \setminus R \]

expressed in either A or B, and integration rules \( p_R \in R \), expressed in code, annotations, XML, or some other form. We introduce and refer to this model as a **CRX model** (Figure 4.3).

Interestingly, the CRX model is very close to the original Hyper/J model [Ossher et al., 1995], taking this evolutionary path a full circle. The introduction of the CRX model as a general AOP model that explains both Hyper/J-like and AspectJ-like weaving systems is a contribution of this chapter and a direct result of a top-down approach.

### 4.2 Analysis

This section explains the fundamentals of the concern integration process at an abstract level. We begin by articulating what a **weaving problem** is, and what is a solution, termed a weaving plan, to a weaving problem. We then present a high-level model of a weaving process that explains the external behavior of a weaving system.
4.2.1 The Weaving Problem

A weaving system implements a solution to a problem of concern integration. Intuitively, before separate concern elements can be “computed” they need to be woven together to produce a “computable” composed element. That is the job of a weaving system.

Per program, the problem of concern integration is to map the modularized program $P$, in which concerns are separated, to a composed program $\varphi$, in which the concerns are woven (Figure 4.4). We denote the mapping by:

$$P \xrightarrow{\text{weave}} \varphi$$

Abstractly, a program is either code or computation. A concern program, $P = \{c_1, \ldots, c_n\}$ is a set of $n$ program elements, $c_j \in C$, $j = 1, \ldots, n$, where $C$ is a domain of concern elements. A modularized concern program $P$ is partitioned into pairwise-disjointed concern modules. For example, in Figure 4.4, $P = M_1 \cup M_2 \cup M_3$.

A composed program, $\varphi = \{x_1, \ldots, x_m\}$ is a set of $m$ program elements, $x_i \in X$, $i = 1, \ldots, m$, where $X$ is a domain of composed elements and where the crosscutting occurs.

4.2.2 The Weaving Plan

A solution to a concern integration problem is a weaving plan. A weaving plan establishes a mapping between elements of the concern program $P$ and elements of the composed program $\varphi$. The weaving plan is a set of pairs,

$$wp = \{(\text{match}_i, \text{mix}_i) : x_i \in \varphi\}$$
where for every element $x_i$ in $\varphi$, $match_i \in 2^P$ is a subset of program elements, $mix_i : 2^P \rightarrow \varphi$ is a constructor, and the weaving plan satisfies the condition that $x_i \in \varphi$ is constructed by applying $mix_i$ to $match_i$. We denote this mapping by:

$$match_i \overset{mix_i}{\rightarrow} x_i$$

4.2.3 The Weaving Process

We model weaving as a rule-driven concern integration process. In this model, a plan $wp$ for integrating $P$ concerns into $\varphi$ is specified by a set of integration rules,

$$\varphi = \{r_1, \ldots, r_k\}$$
where $\mathcal{R}$ is a domain of integration rules, and $r_i \in \mathcal{R}, i = 1, \ldots, k$. A weaving system implements a **CRX weaving process** (Figure 4.5). At each step of the process, the system accesses elements of $P$ (the $C$-register), elements of $\varphi$ (the $\mathcal{R}$-register), and elements of the so-far-composed program $\varphi$ (the $X$-register). The system produces new elements of the composed program $\varphi$, and writes them to the $X$-register.

In this work we only consider weaving processes that transform a composed program strictly by adding new elements. It should be understood, however, that the composed program elements may be organized into a hierarchical structure, e.g., an Abstract Syntax Tree (AST); and that injection of a new element into this structure may update its enclosing elements. For example, consider a process of composing an AST and assume that a so-far-composed tree contains a method node $x_m$ within a class node $x_c$. Addition of a statement $x_s$ into the tree as a child of $x_m$ updates $x_m$ and $x_c$ elements within the tree. Specifically, $x_m$ is replaced with $x'_m$ that differs from the original one in that it contains $x_s$; and $x_c$ is replaced with $x'_c$ that differs from the original one only by containing $x'_m$ instead of $x_m$. In our analysis we ignore these updates as they are orthogonal to the process of weaving.

A weaving process is said to be **nonreactive** (Section 4.3) if does not depend on the state of the $X$-register. A weaving process is said to be **reactive** (Section 4.4) if it requires read-access to the $X$-register in order to produce the composed elements. In a nonreactive process, the integration rules $\varphi$ directly define the weaving plan $wp$. Given the concern program $P$, a nonreactive weaving system builds $wp$ from $\varphi$ in one step:

$$\varphi \xrightarrow{P} wp$$

In contrast, a reactive process maps the rules to the weaving plan in a step-wise manner. At each step $i$ of the process, a reactive weaving system computes $match_i$ and $mix_i$ from $P$, $\varphi$, $\rho$, and $\varphi_i$. (Note: The specific details of how this is done are not provided in the excerpt.)
and the current composed program:

\[
\langle \varrho, \{\} \rangle \xrightarrow{P} \langle \text{match}_1, \text{mix}_1 \rangle
\]
\[
\langle \varrho, \{x_1\} \rangle \xrightarrow{P} \langle \text{match}_2, \text{mix}_2 \rangle
\]
\[
\langle \varrho, \{x_1, x_2\} \rangle \xrightarrow{P} \langle \text{match}_3, \text{mix}_3 \rangle
\]
\[
\ldots
\]
\[
\langle \varrho, \{x_1, x_2, \ldots, x_{i-1}\} \rangle \xrightarrow{P} \langle \text{match}_i, \text{mix}_i \rangle
\]
\[
\ldots
\]

The top level CRX process is the most general model of a weaving system. In the next two sections we describe two specializations of weaving process. We explain the semantics of these models by mapping them to well known AOP languages.

### 4.2.4 Notation

We describe the weaving process using a sequence of data flow diagrams (DFDs) [Yourdon, 1989, Ross, 1977, Wieringa, 2003]. A DFD is a graph that models the data flow communication among processes and data stores. Circles represent transformations. Pairs of parallel lines represent data stores. Arrows represent data. Collectively, a sequence of DFDs specifies the behavior of a weaving system in terms of its internal processes that transform data and the type of data being transformed. At the top level, a context diagram describes the overall system and its data flow interfaces with its environment. The context diagram is then decomposed into level-1 processes, which are then decomposed into level 2 processes, and so on.

We use the standard DFD convention for labeling processes. A context level DFD is numbered \(n.0\). Level-1 processes are labeled \(n.1, n.2\), etc. Successive nested levels follow a similar numbering convention. This convention provides for easy tractability. We use the leading number \(n\) to relate a sequence of diagrams to the section where they are discussed. The first time a process is mentioned in the text, we indicate its label in parentheses.

Note that a DFD does not specify when communications take place. In particular the labels are just identification tags and do not imply any specific processing order.
4.3 Design of a Nonreactive Weaving Process

At each step of the weaving process, a nonreactive weaving system computes an element (or several elements) of the composed program $\varphi$ only from elements of $P$ and $\varnothing$. In other words, the integration rules of a nonreactive weaving system do not consult the state of the $X$-register in order to specify composed elements. In this section we informally introduce the inner working of a nonreactive weaving system through an overview of the weaving process in Hyper/J.

The weaving system of Hyper/J [Ossher et al., 1995, Tarr et al., 1999] is essentially a Java program transformer. In the model terms, hyperspace is the input program $P$; hyperslices corresponding to concerns, e.g., $M_1, M_2, M_3$ in Figure 4.4; a hypermodule is the composed program $\varphi$; and composition rules are the integration rules $\varnothing$ (Figure 4.6). $P$ keeps the concerns separate, but does not specify a coherent behavior. The process generates $\varphi$ with the desired
behavior by integrating the concerns together according to the composition rules ϕ. In general, the rules define each hypermodule unit as a composition of multiple hyperspace units.

The weaving system of Hyper/J is modeled by the COMPOSE (3.0) process (Figure 4.6). The process reads hyperspace units P and composition rules ϕ, and produces hypermodule program ϕ. The hypermodule program is an executable Java program. The meaning of the input program P in Hyper/J is the meaning of the hypermodule program ϕ in Java.

The DFD in Figure 4.7 is an explosion of COMPOSE (3.0). It realizes the two-step composition process of Hyper/J (Section 2.1.4). The first step is realized by the EXPAND (3.1) process, which expands the composition rules against the hyperspace unit tree into composition clauses. At the second step, the composition clauses and the hyperspace units are passed to the WEAVE (3.2) process, which composes the result hypermodule.

In terms of the CRX model, a composition clause defines a weaving plan element. At each step (i) of the weaving process, WEAVE (3.2) selects and applies a clause. The composition clause specifies what hyperspace units to combine (matchi) and how to combine them (mixi). The process completes the step by composing a new hypermodule unit (composed element xi) and committing it to the composed program. The process terminates after all clauses were applied.

For example, consider the Hyper/J program shown in Section 2.1.4. The Person class in Listing 2.5 describes a person from a personal perspective; an the Person class in Listing 2.6 gives a tax perspective. The Person hyperspace specification given in Listing 2.7 defines two hyperslices, namely, PersonalView and TaxView. The composition rules in Listing 2.8 specify integration of the two hyperslices into a Result hypermodule under the mergeByName composition strategy.

The Hyper/J weaving system weaves this program in two steps. At the first step, the EXPAND (3.1) process of the system reads the hyperslices and the composition rules, and outputs the composition clauses shown in Listing 2.9. At the second step, the WEAVE (3.2) process executes the clauses one-by-one, and outputs results into the Result hypermodule.
A weaving system of a nonreactive AOP language $\text{Base} \times \text{Ext}$ thus realizes only the semantics of the aspect extension $\text{Ext}$, i.e., the weaving system is the aspect mechanism $\mathcal{M}$ of $\text{Ext}$. For example, \textit{WEAVE} (3.2) models the aspect mechanism of Hyper/J. The aspect mechanism weaves an input concern program $P$, and produces the executable composed program $\wp$. Integration rules $\varrho$ of a nonreactive aspect program define \textit{constructor} functions for each element of the composed program $\wp$. The composed program is then executed by the base mechanism. The nonreactive AOP language thus naturally separates its aspect and base mechanisms, and applies them sequentially.

4.4 Design of a Reactive Weaving Process

In this section we model a reactive CRX process. We present the model through the explanation of AspectJ’s pointcut and advice (PA) weaving system. The context level DFD in Figure 4.8 depicts a reactive weaving process of a PA weaving system. The PA weaving system realizes a program execution process. Given a concern program $P$, integration rules $\varrho$, and a computation trace $\wp$, \textit{WEAVE} (4.0) executes the program by constructing, transforming, and running computations.

$P$ and $\varrho$ are constructed from an input program string. $P$ consists of all the methods and advice-body expressions found in either the base (e.g., Java classes) or the aspect (e.g., AspectJ aspects) parts of the input program. $\varrho$ is constructed from three sources:

1. pointcut designators;
2. pointcut-to-advice mapping; and
3. advice-to-type mapping.

The first two elements provide collaboratively a mapping from join points to advice-body expressions. This mapping is defined on a domain of join point descriptions which includes dynamic, static, and lexical context of a join point computation. The join point description domain essentially defines a (statically known) set of all the potential join points. The last element
provides a mapping from an advice-body expression to its advice type (\texttt{before}, \texttt{after}, or \texttt{around}), which is necessary for proper weaving.

\(\wp\) is a vector of computations that constitutes the composed program. \texttt{W\!E\!A\!V\!E} executes the program \(\wp\) by constructing computations and running them. At each step of the process, \texttt{W\!E\!A\!V\!E} reads in the most recent computation in the trace, and executes it. The execution produces either a null-computation that wraps around a value or a set of sub-computations that extend \(\wp\).

Figure 4.9 is an explosion of \texttt{W\!E\!A\!V\!E} (4.0). \texttt{ADVISE} (4.1) and \texttt{E\!V\!A\!L\!U\!A\!T\!E} (4.2) are level-1 subprocesses that realize aspect mechanism and base mechanism components of the weaving system, respectively. \texttt{E\!V\!A\!L\!U\!A\!T\!E} realizes an \texttt{expression evaluator} (a Java interpreter). This
process executes the most recent \( \wp \) computation, and generally produces one or more join point computations \( x_{jp} \). The ADVISE process intercepts a join point computation \( x_{jp} \), and transforms it by wrapping it with advice computations. The computation produced by the ADVISE process replaces \( x_{jp} \) in the program execution.

The PA weaving system represents a join point computation \( x_{jp} \) as a join point context \( ctx_{jp} \). The context provides the type of the join point computation (e.g., method call, method execution, constructor execution), values drawn from its evaluation context, and related static and lexical data. The domain of contexts \( Ctx \) defines an interface between the base mechanism and the aspect mechanism of the weaving system. The join point context \( ctx_{jp} \in Ctx \) is a means by which EVALUATE passes information about the program state to ADVISE. The ADVISE process then uses this information for selecting appropriate advice.

A further breakdown of the PA’s weaving process is depicted in Figure 4.10. Four sub-processes constitute a reactive weaving process. MATCH (4.1.1) and MIX (4.1.2) realize advice selection and advice weaving, respectively. MATCH selects pieces of advice by applying the \( jq \rightarrow \text{advice} \) mapping to \( ctx_{jp} \). The process returns the selected advice-body expressions as advice computations \( x_a \). MIX combines the advice computations with a join point computation

---

**Figure 4.10: DFD 4.1: ADVISE and DFD 4.2: EVALUATE**

---
$x_{jp}$ obtained from COMPUTE (4.2.1). The computation composed by MIX extends the execution trace $\varnothing$.

The CONTROL (4.2.2) process runs the most recent computation in $\varnothing$. In general, the computation is built from a complex expression. Running the computation requires evaluation of the expression’s subterms. CONTROL represents subterms as instructions, and delegates their evaluation to COMPUTE. Since the process runs MIX-computations, the instruction set may include both base language (Java) instructions (e.g., method invocation instruction), and aspect-specific (AspectJ) instructions (e.g., advice execution instruction). The COMPUTE process takes an instruction $instr$ and produces a join point computation $x_{jp}$. The process also constructs a join point context $ctx_{jp}$ that describes the computation.

In terms of the weaving problem, at each step $i$ of the weaving process COMPUTE and MATCH provide the $match_i$ elements:

$$match_i = \pi_a^i + \{x_{jp}^i\}$$

The join point element $x_{jp}^i$ is built by COMPUTE; and the advice elements $\pi_a^i$ are selected by MATCH. The MIX process realizes the $mix_i$ function that builds a composed program element $x_i$.

The MATCH and MIX processes apply the integration rules $\varnothing$, i.e., for each element $x_i$ of the composed program $\varnothing$ the processes provide advice elements $\pi_a^i$, and an advice weaving function $mix_i$. Thus, the reactive integration rules (in contrast with nonreactive rules) do not specify all the elements of the $match_i$ set. Furthermore, because $match_i = \pi_a^i + x_{jp}^i$, the rules specify a program element transformer function $\tau_i : X \rightarrow X'$, defined as:

$$\tau_i x_{jp}^i = mix_i (\pi_a^i + \{x_{jp}^i\}) = x_i$$

This shows an important difference between reactive and nonreactive aspect extensions: integration rules of a reactive extension define program element transformers, while integration rules of a nonreactive extension define program element constructors.
4.5 The Duality of Open Classes

Open Classes (OC) is a weaving system found in AspectJ that allows aspects to change the structure of a concern program. The system realizes the semantics of inter-type declarations (Section 2.1.1.2). An inter-type declaration construct associates a target Java type (what to change) with an OC effect (how to change it).

4.5.1 A Nonreactive OC Weaving Process

We can explain the OC weaving system in terms of a nonreactive CRX weaving process (Figure 4.11). The concern program $P$ contains OC effects and base program types (including aspect classes); the integration rules $\varphi$ map target types to the effects; and $\wp$ is a Java program where the types and the effects are combined.

The WEAVE (5.1) process iterates over $P$ types. For each type, WEAVE selects the relevant OC effects using the $\varphi$-specified mappings. Then it transforms the type by applying the effects; and finally it commits the transformation result to the composed program.

4.5.2 A Reactive OC Weaving Process

Interestingly, the OC weaving system can also be explained in terms of a static reactive CRX weaving process that iteratively constructs an AST of a composed program. For clarity, we focus on a simplified representation of AST that has four types of nodes, namely (from parent to children) root, type name, type declaration, and type member (Figure 4.12).
The DFD diagram in Figure 4.13 depicts an (alternative) reactive model for OC. The model combines three processes: MATCH (5.2), MIX (5.3), and SELECT (5.4). Initially, the composed program $\varphi$ contains only a single root node marked as open. At each step, the weaving process selects an open parent node in $\varphi$, populating its children with AST nodes drawn from $P$. That is, SELECT selects the parent from $\varphi$ and the parent’s “base” child nodes $child^*_B$ from the AST of the concern program $P$. MATCH selects OC effects associated with parent, and translates them to $child^*_A$ nodes.\footnote{\textit{child}^*_A are \textit{parent}’s children defined by inter-type declarations. For example, Vector Point.observers defines the observers field as a child of the Point class declaration.} The MIX process integrates the base and the
“advice” children together, and commits the result to the composed program. The child nodes are marked open, with the exception of type member nodes, which are marked closed. Weaving continues until all $\wp$ nodes are closed.

A reactive OC resembles PA. An open node parent corresponds to a join point context $ctx_{jp}$, $\text{child}^*_B$ correspond to a join point computation $x_{jp}$, and $\text{child}^*_A$ correspond to advice computations $\overline{x}_a$.

The nonreactive and reactive alternative OC models illustrate different understandings of OC integration rules. OC neither changes type names nor adds or removes types. The concern program types always end up in the composed program under the same name. This allows one to interpret the OC weaving plan mappings in two ways: as a mapping between concern types and effects, or as a mapping between composed types and effects. The first interpretation implies a nonreactive weaving process; and the second a reactive one.
4.6 Composing Aspect Mechanisms

In terms of the CRX model, the aspect extension composition problem can be reformulated as a problem of composing multi-extension integration rules (Figure 4.14) as follows:

**PROBLEM 2 (INTEGRATION RULES COMPOSITION)** Given a concern program $P$ and a set of integration rule programs

$$\mathcal{I} = \{\varrho_1, \varrho_2, \ldots, \varrho_n\}$$

written in $n$ different aspect extensions, build a composed program $\varphi$ by collaboratively applying the integration rules to compute $\text{match}_i$ and $\text{mix}_i$ for each element $x_i$ of $\varphi$, such that

$$\text{match}_i \xrightarrow{\text{mix}_i} x_i$$

4.6.1 Nonreactive extensions

Nonreactive aspect extensions are trivially composable because their integration rules do not collaborate. A nonreactive integration rule program specifies each element of a composed program as a composition of concern program elements. Consider $\varrho_i$ and $\varrho_j$, two integration rule programs for the same concern program $P$ that are written in different nonreactive aspect extensions. The $\varrho_i$ and $\varrho_j$ rule programs respectively specify different composed programs $\varphi_i$ and $\varphi_j$: $\varrho_i$ ($\varrho_j$) specifies each element of $\varphi_i$ ($\varphi_j$) as a composition of $P$ elements. $\varrho_i$ ($\varrho_j$) is defined (in the context of $P$) only for the elements of the $\varphi_i$ ($\varphi_j$) program. A collaborative application of $\varrho_i$ and $\varrho_j$ to $P$ thus results in a composed program $\varphi = \varphi_i + \varphi_j$.

4.6.2 Reactive extensions

The problem of composing $n$ compatible reactive aspect extensions\(^3\) is solvable because they define weaving as a transformation of composed program elements. The extensions thus

---

\(^3\) reactive aspect extensions are compatible if they can be defined using the same $X$, $C$, and $Ctx$ domains.
can be composed by composing their transformations at every join point (Figure 4.15).

More formally, let $\bar{\varrho} = \{\varrho_1, \varrho_2, \ldots, \varrho_n\}$ be a set of integration rule programs for the concern program $P$ that are written in $n$ different reactive aspect extensions. Each step $(i)$ of a reactive weaving process provides a join point element $x_{jp}^i$ and a join point context $ctx_{jp}^i$. For this step an application of an integration rule program $\varrho_j \in \bar{\varrho}$ yields a transformation function $\tau_j^i : X \rightarrow X$. A composition of the integration rule programs at $x_{jp}^i$ then can be realized as a composition of their transformation functions (Figure 4.15, the COMBINE process):

$$\tau_1^i \circ \cdots \circ (\tau_n^{n-1} \circ (\tau_n^1 x_{jp}^i)) \cdots = x_i$$

where $x_i$ is a composed program element that combines advice of all the aspect extensions.

The transformer composition approach not just illustrates the nature of aspect extension composability, but also solves the problem of composing OC-like static reactive aspect extensions. The solution enables collaborative weaving of multi-extension inter-type declarations into the same program classes. The problem of detecting and resolving conflicts between
the introduced declarations is studied elsewhere [Havinga et al., 2006]. This problem is not a unique attribute of the aspect extension composition problem (e.g., the conflicts may be found in AspectJ programs) and is outside the scope of this research.

The composition of transformers, however, is not a complete solution to the problem of composing dynamic reactive aspect extensions. Real-world compositions of dynamic reactive aspect extensions generally require more sophisticated solutions.

4.7 Beyond Conceptual Models

The CRX model affords deeper understanding of a weaving system and its internal weaving process beyond the ABX model and the ASB framework.

4.7.1 Abstract Domains

In CRX, the abstract AOP language representation overcomes the syntax dependency problem found in ABX. CRX defines the weaving system in terms of concern elements (domain C) and integration rules (domain R). These domains abstract away from any concrete representation. A program in a concrete AOP language syntax, e.g., in AspectJ, can be preprocessed, e.g., by a READ (6.0) process (Figure 4.16) that reads $p_A$ and $p_B$ in and writes $P$ and $\varrho$ out, which in turn can be used by WEAVE (4.0, Figure 4.8). Consequently, the CRX weaving process model generalizes over asymmetric (e.g., AspectJ) and symmetric (e.g., Hyper/J, AspectWerkz, and Classpects) AOP languages uniformly.
4.7.2 The Weaving Process as a Unifying Concept

The CRX process model reveals that join points are only used in reactive weaving processes. Nonreactive processes do not use join points because they do not read from the composed program register \( X \). In other words, the existence of a join point abstraction can be used to **distinguish** rather than **unify** various AOP languages, e.g., PA is a join point-based weaving system, CMP is not.

A CRX weaving process model was constructed top-down using abstract terms. The abstract terminology is useful for comparing different weaving processes. Zooming out highlights the similarities. Zooming in exposes differences. For example, at the top level both CMP and PA realize the same CRX weaving process. However, they fall into two different categories. CMP is a nonreactive, whereas PA is a reactive weaving system.

4.7.3 Cyclic Dependency in AspectJ

AspectJ has a cyclic dependency between a computation in \( X \) and the join point in \( X_{JP} \) that describes that computation. The CRX weaving model (recall Figure 4.10) explains how the cyclic dependency is resolved in the PA weaving process. Specifically, the join point context \( ctx_{jp} \) describes the join point computation \( x_{jp} \) rather than the composed program computation. Thus, the join point is created before the composed computation is constructed.

4.7.4 Advising Advice Execution

ASB’s view that PA selects and merges base and aspect program elements at each join point fails to explain weaving at advice execution join points. Clearly, at advice execution join points, **no** base element should be selected. The role of base is played by a piece of **advised** advice. ASB does not generalize over advice execution join points because it explains **advised** elements strictly as **base**.

In contrast, the CRX weaving model of PA (Figure 4.10) distinguishes between a join
point computation (an advised computation), and an advice computation. The join point computation \( x_{jp} \) is produced by the evaluator's \textsc{Compute} process as a result of running \( \varphi \) computations. Since \( \varphi \) generally contains previously woven advice computations, the join point computation might be an advice execution computation. Thus, the CRX weaving process model also explains how advice executions are advised.

### 4.7.5 Untangling the Base and the Aspect Mechanisms

The reactive CRX model explicitly separates base and aspect mechanisms of the weaving system: \textsc{Evaluate} (interpreting process) realizes the base mechanism, and \textsc{Advise} (advising process) realizes the aspect mechanism. \textsc{Evaluate} selects and computes the meaning of advised elements (e.g., methods). \textsc{Advise} selects advice, computes their meaning, and weaves them. In contrast, ASB modifies the interpreter to include aspectual operations. For example, in ASB's PA weaving system, calls to \texttt{lookup-advice} are scattered throughout the interpreter and tangled with calls to \texttt{lookup-method} [Masuhara and Kiczales, 2003].

### 4.8 Top-Down Classification

The top-down analysis yields a process-oriented classification of weaving systems. At the top level, we distinguish between a \textit{reactive} and a \textit{nonreactive} weaving system. In this section we compare this classification to existing categorizations of AOP languages. We show that our classification is different, and explain its usefulness.

#### 4.8.1 Symmetric versus Asymmetric Languages

Today, existing AOP languages are sometimes categorized using the \textit{syntactic} property of symmetry. Under this view, AspectJ is \textit{asymmetric}—an AspectJ program consists of a base Java program and a set of aspect definitions. Hyper/J is said to be \textit{symmetric}—all concerns are written in Java.
Unfortunately, this view does not provide much insight into the essential difference between AOP languages. It fails to explain how Hyper/J differs from AspectJ on a *semantical* level. Worse yet, under the symmetry-based view, “symmetrical” AspectJ-like languages (e.g., AspectWerkz and Classpects [Rajan and Sullivan, 2005]) fall into the category of Hyper/J, instead of sharing a category with AspectJ.

Our approach abstracts away from the syntax of a language, and focuses instead on the weaving process that realizes its semantics. We represent an input AOP program in a syntax-independent manner, through sets of concern elements and integration rules.

We explain the working of a weaving system according to its semantical operation. The reactive PA weaving process (Figure 4.10) explains AspectWerkz and Classpects in exactly the same manner as it explains AspectJ. Our classification identifies that AspectWerkz, Classpects, and AspectJ are all reactive weaving systems, and contrasts them with the nonreactive Hyper/J.

### 4.8.2 Static versus Dynamic Languages

Another classification approach categorizes AOP languages as *static* or *dynamic*. A static language expresses a weaving process over an abstract syntax tree (AST). For example, Hyper/J dictates the construction of a hypermodule unit tree from a hyperspace unit tree, where units represent AST nodes. Hyper/J is often referred to as a *static* AOP language. A dynamic AOP language allows the expression of concern integration logic over computations, using runtime abstractions. For example, advice weaving in AspectJ is defined in terms of join points in the program execution. AspectJ is often said to be a *dynamic* AOP language.

In our approach the weaving problem and the general CRX weaving process model are abstracted in terms of concern and composed elements. Identifying the composed and concern elements as AST nodes yields a weaving model for static languages. Defining the elements to be computations yields a weaving model for dynamic languages. Although it is tempting to think that nonreactive weaving systems are always static, and reactive weaving systems are always dynamic, it is not so. Specifically, the *static* OC weaving system can realize a *reactive*
weaving process. A reactive–nonreactive classification is, therefore, fundamentally different than a static–dynamic one.

4.8.3 Static versus Dynamic Semantics

Aside from the static–dynamic classification of AOP languages, there is an analogous classification of AOP language semantics. Semantics for dynamic AOP language can be specified in terms of code transformation. It is also possible for static languages or weaving systems to be given dynamic semantics.

In our approach, a weaving process always maps to a specific semantics, either static or dynamic. Alternative semantics for the same AOP language are normally realized by different weaving processes. For example, compilation semantics for AspectJ can be realized by a static nonreactive weaving system; dynamic semantics for AspectJ can be realized by a dynamic reactive weaving system.

4.9 Top-Down Construction

A property of our top-down approach for modeling a weaving process is the ability to systematically construct new weaving processes. The processes can be derived in a step-wise refinement of the general CRX weaving process model (Figure 4.5). For example, the reactive PA weaving process model (Figure 4.10), which refines the general CRX weaving process model, can be further specialized to produce a concrete weaving process specification.

4.9.1 Component-Based Design

In our approach a weaving system is modeled by a set of collaborating subprocesses. A component-based design for the weaving system would support its decomposition into distinct processes. In such a component-based design, it would be possible to replace and reuse subprocesses, thus facilitating weaving system evolution [Lorenz and Vlissides, 2001]. For example, the reactive PA weaving process model decouples the interpreter functionality from the
aspectual advice selection and weaving processes [Lorenz and Vlissides, 2003]. The interpreter operations are realized by the CONTROL (4.2.2) and COMPUTE (4.2.1). MATCH (4.1.1) and MIX (4.1.2) realize the match and mix aspectual functionality.

Decomposing the weaving system into subcomponents allows to replace one component at a time. Consider two reasonable enhancements to AspectJ: extending the pointcut designators with new regular expressions, and adding new types of advice. Each enhancement requires replacement of exactly one component. Extending the original MATCH component to handle new regular expressions would enable the pointcut designators enhancement. To extend the advice type set, the MIX component should be updated.

4.9.2 New AOP Functionality

Most (if not all) the weaving processes that exist to date construct the composed program by extending it with new elements. For example, the advice weaving process in PA simply extends, at each step, the composed program’s computation trace with new computations. AspectJ never transforms computations within the composed program (i.e., an already executed or currently active computations). Similarly, the weaving processes of CMP and OC never change elements of the composed program.

The general CRX weaving process model (Figure 4.5) lends itself toward inventing novel weaving functionality beyond current practices. The model defines a weaving system as a composed program transformer. At each step of the weaving process, the system transforms the composed program $\varphi$. Under this view, the process may include steps that actually modify or replace elements in $\varphi$. For example, one can imagine a dynamic reactive process that would update currently active computations (e.g., transform the current continuation). Another example would be a static reactive process that can extend $\varphi$ with new AST nodes, and transform previously composed AST nodes.
4.10 Summary

This chapter characterizes the design space for weaving processes of AOP languages. A weaving system is an implementation of an AOP language semantics. We model weaving as the process of concern integration, and define the weaving process in an abstract way, independent of any specific AOP language. The abstract model describes the design space, and concrete weaving systems are derived top-down from the abstract model. We distinguish between reactive and nonreactive weaving processes. This taxonomy provides a common framework for comparing and contrasting different AOP languages.

We offer a new CRX model as the next evolutionary step in modeling AOP languages. In CRX, integration rules in $R$ govern the process of integrating concerns in $C$ into crosscutting structures in $X$. We formalize the CRX model as a weaving problem over corresponding abstract domains $R$, $C$, and $X$. A CRX weaving process executes a solution to a weaving problem.

The weaving processes are classified as either reactive or nonreactive, instead of the more classic static–dynamic or symmetric–asymmetric classifications of AOP languages. We design a nonreactive CRX process for the static CMP weaving system of the symmetric Hyper/J language. We design a reactive CRX process for the dynamic PA weaving system of the asymmetric AspectJ language. We also design both a nonreactive and a reactive CRX processes for the static OC weaving system of AspectJ.

The top-down model overcomes many limitations in other models in terms of abstraction, precision, and uniformity. The model was not generalized from a particular set of AOP languages, yet existing AOP languages can be mapped, analyzed, and explained in terms of this model. In contrast, other models are generally constructed bottom-up by analyzing existing weaving systems. The generality of these bottom-up models is illustrated by expressing in terms of the model only the weaving systems used to derive them.

The design space for CRX weaving processes can help not only classify existing AOP languages but also develop new ones. It can guide the implementor of new weaving systems in
existing classes. It can also guide the designer when weaving systems implementing new kinds of weaving are needed. We used the CRX model in researching third-party composition of aspect mechanisms. We used the model to formalize the aspect extension composition problem and scope its domain to reactive aspect extensions.

We believe that the CRX model is a good way to explain and teach AOP [Lorenz and Kojarski, 2006a]. Data flow analysis is effective for conceptual level modeling, and a DFD provides an easy to understand graphical representation of a system [Yourdon, 1989].
Chapter 5

The Composition Specification Problem

A key challenge in specifying a composition of multiple aspect extensions is identifying and resolving adverse feature interactions. These interactions occur due to the incompatible and inconsistent treatment of aspects, join points, and advice across different aspect extensions. In this chapter, we analyze the root cause of this feature interaction problem. We classify common features of aspect extensions, describe how these features may interact when using different extensions in tandem, and concretely illustrate how these interactions may be resolved. The chapter contributes a methodology for building a composition specification. The methodology allows AOP users and tool developers to reason about the occurrence of such adverse and unexpected feature interactions, and to apply several patterns for resolving these problems.

The analysis of the feature interaction problem focuses on a dominant family of aspect extensions known as join point and advice (J&A). Section 5.1 argues why J&A family is the most practical and reasonable space to investigate. Section 5.2 then abstracts over J&A extensions by identifying their common features. The section captures the similarities as abstract features. The abstract features define a model of a J&A extension called an abstract extension.

Interaction patterns are interactions between abstract features in a composition of abstract extensions. Section 5.3 shows that the interaction patterns provide a convenient abstraction for reasoning about, identifying feature interactions, and specifying arbitrary compositions of concrete J&A extensions. An in-depth analysis of the interaction patterns is given in Section 5.4.
and Section 5.5.

A pattern-based methodology for specifying an aspect extension composition is presented in Section 5.6. The section illustrates the methodology by specifying a composition of COOL and AspectJ, a composition of AspectJ and AspectWerkz, and a composition of COOL, AspectJ, and AspectWerkz. Section 5.7 applies the methodology to identify and formulate feature interaction problems in existing compositions and composition tools.

5.1 Scope

The CRX model provides a formal basis for reasoning about the aspect extension composition problem. A refinement of the general CRX weaving process model (WPM) yields an abstract aspect mechanism, e.g., a reactive or a non-reactive WPM. The general composition problem then can be reduced to the problem of composing abstract aspect mechanisms. Furthermore, the model scopes the domain of the aspect extension composition problem to dynamic reactive aspect extensions: the non-reactive extension are trivially composable; and the problem of composing static reactive extensions (e.g., Open Classes) is easy to solve (Section 4.6.2).

The scope of this dissertation is the problem of composing join point and advice (J&A) extensions. In a J&A extension a join point represents a computation, advice is matched against computation’s context, and weaving transforms the computation itself.¹ There are three types of computation transformers that the J&A extensions support, namely before, around, and after. The before (after) transformers run advice computations before (after) the original computation. The around transformers run the advice computation instead of the original computation.

To the best of our knowledge, all existing dynamic reactive aspect extensions are J&A extensions, including AspectJ [Kiczales et al., 2001], AspectWerkz [Bonér, 2004], COOL [Lopes and Kiczales, 1997, Lopes, 1997], KALA [Fabry and D’Hondt, 2006], CaesarJ [Aracic et al., 2006], Spring [Spring], Classpects [Rajan and Sullivan, 2005], JBoss-AOP [JBoss], JAsCo [Van-

¹ Proposals to match advice against the computation itself (e.g., against “future” join points) are known to be not just severely inefficient, but also severely problematic [Klose and Ostermann, 2005].
derperren et al., 2005], and ComposeJ [Wichman, 1999]. The problem of inventing new types of aspect extensions is outside the scope of this research. The family of J&A extensions is thus an adequate and the most practical space to investigate.

We use COOL, AspectJ, and AspectWerkz to analyze the composition problem, and concretely illustrate a composition approach. Various combinations of these extensions present an interesting case study with a representative complexity. A composition of COOL with AspectJ is a representative example of composing domain-specific and general-purpose aspect extensions: the two are sufficiently different, requiring interesting design decisions to make them plug together. A composition of AspectJ and AspectWerkz represents the problem of composing general-purpose aspect extensions. A composition of all three extensions illustrates a scalability of the composition approach.

5.2 Features

From a feature interaction perspective, an aspect extension introduces features that incrementally extend the base language specification. The problem of specifying the behavior of a multi-extension composition is the problem of identifying and resolving the interactions between features of the composed extensions.

This chapter identifies the abstract features that are common to all J&A extensions [Kojarski and Lorenz, 2006]. We refer to the feature model (the set of abstract features) as an abstract extension. A virtual composition of such abstract extensions provides a conceptual framework for reasoning about concrete compositions of concrete extensions. We refer to the interactions between abstract features in a virtual composition as feature interaction patterns.

We distinguish between three related definitions commonly associated with the term join point. A join point computation is a “point” in the execution. A join point type is an extension-specific data structure for describing a computation. A join point instance is an internal description of a particular computation.
5.2.1 Join Point Features

We identify two groups of interaction-related abstract features. The first group includes join point features that characterize the ability to observe a program execution. The features are: granularity, type, visibility, and history.

Join Point Granularity. The join point granularity feature specifies what kinds of computations may be intercepted by the extension. The granularity is the consolidation of base language computations (base granularity) and extension-specific computations (aspect granularity). In AspectJ, for example, the base granularity includes certain computations in Java (method call, method execution, etc.); the aspect granularity includes only advice execution computations. In COOL, the base granularity includes only method invocation computations in Java; and the aspect granularity is empty.

Join Point Type. The join point type feature defines the extension-specific data structure for describing computations. In AspectJ this structure includes the kind of computation (e.g., method call) and its dynamic, static, and lexical contexts (e.g., this and target objects, signature of the target member, and location in the source file). In COOL, the description contains only the signature of the target method.

Join Point Visibility. The join point visibility feature maps join point computations to join point instances. The feature classifies computations as either visible or invisible; and instantiates join points for the visible ones. The join point visibility feature of AspectJ, for example, filters out all the computations within the lexical scope of an if pointcut expression; and constructs join points for all the rest. All method call, get, and set computations within the if pointcut expression are invisible, although subcomputations in their control flow (e.g., executions of methods that are called from the pointcut expression) are visible.
Join Point History and Genealogy. The join point history and genealogy feature stores join point instances, and establishes their relation to provide a genealogy of join points. Intuitively, the feature builds an extension-specific representation of the program execution. For example, AspectJ represents a program execution as a join point stack. The stack defines a dynamic context relation over join points: join points pushed on the top of the stack are said to be in the dynamic context of those below them. An enclosing join point relation associates each join point with its immediate parent on the stack.

5.2.2 Advice Features

The second group includes advice features that shape the capability to modify a program. The features are: advice type, join point advisability, advice ordering, and advice execution.

Advice Type. The advice type feature defines the sorts of advice weaving that an aspect extension supports. For example, AspectJ and AspectWerkz support before, after, and around advice. Identifying the advice types in COOL, on the other hand, is more of a challenge. COOL defines two blocks of operations. The first block is weaved before a method invocation. It comprises operations defined by mutex, selfex, requires, and on_entry expressions. We refer to this as the lock block. The second block is weaved after a method invocation and includes on_exit expressions. We refer to this as the unlock block. The semantics of COOL specify that both the lock and unlock blocks are synchronized. A synchronized block is never executed concurrently with another synchronized block. Therefore, it is reasonable to identify lock and unlock as advice types in COOL.

Join Point Advisability. The join point advisability feature associates advice with a join point. The association logic is normally based on the join point history. For example, in AspectJ an advice is selected if its pointcut matches the join point stack. Besides the core advice
selection logic, the feature may define advising constraints for various types of join points, or even for specific join points. For example, the join point advisability feature of AspectJ restricts the advisability of handler join points to before advice only; and filters around advice at initialization and preinitialization join points.

The join point visibility and join point advisability features are not interchangeable. In general, filtering a join point is not equivalent to filtering all the advice for that join point. In the first case, the join point imposes no effect on the program execution. In the second case, the join point is reflected in the join point history, and can potentially affect the execution at a future join point. For example, if AspectJ were to filter join points within if pointcut expressions by disallowing advice, then these join points could still be accessed via a cflow pointcut designator.

**Advice Ordering.** The advice ordering feature prescribes the semantics for sorting advice that is selected at a join point into a specific execution order. For example, AspectJ orders pieces of advice according to their type, their lexical location in the aspect definition, and a precedence relationship over aspects.

**Advice Execution.** The advice execution feature controls how an extension observes the execution of its own advice. Specifically, the feature determines how advice computations are built. Once built, the computations can be recursively intercepted and transformed by the extension, thus allowing aspects to advise each other. For example, the advice execution feature of AspectJ executes an advice as an advice execution computation. The computation is then intercepted by the join point visibility feature and advised by AspectJ as an advice execution join point. On the other hand, the advice execution feature of COOL is empty, because a COOL coordinator can only advise Java methods.
5.3 Patterns of Interaction

An abstract extension is a convinient and useful abstraction for reasoning about multiple compositions of aspect extensions. All possible compositions of J&A extensions can be analyzed by examining a single composition of abstract extensions. Most importantly, interactions between abstract features of the composed abstract extensions represent feature interactions in all matching compositions of concrete aspect extensions. We refer to the interactions between abstract features as feature interaction patterns.

For illustration, we use a composition of COOL and AspectJ. Figure 5.1 presents a feature interaction matrix of the composition where cross signs and circle symbols represent feature interactions. The matrix categorizes the interactions as influencing/inducing, genuine/incidental, and foreign advising/co-advising.
5.3.1 Influencing versus Inducing

Features of composed aspect extensions interact by influencing each other, and by inducing new emergent composition-level features. Figure 5.1 represents interactions that influence COOL and AspectJ features by black and white symbols, respectively; and gray symbols represent the inducing interactions. For example, the advice execution feature of COOL influences AspectJ features (e.g., join point visibility, join point history and genealogy, and join point advisability) to advise COOL aspects. The interaction between the advice ordering features of COOL and AspectJ induces the emergent advice ordering feature of the composition. The emergent feature orders COOL advice relatively to AspectJ advice.

5.3.2 Genuine versus Incidental

The genuine interactions (represented by cross signs in Figure 5.1) signify unspecified behavior across interacting and emergent features. These interactions are unavoidable: the unspecified behavior is a natural phenomenon of the composition process. For example, neither AspectJ nor COOL define how their granularities relate. A composition of AspectJ and COOL must specify this relation thus resolving the genuine interaction between the join point granularity features.

In contrast, an incidental interaction (represented as a circle in Figure 5.1) has a default empty resolution that is appropriate for most compositions. A particular composition specification may override the default behavior, but does not necessarily needs to. For example, the join point advisability features of AspectJ and COOL are orthogonal, and wouldn’t normally interact: each feature controls selection of advice from within its own aspects. However, it is possible to define a composition with a non-trivial interaction behavior, e.g., for maintaining mutual exclusion property over COOL and AspectJ aspects.
public aspect Logger {
    pointcut scope(): !cflow(within(Logger));

    before(): scope() {
        System.out.println("before " + thisJoinPoint);
    }

    Object around(): scope() {
        System.out.println("around" + thisJoinPoint);
        return proceed();
    }

    after(): scope() {
        System.out.println("after" + thisJoinPoint);
    }
}

Listing 5.1: A logger aspect in AspectJ

5.3.3 Foreign Advising versus Co-advising

In a multi-extension program, an aspect interacts with foreign aspects by advising join points in their execution, and by collaboratively advising the same program join points. We refer to the first kind of interaction as foreign advising, and to the second one as co-advising.

The foreign advising interaction determines how aspects in one extension advise foreign aspects in other extensions. In Figure 5.1 the foreign advising interactions are marked by thick borders, e.g., an interaction between the advice execution feature of COOL and the advice ordering feature of AspectJ. Particularly, in a composition of COOL and AspectJ the foreign advising interaction controls the weaving of AspectJ advice into foreign COOL coordinators, and the weaving of COOL advice into foreign AspectJ aspects. For example, consider running a Logger aspect in AspectJ (Listing 5.1) together with the Stack coordinator in COOL (Listing 2.4). Logger logs all join points in a program execution, including join points within executions of the Stack coordinator. Neither the AspectJ nor the COOL specification define how AspectJ aspects advise COOL coordinators. A resolution of the foreign advising interaction must determine what join points Logger advises within the Stack coordinator, and how.
Co-advising is the application of multiple pieces of advice to the same join point. Generally, co-advising can be resolved in any conceivable way or even arbitrarily. Practically, however, co-advising is usually an advice scheduling problem that requires the varies aspect extensions to coordinate the weaving of advice into the same join point. For example, consider again running the Logger aspect in AspectJ (Listing 5.1) together with the Stack coordinator in COOL (Listing 2.4) and the Stack class in Java (Listing 2.3) in the composition of COOL and AspectJ. The Logger and the Stack coordinator collaboratively advise executions of the Stack methods. However, it is unspecified how the aspect and the coordinator interact. A resolution of the co-advising interaction of the composition must determine in what order the pieces of advice of the aspect and the coordinator execute.

The next two sections give an in-depth analysis of genuine interaction patterns. The problem of identifying and resolving the genuine interactions is a crucial element of the composition problem. Failure to handle these interactions leads to unexpected and often incorrect behavior in multi-extension aspect programs. In contrast, incidental interactions do not introduce ambiguity into the aspect programs.

There are two main groups of genuine interaction patterns, namely inducing co-advising patterns and influencing foreign advising patterns. We present each pattern by listing the interacting abstract features and discussing how the interaction can be resolved.

5.4 Co-advising Patterns

This section analyzes three co-advising patterns, namely (1) emergent join point granularity, (2) emergent advice type, and (3) emergent advice ordering. Each pattern encloses interactions between same-name features of the composed extensions.

Emergent Join Point Granularity and Type. The emergent join point granularity feature normalizes and consolidates the individual granularities to enable a common reference for mutual interaction. Unfortunately, this process is generally ambiguous. The problem occurs be-
cause the join point granularity features of the individual extensions are given in different terms
and at different levels of detail; and they may be normalized in more than one way. This ambi-
guity must be resolved in the composition.

The AspectJ granularity feature includes method execution and method call computa-
tions. The COOL granularity feature is less detailed and only includes method invocation com-
putations. Thus, there are at least three ways to normalize the granularities of COOL and As-
pectJ. A method invocation can be equated with a method call, or a method execution, or it can
be mapped to a computation between the call and the execution (e.g., nested within a method
dispatch computation, but around a method execution computation).

Each of the alternative normalizations specifies a unique composition behavior. Equating
method invocation with method call join point computations forces coordinators in COOL to
synchronize AspectJ advice at method execution join points. It also allows the composition to
choose whether or not the coordinators synchronize AspectJ advice at method call join points
(by ordering the advice of COOL and AspectJ). In contrast, equating method invocation with
method execution join point computations prevents the COOL coordinators from synchronizing
AspectJ advice at method call join points, and allows the composition to choose whether or not
the coordinators synchronize AspectJ advice at method execution join points. Finally, mapping
method invocation to a computation in-between AspectJ’s method call and method execution
prevents COOL from synchronizing AspectJ advice at method call join points, and forces the
coordinators in COOL to synchronize AspectJ advice at method execution join points.

The emergent granularity may also include computation types that the individual gran-
ularities do not explicitly define. Consider lock, unlock, on_entry, and on_exit computa-
tions that are specific to COOL. The first two computation types respectively represent executions
of COOL’s lock and unlock advice; and the on_entry and on_exit computations represent execu-
tions of on_entry and on_exit statement blocks. Although these computations are not in
the granularity of COOL they may be advisable in a composition of COOL and AspectJ (e.g., as
advice-execution join points).
Emergent Advice Type. The advice type feature of the composition normalizes the set of advice types defined by the individual extensions. However, the normalization of advice types is generally ambiguous. Different normalizations yield different composition specifications. This ambiguity constitutes the unspecified behavior found in the composition.

The normalization of COOL and AspectJ advice types is a good example. The first challenge is to identify correctly the COOL advice types (recall Section 5.2.2). Once the advice types are identified, the problem of normalizing advice types in the composition of COOL and AspectJ is the problem of matching the lock and unlock advice types of COOL with the before, after, and around advice types of AspectJ. One option is to equate lock and unlock with before and after, respectively. Another option is to equate a pair of lock and unlock advice blocks with a single around advice. A third option is to consider COOL and AspectJ advice types distinct, and allow COOL advice to dominate over AspectJ advice (i.e., to always run lock advice before AspectJ’s advice and unlock advice after AspectJ’s advice). This ambiguity illustrates the general problem of advice type feature interaction in a composition of multiple aspect extensions.

Emergent Advice Ordering. A single join point computation can generally match advice written in several extensions. The advice ordering feature of the composition (not to be confused with the advice ordering feature of an individual extension) defines the ordering of selected multi-extension advice.

The ordering of multi-extension advice is unknown at the level of the individual extensions. Rather, this feature emerges in the combination of the various advice ordering features and is thus unspecified.

Consider a composition of AspectJ and AspectWerkz. For this composition, the advice ordering feature interaction problem has many alternative solutions that exhibit considerably different behavior. For example, one option is (a) to execute before advice of both extensions first; then to wrap their around advice over each other; and finally execute the remaining after advice. Another option is (b) to run AspectWerkz’s advice only when AspectJ proceeds to the
base program. In this case, AspectWerkz advice are nested within the execution of AspectJ around advice.

Adding an extension to the composition generally increases a number of alternative solutions. For example, consider a composition of AspectJ, AspectWerkz and COOL. Assuming that COOL method invocation join points are equated with method execution join points, and COOL lock and unlock advice types are equated with before and after advice types of AspectJ/AspectWerkz, each ordering of AspectJ and AspectWerkz advice corresponds to multiple combinations of AspectJ, AspectWerkz, and COOL advice. For example, if AspectJ and AspectWerkz advice are ordered as suggested by option (a), the lock and unlock advice may be ordered to (i) synchronize all AspectJ/AspectWerkz advice; (ii) synchronize only around advice; (iii) synchronize only Java methods, and so on.

5.5 Foreign Advising Patterns

For each extension, the foreign advising patterns comprise interactions where (1) an advice execution feature influences the foreign join point visibility, join point history and genealogy, join point advisability, and advice ordering features; (2) a join point granularity feature influences the foreign join point visibility, join point history and genealogy, join point advisability, advice ordering features; and (3) the join point granularity feature influences the foreign advice execution features.

The first two groups of interactions are closely related and can be joined. The join point granularity feature influences the foreign features to handle its extension-specific computation types. The advice execution feature influences the foreign features to handle computations within its aspects. However, the extension-specific computation types are pertinent only to the advice execution computations. The second group of interactions is thus more general than the first one.

This section analyzes most important interactions from the second and the third groups.
Specifically, we analyze four interaction patterns, namely advice execution, join point visibility, join point history and genealogy, and join point advisability.

**Advice Execution Interaction.** An aspect extension introduces extension-specific terms and expressions. In a multi-extension AOP language, these extension-specific terms and expressions can be advised by foreign aspects. However, it is unspecified how these terms and expressions are observed by the foreign extensions.

Consider deploying the Stack (Listing 2.4) coordinator in a multi-extension composition of COOL and AspectJ. An execution of the coordinator locks and unlocks a target Java method, reads and writes to the empty, full, and len instance variables of the coordinator, and reads from the buf field of the coordinated Stack Java object. In COOL these operations are not join point computations because the granularity of COOL is limited to method invocations only. The composition of COOL and AspectJ has a finer join point granularity that might intersect with COOL coordinators. This poses the question: what join point computations in the emergent granularity does the Stack coordinator contain?

The problem can be attributed to the interaction between the advice execution feature of an aspect extension and the emergent granularity feature of the multi-extension composition. The advice execution feature controls how the extension observes execution of its own aspects in the extension’s granularity. In a multi-extension composition, the aspect can generally be advised by foreign extensions at the emergent granularity level. Because the emergent granularity is generally finer than the extension’s granularity, it is undefined what join point computations in the emergent granularity the aspect contains.

To resolve this interaction, a composition designer must refine the advice execution feature to build join point computations in the emergent granularity domain. In general, however, the feature can be refined in multiple alternative ways. The point of variability is the inherent difference between base language and aspect language terms and concepts. In the example, the composition designer should decide whether COOL coordinators behave like Java classes;
whether static initializations of the coordinator classes generate static initialization computations; whether instantiations of the coordinator objects generate preinitialization, constructor execution, and initialization computations; whether COOL condition fields can be treated as boolean Java fields; what join point computations are found in on_entry, on_exit, and requires expressions; and whether or not other COOL constructs (e.g., mutex) produce Java join point computations.

Join Point Visibility Interaction. The join point visibility feature of an aspect extension maps join point computations to extension-specific join point instances. In a multi-extension composition the mapping is undefined over join point computations that are produced by foreign aspects.

Consider now the deployment of the Stack coordinator (Listing 2.4) together with AJAccessLogger (Listing 5.2) in a composition of COOL and AspectJ. Stack accesses the fields of the coordinator and the coordinated object, and AJAccessLogger logs access to instance variables. Even if we assume that the advice execution feature of COOL is defined to generate field access (get and set) join point computations in executions of requires, on_entry, and on_exit expressions, it is still unspecified what join points AJAccessLogger logs in the Stack coordinator.

The problem can be attributed to the interaction between the join point visibility feature of AspectJ and the advice execution feature of COOL. The interaction is resolved by extending the join point visibility feature to classify and instantiate join points in coordinators. In general,
the feature can be extended in many possible ways. In the example, the join point visibility feature can be defined to construct AspectJ join points for all join point computations in COOL coordinators; it can be defined to ignore \texttt{get} and \texttt{set} computations that access the \texttt{condition} fields of a coordinator; or it can be defined to ignore all COOL computations. In the first case, \texttt{AJAccessLogger} would log any \texttt{set} and \texttt{get} field access within the \texttt{Stack} coordinator. In the second case, the aspect would not log \texttt{get} and \texttt{set} access to the full and empty fields. In the third case, the aspect would not advise the coordinator at all.

Another point of variability is dealing with join point computations that are specific to foreign extensions. Let’s assume that the emergent granularity includes COOL-specific \texttt{lock}, \texttt{unlock}, \texttt{on.entry}, and \texttt{on.exit} computations. Consider deploying the \texttt{Stack} coordinator together with the \texttt{AJAdviceLogger} (Listing 3.1) aspect. The \texttt{AJAdviceLogger} aspect logs all advice-execution join points in the program. The composition specification should define how, if at all, \texttt{AJAdviceLogger} logs executions of the \texttt{Stack} coordinator.

One option is to instantiate advice-execution join points for the \texttt{lock} and \texttt{unlock} computations. Another option is advise the \texttt{on.entry} and \texttt{on.exit} computations as advice-execution join points. Yet another option is to built no advice-execution join points within COOL coordinators whatsoever.

\textbf{Join Point History Interaction.} In a single extension, the join point history and genealogy feature establishes and maintains a relation over the join points found in the aspect and base programs. In a multi-extension composition the feature is unspecified over join points that the extension observes in foreign aspects. We call these foreign join points.

Consider the \texttt{AJStackExecScope} aspect (Listing 5.3). The \textbf{dynamic context} relation in AspectJ provides the semantics to the \texttt{cflow} pointcut. \texttt{AJStackExecScope}’s \texttt{before} advice logs all join points in the dynamic context (control flow) of the execution of a \texttt{Stack} method (except for the join points in the control flow of the advice).\footnote{Join points in the control flow of the advice are also in the control flow of the method execution, but they are excluded to prevent an infinite loop.}

Join points in the control flow of the advice are also in the control flow of the method execution, but they are excluded to prevent an infinite loop.
Next, consider the `AJStackLogger` aspect (Listing 5.4). In AspectJ, advice is executed in the dynamic context of the join point it advises. `AJStackLogger` advises executions of `Stack` methods. If `AJStackExecScope` and `AJStackLogger` are deployed together then the former will always advise join points inside the latter.

In a composition of AspectJ and COOL, however, the dynamic context relation of AspectJ is unspecified over join points found in coordinators. Let’s assume that aspects advise `get` and `set` join points in coordinators. The `Stack` (Listing 2.4) coordinator runs COOL expressions on every execution of `pop` and `push`. When the coordinator and the `AJStackExecScope` aspect are deployed simultaneously, the join point history and genealogy feature of AspectJ can treat the `Stack` coordinator in one of several ways.

One option is to run the `requires`, `on_entry`, and `on_exit` expressions in the dynamic context of AspectJ’s `method execution` join points. In that case, `AJStackExecScope` logs join points in the coordinator. An alternative is to run the expressions outside the dynamic context of the `method execution` join points. In that case, the `AJStackExecScope` aspect does not advise the coordinator.
Join Point Advisability Interaction. The join point advisability feature selects advice at a join point. In a composition of multiple extensions, however, the feature is undefined over foreign join points.

The construction of foreign join points is resolved in the join point visibility interaction. The join point advisability interaction is therefore an interaction between the join point advisability and join point visibility features of the aspect extension. The interaction is resolved by extending the join point advisability feature to select advice at the foreign join points.

For example, assume that the join point visibility feature of AspectJ constructs get and set join points for every access to any field of a COOL coordinator or a coordinated object (from within the coordinator). The interaction between the join point advisability and join point visibility features of AspectJ then raises the question of how AspectJ advises these join points.

The interaction can be resolved in at least three reasonable ways. First, AspectJ may treat these join points as regular field access without constraining the kind of advice. A second option is to allow AspectJ aspects to advise only a field access that targets a field of a coordinated object (e.g., access to Stack.buf), while hiding any access to an internal field of a coordinator (e.g., empty, full, len). This solution respects the differences between COOL and Java objects, while allowing AspectJ aspects to advise any access to a Java field. A third option is to allow AspectJ aspects to advise an access to a field of a coordinator with before advice only. In this manner, the internal synchronization logic of COOL coordinators cannot be overridden by AspectJ, while AspectJ aspects can still monitor the coordinator.

5.6 Building a Composition Specification

The feature interaction approach allows a language designer to build a complete specification of a multi-extension AOP language in three steps. The first step applies the interaction patterns to mechanically identify interactions between features of the composed extensions. The second step formulates one or more questions for each interaction. Finally, the composit-
tion specification is constructed by answering the questions.

The section illustrates the approach by specifying three multi-extension AOP languages, namely COOLAJ, AJW, and COOLAJW. COOLAJ combines the domain-specific COOL and the general-purpose AspectJ extensions. AJW combines AspectJ and AspectWerkz, two general-purpose aspect extensions. COOLAJW is a composition of COOL, AspectJ, and AspectWerkz. The section also discusses applicability of the approach to other multi-extension compositions.

5.6.1 The COOLAJ Specification

We illustrate the feature interaction approach by building a specification of COOLAJ, a multi-extension AOP language that combines COOL and AspectJ. The approach provides a feature interaction matrix (e.g., the matrix in Figure 5.1) that lists all the possible interactions in the composition of abstract extensions. A concrete composition specification normally addresses a subset of the matrix interactions that must include all non-empty genuine interactions, and may also include some incidental interactions. For example, the COOLAJ specification (Figure 5.2) addresses eleven genuine interactions: all the interactions generated by the patterns except the interactions that influence the advice ordering and the join point history and genealogy features of COOL. The two interactions are excluded because the join point history and genealogy feature is empty, and the advice ordering feature is immutable.

The identified interactions can be easily expressed as questions in terms of abstract extensions, e.g.:

- What is the emergent granularity of the composition?
- What is the order of multi-extension advice that apply to the same join point?
- What emergent granularity computations are specified by a COOL coordinator?
- How does AspectJ define the history relation over coordinator’s join points?

The specification formulates and answers these questions in extension-specific terms.
The following questions and answers form the COOLAJ specification:

1. **Emergent co-advising interactions.** The emergent granularity, emergent advice types, and emergent advice ordering interactions respectively raise the following questions:

   a. **What is the emergent granularity of the composition?** The emergent granularity extends the granularity of AspectJ with lock and unlock COOL computations. Method invocation computations of COOL are equated with method execution computations of AspectJ.

   b. **How COOL advice types correspond to AspectJ advice types?** COOL advice types are lock and unlock. lock is equated with before, and unlock is equated with the after advice type of AspectJ.

   c. **What is the order of multi-extension advice?** The lock (unlock) advice of COOL is executed before (after) the before, around, and after advice of AspectJ.

2. **Advice execution interactions** influencing COOL and AspectJ, respectively:
(a) **What join point computations are specified by a COOL coordinator?** The coordinator advises a target method with lock and unlock advice. An execution of the lock (unlock) advice yields a lock (unlock) computation. Within the advice, COOL exposes get and set computations as defined by requires, on_entry, and on_exit expressions. An access to a field of the coordinated object generates a get computation. However, an implicit access to a field of the coordinator object that stores a reference to the coordinated object does not yield an advisable computation. A coordinator defines no other computations (e.g., initialization, constructor execution, static initialization).

(b) **What join point computations are specified by an AspectJ aspect?** As defined by the AspectJ specification.

(3) **Join point visibility interactions** that respectively influence features of AspectJ and COOL:

(a) **What join points does AspectJ instantiate for coordinator’s computations?** A lock (unlock) computation is mapped to an advice execution join point. A get (set) computation is represented by a field-get (field-set) join point.

(b) **What join points does COOL instantiate for aspect’s computations?** COOL treats the AspectJ aspects as Java classes. It builds method-invocation join points for method execution computations that originate from the aspects.

(4) **Join point advisability interactions** that influence features of AspectJ and COOL, respectively:

(a) **How does AspectJ advise join points that originate from COOL coordinators?** AspectJ aspect advises COOL join points in a regular manner with the following exceptions:
• an access (read or write) to a condition field can only be advised with before or after advice. This way aspects cannot override values of these fields, but are still able to observe their access patterns. This restriction is important for protecting the synchronization logic of a coordinator.

• An execution of a lock or an unlock computation is advisable by aspects as an advice-execution join point. However, aspects are limited to advising these join points with before and after advice only. This restriction ensures that the locking and unlocking operations imposed by coordinators are not overridden by aspects, and always apply in the correct order.

(b) **How does COOL advise join points that originate from AspectJ aspects?** In a regular manner.

(5) **Join point history interactions** influencing AspectJ:

(a) **How does AspectJ define enclosing join point relation over coordinator’s join points?** The advice-execution join points that represent lock and unlock computations are enclosing for the corresponding field-get and field-set join points of the coordinator.

(b) **How does AspectJ define dynamic context join point relation over coordinator’s join points?** COOL advice executes in the control flow of the method-execution join point it advises.

(6) **Advice ordering interactions** influencing AspectJ:

(a) **What is the order of AspectJ advice at coordinator’s join points?** AspectJ orders advice at COOL join points in a regular manner.

### 5.6.2 The AJW Specification

AJW is an AspectJ/5-like composition of AspectJ and AspectWerkz. Its is designed
to treat AspectJ and AspectWerkz aspects uniformly. AJW is specified similarly to COOLAJ: the specification resolves all non-empty genuine feature interactions that are generated by the interaction patterns for AspectJ and AspectWerkz. The specification establishes the uniformity of AspectJ and AspectWerkz by equating their similarities in the emergent features (e.g., equating types of join point computations and advice types), and directing individual features (e.g., join point visibility, join point history and genealogy, join point advisability) to handle foreign aspects as their own. More important, the feature interaction analysis also identifies important differences between seemingly-identical AspectJ and AspectWerkz extensions, and the specification explicitly addresses them:

(1) **Emergent co-advising interactions.**

(a) **What is the emergent granularity of the composition?** The emergent granularity extends the granularity of AspectJ with advice method execution computations of AspectWerkz. The emergent granularity equates respective (same-name) computations of AspectJ and AspectWerkz (e.g., method call).

(b) **How AspectWerkz and AspectJ advice types correspond?** The emergent advice type feature equates corresponding advice types of AspectJ and AspectWerkz.

(c) **What is the order of multi-extension advice?** AspectWerkz and AspectJ order advice differently. AspectWerkz orders advice only according to their types, i.e., before advice always executes first, then around advice, and finally after advice. In AspectJ, the lexical order of advice supersedes advice type ordering: an around advice takes precedence over (“wraps” around) after advice that are defined (lexically) before it, and over before advice that are defined after it. AJW preserves partial order of AspectJ and AspectWerkz advice in a multi-extension advice set. It orders before advice of AspectWerkz to run before the first around advice of AspectJ; around advice of AspectWerkz to run after the last before advice of AspectJ; and after advice of AspectWerkz to run not before all around
advice of AspectJ are executed.

(2) **Advice execution interactions** influencing AspectWerkz and AspectJ, respectively:

(a) **What join point computations are specified by an AspectWerkz aspect?** In AspectWerkz an advice is an annotated method with a dual purpose (and a pointcut can also be defined using a method). The method may be executed implicitly when playing the role of an advice (or a pointcut). It may also be invoked explicitly as a regular Java method. Implicit executions of the advice methods of AspectWerkz yield *advice method execution* computations. Implicit executions of the pointcut methods are not join point computations, including computations that lexically reside within the pointcut methods. Explicit executions of the pointcut and advice methods are treated as regular Java method executions. An AspectWerkz aspect also exhibits other computations that are pertinent to Java classes (e.g., *initialization*, *constructor execution*, *static initialization*).

(b) **What join point computations are specified by an AspectJ aspect?** As defined by the AspectJ specification.

(3) **Join point visibility interactions** that respectively influence features of AspectJ and AspectWerkz:

(a) **What join points does AspectJ instantiate for AspectWerkz aspect’s computations?** An *advice method execution* computation of AspectWerkz is mapped to an *advice-execution* join point of AspectJ. The other computations are mapped to their respective AspectJ join points in a regular manner.

(b) **What join points does AspectWerkz instantiate for AJ aspect’s computations?** AspectWerkz maps the AspectJ aspect’s computations to its respective join points in a regular manner (e.g., it maps *method call* computations to *method-call* join points).
(4) **Join point advisability, join point history, and advice ordering interactions** are resolved by applying to foreign join points the same rules that AspectJ and AspectWerkz use for advising join points within their own aspects.

### 5.6.3 The COOLAJW Specification

In this section we apply the interaction patterns approach to specify a multi-extension AOP language COOLAJW that composes three extensions, namely AspectJ, AspectWerkz and COOL. For this composition, the patterns generate a set of 26 interactions to be analyzed. Concretely, the three emergent patterns generate one feature interaction each, the advice execution pattern yields three interactions, the join point visibility and the join point advisability patterns generate six interactions each (e.g., interactions between the join point visibility feature of COOL/AspectJ/AspectWerkz and the advice execution features of AspectJ and AspectWerkz (COOL and AspectWerkz; AspectJ and COOL)); and the join point history and genealogy and the advice ordering patterns generate four interactions each.

Intuitively, the COOLAJW language is a “merger” of COOLAJ and AJW, i.e., COOLAJW defines interactions between COOL and AspectJ (AspectWerkz) aspects similarly to COOLAJ; and the interactions between AspectJ and AspectWerkz aspects are specified similarly to AJW. The COOLAJW specification thus reuses the COOLAJ and AJW specifications whenever possible:

1. **Emergent co-advising interactions.**

   (a) **What is the emergent granularity of the composition?** The emergent granularity extends the granularity of AspectJ with *lock* and *unlock* computations of COOL and *advice method execution* computations of AspectWerkz. The emergent granularity equates COOL’s *method invocations* with *method executions*. The join point computations of AspectWerkz are equated with the same-name computations of AspectJ.
(b) **How advice types of COOL, AspectJ, and AspectWerkz correspond to each other?** COOLAJW equates same-name advice types of AspectJ and AspectWerkz. The lock and unlock advice types of COOL are respectively equated with before and after.

(c) **What is the order of multi-extension advice?** Ordering of AspectJ and AspectWerkz advice is the same as in AJW. COOL advice dominates over advice of AspectJ and AspectWerkz.

(2) **Advice execution interactions** that influence AspectJ and AspectWerkz are resolved in compliance with the AJW specification. The advice execution behavior of COOL complies with the COOLAJ specification.

(3) **Join point visibility interactions**

   (a) **What join points does COOL (AspectJ) instantiate for AspectJ (COOL) aspect’s computations?** As in COOLAJ.

   (b) **What join points does AspectJ (AspectWerkz) instantiate for AspectWerkz (AspectJ) aspect’s computations?** As in AJW.

   (c) **What join points does AspectWerkz instantiate for coordinator’s computations?** A `get` (`set`) computation is represented by a `field-get` (`field-set`) join point. A `lock` (`unlock`) computations yield no join points.

   (d) **What join points does COOL instantiate for AspectWerkz aspect’s computations?** COOL treats the AspectWerkz aspects as Java classes. It builds `method-invocation` join points for `method execution` computations that originate from the aspects. Implicit executions of the aspect advice methods do not yield COOL join points.

(4) **Join point advisability interactions**
(a) How does COOL (AspectJ) advise join points that originate from AspectJ (COOL) aspects? As in COOLJ.

(b) How does AspectWerkz (AspectJ) advise join points that originate from AspectJ (AspectWerkz) aspects? As in AJW.

(c) How does AspectWerkz advise join points that originate from COOL coordinators? An around advice of AspectWerkz cannot be imposed over an access (read or write) to a **condition** field of a coordinator. The other coordinator’s join points are advised by AspectWerkz in a regular manner.

(d) How does COOL advise join points that originate from AspectWerkz aspects? In a regular manner.

(5) **Join point history interactions** influencing AspectJ:

(a) How does AspectJ define enclosing join point and dynamic context relations over coordinator’s join points? As in COOLJ.

(b) How does AspectJ (AspectWerkz) define enclosing join point and dynamic context relations over AspectWerkz (AspectJ) aspect’s join points? As in AJW.

(c) How does AspectWerkz define enclosing join point and dynamic context relations over coordinator’s join points? AspectWerkz does not provide enclosing join points for COOL coordinator’s join points. COOL advice executes in the control flow of the **method-execution** join point it advises.

(6) **Advice ordering interactions**

(a) What is the order of AspectWerkz advice at coordinator’s join points? AspectWerkz orders advice at coordinator’s join points in a regular manner.

(b) What is the order of AspectJ advice at coordinator’s join points? As in COOLJ.
What is the order of AspectJ (AspectWerkz) advice at AspectWerkz (AspectJ) aspect’s join points? As in AJW.

5.6.4 Scalability and Completeness

A composition of \( n \) extensions generates \( O(n^2) \) genuine interactions to be resolved: three co-advising patterns yield three interactions, the advice execution pattern yields \( n \) interactions, and the other patterns yield at most \( n(n-1) \) interactions each. The approach provides a convenient abstraction for reasoning about feature interactions, and for specifying a large set of multi-extension compositions. To the best of our knowledge, the set of abstract features generalizes well over features of the analyzed aspect extensions. The abstract features, however, do not exhaustively model all possible J&A extensions: this goal is a moving target as long as AOP has no definition. If a certain extension (e.g., a future J&A extension) exhibits features that are not captured by the abstract extension, then the approach can be extended to adopt this new features.

The analysis of the feature interaction patterns considers only reasonable interaction scenarios. We believe that the patterns catch all feature interactions in most of J&A extensions compositions. Of course, some compositions may exhibit feature interactions beyond the identified patterns, e.g., incidental interactions. These cases should be resolved on a case by case basis by analyzing the composed extensions directly.

5.7 Analyzing Existing Compositions

The interaction patterns are useful for identifying and explaining adverse feature interactions in existing compositions of aspect extensions. Applying the analysis to AspectJ/5, Reflex, and XAspects reveals that they fail to adequately resolve feature interactions in multi-extension compositions. All three systems exhibit an unexpected behavior of aspects “misadvising” foreign aspects. AspectJ/5 fails to address a particular instance of the advice execution feature
interaction; and Reflex and XAspects fail to address the feature interaction problem in general.

**Reflex.** In Reflex there is ample support for configuring co-advising at the *aspect level*. A programmer can resolve interactions between aspects in a specific aspect program. Unfortunately, Reflex does not support resolution of *language-level* interactions, i.e., the interactions between aspect extensions that affect the behavior of all multi-extension programs.

Most notably, the lack of support for resolving advice execution interactions renders the Reflex framework inapplicable for composing aspect extensions in general. The framework implements the composition by translating source aspects in foreign extensions to aspects in a common target language. The translation introduces and exposes in the target aspects synthetic join points that do not exist in the source. However, in Reflex, foreign aspects cannot distinguish the synthetic from the genuine join points. Moreover, Reflex provides no composition rules for changing the foreign advising behavior, thus preventing the integrator from being able to correct the faulty resolution of this feature interaction. As a result, multi-extension aspect programs compiled in Reflex may exhibit incorrect behavior [Kojarski and Lorenz, 2005, 2007b].

**XAspects.** Similar to Reflex, the XAspects framework does not provide adequate support for resolving feature interactions in the multi-extension AOP language. We analyzed the support for COOL and AspectJ in XAspects. We found that the major problem of the composition lies in a failure to resolve the interaction between the advice execution feature of COOL and the emergent granularity feature of the composition. Because of that interaction, COOL coordinators expose unexpected join point computations that can be erroneously advised by AspectJ aspects. In some multi-extension programs this interaction causes an unexpected deadlock [Kojarski and Lorenz, 2005]. Most of the other interactions are also not addressed in XAspects.

**AspectJ/5.** AspectJ/5 [Colyer, 2005a] is a merger of AspectJ and AspectWerkz. AspectJ and AspectWerkz are very similar semantically, diverging mainly at the syntactical level. AspectJ differentiates syntactically between a method, an advice, and a pointcut; whereas AspectWerkz
We tested AspectJ/5 using a set of AspectJ and AspectWerkz aspects that address all the interactions that are generated from the interaction patterns. Running these aspects in AspectJ/5 revealed that:

- Explicit executions of either pointcut or advice methods in AspectWerkz aspects do not generate method execution join points;

- Advice execution join points are generated for both implicit and explicit executions of the advice methods;

- No join points are generated within the body of a pointcut method, even if it is executed explicitly.

This unexpected behavior indicates a failure of AspectJ/5 to identify and address the interactions between the advice execution feature of AspectWerkz and the emergent granularity feature of AspectJ/5.

**5.8 Summary**

The chapter analyzes the feature interaction problem in the composition of aspect extensions. This problem is the reason why the task of specifying a multi-extension AOP language is so complex, error prone, and unintuitive. Failing to address feature interactions normally results in unexpected behavior in multi-extension aspect programs.

To tackle the problem of specifying a composition of reactive aspect extensions, we took a feature interaction approach. In this approach, an aspect extension is viewed as a set of features that extend the base language; a composition of aspect extensions is specified by identifying and resolving all interactions between features of the composed extensions.

The approach uses the top-down model to identify commonalities between reactive aspect extensions, and capture them as abstract features. The set of identified abstract features
forms an **abstract extension**. A composition of abstract extensions provides a convenient and useful abstraction for reasoning about arbitrary compositions of concrete extensions. Analysis of this composition identifies a set of feature interaction patterns, i.e., interactions between abstract features of the composed abstract extensions.

Given a set of reactive extensions, a specification of their composition is built in three steps. The first step generates mechanically from the feature interaction patterns a set of questions that identify feature interactions in abstract terms. The second step instantiates those questions to generate concrete extension-specific questions. Finally, a specification is constructed by answering the concrete questions. The approach is useful for specifying the behavior of new compositions of existing aspect extensions. To illustrate that we identified and resolved feature interaction problems in hypothetical compositions of AspectJ, AspectWerkz, and COOL.
Chapter 6

The Composition Implementation Problem

In this chapter we address the composition implementation problem: design a composition framework such that, given $n$ aspect mechanisms and a composition specification, plugging them into the framework implements the composition under the specification.

The main contribution of this chapter is an Aspect co-Weaving System for Composing Multiple Extensions (AWESOME). AWESOME is a practical composition framework for implementing a multi-extension AOP language by plugging together independently developed aspect mechanisms. We build the framework against the following design requirements:

- **Maximum code reuse.** The framework should provide libraries and abstractions that support rapid development of individual aspect mechanisms. Components that are common to all aspect mechanisms should be reused whenever possible. New aspect mechanisms should implement only the necessary extension-specific operations. Avoiding unnecessary code repetitions in constructing different aspect mechanisms also improves their reliability.

- **Auto-configuration.** The framework must support automatic composition of multiple aspect mechanisms into a weaving system that exhibits a reasonable default behavior, thus avoiding where possible the tedious task of resolving all the interactions explicitly.

- **Manual override.** Although the default weaving system behavior is reasonable, a
composition specification might require to resolve some of the feature interactions differently. The framework must allow the language designer to override parts of the default configuration in order to comply with the specification.

- **Minimum performance overhead.** To be practically useful, the framework must construct and compose aspect mechanisms without inflicting performance degradation in the runtime of aspect programs.

To address the first requirement AWESOME provides a component-based and aspect-oriented architecture that facilitates the development and integration of aspect mechanisms. To be scalable, the framework provides a default resolution of feature interactions in the composition. To be general, the framework provides means for customizing the composition behavior. Furthermore, to be practically useful, AWESOME supports compile-time weaving scheme that imposes no framework-associated overhead on the runtime performance of compiled aspect programs.

In the sequel we refer to a compile-time weaving system as a weaver; to a compile-time weaving system for a multi-extension AOP language as a multi-weaver; and call an aspect mechanism component for the framework a weaver plugin, or just plugin. In this terms, AWESOME supports composition of independent weaver plugins into a multi-weaver under a given multi-extension AOP language specification.

We start by formulating the design requirements for an aspect extension composition framework. The requirements are formulated in feature interaction terms. By way of background we then explain how individual weavers work. In Section 6.2 we review the working of the ajc weaver for AspectJ, and in Section 6.3 we describe the working of a weaver for COOL. The objective of this overview is to provide the reader with a familiarity with the internal components of a weaver and to introduce the necessary terminology.

A novel aspect-oriented architecture for the framework is presented in Section 6.4. In Section 6.5 we refactor the implementation of the AspectJ and the COOL weavers to reflect this
new AOP design.

6.1 System Design Requirements

The scope of the thesis study is the problem of composing J&A extensions. The goal of this chapter is to build an aspect mechanism composition framework for implementing a multi-extension AOP language. Given a set of J&A aspect mechanisms and a composition specification, the framework should construct a weaving system for the specified multi-extension AOP language. In this section we formulate four design requirements for the composition implementation. The requirements are: granularity, decoupling, composability, and customizability.

6.1.1 Granularity

J&A extensions differ in their join point granularities. A granularity of an extension combines two parts, namely a base granularity and an aspect granularity. The base granularity includes join point computations that the extension observes in a base program; and the aspect granularity represents concepts and abstractions that are specific to extension’s aspects. For example, the aspect granularity of AspectJ includes only advice execution computations, and the base granularity includes all the other AspectJ join points.

The framework must support AspectJ-compatible J&A extensions. A J&A extension is AspectJ-compatible if its base granularity can be reduced to the base granularity of AspectJ. Most of the current extensions are AspectJ-compatible, including COOL, AspectWerkz [Bonér, 2004], CaesarJ [Aracic et al., 2006], and Spring. Moreover, all the current composition tools are AspectJ-compatible as well. AspectJ/5 and XAspects are AspectJ-compatible by construction: the first one is a composition of AspectJ and AspectWerkz; and the second one composes extensions by translation to AspectJ. The base granularity of Reflex is also within the granularity of AspectJ. The base granularity of AspectJ is thus a good choice for the framework.
The granularity requirement does not impose a fundamental restriction on the framework’s applicability. The problem of extending the framework’s granularity is orthogonal to the problem of composing aspect extensions. A change in the framework’s granularity does not affect principles and mechanisms that enable construction of multi-extension AOP languages. For example, it would be easy to extend the framework’s granularity beyond AspectJ, e.g., by including statement-level join points.

6.1.2 Decoupling.

The composition framework should decouple abstractions that are common to all AspectJ-compatible aspect extensions from abstractions that are extension-specific. This reduces the responsibility of the individual aspect mechanism to implementing only the extension-specific weaving operations. By reusing the framework’s abstractions as much as possible, the development of new aspect mechanisms is drastically simplified.

6.1.3 Composability

The framework should support the composition of multiple aspect mechanisms into a default multi-weaver that resolves interactions automatically in a well-defined and reasonable way. This requirement targets the scalability problem that is inherent to the feature interaction problem. It enables an extensible and scalable solution to the problem of composing aspect mechanisms.

We define the default multi-weaver behavior according to the three principles, namely preserving behavior of individual weavers, default co-advising and default foreign advising behavior:

**Preserving behavior of individual weavers.** A default multi-weaver preserves the behavior of the individual weavers as observed when weaving their respective single-extension programs. For example, a multi-weaver for a composition of COOL and AspectJ would weave pure AspectJ
(COOL) programs in exactly the same manner as a stand-alone AspectJ (COOL) weaver would have.

**Default co-advising.** The default co-advising behavior defines emergent join point granularity, emergent advice types, and multi-extension advise matching and ordering:

- **Emergent join point granularity.** An emergent granularity of a multi-extension composition is a union of the framework’s granularity and aspect granularities of the composed aspect extensions.\(^1\) The default resolution of the granularity interaction maps base granularities of the aspect extensions to the framework’s granularity. For each extension, the mapping is specified individually at a weaver plugin development time. For example, a designer of a COOL plugin may resolve the interaction by mapping *method invocation* join points of COOL to the *method execution* join points of the framework.

- **Emergent advice type.** The emergent advice type feature relies on the advice types being similar in all J&A extensions. An aspect advises a program by transforming computations at certain join points. There are three types of transformations: to add advice before a join point computation, to add the advice after the computation, and to introduce the advice instead of (around) the computation. The framework reflects that by introducing three common advice types, namely *before*, *after*, and *around*.

  The default resolution of the advice type interaction maps advice types of the composed weavers to the framework’s advice types. For each weaver, the mapping is specified individually at the weaver development time. For example, the designer of the COOL plugin may map *lock* and *unlock* advice types of COOL to *before* and *after* advice types of the framework.

- **Emergent advice ordering.** When the multi-weaver selects multi-extension advice at a join point, the default multi-extension ordering behavior is to run multi-extension

\(^1\) An intersection of aspect granularities of different aspect extensions is empty.
before advice first, then multi-extension around advice, and finally the multi-extension after advice. For example, a default multi-weaver for a composition of COOL and AspectJ would order lock and unlock (COOL advice) to execute before and after AspectJ’s around advice, respectively. The multi-weaver also preserves a partial order of same-extension advice within the selected multi-extension advice.

- Emergent join point advisability. The default policy is to unify the individual matching results of the composed extensions. The selected multi-extension advice include all pieces of advice that match the join point under the semantics of their extensions. Each individual aspect weaver selects advice only from its own aspects, and does not interfere with the matching in foreign weavers.

Default foreign advising. The default foreign advising behavior resolves advice execution, join point visibility, join point history and genealogy, join point advisability, and advice ordering interactions:

- Advice execution. Syntactically, an aspect is a mixture of Java code and extension-specific terms. The default advice execution behavior of an aspect weaver is to build framework’s join point computations for all Java statements within its aspects, and only for those statements. For example, a COOL coordinator embeds Java expressions within requires, on_exit, and on_entry constructs. A COOL weaver plugin then builds field get and field set computations for read and write field access within these expressions, respectively.

- Join point visibility, join point history and genealogy, join point advisability, and advice ordering. In a default composition these features handle foreign join points as if they were join points in base program classes. For example, in the default composition of COOL and AspectJ, AspectJ aspects advise COOL coordinators by transforming get and set join point computations within requires, on_exit, and on_entry
constructs. The join point visibility feature of AspectJ then instantiates \textit{field-get} and \textit{field-set} join points for the corresponding computations. The join point history and genealogy feature sets the constructed join points into the control flow of a \textit{method-execution} join point that the coordinator advised; and the join point advisability and advice ordering features respectively select and order AspectJ advice. Under this behavior, the \texttt{Logger} aspect (Listing 5.1) advises all field access join points within \texttt{requires}, \texttt{on_exit}, and \texttt{on_entry} expressions of the \texttt{Stack} coordinator (Listing 2.4).

6.1.4 Customizability

Although the default multi-weaver implements a reasonable behavior, a composition specification may define special foreign advising and co-advising behavior. For example, a foreign advising specification for a composition of \texttt{COOL} and AspectJ might choose to allow AspectJ aspects to advise \texttt{COOL}’s \texttt{(un)lock} computations as \texttt{advice-execution} join points. Hence, the framework must also allow the language designer to configure the multi-weaver to comply with the composition-specific foreign advising and co-advising specifications.

6.2 An Aspect Compiler

An \texttt{aspect compiler} compiles aspect programs into an executable. We start by describing the high-level architecture of an aspect compiler. For concreteness, we describe the \texttt{ajc} compiler for AspectJ.

In general, an aspect compiler has a front-end and a back-end. The \texttt{front-end} translates aspects to (annotated) classes in the base language. For example, the front-end of \texttt{ajc} translates aspects written in AspectJ (.java and .aj files) to annotated classes in Java.\footnote{More precisely, the target classes are expressed in the Java Virtual Machine (JVM) bytecode language.} The translation process in \texttt{ajc} is mostly straightforward: an aspect is translated to a Java class with the same name; an advice declaration is transformed into a method declaration with the same body. The
compiled advice method is also annotated with attributes that store its aspect-specific data (e.g.,
pointcut declarations). The annotations distinguish aspect classes from other Java classes, and
provide pointcut designators for advice methods.

The **back-end** implements the semantics of the aspect extension. The semantics define
the meaning of advice weaving in terms of computations. A **computation** in this context is a
block of program execution, e.g., a method execution. It encapsulates a sequence of operations
that define a behavior and a dynamic context that includes all arguments and other values ac-
cessible by the computation. An advice is a **computation transformer** [Hilsdale and Hugunin,
computation and produces a transformed computation that runs the advice body before, after,
or instead of the original computation.

In chapter 4 the weaving semantics of AspectJ was modeled using a DFD with four pro-
cesses, namely **CONTROL**, **COMPUTE**, **MATCH**, and **MIX** (Figure 4.10). To model an AspectJ
compiler, we further decompose the **MATCH** process into three subprocesses, namely **RELATE**, **SELECT**, and **ORDER** as shown in Figure 6.1. The **RELATE** process relates a join point descrip-
tion \( ctx_{jp} \) to other join points by organizing them into a stack. It then passes the stack to the
**SELECT** process as a program representation object \( repr_{jp} \). The **SELECT** process selects the set
of pieces of advice \( x_{a}^{*} \) by matching their pointcuts against \( repr_{jp} \). Lastly, **ORDER** orders the
selected advice into a \( \pi_{a} \) vector.

While the extension’s semantics define the weaving in terms of dynamic runtime abstrac-
tions, a compile-time weaver implements the semantics statically by transforming the base and
aspect classes. The weaver transforms a computation by transforming its **shadow**, a body of
code that defines a computation’s behavior. For example, the \( ajc \) weaver transforms Java byte-
code. At the bytecode level, an advisable shadow maps to a continuous block of instructions
with a well-defined begin and end.

Intuitively, the aspect compiler realizes the “static part” of each of the semantical pro-
cesses in Figure 6.1, and projects the dynamic operations onto the woven code. For example,
the **COMPUTE** process builds join point computations from program methods, and describes them using computation context objects. The aspect compiler realizes the static part of **COMPUTE** by identifying the shadows of the join point computations in the program methods; and by describing each shadow using its static and lexical context data. The compiler expresses the dynamic **COMPUTE** operations as Java statements and injects them at join point shadows during the weaving process. The woven statements then build join point descriptions at runtime thus realizing the **COMPUTE** semantics.

We use the semantical abstractions to model a compile-time weaver as an abstract **weaving process** that comprises five subprocesses, namely **reify**, **match**, **order**, **relate** and **mix** (Listing 6.1). The **reify** corresponds to the **COMPUTE** and **CONTROL** processes of the semantical model; **relate**, **match**, and **order** realize the static operations of **RELATE**, **SELECT**, and **ORDER**, respectively; and **mix** implements the computation transformation op-

---

**Figure 6.1: DFD 4.1.1.1: RELATE, DFD 4.1.1.2: SELECT, and DFD 4.1.1.3: ORDER**
public void weaveClass(ClassFile cf) {
    Shadow[] shadows = reify(cf);
    for (Shadow shadow : shadows) {
        Advice[] advs = order(shadow, match(shadow));
        advs = relate(shadow, advs);
        mix(shadow, advs);
    }
}

Listing 6.1: A weaver

The reify process takes as input a class file and constructs a weaver-specific representation of the class. For example, the AspectJ weaver represents a class as a set of computation shadows. Its reify process examines the input Java class cf, and identifies all the shadows that can possibly be advised. Each shadow references a list of instructions embedded in one of cf’s methods (the body of the shadow), and provides static and lexical descriptions of these instructions (the static context of the shadow).

For example, the reify function of the AspectJ weaver is shown in Listing 6.2. The function represents every method in the input class cf, as a list of shadows, and accumulates all identified shadows in the result shadow array. The function distinguishes four types of methods, namely synthetic methods, advice methods, constructors, and regular methods. The synthetic methods (introduced into the aspect classes by the AspectJ front-end) are filtered from the shadow representation of the class; an advice method is represented by an advice execution shadow; and a constructor (regular method) yields a constructor execution shadow (method execution shadow). The shadows are constructed by Shadow.make methods.

For each method (except synthetic methods), reify examines its body and constructs shadows for every advisable instruction. The instruction-level shadows are constructed by a getInstrShadow function (Listing 6.3). addAll is an auxiliary method that takes two argument
arrays, and concatenates them by appending the second one to the first one.

The **match** process associates elements of the program representation (shadows) with pieces of advice. The **match** function of the AspectJ weaver is shown in Listing 6.4. In AspectJ, the weaver selects the set of advice by matching the description of the shadow (the static context) against the static part of the advice pointcuts. In Listing 6.4, the **matchPct**
function selects pointcuts that match the shadow; and the \texttt{pctToAdv} function returns advice that are associated with the pointcut.

The \texttt{order} process sorts and orders all pieces of advice that match the same shadow into a correct application order. The \texttt{ajc} weaver orders the pieces of advice according to the rules defined by the AspectJ language semantics.

The \texttt{relate} process implements semantics of join point relations. The relations are defined over join point instances and must be maintained at run-time. For example, the dynamic context relation of AspectJ is an abstraction of a program’s call stack; it must be updated as program executes. \texttt{relate} implements the relations by adding auxiliary advice to the advice list. Executions of the auxiliary advice then maintain the join point relations at run-time. For example, \texttt{ajc} implements the dynamic context relation over join points by weaving \texttt{CFlowPush} and \texttt{CFlowPop} advice around the program shadows.

Listing 6.5 shows the \texttt{relate} function of the AspectJ weaver. \texttt{ajc} weaves the \texttt{CFlowPush} and the \texttt{CFlowPop} advice if the shadow matches a \texttt{cflow} pointcut. The \texttt{hasCFlowPct} function tests whether a \texttt{cflow} pointcut matches the shadow. If it does, \texttt{relate} constructs the \texttt{CFlowPush} and the \texttt{CFlowPop} advice for the shadow (using \texttt{makeCFlowPush} and \texttt{makeCFlowPop} constructors, respectively) and appends them to the advice array. The \texttt{cflow} advice is woven last thus dominating the other AspectJ advice at run-time. This ensures that the AspectJ advice executes in a control flow of a join point it advises.
Advice[] relate(Shadow shadow, Advice[] advs) {
    Pointcut[] pointcuts = matchPct(shadow);
    for (Pointcut pointcut: pointcuts)
        if (hasCFlowPct(shadow, pointcut)) {
            Advice cflowPush = makeCFlowPush(shadow);
            Advice cflowPop = makeCFlowPop(shadow);
            return addAll(advs, new Advice[]{cflowPush,cflowPop});
        }
    return advs;
}

Listing 6.5: History Relations in AspectJ

The **mix** process transforms an actual body of a shadow by introducing code of advice (or calls to advice) that match this shadow. The AspectJ weaver transforms the shadow’s instruction list by sequentially introducing calls to the advice methods before, after, or instead of the original code. The advice pieces are woven in by sequentially transforming the body of the shadow. An advice then injects new code inside the body of the shadow, immediately before, immediately after, or instead of the original code. The transformation considers instructions that were woven earlier as if they were a part of the original shadow. This way the advice pieces “wrap” around each other in the transformed shadow.

In the abstract extension terms, shadow type hierarchy realizes the join point granularity feature. The **reify** process realizes the join point visibility and advice execution features of the extension. The process instantiates join point shadows from class file instructions; and partitions aspect classes into advisable shadows. The **match**, **order**, and **relate** processes realize the join point advisability, advice ordering, and join point history and genealogy features, respectively. The advice type hierarchy realizes the advice type feature.

The five processes provide a high-level description of the **advice weaving** semantics. A concrete weaver may also realize other kinds of transformations. For example, the **ajc** weaver implements intertype declarations and advice weaving in two separate steps (Listing 6.6). First, the weaver extends and transforms the class cf by applying the intertype declarations (the call to **applyIntroductions** in Listing 6.6). Once the declarations are applied, the weaver calls
public void AJWeaver(ClassFile cf) {
    applyIntroductions(cf);
    weaveClass(cf);
}

Listing 6.6: The AspectJ weaver

weaveClass, which implements the advice weaving behavior. The additional transformations are normally static in nature, and do not interfere with the dynamic advice weaving behavior.

6.3 A Compiler for COOL

The architecture of a compiler for COOL is similar to that of ajc. The front-end translates coordinators in COOL to classes in Java. The back-end instruments the program with calls to methods of coordinator classes. This section explains by example the internal workings of the COOL compiler.

6.3.1 Front-end Translation

Consider the Stack class (Listing 2.3) and the Stack coordinator (Listing 2.4). The coordinator imposes the synchronization logic over push and pop in an aspect-oriented manner. The Stack methods are not synchronized. But in the presence of the Stack coordinator, the stack object operates correctly even when multiple client threads execute methods simultaneously.

The COOL front-end translates a coordinator in COOL to a coordinator class in Java. The name of the class is obtained by appending “Coord” to the name of the aspect, e.g., StackCoord.

StackCoord (Listing 6.7) implements the synchronization logic via special synchronized methods and instance variables. The class provides a pair of lock_ and unlock_ methods and an instance variable for every method that is advised by the coordinator. Specifically, the synchronization for the Stack.push method is realized by lock_push and
unlock_push. Similarly, the synchronization logic for Stack.pop is realized by lock_pop and unlock_pop. At any point of the execution, the pushState (popState) instance variable stores all threads that are currently executing the push (pop) method on the coordi-

Listing 6.7: A translated COOL coordinator class
```java
public class Stack {
    public Stack(int capacity) {
        buf = new Object[capacity];
        _coord = new StackCoord();
    }
    public void push(Object obj) {
        _coord.lock_push(this);
        try{
            buf[ind] = obj;
            ind++;
        } finally {_coord.unlock_push(this);}
    }
    public Object pop() {/*omitted*/}
    public Object[] _buf() {return buf;}
    private Object[] buf;
    private int ind = 0;
    private StackCoord _coord;
}
```

Listing 6.8: A synchronized bounded stack

A synchronized bounded stack maintains an array of objects. The coordinator class also includes all fields of its coordinator.

The lock_ methods implement the semantics for mutex, selfex, and requires, and run on_entry blocks. A while loop suspends the execution of the current thread if a guard condition is violated. Specifically, the while loop in the lock_push method suspends the execution of the current thread (by invoking wait on the coordinator object) so long as either one of the requires, selfex, or mutex conditions is in violation. The requires condition is checked by the !(!full) expression (line 4). selfex fails if push is run by another thread (line 5); and mutex fails if pop is run in parallel (line 6). If all the guard conditions are satisfied, the thread executes all the existing on_entry statements, and locks the coordinated push method by adding its Thread object to the pushState list (line 9).

The unlock_ methods unlock the coordinated method and run the on_exit statements. Specifically, unlock_push unlocks the coordinated push method by removing the current Thread object from the pushState list (line 13). It then executes the on_exit statement (lines 14 – 17) and notifies the other threads waiting on the lock that the coordinated
method is free (line 18). Note that accesses to the coordinated object fields (instance variables) are translated into method calls on the coordinated object. Specifically, access to the buf field of the coordinated object is translated into a _buf() method call (line 16). This is the way in which the coordinator class gains access to protected or private fields of the coordinated class. The method is generated in the coordinated class by the COOL weaver, and simply returns the value of the corresponding field.

6.3.2 Back-end Weaving

The COOL weaver applies four kinds of transformations to a coordinated class, namely method transformation, constructor transformation, field introduction, and method introduction. When applied to the non-synchronized Stack (Listing 2.3), these transformations yield a synchronized stack (Listing 6.8). The weaver associates a coordinator with a coordinatee by introducing a _coord field in the coordinated class (line 17), and adding an initialization statement in the constructor (line 4). The weaver also introduces public getter methods (_buf()) for protected and private fields of the coordinated class that need to be accessed by the coordinator.

The weaver transforms the coordinated methods by introducing calls to the coordinator’s lock_ and unlock_ methods before and after the original body. To ensure invocation of the unlock_ method, the weaver also introduces a try-finally block around the original body.

In sum, the COOL weaver realizes the COOLWeaver algorithm (Listing 6.9). Given a class file cf to be transformed, the weaver searches for its coordinator (findAspect, line 2).\(^3\) If found, the weaver introduces a coordinator field (addCoordField, line 4), transforms the constructors to initialize that field (transformConstructor, line 5), and generates getter methods for protected and private cf fields that are read by the coordinator (addGetterMethods, line 6).

Then, the weaver synchronizes the methods of cf by imposing locking and unlocking advice before and after their bodies, respectively. Advice weaving in COOL follows the same

\(^3\) In COOL, each class can be associated with at most one coordinator.
```java
public void COOLWeaver(ClassFile cf) {
    ClassFile coordAspect = findAspect(cf);
    if (coordAspect!=null) {
        addCoordField(cf, coordAspect);
        transformConstructor(cf, coordAspect);
        addGetterMethods(cf, coordAspect);
        weaveClass(cf);
    }
}

Method[] reify(ClassFile cf) {cf.getMethods();}

Method[] match(Method shadow) {
    ClassFile coordAspect = findAspect(  
        shadow.getClass());
    Method lock = findLock(  
        shadow.getSignature(), coordAspect);
    Method unlock = findUnlock(  
        shadow.getSignature(), coordAspect);
    if (lock==null) return new Method[0];
    return new Method[]{lock, unlock};
}

Method[] order(Method shadow, Method[] advs) {
    return advs;
}

Method[] relate(Method shadow, Method[] advs) {
    return advs;
}

void mix(Method shadow, Method[] advs) {
    if (advs.length>0) {
        addCallBefore(shadow, advs[0].getSignature());
        addCallAfter(shadow, advs[1].getSignature());
    }
}
```

Listing 6.9: The COOL weaver

five-process model as in AspectJ (call to `weaveClass`, line 7). In terms of this five-process model, a shadow in COOL is a method of `cf`, and the advice are the `lock_` and `unlock_` methods of the coordinator class.
The **reify** process of the COOL weaver represents an input class file as a set of methods (the **reify** method, line 11). The **match** process uses the signature of a yet-to-be-coordinated method to select a pair of **lock_** and **unlock_** advice methods (the **match** method, lines 13 – 22). For every coordinated method, the weaver finds the corresponding **lock_** and **unlock_** methods in the coordinator class (findLock, line 16; findUnlock, line 18). The **order** and the **relate** processes of the weaver are empty (the **order** method, lines 24 – 26 and the **relate** method, lines 29 – 31). Lastly, the **mix** process (**mix**, lines 33 – 38) introduces a call to the **lock_** method before the method body (addCallBefore, line 35), and a call to **unlock_** after the method body (addCallAfter, line 36).

### 6.4 An Aspect-Oriented Architecture

This section introduces a practical component-based and aspect-oriented architecture that facilitates the development of aspect weavers, and supports the integration of independently developed aspect weavers into a multi-weaver. The architecture is introduced in three steps. The first step explains the design decisions for decoupling **extension-specific** from **common** components. We refer to the extension-specific components as the **aspect mechanism**, and to the common components as the **platform**. The platform is implemented once; it provides facilities that, if reused, significantly ease the development of new weavers. Second, we present the design principles and decisions that allow the multiple aspect mechanisms and the platform to be automatically composed into a default multi-weaver. Third, we present a solution to the multi-weaver customizability problem. The architecture provides support for configuring the default multi-weaver to comply with a specialized composition specification.

#### 6.4.1 Decoupling

We use the five-process weaver model [Kojarski and Lorenz, 2006] to identify the **extension-specific** and the **common** weaver components.
The **reify** process of a weaver constructs shadows for the base language classes and the extension aspects. As a part of its functionality, the process realizes a base representation function, i.e., a function that builds shadows for base program classes. Because the weaver is AspectJ-compatible, it can be realized using the base shadow domain and the base representation function of AspectJ.\(^4\) Thus, the base shadow domain and the base representation function of AspectJ can be shared by all AspectJ-compatible weavers.

If a weaver’s extension does not allow an aspect to advise aspects (e.g., a coordinator in COOL cannot advise other coordinators), then the common base representation function realizes the weaver’s **reify** process in full. However, in the more general case (e.g., in AspectJ, an aspect can advise itself, as well as other aspects), a weaver needs to realize an extension-specific representation function, i.e., a function that builds shadows for aspects. The **reify** process of the weaver is thus a composition of the **common** base representation function and the **specific** representation function for aspects.

The **match**, **order** and **relate** processes of a weaver are **extension-specific**. An individual weaver matches advice in its own aspects, orders only extension-specific advice, and relates only extension’s join points. The **mix** process weaves the ordered pieces of advice by transforming the shadow. Since an advice defines a shadow transformer function, **mix** can be modeled as a **common** extension-independent process that iteratively applies the advice transformers to the shadow.

Figure 6.2 depicts the design of an AspectJ weaver as a composition of common and AspectJ-specific components. There are two main architectural parts: (a) a **platform** that provides **common** facilities; (b) a **mechanism** that implements **extension-specific** behavior. The platform’s behavior is realized by the **Platform** class. The mechanism is realized by an aspect that implements the **Mechanism** interface. The dashed arc in the figure denotes advising.

---
\(^4\) The base shadow domain of AspectJ includes all shadows except for advice-execution.
Platform The platform provides the base shadow domain. Its methods `reify`, `mix`, and `weaveClass` implement the common weaver’s operations: `reify` uses base shadows to represent base classes; `mix` weaves advice at each shadow; and `weaveClass` implements the high-level weaving algorithm (Listing 6.1).

Mechanism Each mechanism is realized as an aspect that implements the Mechanism interface by realizing the extension-specific match, order, and relate processes via the implementation of the `match`, `order`, and `relate` methods, respectively. If a mechanism’s extension allows an aspect to advise other aspects, then the mechanism realizes the extension-specific representation function as an `around` advice that refines executions of the platform’s `reify` method. The methods and the advice are implemented with the following conception: the mechanism uses base shadows as a representation domain for base classes; the advice uses base and extension-specific shadows as a representation domain for aspects;\(^5\) and the advice defers to the platform the representation of base classes.

\(^5\) Extension-specific shadows represent constructs, declarations, and expressions that are specific to the extension, e.g., the *advice-execution* shadow in AspectJ.
6.4.2 Composability

The architecture supports the compositions of multiple aspect mechanisms into a multi-weaver. Figure 6.3 illustrates the extended architecture by showing a default multi-weaver for a composition of COOL and AspectJ. In the extended architecture, the multi-weaver is realized by the platform. The platform mediates between the composed mechanisms, and manages their collaborative application. Furthermore, the Mechanism interface is replaced with the abstract Mechanism aspect, which defines the abstract methods match, order, and relate. In addition, the aspect advises the Platform’s constructor to register the mechanisms with the platform.  

At an abstract level, the multi-weaver implements the same high-level weaving process as a stand-alone weaver (Listing 6.1). The five subprocesses of the multi-weaver are built by integrating and unifying the corresponding processes of the individual weavers. The reify process of the multi-weaver represents base classes, and aspects that are written in different extensions. The match and order processes of the multi-weaver select and order multi-extension advice, respectively. The relate multi-weaver process relates join point instances of all the composed extensions. In the extended architecture, Platform provides the methods match, order, and relate to realize these processes. The mix process weaves the ordered

---

6 This behavior could have been realized in an object-oriented manner, but AOP enables a more elegant design.
multi-extension advice by transforming the shadow.

We enable composability of aspect mechanisms by introducing additional design principles:

**Mandatory Aspect Representation**  To enable a default foreign advising behavior, an aspect mechanism **must** realize an extension-specific aspect representation function, even if the function is **not** normally required for its own stand-alone operation. This policy ensures that a multi-weaver builds shadows for all aspects in a multi-extension program, thus letting an aspect observe and advise Java shadows in any foreign aspect. For example, a stand-alone weaver for COOL does not advise coordinators, and thus does not need to represent them. A multi-weaver for a composition of COOL and AspectJ, in contrast, uses the COOL representation function for exposing the coordinators to AspectJ aspects. The aspects can then advise Java shadows within the coordinators.

Under this policy, the multi-weaver exhibits a default resolution of the advice execution interactions. It also resolves the other foreign advising interactions as desired: the **match**, **order**, and **relate** processes of the individual weavers advise foreign shadows as regular base shadows.

Intuitively, the aspect representation function provides the most fine-grained representation of an aspect that includes all base shadows for its Java fragments, and dedicated extension-specific shadows for all the extension-specific computations. For example, a function for representing coordinators must build shadows for all Java fragments within a coordinator’s code (e.g., Java expressions **within** a **requires** statement), and **on_entry**, **on_exit**, **requires**, **lock**, and **unlock** shadows for all the respective computations.

Because the individual weavers realize all extension-specific shadow types, the default multi-weaver exhibits a desired resolution of the emergent granularity interaction. Specifically, the base granularities of the composed weavers are normalized appropriately; and the emergent granularity is a union of the framework’s granularity and the aspect granularities of the
composed weavers.

**Parallel Matching of Multi-extension Advice** To enable a default multi-extension advice matching behavior, the `match` method of the platform should run the `match` methods of the composed mechanisms in parallel. The multi-extension advice selected at a shadow is then a list of extension-specific advice sets that are produced by the composed aspect mechanisms. The emergent join point advisability feature of the default multi-weaver thus exhibits a desired behavior.

**Uniform Advice Types** To enable a default multi-extension advice ordering behavior, an aspect mechanism should partition advice into three ordered sets, namely before, around, and after. In terms of the architecture, the `order` method of a mechanism returns a list of three advice arrays, the first contains *before* advice, the second contains *around* advice, and the third contains *after* advice. The platform’s `order` method then runs the `order` methods of the composed mechanisms in parallel, and linearizes their results into a single advice vector in accordance with the default multi-extension ordering policy. The emergent advice ordering feature of the default multi-weaver thus exhibits a desired behavior.

A composition of the aspect mechanisms with the platform produces a deafult multi-weaver with a desired behavior. The aspect representation principle enables a default foreign advising behavior, and the other principles enable a default co-advising behavior. The multi-weaver uses as its common shadow domain (i.e., emergent granularity) the union of the common base shadow domain (i.e., framework’s granularity) and all the extension-specific shadow domains.\(^7\) It represents the multi-extension aspects and the base classes as appropriate for the composed mechanisms. It has a well-defined multi-extension advice matching and weaving behavior; and it uses the `order` method of the platform for ordering advice.

---

\(^7\) For simplicity, we assume that the intersection of extension-specific shadow domains is empty [Kojarski and Lorenz, 2005].
6.4.3 Customizability

Of course, the default behavior of the multi-weaver may differ from the actually desired one. The specification may dictate foreign advising and co-advising rules that involve several extensions. For example, the specification may require an aspect in AspectJ to advise executions of lock and unlock in COOL as advice-execution join points. Generally, the foreign advising rules alter the semantics of the individual extension for advising foreign aspects. The co-advising rules specify a collaborative behavior for multi-extension aspects that advise the same join point. These rules are composition-specific and thus cannot be defined on the level of an individual extension.

To this end, the architecture provides a Config aspect that customizes the behavior of the default multi-weaver (Figure 6.4). The configuration aspect implements the composition-specific foreign advising behavior by extending and overriding the match, order, and relate methods of the aspect mechanisms. The aspect specializes the co-advising behavior by advising the match, order, and relate methods of the platform.

The architecture thus supports the construction of a multi-weaver with a customized behavior. The multi-weaver reify method recognizes and represents properly aspects of all the composed extensions using a common shadow domain. The adapted match and order methods of the individual mechanisms select and order extension-specific advice according to the
foreign advising specification; and the adapted \textit{relate} methods realize the specified behavior over foreign join point shadows. The customized multi-weaver \textit{match} and \textit{order} methods select and order the multi-extension advice in accordance with the co-advising specification:

\textbf{Emergent join point granularity and advice type.} In the architecture, the emergent granularity of a composition is defined by the \textit{common shadow domain} that combines a base shadow domain of the platform (base granularity) with extension-specific shadow domains (aspect granularities) of the composed aspect mechanisms. The emergent advice type interaction is resolved by partitioning extension-specific advice types into three groups, namely \textit{before}, \textit{after}, and \textit{around}. The partitioning normalizes the advice types by equating all types within the same group. The join point granularity and the advice type interactions are controlled by an aspect mechanism designer. The designer implements the mechanism using the base shadow domain, the extension-specific shadows, and the three advice groups.

\textbf{Emergent advice ordering and join point advisability.} The emergent join point advisability feature selects multi-extension advice at a join point, and the emergent advice ordering feature sorts the advice in a desired execution order. The \textit{match} and the \textit{order} methods of the platform respectively realize default multi-extension advice selection and ordering behavior. The architecture allows a multi-extension AOP language designer to alter the default behavior. A specialized advisability and ordering behavior is realized by the \textit{Config} aspect that overrides the \textit{match} and the \textit{order} methods with \textit{around} advice. For example, \textit{Config} realizes the multi-extension advice ordering specification of COOLAJ by overriding the \textit{order} method with a desired behavior.

\textbf{Advice execution.} Advice execution is a foreign advising interaction between the advice execution feature of an aspect extension and the emergent granularity feature of a composition. The architecture delegates a resolution of the interaction to the aspect mechanism designer. The designer builds an aspect mechanism in accordance with the mandatory aspect representation
principle. The principle requires that an aspect mechanism provides a function for representing its aspects using the base and extension-specific shadow domains. The function is realized as advice around executions of the `reify` method of the platform. When plugged together, multiple mechanisms collectively advise the `reify` method. The advised method then adequately represents all aspects in a program.

**Join point visibility.** The join point visibility interaction maps join point computations within foreign aspects to extension’s join points, e.g., the COOLJ specification maps the `lock` and `unlock` computations of COOL to *advice-execution* join points of AspectJ. In the architecture, the interaction is resolved by the multi-extension AOP language designer. The `Config` aspect allows the language designer to specialize the join point visibility feature of an aspect mechanism by advising calls to its `match`, `order`, and `relate` methods. `Config` filters or masks foreign join point shadows before the mechanism advises them. For example, `Config` implements the COOLJ behavior by advising calls to the AspectJ mechanism methods, and masking `lock` and `unlock` shadows as *advice-execution* shadows.

**Join point advisability.** The advisability feature of an aspect mechanism controls advice selection behavior. The advisability interaction influences the feature by specifying advice selection strategy at foreign join points, e.g., the COOLJ specification prohibits AspectJ from advising COOL’s `lock` and `unlock` computations with *around* advice. In the architecture, the interaction is controlled by the multi-extension AOP language designer. The advisability feature of the mechanism is realizes by its `match` method. To resolve the interaction, the language designer provides the `Config` aspect that overrides the method with *around* advice. For example, `Config` implements the COOLJ behavior by advising executions of the AspectJ’s `match` method at `lock` and `unlock` shadows, and filtering *around* advice from the originally selected advice list.

**Join point history.** The join point history and genealogy feature of an aspect extension estab-
lishes and maintains relations over its join points, e.g., the dynamic context relation in AspectJ.

In a multi-extension composition the feature is undefined over foreign aspects. The composition specification resolves the interaction by extending the relations over the foreign extensions. For example, the COOLAJ specification defines that COOL advice is executed in the control flow of a method-execution join point it advises.

An aspect weaver implements the feature by weaving auxiliary advice around program shadows, e.g., CFlowPush and CFlowPop advice of the AspectJ weaver. In the architecture, the feature of an aspect mechanism is realizes by the relate method. To resolve the interaction, the multi-extension AOP language designer provides the Config aspect that overrides the method with around advice. For example, Config realizes the interaction between COOL and AspectJ by advising executions of the AspectJ’s relate method, and adding CFlowPush and CFlowPop advice appropriately to the multi-extension advice list.

In sum, the architecture (Figure 6.4) establishes the fundamental principles for designing composable aspect mechanisms. In Section 6.5 we apply these principles to build a concrete co-weaving system for composing multiple extensions (AWESOME). Using AWESOME we implement aspect mechanisms for COOL, AspectJ, and AspectWerkz, and combine these mechanisms to produce weavers for the COOLAJ, AJW, and COOLAJW AOP languages.

6.5 Implementation by Refactoring AspectJ

As a proof of concept, we realize the weaving system and the mechanisms by refactoring the ajc compiler for AspectJ/5, and reimplementing the original weaving algorithm for COOL. In the ajc code, shared, AspectJ-specific, and AspectWerkz-specific operations are intertwined. Through refactoring we untangled and separated these three kinds of operations, moving the ones in common to the Platform and modularizing the AspectJ and AspectWerkz ones in the AspectJ and AspectWerkz mechanisms, respectively. The original weaving algorithm for

---

8 We also moved the shadow transformation functionality in the Shadow.transform method to the Advice class; and resolved the AspectJ/5 feature interaction problem by enhancing the AspectWerkz mechanism.
COOL uses method-level representation of Java classes and does represent coordinators. We implemented the COOL weaver using base AspectJ shadows as a base shadow domain. Our implementation uses method-execution shadows for advice matching and weaving; and provides a shadow representation for the coordinator classes.

The main ajc class runs AspectJ’s front-end and eventually weaves bytecode classes by invoking the weave method on the org.aspectj.weaver.bcel.BcelClassWeaver class. We modified this method to call instead the Platform.weaveClass method for weaving. This permitted to “plug” a specific multi-weaver into ajc by putting a corresponding implementation of the Platform class on the class path, and running the AspectJ compiler as usual.

### 6.5.1 Implementing a Platform

The platform is realized by the Platform class. A list of plugged aspect mechanisms is stored in the mechanisms instance variable. weaveClass is a TEMPLATE METHOD [Gamma et al., 1995] that implements the weaving process (Listing 6.1) using reify, match, order, relate and mix.

Listing 6.10 shows implementation of the reify process. The top-level reify method represents a Java class as a set of shadows. We implemented it by factoring out all operations that represent aspects from the representation function reify of the original AspectJ weaver. For improving code reusability we also extended the Platform’s interface to include method-level, block-level, and instruction-level reify functions.

The top-level function builds a shadow representation of a class by invoking the method-level reify function for all the class’ methods. The method-level reify function builds method’s enclosing shadow and invokes block-level reify to identify shadows within the method body. Block-level reify invokes its instruction-level counterpart for all the block’s instructions; and the instruction-level reify function terminates the reify invocation chain by returning a shadow.
The \texttt{match} and \texttt{order} methods are shown in Listing 6.11. The \texttt{match} method selects advice by calling its \texttt{match} counterparts in the individual mechanisms. The \texttt{order} method calls the \texttt{order} methods of the individual mechanisms, and then linearizes the multi-extension advice. The \texttt{order} method schedules \texttt{around} advice to be woven first, so that \texttt{before} and \texttt{after} advice “wrap” any \texttt{around} advice at the same shadow. Note that weaving order is not the same as execution order.

Listing 6.12 shows \texttt{relate} and \texttt{mix} methods. The \texttt{relate} method shown in realizes the join point relations by calling its \texttt{relate} counterparts in the individual mechanisms.
public Advice[][] match(Shadow shadow) {
    Advice[][] result = new Advice[mechanisms.length][0];
    for (int i=0; i<mechanisms.length; i++)
        result[i] = mechanisms[i].match(shadow);
    return result;
}

public Advice[] order(Shadow shadow, Advice[][] multiAdvs) {
    Advice[] bfAdv = new Advice[0];
    Advice[] ardAdv = new Advice[0];
    Advice[] afAdv = new Advice[0];
    for (int i=0; i<mechanisms.length; i++) {
        Advice[][] mechAdvs =
            mechanisms[i].order(shadow, multiAdvs[i]);
        bfAdv = addAll(bfAdv, mechAdvs[0]);
        ardAdv = addAll(ardAdv, mechAdvs[1]);
        afAdv = addAll(afAdv, mechAdvs[2]);
    }
    return addAll(ardAdv, addAll(bfAdv, afAdv));
}

Listing 6.11: match and order methods

public Advice[] relate(Shadow shadow, Advice[] advs) {
    for (int i=0; i<mechanisms.length; i++)
        advs = mechanisms[i].relate(shadow, advs);
    return advs;
}

void mix(Shadow shadow, Advice[] advs) {
    for (Advice a: advs) a.transform(shadow);
}

Listing 6.12: relate and mix functions

The mix method sequentially applies advice transformers to the shadow. A transformer (the
transform method) integrates the advice instructions into the shadow.

6.5.2 An Abstract Aspect Mechanism

We implemented the mechanisms for AspectJ, AspectWerkz, and COOL as aspects that
The *after* advice ensures that aspect mechanism instances are created and plugged into the platform as soon as the platform is instantiated. The concrete mechanisms (*AJMechanism*, *AWMechanism*, and *COOLMechanism*) provide an implementation for *match*, *order*, *relate* and override the *Platform.reify* methods by advising them with *around* advice.

### 6.5.3 Implementing an AspectJ Mechanism

Implementation of the AspectJ mechanism is presented in Listing 6.14. The *AJMechanism* aspect advises the method-level and instruction-level *reify* functions of the platform. The advice to the method-level function builds AspectJ-specific *advice-execution* shadows for advice methods of aspect classes, and delegates other shadow identification operations to the platform.

In the listing, an *isAJAspect* (*isAJAdvice*) method determines whether or not the argument class (method) represents an AspectJ’s aspect (advice); *makeAExecShadow* builds an *advice-execution* shadow (an *AJAExecShadow* object); and calls to *isAJSynthetic* allow the mechanism to identify and filter out synthetic methods generated by the AspectJ front-end.

The advice to the instruction-level *reify* function filters out instructions that implement *proceed* calls. It calls the *isAJProceedCall* method that determines whether an argument instruction represents a *proceed* call in the original aspect code.

The mechanism also extends the multi-weaver with an intertype declaration mechanism by imposing *before* advice on the *weaveClass* method. The *match*, *order*, and *relate*
public aspect AJMechanism {

  Shadow[] around(Method method, Platform platform):
    args(method) && this(platform) && execution(Shadow[] Platform.reify(Method))
    {
      ClassFile cf = method.getClassFile();
      if (isAJSynthetic(method)) return new Shadow[0];
      if (lisAJAspect(cf) || isAJAdvice(method)) return proceed(method);
      AJAExecShadow encl = makeAExecShadow(method);
      Shadow[] bodyShadows = platform.reify(encl, method.getBody());
      return addAll(new Shadow[]{encl}, bodyShadows);
    }

  Shadow[] around(Instruction i):
    args(i) &&
    execution(Shadow Platform.reify(Shadow, Instruction))
    {
      ClassFile cf = i.getMethod().getClassFile();
      if (isAJAspect(cf) && isAJProceedCall(i)) return null;
      return proceed(i);
    }

  before(ClassFile cf):
    args(cf) &&
    execution(void Platform.weaveClass(ClassFile))
    {
      applyIntroductions(cf);
    }

  /* Other implementation details are omitted */
}

Listing 6.14: The AspectJ mechanism

methods are copied from the original code. We omit them, as well as isAJAspect, isAJAdvice,
makeAExecShadow, and isAJProceedCall.

6.5.4 Implementing an AspectWerkz Mechanism

We implement the mechanism of AspectWerkz as an AWMechanism aspect (Listing 6.15).
Since ajc translates AspectJ and AspectWerkz aspects to the same intermediate representa-
tion (i.e., annotated Java classes), implementations of the AspectWerkz and AspectJ mecha-
nisms in AWESOME are very similar. Specifically, the match and relate methods of the
AWMechanism and the AJMechanism aspects are nearly identical.
However, the two mechanisms implement their \texttt{reify} advice and \texttt{order} methods differently. In AspectWerkz, an advice-annotated method has a dual purpose, i.e., it can be executed implicitly as advice, or explicitly as a method. An implicit execution should yield an \textit{method advice-execution} join point, and an explicit execution should generate a \textit{method-execution} join point. \texttt{AWMechanism} implements this behavior by explicitly separating advice and method roles of the annotated method. Prior to shadow identification step, the mechanism duplicates the annotated method, and uses its original copy for explicit executions, and the duplicate copy for implicit executions.

Specifically, the advice around top-level \texttt{reify} method uses the \texttt{isAWAspect} function to determine whether or not the argument class represents an AspectWerkz aspect. \texttt{reifyAWAspect} then constructs shadow representation of the aspect class similarly to the \texttt{reifyAJAspect} method of the AspectJ mechanism; and \texttt{dupAdvMethods} duplicates advice-annotated methods of the aspect class.

The \texttt{dupAdvMethods} method uses \texttt{copyMethod} to build a copy of a method object; \texttt{Method.setName} to rename a method; \texttt{ClassFile.addMethod} to add the method to the class file; and \texttt{removeAdvAnnotations} to remove advice annotations from a method declaration (thus dismissing its role as advice).

Similarly to the AspectJ mechanism, the \texttt{AWMechanism} aspect also advises method-level and instruction-level \texttt{reify} functions of the platform. Advice to the method-level function builds \texttt{AWAExecShadow} objects to represent executions of AspectWerkz advice methods; and the advice to the instruction-level function filters out \texttt{proceed} calls.

In contrast to AspectJ, where lexical positions of advice definitions in an aspect file affect order of advice, AspectWerkz orders advice only with respect to their types. In the \texttt{order} method, the \texttt{AWBeforeAdvice}, \texttt{AWAroundAdvice}, and \texttt{AWAfterAdvice} classes respectively implement \texttt{before}, \texttt{around}, and \texttt{after} advice of AspectWerkz; and the \texttt{orderTypedAdvice} method selects advice pieces of a specific type.
public aspect AWMechanism {

  Shadow[] around(ClassFile cf): args(cf) &&
  execution(Shadow[] Platform.reify(ClassFile)) {
    if (isAWAspect(cf)) dupAdvMethods(cf);
    return proceed(cf);
  }

  private void dupAdvMethods(ClassFile cf) {
    Method[] methods = cf.getMethods();
    for(int i=0;i<methods.length;i++) {
      Method m = methods[i];
      if (!isAdvice(m)) continue;
      Method advMeth = copyMethod(m);
      advMeth.setName("adv_"+i);
      cf.addMethod(advMeth);
      removeAdvAnnotations(m);
    }
  }

  public Advice[][] order(Shadow shd, Advice[] advArr) {
    Advice[][] result = new Advice[3][0];
    result[0] = orderTypedAdvice(advArr, AWBeforeAdvice.class);
    result[1] = orderTypedAdvice(advArr, AWAroundAdvice.class);
    result[2] = orderTypedAdvice(advArr, AWAfterAdvice.class);
    return result;
  }

  private Advice[] orderTypedAdvice(Advice[] advArr, Class advType) {
    List tmp = new ArrayList();
    for(Advice adv:advArr)
      if (advType.isInstance(adv)) tmp.add(adv);
    return (Advice[])tmp.toArray(new Advice[]);
  }

  /* Other implementation details are omitted */
}

Listing 6.15: The AspectWerkz mechanism

6.5.5 Implementing a COOL Mechanism

The refactoring of the COOL mechanism includes a change to the front-end for translating source COOL coordinators into annotated Java classes. The annotations mark the lock_
public aspect COOLMechanism {
  
  after(ClassFile cf): args(cf) &&
  execution(void Platform.weaveClass(ClassFile)) {
    ClassFile coordAspect = findAspect(cf);
    if (coordAspect!=null) {
      addCoordField(cf, coordAspect);
      transformConstructor(cf, coordAspect);
      addGetterMethods(cf, coordAspect);
    }
  }
  
  /* the rest of the implementation is omitted */
}

Listing 6.16: Implementation of the COOL mechanism

and unlock_ methods of the coordinator class and identify the requires, on_entry, and on_exit instruction blocks within these methods.

The COOL mechanism introduces shadow types for lock, unlock, requires, on_enter, and on_exit computations. The lock and unlock shadows represent executions of the lock_ and unlock_ methods. The requires, on_enter, and on_exit shadows represent executions of the corresponding COOL expressions and statements. They map to blocks of instructions within the lock_ and unlock_ methods. The bodies of the requires, on_entry and on_exit constructs are Java expressions and statements. The mechanism represents them using the base shadow domain (field-get and field-set shadows).

The COOLMechanism aspect advises the weaveClass and the reify methods of the platform. The after advice to the weaveClass method introduces into a coordinated (target) class a coordinator field and getter methods, and transforms the constructor of the class (Listing 6.16).

The advice around the reify method is similar to the corresponding advice in the AJMechanism aspect: if the argument class is a COOL coordinator class, then the advice provides a shadow representation for it; otherwise, the advice proceeds.

The COOL mechanism also provides an implementation for match, order and relate.
match selects lock and unlock pieces of advice by matching the coordinator classes against the method-execution shadows. The order method schedules the lock advice to run before the unlock advice. The relate method is empty: it returns an argument advice array.

6.6 Summary

The chapter provides a solution to the composition implementation problem. It presents AWESOME - a practical framework for composing aspect mechanisms that is:

- **Pluggable**: enables third-party composition of aspect mechanisms;

- **Customizable**: provides means for customizing the behavior of the constructed multi-extension mechanism to cater for the specification of the multi-extension AOPL; and

- **Efficient**: employs a compile-time weaving scheme.

The main architectural parts of the framework are (1) a platform that provides common operations and interfaces; (2) an aspect mechanism that implements an extension-specific behavior; and (c) a configuration that resolves feature interactions. Aspect mechanisms are independently developed against the common platform, and then composed into a default multi-weaver. The default multi-weaver resolves feature interactions in a pre-defined, standard manner. The configuration customizes the default multi-mechanism to implement a composition-specific feature interactions behavior.

In the next chapter we demonstrate and test the framework on real-world languages. We use it to implement COOLAJ, AJW, and COOLAJW multi-extension AOP languages. The first language composes AspectJ and COOL extensions; AJW is a composition of AspectJ and AspectWerkz; and COOLAJW composes AspectJ, AspectWerkz, and COOL.
Chapter 7

Evaluation

This chapter evaluates pluggability, composability, customizability, and efficiency properties of the framework, and compares AWESOME to the existing aspect extension composition tools. We evaluate the framework by:

- Demonstrating support for the AspectJ [Lopes and Kiczales, 1998, Kiczales et al., 2001] AOP language. AspectJ is a representative J&A extension. Many AOP languages mimic AspectJ conceptually, and often realize features that are similar to those of AspectJ. We demonstrate the framework’s support for AspectJ by constructing an AWESOME AspectJ weaver and comparing it to ajc [Hilsdale and Hugunin, 2004], the standard AspectJ compiler.

- Demonstrating support for the COOL [Lopes and Kiczales, 1997, Lopes, 1997] AOP language. COOL is one of the first domain-specific AOP languages that has been shown to be practically useful [Walker et al., 1998, 1999, Murphy et al., 1999, 2001b, Walker et al., 1995]. COOL is also small enough to permit evaluation by direct inspection of the implementation and comparison with the semantics of the language [Lopes and Kiczales, 1997, Lopes, 1997]. In this chapter we build an AWESOME weaver for COOL and compare it to the standard COOL weaving algorithm (Section 6.3).

- Demonstrating support for a multi-extension AOP language that integrates AspectJ
and COOL. Chapter 5 illustrates that the composition of AspectJ and COOL is far from trivial and requires a general consideration of the feature interaction problem in a composition of multiple aspect extensions. Demonstrating that the framework can make a plugin for AspectJ work in concert with a plugin for COOL is therefore a meaningful benchmark. In this chapter we construct and test an AWESOME multi-weaver for COOLAJ.

- Demonstrating support for a multi-extension AOP language that integrates AspectJ and AspectWerkz. The problem of composing AspectJ and AspectWerkz is a representative case of the more general problem of composing general-purpose J&A extensions. Moreover, this problem is the only one that has a working solution instance, i.e., AspectJ/5. In this chapter we build an AWESOME weaver for AJW, test it against the AJW specification (Section 5.6.2), and compare it to the ajc AspectJ/5 compiler.

- Demonstrating support for a composition of \( k > 2 \) aspect extensions. While the composition of AspectJ and COOL and the composition of AspectJ and AspectWerkz are representative challenges, it is necessary to also consider scalability of the framework. To evaluate that we build and test an AWESOME weaver for COOLAJW.

In Section 7.1 we informally specify and describe AWESOME implementations of multi-weavers for COOLAJ, AJW, and COOLAJW multi-extension AOP languages. COOLAJ combines COOL and AspectJ, AJW combines AspectJ and AspectWerkz, and COOLAJW composes COOL, AspectJ, and AspectWerkz.

Section 7.2 evaluates pluggability of the framework. It demonstrates that the framework supports third-party composition of aspect weavers from weaver plugins, platform, and configuration modules. Section 7.3 tests behavior of the framework weavers by comparing single-extension weavers to the standard aspect compilers; and validating behavior of multi-extension weavers against specifications of their respective multi-extension AOP languages. Section 7.4 evaluates the efficiency of the framework’s weaving scheme. It compares run-time
performance of aspect programs that are woven by the framework to run-time performance of
the same programs woven by standard aspect weavers and weaving algorithms (e.g., \texttt{ajc AspectJ/5} compiler, weaving alghorithm for COOL, manual weaving algorithms for COOLAJ and
COOLAJW). Section 7.5 summarizes the evaluation results and compares pluggability, com-
posability, customizability, and efficiency properties of the AWESOME, Reflex, and XAspects
frameworks.

7.1 Case Study: Composing AspectJ, AspectWerkz, and COOL

In this section we use the AWESOME framework to build three multi-extension AOP
languages, namely COOLAJ, AJW, and COOLAJW.

7.1.1 Implementing COOLAJ

Plugging the \texttt{AJMechanism} and the \texttt{COOLMechanism} aspects into the composition
Platform produces a multi-weaver with a default behavior. It lets aspects advise join points
within \texttt{requires, on_entry, and on_exit} expressions of coordinators. It lets coordina-
tors synchronize methods that are defined within aspects; and it allows coordinators and aspects
to co-advice the same method. Although this default behavior is reasonable, a specific multi-
extension composition of AspectJ and COOL may require different semantics. In this section
we implement a weaver for COOLAJ, a multi-extension AOP language that combines COOL
and AspectJ (Section 5.6.1). The implementation customizes the default multi-weaver using the
CoolaJConfig aspect.

The specification for COOLAJ is independent of the AWESOME architecture. It is based
only on the syntax and semantics of the AspectJ and COOL languages; not on their implemen-
tation. COOLAJ is specified as a conservative composition of AspectJ and COOL, i.e., it follows
as much as possible the original semantics of AspectJ and COOL. Specifically, in COOLAJ an
aspect is woven into classes and aspects according to the weaving semantics of AspectJ. Sim-
ilarly, a coordinator is woven into classes according to the weaving semantics of COOL. The specification for COOLAJ differs from the default multi-weaver in resolving emergent advice ordering, join point visibility, and join point advisability interactions. We realized a multi-weaver for COOLAJ by providing a CoolaJConfig aspect with three pieces of advice, one implementing the emergent advice ordering rules, and the other two realizing the visibility and advisability interactions.

**Emergent Advice Ordering.** In COOLAJ, COOL advice dominates AspectJ advice. CoolaJConfig implements the advice ordering rules of COOLAJ by advising the Platform.order method as shown in Listing 7.1. The advice orders COOL advice to be woven last thus “wrapping” around the AspectJ advice.

In the listing, LockAdv and UnlockAdv classes respectively implement lock and unlock advice of COOL; elTypePos returns a first position of an object of a given class in the array; and the move method moves an element of an array from one position to another. Specifically, move(advs, fromPos, toPos) moves an element at the position fromPos of the advs array to the position toPos, and shifts elements between fromPos (exclusively) and toPos (inclusively) to the left, if fromPos < toPos, or to the right, if fromPos > toPos. mvToEnd is an auxiliary method that is used to move COOL advice to the end of the multi-extension advice array. The
Advice[] around(Shadow shadow): args(Shadow shadow) &&
        (call(Advice[] AJMechanism.match(..)) ||
         call(Advice[] AJMechanism.order(..)) ||
         call(Advice[] AJMechanism.relate(..))) {
            return proceed(isLockOrUnlock(shadow) ?
                           maskAsAJAExec(shadow) : shadow);
        }

Listing 7.2: Normalizing multi-extension join points in COOLAJ

Advice[] around(Shadow shadow): args(Shadow shadow) &&
        execution(Advice[] AJMechanism.match(..)) {
            Advice[] advs = proceed(shadow);
            if (isCondFieldAccess(shadow) ||
                isLockOrUnlock(shadow))
                advs = removeElType(advs, AJAroundAdvice.class);
            return advs;
        }

Listing 7.3: Restricting advisability of foreign join points in COOLAJ

COOL advice is scheduled to be woven the last, thus dominating the AspectJ advice at run-time.

Join Point Visibility. To allow aspects to advise lock and unlock computations as advice-executions, CoolaJConfig normalizes lock and unlock shadows of COOL with advice-execution shadows of AspectJ by advising calls to the match, order, and relate methods of the AJMechanism aspect.

The implementation is shown in Listing 7.2. In the listing, isLockOrUnlock tests if shadow is a lock or unlock shadow, and maskAsAJAExec masks the COOL shadow as an AspectJ’s advice-execution shadow. As a result, AspectJ advises the lock and unlock shadows as if they were advice-execution shadows.

Join Point Advisability CoolaJConfig restricts the advising of join points within COOL coordinators by advising the executions of the AJMechanism.match method as shown in Listing 7.3. In the listing, the AJAroundAdvice class implements AspectJ’s around advice, isCondFieldAccess checks if the shadow represent access to a condition field of a COOL co-
ordinator, and removeElType removes all elements of a given type from the array.

7.1.2 Implementing AJW

AWESOME makes it very easy to implement AJW, the multi-extension AOP language specified in Section 5.6.2. Plugging AJMechanism and AWMechanism aspects into the platform does most of the work. The default multi-weaver complies with the AJW specification in resolving all but the join point visibility interaction. E.g., the multi-weaver preserves a partial order of AspectJ and AspectWerkz advice, and co-orders multi-extension advice as desired.

The default multi-weaver resolves the join point visibility interactions between the extensions by allowing AspectJ aspects to advise all Java shadows in AspectWerkz aspects, and vice versa. The AJW specification, however, requires AspectJ aspect to also advise implicit executions of AspectWerkz’s advice methods as advice-execution join points. In the default multi-weaver, however, AspectWerkz advice are reified using the AWAExecShadow shadow class. This class is unknown to the AspectJ mechanism that uses the AJAExecShadow objects.

To allow AspectJ aspects to advise executions of AspectWerkz advice as advice-execution join points we customize the default multi-weaver with a AJWConfig aspect. AJWConfig masks AWAExecShadow objects as AJAExecShadow objects for the match, order, and relate methods of AJMechanism (Listing 7.4). In the listing, maskAWAExecAsAJAExec masks AWAExecShadow objects as AJAExecShadow shadows.
7.1.3 Implementing COOLAJW

COOLAJW is a multi-extension AOP language that combines COOL, AspectJ, and AspectWerkz. Section 5.6.3 defines this language as a “merger” of the COOLAJ and AJW multi-extension AOP languages. We implement COOLAJW in AWESOME by plugging COOLMechanism, AJMechanism, and AWMechanism to the platform, and customizing the constructed default multi-weaver with the CoolaJConfig, AJWConfig, and CoolaJWConfig configuration aspects.

Behavior of the default multi-weaver differs from the COOLAJW specification in resolving emergent advice ordering, join point visibility, and join point advisability interactions. Because COOLAJW is a “merger” of COOLAJ and AJW, their configuration aspects (CoolaJConfig and AJWConfig) also work for resolving COOLAJW interactions. When woven with the default multi-weaver, CoolaJConfig properly customizes foreign interactions between the COOL and AspectJ features, and provides desired multi-extension advice ordering behavior. Similarly, AJWConfig properly resolves interactions between the AspectJ and AspectWerkz features.

The two configuration aspects, however, do not customize interactions between the features of COOL and AspectWerkz. We provide a CoolaJWConfig aspect for customizing these interactions. Specifically, CoolaJWConfig restricts the advising of AspectWerkz join points within COOL coordinators by advising the executions of the AWMechanism.match method. Listing 7.5 shows the corresponding advice. In the listing, the AWAroundAdvice class implements AspectWerkz’s around advice.

7.2 Third-party Composition

We evaluated the pluggability feature of AWESOME by constructing six different weavers from the same building blocks. The building blocks are jar files containing compiled aspects and classes. platform.jar is the stripped down platform containing the Platform
Advice[] around(Shadow shadow): args(shadow) &&
    execution(Advice[] AWMechanism.match(..)) {
    Advice[] advs = proceed(shadow);
    if (isCondFieldAccess(shadow))
        advs = removeElType(advs, AWAroundAdvice.class);
    return advs;
}

Listing 7.5: Configuring the COOLAJW weaver

class and the abstract Mechanism aspect. The jars ajm.jar, awm.jar, and coolm.jar
contain the concrete independently developed aspects AJMechanism, AWMechanism, and
COOLMechanism for AspectJ, AspectWerkz, and COOL, respectively. coolajConf.jar
contains the CoolaJConfig aspect for customizing the composition of COOL and AspectJ;
ajwConf.jar contains the AJWConfig aspect for customizing the composition of AspectJ
and AspectWerkz; and coolajwConf.jar contains the CoolaJWConfig, CoolaJConfig,
and AJWConfig aspects for customizing the composition of COOL, AspectJ, and AspectWerkz.

We verified that it is possible, using the command line, to construct weavers for AspectJ,
AspectWerkz, COOL, COOLAJ, AJW, and COOLAJW from the seven building blocks. We
constructed a stand-alone AspectJ weaver, named ajx, by plugging just the AspectJ mechanism
into the platform. The command line is (Figure 7.1):

    ajc -inpath platform.jar; ajm.jar -outjar ajx.jar

where ajc is the original (non-refactored) version of the AspectJ compiler. The inpath option
directs ajc to weave classes within jar files. The outjar option directs the compiler to save
the woven classes into a separate jar file.

To construct a stand-alone COOL weaver, named coolx, we plugged only the COOL
mechanism (Figure 7.2):

    ajc -inpath platform.jar; coolm.jar -outjar coolx.jar

Similarly, to construct a stand-alone AspectWerkz weaver, named awx, we plugged the
AspectWerkz mechanism:
To construct an AJW multi-weaver, named \texttt{ajwx}, that combines AspectJ and AspectWerkz, we ran (Figure 7.3):

\begin{verbatim}
ajc -inpath platform.jar;ajm.jar;awm.jar -outjar ajwx.jar
\end{verbatim}

To construct a COOLAJ multi-weaver, named \texttt{coolajx}, that combines COOL and AspectJ, we ran (Figure 7.4):

\begin{verbatim}
ajc -inpath platform.jar;coolm.jar;ajm.jar;
coolajConf.jar -outjar coolajx.jar
\end{verbatim}

To construct a COOLAJW multi-weaver, named \texttt{coolajwx}, that combines COOL, AspectJ, and AspectWerkz, we ran (Figure 7.5):

\begin{verbatim}
ajc -inpath platform.jar;coolm.jar;ajm.jar;awm.jar;
coolajwConf.jar -outjar coolajwx.jar
\end{verbatim}

To compile and run a multi-extension aspect program, a file with unwoven bytecode \texttt{unwoven.jar} (an unwoven program including aspect and base classes) was passed to the multi-weaver to produce a woven file:

\begin{verbatim}
java -cp <weaver>.jar;aspectjtools.jar
org.aspectj.tools.ajc.Main -inpath
unwoven.jar -outjar woven.jar
\end{verbatim}
where `<weaver>` was one of `ajx, awx, coolx, coolajx, ajwx, or coolajwx`; and `woven.jar` is the woven bytecode program that can be run on a JVM as a regular Java program.

### 7.3 Testing

We tested the six weavers to determine with high confidence that indeed they implement semantics of their multi-extension AOP languages, i.e., that `ajx` implements the semantics of AspectJ; `awx` implements the semantics of AspectWerkz; `coolx` implements the semantics of COOL; `ajwx` realizes the specification for AJW; `coolajx` realizes the specification for COOLAJ; and `coolajwx` realizes the specification for COOLAJW. We did this by observing the runtime behavior of test programs; by inspecting their woven bytecode; by analyzing join point traces; and, when possible, by comparing the results to programs compiled with standard aspect compilers and algorithms. Because the framework is based on `ajc`, which is assumed
correct, we focused testing on a coverage of the newly introduced and refactored behavior.

The multi-extension AOP languages are defined inductively from AspectJ, AspectWerkz, and COOL. For example, COOLAJ is defined as a composition of COOL and AspectJ; and COOLAJW is a composition of COOLAJ and AJW. Since a multi-extension AOP language preserves semantics of its sublanguages (e.g., meaning of an AspectJ program in COOLAJ, AJW, and COOLAJW is the same as in AspectJ), its weaver must implement all their specifications. E.g., coolajwx must implement AspectJ, AspectWerkz, COOL, AJW, and COOLAJ languages.

We therefore grouped tests by six languages and reused them for verifying different weavers (Table 7.1). A weaver is verified using tests for all the AOP languages it implements, e.g., coolajx tests include AspectJ, COOL, and COOLAJ programs; and coolajwx is verified using tests in all the six languages.

<table>
<thead>
<tr>
<th>Test Group</th>
<th>ajx</th>
<th>awx</th>
<th>coolx</th>
<th>ajwx</th>
<th>coolajx</th>
<th>coolajwx</th>
</tr>
</thead>
<tbody>
<tr>
<td>AspectJ</td>
<td>+</td>
<td>+</td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>AspectWerkz</td>
<td></td>
<td></td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COOL</td>
<td></td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AJW</td>
<td></td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COOLAJ</td>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COOLAJW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+</td>
</tr>
</tbody>
</table>

Table 7.1: Test Plan

Test cases are constructed from seven modules, namely the Stack.java (Listing 2.3) class in Java; the Stack.cool (Listing 2.4) coordinator in COOL; AJLogAll.aj, AJTouchAll.aj, and LogAdviceOnStack.aj (Listing 7.6) aspects in AspectJ; and AWLogAll.java and AWTouchAll.java aspects in AspectWerkz. The AJLogAll aspect advises all the join points in a program except those within the aspect itself (to prevent an infinite loop)\(^1\) with before, around, and after advice, and logs the join points to a file. AJTouchAll also advises everything but itself,\(^2\) but just “touches” the join points with empty advice, i.e., its before and after advice are empty, and the around advice does nothing but proceeds. AWLogAll and AWTouchAll

\(^1\) AJLogAll advises `cflow(within(AJLogAll))` join points
\(^2\) AJTouchAll advises `cflow(within(AJTouchAll))`
are AspectWerkz versions of the AJLogAll and AJTouchAll aspects, respectively.

The LogAdviceOnStack aspect logs all advice (except for its own) at Stack method-execution join points. The

cflow(tgt) && !cflowbelow(tgt())

pointcut selects not only tgt() join points, but also join points within aspects that advise the tgt() join points. In particular, LogAdviceOnStack would advise join points within the AJLogAll aspect when the two are used together with the Stack class.

Using different configurations of the test modules we’ve built 13 tests cases. Table 7.2 shows the test cases and maps them to the test modules and test groups. The table represents the modules as columns, rows represent the test cases and test groups, and + indicates inclusion of a module into a test case. For space considerations the table refers to the test modules via nicknames: Log and Touch subcolumns of the AspectJ (AspectWerkz) section respectively correspond to the AJLogAll.aj (AWLogAll.java) and AJTouchAll.aj (AWTouchAll.java) modules; LogAdvice is a nickname for LogAdviceOnStack.java; and COOL’s Stack column denotes the Stack.cool coordinator. The Stack.java class is not shown in the table because it participates in all the test cases.

Tests in AspectJ, AspectWerkz, and COOL groups compare framework’s weavers to the ajc, aspectwerkz, and Xcool compilers and algorithms, respectively (Xcool is a standard compilation algorithm for COOL that is applied manually). Tests for multi-extension AOP lan-

```java
public aspect LogAdviceOnStack {
    pointcut scope():
        !cflow(within(LogAdviceOnStack));
    pointcut tgt(): execution(* Stack.*(..));
    before(): scope() &&
        cflow(tgt()) && !cflowbelow(tgt()) {
            System.out.println(thisJoinPoint);
        }
}
```

Listing 7.6: LogAdviceOnStack
<table>
<thead>
<tr>
<th>Group</th>
<th>Test Case</th>
<th>AspectJ</th>
<th>AspectWerkz</th>
<th>COOL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Log</td>
<td>Touch</td>
<td>Log</td>
</tr>
<tr>
<td>AspectJ</td>
<td>compare to $ajc$</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>AspectWerkz</td>
<td>compare to $aspectwerkz$</td>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>COOL</td>
<td>compare to $xcool$</td>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>AJW</td>
<td>$aj \xrightarrow{adv} aw$</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$aw \xrightarrow{adv} aj$</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$+{aj, aw}$</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>COOLAJ</td>
<td>$aj \xrightarrow{adv} cool$</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>$+{aj, cool}$</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>COOLAJW</td>
<td>$aw \xrightarrow{adv} cool$</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>$+{aw, cool}$</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>$aj \xrightarrow{adv} {aw, cool}$</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>$aw \xrightarrow{adv} {aj, cool}$</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>$+{aj, aw, cool}$</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 7.2: Test Cases

Languages comprise foreign advising and co-advising cases. An $\xrightarrow{adv}$ arrow from an adviser extension to advisee extension(s) denotes a foreign advising test case, e.g., the $aj \xrightarrow{adv} cool$ case verifies behavior of AspectJ aspects in advising COOL coordinators; and $aw \xrightarrow{adv} \{cool, aj\}$ verifies how AspectWerkz aspects advise COOL coordinators and AspectJ aspects. Co-advising tests are denoted using $+$ symbol, e.g., $+\{aj, aw, cool\}$ verifies co-advising behavior of AspectJ, AspectWerkz, and COOL aspects.

7.3.1 Testing AspectJ

We evaluate AspectJ weavers by comparing framework-woven bytecode to $ajc$-woven bytecode. In fact, the main difference between the framework’s AspectJ plugin and $ajc$ is in the design and implementation of the $reify$ process. In the implementation of the AspectJ plugin we disentangled the monolithic $reify$ process of $ajc$ into a common platform $reify$ method and an AspectJ-specific advice of the AspectJ mechanism. The other processes were either left unchanged (e.g., $match$ and $order$), or undergone a coarse-grained (and assumed behavior-
preserving) transformations (e.g., \texttt{relate,mix}). Thus, we hypothesize that the AspectJ plugin is a behavior-preserving refactoring of \texttt{ajc}, if the \texttt{reify} processes of the two exhibit the same behavior, i.e., given a Java class or an AspectJ aspect they build identical shadow representations.

To test the \texttt{reify} process and reason about its shadow representation, we generated an exhaustive join point trace by weaving together three classes and aspects: \texttt{Stack.java} (Listing 2.3); \texttt{AJLogAll.aj}, and \texttt{AJTouchAll.aj}. The woven bytecode is run by a main program that creates a \texttt{Stack} object and invokes \texttt{push} and \texttt{pop} in a single thread. The execution produces an exhaustive trace of the join points within \texttt{Stack} and within \texttt{AJTouchAll}. This trace provides a good insight into behavior of the \texttt{reify} process, because it covers almost all types of join points and includes join points within both Java classes and AspectJ aspects.

We applied \texttt{ajx}, \texttt{ajwx}, \texttt{coolajx}, \texttt{coolajwx}, and \texttt{ajc} to the test program and obtained five woven bytecode programs. We executed the bytecode programs one by one using the same main program and obtained five identical join point traces. We therefore conclude that, at least on this benchmark example, the four framework weavers behave the same as \texttt{ajc}, and are likely to exhibit an \texttt{ajc}-equivalent behavior for AspectJ programs in general.

### 7.3.2 Testing AspectWerkz

We tested the \texttt{awx}, \texttt{ajwx}, and \texttt{coolajwx} framework weavers by comparing them to \texttt{aspectwerkz}, the standard AspectWerkz compiler. Specifically, we compared framework-woven bytecode to \texttt{aspectwerkz}-woven bytecode. Because the framework’s AspectWerkz plugin is built by refactoring \texttt{ajc} we scoped testing to differences between AspectJ and AspectWerkz, and between \texttt{ajc} and the AspectWerkz plugin. In terms of aspect extension features, the two extensions differ in their join point granularity, advice execution, and advice ordering features. In terms of the weaver processes, the plugin and \texttt{aspectwerkz} differ in their \texttt{reify}, \texttt{match} and \texttt{order} processes. Thus, we hypothesize that each of the three framework weavers implements semantics of AspectWerkz if (1) an AspectWerkz aspect observes and advises the
same join points in framework-woven and aspectwerkz-woven code, i.e., reify and match
processes of the AspectWerkz plugin exhibit the same shadow identification and advice selec-
tion behavior as aspectwerkz; and (2) the order of advice application in the framework-woven
bytecode is the same as in aspectwerkz, i.e., the order process of the AspectWerkz plugin
exhibits the same advice ordering behavior as aspectwerkz.

We tested the three AspectWerkz weavers in the same manner as we tested the AspectJ
weavers, i.e., by comparing join point traces of framework-woven and aspectwerkz-woven
programs. We applied the framework weavers and aspectwerkz to the test AspectWerkz pro-
gram and obtained five woven bytecode programs. We then executed the bytecode programs one
by one using the same main program that invokes all the methods of Stack and AWTouchAll
classes. The executions produced five identical join point traces thus indicating (on this bench-
mark example) that the reify and match processes of the framework weavers exhibit correct
behavior.

To test the order process we repeated the weaving and execution steps for every possi-
ble order of before, around, and after advice definitions in the AWTouchAll.java file. At
each iteration we obtained identical join point traces. We therefore conclude that, at least on
this benchmark example, awx, ajwx, and coolajwx behave the same as aspectwerkz, and are
likely to exhibit an aspectwerkz-equivalent behavior for AspectWerkz programs in general.

7.3.3 Testing COOL

We tested whether the runtime behavior of framework-woven COOL programs complies
with the standard weaving algorithm of COOL [Lopes, 1997] that we refer to as Xcoolc. We
applied coolx, coolajx, and coolajwx to the COOL test program and obtained three woven
bytecode programs. The test program comprises Stack.java (Listing 2.3) and Stack.cool (List-
ing 2.4), and employs all the features of COOL (i.e., selfex, mutex, requires, on_entry,
on_exit, and access to private field of a coordinated class from a coordinator aspect). We
tested the woven programs by observing their run-time behavior, and by inspecting their byte-
code. As part of testing we also inspected the StackCoord.java file that was constructed by the COOL front-end from Stack.cool.

To test the runtime behavior we executed the woven programs one by one using the same main program. The main program creates a Stack instance with a very small capacity (size of 5), and invokes its methods concurrently by five reader and five writer threads. A reader thread attempts to remove 5000 objects from the stack, while a writer thread attempts to add 5000 objects onto the stack. All executions completed successfully (i.e., executed all the threads to completion without throwing an exception), indicating, with a high probability, that the woven programs behave correctly. Additional inspection of their bytecode verified that behavior of the woven programs is the same as behavior of the manually-woven class presented in Listing 6.8.

We tested the front-end translator by comparing the generated StackCoord.java coordinator class against the manual translation presented in Listing 6.7. We concluded that the lock_ and unlock_ methods of the generated class encode the same behavior as the corresponding lock_ and unlock_ methods in Listing 6.7.

All the tests succeeded. The coolx, coolajx, and coolajwx weavers exhibited the correct dynamic and compilation semantics for the test COOL program that comprised Stack.java and Stack.cool. We consider the test program to be a representative COOL application since it uses all the features of COOL. We thus conclude with a high degree of confidence that the three framework weavers would generally weave COOL programs correctly.

7.3.4 Testing AJW

We hypothesize that an AJW weaver correctly weaves a program with Java classes, AspectJ and AspectWerkz aspects if it weaves the AspectWerkz aspects into classes and other AspectWerkz aspects according to the semantics of AspectWerkz; weaves the AspectJ aspects into classes and other AspectJ aspects according to the semantics of AspectJ; weaves AspectJ (AspectWerkz) aspects into AspectWerkz (AspectJ) aspects according to the foreign advising specification of AJW; and coordinates the weaving of multi-extension advice according to
the co-advising specifications of AJW. The two AJW weavers that we tested are `ajwx` and `coolajwx`.

We tested the foreign advising behavior of each weaver using a single test program. The program tested the behavior of AspectJ aspects advising AspectWerkz aspects (the \(aj \xrightarrow{adv} aw\) case in Table 7.2), and the behavior of AspectWerkz aspects advising AspectJ aspects (the \(aw \xrightarrow{adv} aj\) case). The program comprised the `Stack.java`, `AJLogAll.aj`, and `AWLogAll.java` modules. We compiled the program using `ajwx` and `coolajwx`; executed the woven programs using a single-threaded main program that invokes `Stack` and `AWLogAll` methods; and validated the produced join point traces against the AJW specification. To test the \(aj \xrightarrow{adv} aw\) and the \(aw \xrightarrow{adv} aj\) behaviors separately, we configured `AJLogAll` and `AWLogAll` to log join points into different files.

Executions of the `ajwx`-woven and `coolajwx`-woven test bytecode with the main program produced identical AspectJ and AspectWerkz join point traces. We analyzed these join point traces, and concluded that they comply with the foreign advising specification of AJW.

We tested the co-advising behavior (the \(+\langle aj, aw\rangle\) case in Table 7.2) on a program that contains `Stack.java`, `AJTouchAll.aj`, `LogAdviceOnStack.aj` (Listing 7.6), and `AWTouchAll.java`. Similarly to the foreign advising case, we compiled the test program using both weavers, executed the woven bytecode programs using the same single-threaded main program, obtained identical join point traces, and validated them against the AJW co-advising specification. The execution trace of a `Stack` method reflected that: (1) execution of `AWTouchAll` and `AJTouchAll` at a `Stack` method execution join point is advised by `LogAdviceOnStack`; and (2) at the same join point, the `before` (`after`) advice of AspectJ and AspectWerkz executes before (after) their `around` advice. The first result shows that `AWTouchAll` advice executes in the control flow of the join point it advises. The second result shows that `AWTouchAll` and `AJTouchAll` advice are properly co-ordered at a join point.

`ajwx` and `coolajwx` passed all the foreign advising and co-advising tests. We verified that they weave the test programs according to the AJW semantics. The test programs provide
a good coverage of the AJW specification. Therefore, we conclude with high confidence that `ajwx` and `coolajwx` weave AJW programs correctly.

### 7.3.5 Testing COOLAJ

We hypothesize that a COOLAJ weaver correctly weaves a program with Java classes, AspectJ aspects and COOL coordinators if it weaves the coordinators into their matching classes according to the semantics of COOL; weaves the aspects into classes and other aspects according to the semantics of AspectJ; weaves aspects into coordinators (the $aj \xrightarrow{adv} cool$ case in Table 7.2) according to the foreign advising specification; and coordinates the weaving of multi-extension advice according to the co-advising specifications of COOLAJ (the $+(aj, cool)$ case in Table 7.2). The two COOLAJ weavers that we tested are `coolajx` and `coolajwx`.

To verify the $aj \xrightarrow{adv} cool$ behavior we compiled the `Stack.java`, `Stack.cool` and `AJLogAll.aj` modules using `coolajx` and `coolajwx`; executed the woven programs using a single-threaded main program that invokes `Stack` methods; and validated the produced (identical) join point traces against the COOLAJ specification. The specification defines the shadow representation of COOL coordinators; the normalization between COOL and AspectJ shadow types; and mapping between AspectJ advice types and coordinator-located shadows. In particular, we verified that the traces contain only expected join points, that they reflect executions of `lock_` and `unlock_` methods as `advice-execution` join points, and that the `around` advice of `AJLogAll` is properly filtered (e.g., not applied at the `advice-execution` join points).

We also verified weaving of AspectJ aspects into Java classes by validating the traces against AspectJ specification. To verify weaving of coordinators we executed the woven bytecode programs using the same multi-threaded main program as we used for testing COOL.

We also tested the ordering of multi-extension advice on a program that contains `Stack.java`, `Stack.cool`, `LogAdviceOnStack.aj`, and `AJTouchAll.aj`. The execution trace of a `Stack` method reflected that: (1) execution of `StackCoord` is advised by `LogAdviceOnStack`; and (2) the

---

3 Coordinators never weave other coordinators.
first and the last advice-execution join points around a Stack method execution join point that are not in the control flow of LogAdviceOnStack are executions of lock_ and unlock_ methods, respectively. The first result shows that COOL advice executes in the control flow of the join point it advises. The second result shows that the COOL advice takes precedence over AspectJ advice at the same join point.

coolajx and coolajwx passed all these tests. We verified that both weavers compiled the input programs according to the COOLAJ semantics. The input programs provide a good coverage of the COOLAJ specification. Therefore, we conclude with high confidence that coolajx and coolajwx weave COOLAJ programs correctly.

7.3.6 Testing COOLAJW

COOLAJW tests fall into two subgroups, first targeting interactions between AspectWerkz and COOL aspects (aw \xrightarrow{adv} cool and +⟨aw, cool⟩ cases from Table 7.2); and the second one testing interactions between AspectWerkz, AspectJ, and COOL aspects (aj \xrightarrow{adv} ⟨aw, cool⟩, aw \xrightarrow{adv} ⟨aj, cool⟩, and +⟨aj, aw, cool⟩ cases from Table 7.2). Tests in the first subgroup validate that given a program with Java classes, AspectWerkz aspects, and COOL coordinators coolajwx weaves the coordinators into their matching classes according to the semantics of COOL; weaves the AspectWerkz aspects into classes and other AspectWerkz aspects according to the semantics of AspectWerkz; weaves AspectWerkz aspects into coordinators according to the foreign advising specification of COOLAJW; and coordinates the weaving of AspectWerkz and COOL advice according to the co-advising specifications of COOLAJW.

In addition to that the second subgroup validates that given a program with Java classes, AspectJ aspects, AspectWerkz aspects, and COOL coordinators coolajwx weaves AspectJ (AspectWerkz) aspects into coordinators and AspectWerkz (AspectJ) aspects according to the foreign advising specification; and coordinates the weaving of AspectJ, AspectWerkz, and COOL advice according to the co-advising specifications of COOLAJW.

Tests in the first subgroup are essentially the same as the COOLAJ tests. The only differ-
ence is that AspectJ aspects are replaced in the test cases with their AspectWerkz counterparts (e.g., using \texttt{AWLogAll} instead of \texttt{AJLogAll}). The details are omitted.

To validate the foreign advising and co-advising behavior in the second subgroup we used various combinations of \texttt{Stack.java}, \texttt{Stack.cool}, \texttt{AJLogAll.aj}, \texttt{AJTouchAll.aj}, \texttt{AWLogAll.java}, and \texttt{AWTouchAll.java} modules. To test the $\text{adv} \rightarrow \langle \text{aw}, \text{cool} \rangle$ case we compiled the \texttt{Stack.java} class together with \texttt{Stack.cool} coordinator, the \texttt{AJLogAll.aj} (\texttt{AWLogAll.java}) and \texttt{AWTouchAll.java} (\texttt{AJTouchAll.aj}) aspects, and executed a single-threaded main program. The main program invoked \texttt{Stack} methods and advice methods of the \texttt{AWTouchAll (AWLogAll)} aspect. We then verified that join point traces constructed by the main program execution comply with the foreign advising specification of \texttt{COOLAJW}.

To test the $+\langle \text{aj}, \text{aw}, \text{cool} \rangle$ case we compiled the \texttt{Stack.java}, \texttt{Stack.cool}, \texttt{AJTouchAll.aj}, \texttt{LogAdviceOnStack.aj}, and \texttt{AWTouchAll.java} modules. The execution trace of a Stack method reflected that: (1) execution of \texttt{AWTouchAll, StackCoord} and \texttt{AJTouchAll} is advised by \texttt{LogAdviceOnStack}; and (2) at the same join point, the \texttt{before (after)} advice of AspectJ and AspectWerkz executes before (after) their \texttt{around} advice; and \texttt{StackCoord} advice dominates over the AspectJ and AspectWerkz advice. The first result shows that AspectWerkz and COOL advice execute in the control flow of the join point they advise. The second result shows that \texttt{AWLogAll, StackCoord, and AJLogAll} advice are properly co-ordered at a join point.

\texttt{coolajwx} passed all the tests. We verified that \texttt{coolajwx} weaved the input programs according to the \texttt{COOLAJW} semantics. The input programs provide a good coverage of the \texttt{COOLAJW} specification. Therefore, we conclude with high confidence that \texttt{coolajwx} performs a correct weaving of \texttt{COOLAJW} programs.
7.4 Performance

We evaluated the efficiency of the framework by comparing the runtime performance of the bytecode produced by the framework weavers to the bytecode woven by standard compilers and weaving algorithms. This is intended to verify that the quality of woven bytecode is unaffected by the improved design of the weaver. Specifically, we validated that the runtime performance of framework-woven bytecode is the same as:

(1) ajc-woven for AspectJ programs;

(2) aspectwerkz-woven for AspectWerkz programs;

(3) ajc-woven$^4$ for AJW programs;

(4) Xcool-woven for COOL programs;

(5) Xajcool-woven for COOLAJ programs;

(6) Xajwcool-woven for COOLAJW programs.

where Xcool, Xajcool, and Xajwcool are manually applied weaving algorithms for COOL, COOLAJ, and COOLAJW, respectively.

In our evaluation we used all the six framework weavers. For each test case, however, all the compatible weavers produced identical woven bytecode programs (e.g., given an AspectJ program ajx, ajwx, coolajx, and coolajwx output the same bytecode). In the test case context terms “framework weaver” and “framework-woven” thus refer to a compatible weaver and its output, respectively.

To test the framework weavers we used four modules, namely the Stack.java class in Java, the AJTouchAll.aj and AWTouchAll.java aspects in AspectJ and AspectWerkz, respectively; and the Stack.cool coordinator in COOL. To evaluate COOL-enabled weavers, we ran a multithreading program that creates a Stack object and invokes its methods using ten writer-reader

---

$^4$ ajc is an AspectJ/5 compiler
threads. A writer-reader thread performs 5000 pairs of push-pop operations. The test program reported the average running time of a thread (in milliseconds) over series of 10 runs. To evaluate other weavers, we ran a single-threaded program that created a Stack object, and invoked its push and pop methods 5000 times each. We measured the average running time of executing the operations over a series of 10 runs.

Figure 7.6 summarizes the measured execution times. Programs compiled with coolx are as efficient as those compiled with Xcool; efficiency of ajx-woven and ajwx-woven bytecode is comparable to ajc-woven programs. Efficiency of woven code produced by the awx, ajwx, and coolajwx weavers is comparable to aspectwerkz-woven programs. Programs compiled with coolajx and coolajwx are within 4% efficiency compared to (optimal) code woven mechanically using the Xajcool and Xajwcool algorithms (and ajc as a back-end compiler). This indicates that the framework design overhead on the performance of the woven bytecode is negligible, i.e., there is almost no overhead to supporting the plugin architecture.
Table 7.3: Comparison of aspect extension composition frameworks.

<table>
<thead>
<tr>
<th>Property / Framework</th>
<th>Reflex</th>
<th>XAspects</th>
<th>AWESOME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composition approach</td>
<td>Reflection</td>
<td>Preprocessing</td>
<td>Compilation</td>
</tr>
<tr>
<td>Composability</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Customizability</td>
<td>-^5</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Efficiency</td>
<td>+^6</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Real-world languages</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

7.5 Summary

The evaluation demonstrated that the AWESOME framework is pluggable, customizable, and produces efficient woven code. In contrast, the current composition frameworks leave much to be desired in terms of composability, customizability, and efficiency (Table 7.3). Reflex [Tantzer and Noyé, 2005] and XAspects [Shonle et al., 2003] do not support the level of composability or customizability that is necessary for resolving foreign advising. These frameworks implement the composition by translating source aspects in foreign extensions to aspects in a common target language. The translation introduces and exposes in the target aspects synthetic join points that do not exist in the source. However, in Reflex and in XAspects, foreign aspects cannot distinguish the synthetic from the genuine join points. Moreover, Reflex and XAspects provide no mechanism (or composition rules) for customizing the foreign advising behavior, thus preventing the integrator from being able to correct the faulty resolution of this feature interaction. As a result, aspect programs compiled in Reflex- and XAspects-based multi-extension weavers may exhibit incorrect behavior [Kojarski and Lorenz, 2005, 2007b].

In Reflex there is ample support for configuring co-advising at the aspect level. A programmer can resolve interactions between aspects in a specific aspect program. AWESOME, on the other hand, supports customizability at the language level. A language designer can resolve the interactions between aspect extensions, thus affecting the behavior of all multi-extension programs.

---

^6 Reflex provides support for resolving interactions between aspects, but not between aspect extensions.

^6 The reflection-based weaving scheme of Reflex may degrade the runtime performance of the woven program.
AWESOME is also the only composition framework to be demonstrated and tested on real-world languages. We demonstrate the construction of AWESOME weavers for the COOLAJ, AJW, and COOLAJW multi-extension AOP languages. To the best of our knowledge, Reflex has not been shown to work with AspectJ. We only found a plugin that implements a limited subset of AspectJ. The plugin, however, does not advise AspectJ aspects correctly, emphasizing the general limitation of the Reflex framework to support foreign advising [Kojarski and Lorenz, 2007b]. Moreover, compile-time weaving scheme used by AWESOME more practical and efficient than the reflection-based schema used by Reflex. Similar to AWESOME, XAspects too explored a composition of COOL and AspectJ. However, the XAspects weaver exhibits incorrect weaving behavior that may result in deadlock [Kojarski and Lorenz, 2005].
Chapter 8

Conclusion

Aspect-oriented programming has over a decade of research and development and industry adoption. There are many AOP tools, languages, and frameworks. They range from widely-used production-quality aspect extensions to research-level aspect mechanisms. New AOP features are continually being proposed.

Currently, there are three ways to realize new features. Some features are realized as addons to extensible aspect compilers (e.g., abc). However, since an extensible compiler typically allows only fine-grained extensions to its AOP language (e.g., AspectJ), this realization method is good mostly for pointcut constructs. Furthermore, the extensible compilers are not designed to handle interactions between newly introduced features, because the supported features do not naturally interfere with each other. Thus, this method is quite limited.

More general AOP features (e.g., thread synchronization, security) require their own aspect extensions. These features are realized using AOP kernels (e.g., Reflex) that introduce primitives for expressing various weaving operations. Unfortunately, the state of the art in AOP kernels is to realize the features as stand-alone compilers. Consequently, it is often impossible to safely use a kernel-based aspect extension with another extension in the same program. As a result, the practical impact of even the most useful new aspect features and extension (e.g., COOL, KALA) is often negligible.

Conceptually, AOP kernels can support a wide range of weaving operations. In practice,
however, AOP kernels lack expressiveness and efficiency that is necessary for implementing complex production-quality aspect extensions. The dominant general-purpose aspect compilers (e.g., ajc) are thus realized from scratch. These implementations are not composable. The extensions cannot be used concurrently in the same program. The user base of even very similar AOP tools (e.g., Spring and AspectJ) are disjoined: programmers have to choose one or the other, and are restricted to the capabilities of the chosen extension.

The current situation in the AOP field resembles the pre-Eclipse state-of-the-art in Integrated Development Environment (IDE) development. There are multiple independent tools that share many similarities; and although some of the tools provide a certain level of extensibility, the tools and their features cannot be used together. A general, practical, and efficient environment for third-party construction and composition of aspect extensions can change the situation in the AOP field in the same way as the Eclipse platform changed IDE development. Similarly to Eclipse, the environment would allow language designers to realize AOP features and aspect extensions as plugins. Just as Eclipse defines standards and interfaces for IDE components, an aspect DSL environment would standardize and simplify development of aspect extensions, both general-purpose and domain-specific. Just as Eclipse integrates IDE tools and allows them to interact, the environment would enable compositions of AOP features and aspect extensions into multi-extension AOP languages.

Providing support for composing independent aspect extensions may have a dramatic effect on the future of AOP. An ability to compose new AOP features with the main stream aspect extensions can accelerate the evolution of aspect mechanisms by making research advances accessible for experimentation and use in realistic settings. Supporting the composition and use of newly developed aspect extensions together with established main stream ones can leverage and broaden their respective impact. Compositions of the main stream extensions with each other will unify their user bases. The ability to program in a multi-extension AOP language can also help compare features of the composed extensions. It can eliminate tradeoffs associated with choosing an extension with the most appropriate features.
But most importantly, the composition capability can help realize the vision of domain-specific aspect languages (aspect DSLs). A current trend in AOP is the usage of a single general-purpose aspect extension, such as AspectJ and AspectWerkz. An Eclipse-like environment for composition can reduce the dominance of general-purpose AOP languages by enabling practical use of aspect DSLs. Under this vision, the aspect DSLs are implemented by third-party providers as plugins to the environment. Language designers or AOP programmers can compose multiple plugins together with general-purpose aspect extensions, and use them collaboratively in the same program.

The main advantage of aspect DSLs over the general-purpose aspect extensions is the level of abstraction and simplicity. Domain-specific aspects are declarative and simple, while general-purpose aspects that implement the same functionality are imperative and sophisticated. In the AOP field these advantages outweigh the difficulty to learn how to program in multiple languages for two reasons. First, the majority of aspect DSLs are likely to share many similarities with the main stream general-purpose AOP languages (e.g., lingual mechanisms of advice selection and binding), and differ mainly in their domain terms and expressions (e.g., selfex). If that is the case, then it should not be difficult for the programmer to quickly learn new languages. Second, the simplicity of domain-specific code reduces the chances for unexpected aspect interactions, by shifting the complexity to the extension level. In an aspect program that is written in a general-purpose aspect extension, these interactions are the responsibility of the aspect programmer, while in the multi-extension AOP language, many of them are resolved by language designer at the aspect extension composition level.

This Ph.D. Thesis lays down the foundation for the envisioned environment by formulating, comprehensively analyzing, and presenting a practical solution to the problem of composing aspect extensions. The formulation of the problem is presented in Chapter 1. The chapter also introduces an approach to the problem and overviews main results of the thesis research. Chapter 2 and Chapter 3 study the context and the nature of the problem. Specifically, Chapter 2 analyzes state of the art in AOP and shows that the aspect extension composition problem
remains unaddressed in the current AOP tools, models, languages, and aspect extension composition methods. Chapter 3 motivates the need for simultaneous use of multiple aspect extensions and explains why it is difficult to compose them. The chapter shows that the difficulty of the problem is due to complex semantical interactions between the composed extensions.

The Thesis approaches the aspect extension composition problem by formulating and solving three subproblems:

- the aspect extension component abstraction problem: model an aspect extension in an abstract, precise and modular way;

- the aspect extension composition specification problem: derive a methodology for specifying semantics of a multi-extension AOP language;

- the aspect extension composition implementation problem: build a pluggable, customizable, and efficient framework for composing aspect extensions.

A solution to the first subproblem is presented in Chapter 4. The chapter introduces a top-down model of AOP that generalizes over aspect mechanisms in a novel way by recognizing concern integration rules as a unique feature of AOP. The model introduces a novel classification of aspect mechanisms as reactive or nonreactive, instead of more traditional classifications of AOP languages (i.e., static–dynamic or symmetric–asymmetric). The model formalizes weaving in an abstract way as the process of concern integration, independently of any specific aspect mechanism. The new formalism enables a detailed analysis of the aspect extension composition problem.

Chapter 5 presents a feature interaction approach to the composition specification problem. In this approach, a multi-extension composition is specified by identifying and resolving interactions between features of the composed extensions. The approach contributes a road map for resolving the feature interaction problem in compositions of aspect extensions. We analyzed existing aspect extensions and derived a set of abstract features and their patterns of interaction.
The patterns and features allow language designers to identify feature interactions in a large set of compositions.

A solution to the composition implementation problem is presented in Chapter 6. The chapter introduces AWESOME - a pluggable, customizable and efficient framework for composing aspect extensions. In AWESOME, an aspect extension is realized as a plugin to the compile-time aspect weaver. AWESOME is pluggable because it supports third-party composition of multiple aspect extension plugins into a multi-extension weaver with a default reasonable behavior. AWESOME is customizable because it provides means for customizing the default behavior to comply with a given composition specification. AWESOME is efficient because the runtime performance of compiled aspect programs is comparable to the performance of binaries that are produced by standard aspect compilers. The framework was tested and evaluated on real-world aspect languages. The evaluation strategies and results are presented in Chapter 7.

AWESOME is presented informally using design-level and implementation-level models of a weaver. Appendix A studies the aspect extension composition problem at the semantical level. It introduces Pluggable AOP - an interpreter-based framework for third-party composition of aspect mechanisms. The framework captures the essentials of the aspect extension composition process in AWESOME.

8.1 Summary of Contribution

This dissertation contributes to both theoretical and practical state of the art in AOP. The theoretical contribution is a novel taxonomy of AOP that organizes aspect mechanisms horizontally, as reactive and non-reactive, and vertically, from the intent of AOP down to its concrete implementations. The practical contribution is AWESOME, a prototyped compilation framework for composing aspect extensions.

The new AOP taxonomy comprises a set of AOP models at conceptual, specification, design, implementation, and semantical levels. The basis of the new AOP taxonomy is the top-
down model of AOP that is presented in Chapter 4. The model formally defines the basic AOP terms (e.g., aspect mechanism, weaving). It derives a high-level conceptual model of an aspect mechanism, and introduces a new reactive/non-reactive classification of aspect mechanisms. The feature interaction approach in Chapter 5 introduces an informal specification of an aspect mechanism as a feature model. The model explains an aspect mechanism as a set of features, and gives a high-level description of their behavior. Intuitively, the feature model explains the working of a conceptual top-down model and describes the internal processes of an aspect mechanism.

Chapter 6 presents the design and implementation-level models in the context of the AWESOME framework. The design model describes the principles for designing composable aspect weavers and specifies the interfaces between the weavers and the composition platform. The implementation-level model maps the design model to the concrete real-world AOP languages.

The semantical model in Appendix A formalizes the aspect mechanism design principles that enable automatic resolution of interactions in a multi-extension composition. The model formulates these principles as visibility of an aspectual effect and encapsulation of weaving. The semantical model maps the high-level conceptual model to the programming language domains.

The main practical contribution of the thesis work is the AWESOME framework. AWESOME is a framework for composing aspect extensions. The framework enables construction and composition of aspect extensions by:

1. guiding the design and the implementation of individual aspect extensions as weaver plugins;

2. providing a road map for specifying a multi-extension AOP language;

3. implementing the multi-extension language by composing the weaver plugins and customizing the constructed multi-weaver to cater for the specification; and by
(4) validating the constructed multi-weaver against the specification of the multi-extension AOP language.

The framework is demonstrated to work with real-world AOP languages; and can be applied to a wide range of extensions.

8.2 Benefits of Approach

The main benefit of the AWESOME composition framework is a plugin composability feature. The framework automatically resolves interactions between the composed plugins in a well-defined reasonable manner. A language designer may then customize the multi-weaver behavior by fine-tuning the default resolution, but does not necessarily needs to. The runtime performance of compiled aspect programs is practically unaffected by the extensible design of the framework, making AWESOME also useful in practice.

The framework also simplifies the creation of new extensions. The plugin development philosophy of AWESOME is “implementing as much as necessary, not more”. Writing a plugin in AWESOME is much simpler than writing a complete compiler.

Another practical benefit is a road map for resolving feature interactions in a multi-extension composition. The value of this road map is in providing users and tool developers with an appropriate abstraction for identifying feature interactions in a large set of compositions. The set of identified features introduces a vocabulary for documenting, understanding, and communicating designs of aspect extension composition frameworks. This road map is also useful for evaluating existing composition tools. This was illustrated by identifying feature interactions in AspectJ/5, Reflex, and XAspects.

8.3 Limitations of Approach

The composition approach is currently limited to the domain of join point and advice (J&A) extensions. This restriction, however, is a reasonable one and not fundamental. It is
reasonable because most of the existing aspect extensions are J&A extensions. Other more dis-
parate aspect-oriented extensions are either non-reactive aspect mechanisms, which are trivially
composable, e.g., Hyper/J [Ossher and Tarr, 2000, Tarr et al., 1999], or they are not “obliv-
ious” [Clifton and Leavens, 2003, Filman and Friedman, 2000, 2005] and can be composed
trivially, e.g., Demeter [Lieberherr and Lorenz, 2005, Lieberherr, 1996]. Aspect-oriented fea-
tures other than advising, such as introductions, are easier to compose because it is easier to
detect and resolve conflicts [Havinga et al., 2006]. It is possible to extend the approach to
support compositions of different kinds of aspect extensions.

Another limitation is due to the fixed set of feature interactions that the approach cur-
rently handles. Although this set is sufficient for composing most of the existing aspect ex-
tensions, future aspect extensions may introduce new features and thus new interactions. This
limitation, however, is not fundamental. The approach can be extended to accommodate a new
feature by identifying patterns of interactions between the new feature and the existing ones,
specifying a default resolution policy for the new interactions, and extending a composition
framework to handle the interactions appropriately.

Some limitations are specific to the AWESOME framework. AWESOME resolves feature
interactions at compile time. As such, the framework provides a restricted support for run-time
resolution of feature interactions. For example, AWESOME does not facilitate resolution of
foreign advising interactions that depends on run-time program values. This is not, however,
a limitation of the approach. A Pluggable AOP framework supports dynamic resolution of
interactions.

AWESOME uses Config aspects to customize the multi-weaver. Although this customiz-
ability mechanism is powerful, it breaks the encapsulation of the composed plugins, and re-
quires knowledge of their internal workings. Ideally, the customizability mechanism should
allow a language designer to configure different weaver plugins in a standard, non-invasive
way. Improving the customizability mechanism is an important direction for the future work.

AWESOME supports configurability at the language level, but does not allow the aspect
programmer to resolve program-level interactions among a concrete set of aspects. Although the language-level customizability is an essential feature of a composition framework, the program-level customizability is useful for specializing certain interactions for a particular program, e.g., for specifying aspect precedence. Although AWESOME does not provide program-level configurability, it can be extended to incorporate this feature. For example, supporting mutual inclusion or mutual exclusion interactions (Section 2.4.1) between aspects can be provided as a specialized multi-weaver matching strategy. In AWESOME, this strategy can be specified in the Config aspect as an advice to the match method of the platform. Extending AWESOME with aspect-level support for fine-tuning co-advising is a topic for future work.

8.4 Future Directions

Directions for the future work include enhancements of the AWESOME framework, design and construction of new domain-specific AOP languages, and new types of aspect mechanisms:

Improving AWESOME. An important enhancement to the AWESOME framework that would improve its usability include introduction of a non-intrusive composition customization mechanism, support for program-level and run-time interactions, extensibility of the customization and composability mechanisms to accommodate for new types of interactions.

Domain specific AOP languages. Certain application domains require concern separation capabilities that are not provided by general-purpose AOPLs. These kinds of crosscutting problems can be addressed by developing domain-specific AOPLs and by combining the developed domain-specific languages with the general-purpose AOPLs.

New aspect mechanisms. The top-down model of AOP can be used to identify new kinds of mechanisms, and to improve the existing ones. For example, non-reactive aspect
mechanisms [Kojarski and Lorenz, 2006] (e.g., Hyper/J) is an important research subject that is not understood well. Current studies of AOP focus mostly on reactive mechanisms (e.g., AspectJ). At the same time, the non-reactive mechanisms are known to be potentially better than reactive ones in disentangling functional system concerns.
Bibliography


Adrian Colyer. AOP@Work: Introducing AspectJ 5. developerWorks, July 12 2005a.

Adrian Colyer. AspectJ. In Filman et al. [2005], pages 123–143.


Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Filman et al. [2005], pages 21–35.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing. Addison-Wesley, 1995.


Wim Vanderperren, Davy Suváe, Bart Verheecke, María Agustina Cibrán, and Viviane Jonckers. Adaptive programming in JAsCo. In Proceedings of the 4th International Conference on


Appendix A

Publications
A.1 Pluggable AOP: Designing aspect mechanisms for third-party composition.
Pluggable AOP — Designing Aspect Mechanisms for Third-party Composition

Sergei Kojarski†
Northeastern University
Boston, Massachusetts 02115-5000, USA
kojarski@cs.virginia.edu

David H. Lorenz‡
University of Virginia
Charlottesville, Virginia 22904-4740, USA
lorenz@cs.virginia.edu

ABSTRACT
Studies of Aspect-Oriented Programming (AOP) usually focus on a language in which a specific aspect extension is integrated with a base language. Languages specified in this manner have a fixed, non-extensible AOP functionality. This paper argues the need for AOP to support the integration and use of multiple domain-specific aspect extensions together. We study the more general case of integrating a base language with a set of third-party aspect extensions for that language. We present a general mixin-based semantic framework for implementing dynamic aspect extensions in such a way that the independently developed aspect mechanisms can be subject to third-party composition and work collaboratively. Principles governing the design of a collaborative aspect mechanism are aspectual effect exposure and implementation hiding.

Categories and Subject Descriptors
D.2.10 [Software Engineering]: Design—Design concepts; D.1.5 [Programming Techniques]: Aspect-oriented Programming; D.3.1 [Programming Languages]: Formal Definitions and Theory; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages; D.2.12 [Software Engineering]: Interoperability

General Terms
Design, Languages, Theory

Keywords
AOP, AOSD, domain-specific aspect language, aspect extension, aspect mechanism, aspectual effect, granularity, software components, CBSE, third-party composition, collaboration, reuse, mixin, semantics, AspectJ, AspectWerkz, COOL.

1. INTRODUCTION
A current trend in Aspect-Oriented Programming (AOP [27]) is the usage of general-purpose AOP languages (AOPLs). However, a general-purpose AOPL lacks the expressiveness to tackle all cases of crosscutting. A solution to unanticipated crosscutting concerns is to create and combine different domain-specific aspect extensions to form new AOP functionality. As of yet, there is no methodology to facilitate this process [43].

Studies of AOP typically consider the semantics for an AOPL that integrates a certain aspect extension, Ext, with a base language, Base. For example, Ext1 might be a (simplified version of) AspectJ [26] and Base (a simplified version of) Java [1]. The semantics for the integration Base × Ext1 is achieved by amending the semantics for the base language. Given a pair of programs ⟨base, aspect1⟩ ∈ Base × Ext1, the amended semantics explains the meaning of base in the presence of aspect1.

Unfortunately, the semantics for the aspect extension and that for the base language become tangled in the process of integration. Consequently, it is difficult to reuse or combine aspect extensions. For each newly introduced aspect extension, say Ext2, the semantics for Base × Ext2 needs to be reworked. Moreover, given the semantics for Base × Ext1 and the semantics for Base × Ext2, the semantics for Base × Ext1 × Ext2 is undefined even though Ext1 and Ext2 are both aspect extensions to the same base language.

In this paper we resolve this difficulty by considering a more general open question:

THE ASPECT EXTENSION COMPOSITION QUESTION: Given a base language, Base, and a set {Ext1, . . . , Extn} of independent aspect extensions to Base, what is the meaning of a program base ∈ Base in the base language in the presence of n aspect programs ⟨aspect1, . . . , aspectn⟩ ∈ Ext1 × · · · × Extn, written in the n different aspect extensions?

Ability to compose distinct aspect extensions offers first and mostly great practical benefits. In Section 2 we discuss these benefits and illustrate that, unfortunately, prevalent implementations are not composable. Addressing the general composition question also provides in the special case where n = 1 a better encapsulation of the semantics for a single aspect extension.

1.1 Combining Two Aspect Extensions
Answering the aspect extension composition question is difficult even for n = 2. Let MyBase be a procedural language, and con-
sider two independent, third-party aspect extensions to MyBase. The first, HisExt1, capable of intercepting procedure calls and similar in flavor to AspectJ. The other, HerExt2, an aspect extension to MyBase capable of intercepting calls to the primitive division operator for catching a division by zero before it even happens (as opposed to catching a division by zero exception after it occurs), a capability that AspectJ lacks.1 Both call interception (e.g., [28]) and checking if a divisor is zero (e.g., [3, 29, 17]) are benchmarks often used in connection with aspects.

W.l.o.g., assume HisExt1 is created before HerExt2 is even conceived. If HisExt1 is to eventually work collaboratively with another aspect extension, e.g., HerExt2, the implementation of HisExt1 must take special care to expose its AOP effect, and only its effect, in terms of MyBase. This is because an aspect2 program written in HerExt2 would need to intercept divisions by zero not only in the base program base but also in advice introduced by an aspect1 program written in HisExt1.

Failing to reify a division by zero in aspect1 might cause a false-negative effect in HerExt2. Meanwhile, aspect2 must not intercept divisions by zero, if any, in the implementation of either HisExt1 or HerExt2. Reifying a division by zero in the implementation mechanism might cause a false-positive effect in HerExt2.

Similarly, aspect1 must intercept not only procedure calls in base but also any matching procedure call introduced by aspect2. aspect1 must not, however, intercept internal procedure calls that are a part of the implementation mechanism of either HisExt1 or HerExt2.

Note that generally aspect extensions present incompatible levels of AOP granularity [31]. In our example, aspect1 is not expressible in HerExt2, and aspect2 is not expressible in HisExt1. Therefore the problem of integrating the two cannot be reduced to translating aspect1 to HerExt2 or translating aspect2 to HisExt1 and using just one aspect extension. This distinguishes our objective from the purpose of frameworks (like XAspects [39]) that rely on the use of a general-purpose AOPL (like AspectJ).

In the sequel, a base mechanism denotes an implementation of a base language semantics, an aspect mechanism denotes an implementation of an aspect extension semantics, and a multi mechanism denotes an implementation of a multi-extension AOPL.

1.2 Objective and Contribution

We describe a general method for implementing the base mechanism and the aspect mechanisms in such a way that multiple, independent aspect mechanisms can be subject to third-party composition and work collaboratively. By third-party composition of aspect mechanisms we mean a semantical framework in which distinct aspect mechanisms can be assembled with the base mechanism into a meaningful multi mechanism without modifying the individual mechanisms. The mechanisms are said to be collaborative units of composition if the semantics of the composed multi mechanism can be derived from the semantics of the mechanisms that comprise it.

More precisely, let \( B \) denote the base mechanism for \( \text{Base} \). Let \( M_1, \ldots, M_n \) denote the aspect mechanisms for \( \text{Ext}_1, \ldots, \text{Ext}_n \), respectively. The aspect mechanism composition problem is to enable the third-party composition of \( M_1, \ldots, M_n \) with \( B \) into a multi mechanism \( A \), in a manner similar to the assembly of software components:

- **Units of independent production.** The aspect mechanisms \( M_1, \ldots, M_n \) are independently defined. The base mechanism \( B \) is defined independently from \( M_1, \ldots, M_n \). To enable the composition, \( M_1, \ldots, M_n \) rely only on \( B \) and have an explicit context dependency only on \( A \).
- **Units of composition.** The mechanisms are subject to third-party composition. The multi mechanism \( A \) for the combined AOP language is constructed (denoted by a \( \boxplus \) combinator) by composing the base mechanism with the aspect mechanisms without altering them: \( A = \boxplus (B, M_1, \ldots, M_n) \).
- **Units of collaboration.** The semantics for the composed multi mechanism \( A \) is the “sum” of the semantics provided by all the mechanisms.

Independence enables third-party development of individual aspect mechanisms; composability enables third-party composition of aspect mechanisms; and collaboration enables the desired behavior in the constructed AOP language.

Specifically, our approach enables third-party composition of dynamic aspect mechanism. We illustrate our solution for expression evaluation semantics. We model each aspect mechanism as a transformation function that revises the evaluation semantics for expressions.

1.3 Outline

In the rest of this paper, we demonstrate our solution to the aspect mechanism composition problem concretely through the implementation of interpreters. The next section motivates the need for composing multiple aspect extensions and demonstrates the lack of integration support in current aspect mechanisms. Section 3 presents a concrete instance of the problem: a base language MyBase with two aspect extensions, HisExt1 and HerExt2. We present their syntax and analyze a runnable programming example implemented in our framework. In Section 4 we present our approach for the general case of integrating \( n \) aspect mechanisms. In Section 5 we revisit the example shown in Section 3 and formally demonstrate our approach by constructing the semantics for MyBase, HisExt1, and HerExt2.

2. MOTIVATION

There is a growing need for the simultaneous use of multiple domain-specific aspect extensions. The need stems mainly from the favorable trade-offs that a domain-specific aspect extension can offer over a general-purpose AOPL:

- **Abstraction.** A general-purpose AOPL offers low-level abstractions for covering a wide range of crosscutting concerns. Domain-specific aspect extensions, in contrast, can offer abstractions more appropriate for the crosscutting cases in the domain at hand, letting the programmer concentrate on the problem, rather than on low-level details.

\(^2\)A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition [41].
• **Granularity.** The granularity of an aspect extension dictates all possible concern effect points within an application. Combining domain-specific aspect extensions allows to overcome the fixed granularity limitation of general-purpose AOPLs [31].

• **Expressiveness vs. Complexity.** The granularity of a general-purpose AOPL exposes a non-linear relationship between the language expressiveness and complexity. An increase in the language granularity significantly increases the language complexity while achieving a relatively small increase in expressiveness. Domain-specific aspect extensions, in contrast, can offer independent diverse ontologies [49].

The need also arises from the sheer abundance of available aspect extensions (and their evolving aspect libraries). For the Java programming language alone there are numerous aspect extensions that are being used in a variety of commercial and research projects. These include: AspectJ (ajc [10] and abc [2]), AspectWerkz [4], COOL [30], JBoss-AOP [25], JAsCo [44], Object Teams [20], Compose [51], to name just a few. ³ Ability to use these aspect extensions together will allow to reuse existing (and future) aspect libraries written for the different aspect extensions.

Unfortunately, little support is provided for the integration of distinct aspect mechanisms. Each aspect mechanism creates its own unique program representation which often excludes foreign aspects. Consequently, interaction between multiple aspect mechanisms operating on a single program can produce unexpected or incoherent results.

### 2.1 Example
Consider a bounded buffer example implemented in Java (Listing 1). Suppose you have three aspect extensions at your disposal:

- **COOL** [30]—a domain-specific aspect extension for expressing coordination of threads;
- **AspectWerkz** [4]—a general-purpose lightweight AOP framework for Java;
- **AspectJ**—a general-purpose aspect extension;

and two concerns to address, namely, a synchronization concern and a tracing concern (e.g., logging or auditing).

#### 2.1.1 COOL versus AspectJ
The synchronization concern can be expressed as a coordinator aspect in COOL (e.g., Listing 2) or alternatively as an aspect in AspectJ (e.g., Listing 3).

The COOL aspect (Listing 2) provides an elegant declarative description of the desired synchronization. The `mutex` exclusion set `{add, remove}` specifies that `add` may not be executed by a thread while `remove` is being executed by a different thread, and vice versa. In addition, the `selfexec` exclusion set prohibits different threads from simultaneously executing either `add` or `remove`.⁴

³For a complete list of commercial and research aspect extensions see [http://www.aosd.net/technology/](http://www.aosd.net/technology/)

⁴However, the same thread is not prohibited from entering both `add` and `remove`.

#### 2.1.2 AspectWerkz + AspectJ
Semantically, the underlying mechanisms of AspectWerkz and AspectJ are essentially equivalent. Yet, their syntactical differences present programmers with a useful choice of alternatives. Recently it was announced that AspectWerkz has joined the AspectJ project.
to bring the key features of AspectWerkz to the AspectJ 5 platform [5]. This merger will allow aspects like those in Listing 4 and Listing 5 to run side by side.

Listing 4 is a simple logging aspect in AspectWerkz. The annotation @Aspect("perJVM") specifies that the AWLogger class is actually a singleton aspect. The annotation @Before call(* .*(...) && !cflow(within(AWLogger))) specifies that the log method is to be called for every method call not in the dynamic control flow of methods in AWLogger.

Listing 5 is an auditing aspect in AspectJ. The toLog() pointcut specifies that every method call should be recorded. The before, after() returning, and after() throwing advice add log messages to the buffer.

Arguably, if AspectWerkz and AspectJ were designed to be composable third-party aspect mechanisms, building AspectJ 5 would have been much easier. Moreover, third-party composition of aspect mechanisms would have made other domain-specific combinations possible, like combining COOL with AspectWerkz and AspectJ.

2.2 Lack of Integration Support
Unfortunately, current aspect mechanisms fail to compose correctly. We demonstrate this failure on the bounded buffer example for two commonly used approaches:

- **Translation.** Aspect programs in different aspect extensions can be translated to a common target aspect extension.
- **Instrumentation.** Aspect mechanisms can be implemented by means of program instrumentation. Such multiple independent aspect mechanisms can be trivially composed by passing the output of one aspect mechanism as the input to another aspect mechanism.

2.2.1 No Behavior-Preserving Translation
The translation approach requires the expressiveness of the target aspect extension to support arbitrary granularity. Even when granularity does not pose a problem, a translation from one aspect language to another will not generally preserve the behavior of the
source aspect program in the presence of other aspects. Consider the synchronization concern implementation in COOL (Listing 2). Translating it to AspectJ (Listing 3) results in an aspect that seems to be a correct substitution for the COOL coordination aspect, but in the presence of the auditing aspect (Listing 5) is actually not.

A property of the COOL synchronization concern is transparency with respect to the AspectJ auditing concerns. There should not be any interference between the two. The COOL aspect does not contain any join points that should be visible to the AspectJ mechanism. This property is not preserved in the translation. Calls to wait (Listing 3, lines 13 and 33) and notifyAll (Listing 3, lines 20 and 41), which do not exist in the COOL code, will nonetheless be unexpectedly reflected by the auditor.\(^5\)

Worse yet, the unexpected join points in the target program may break existing invariants, resulting in our case in a deadlock. An implicit invariant of the COOL aspect is that if both add and remove are not currently executed by some other thread, then the thread can enter and execute them. The AspectJ synchronization aspect, however, violates this invariant. Assume that two threads concurrently access the buffer. The first thread acquires the lock, while the second invokes wait on the BufferSyncAspect object. However, before wait is invoked, the AJAuditor aspect calls BoundedBuffer.\(^6\)add (Listing 5, line 19). The latter call causes the second thread to enter the guarded code again and trigger a second call to wait.\(^6\) Since the second wait call is in the cflow of the auditor, it is not advised, and the thread finally suspends. When the first thread releases the lock, the second thread wakes up after the second wait. It acquires the lock, completes the advice execution, releases the lock, and proceeds to the first wait invocation. At this point, the buffer is not locked; the second thread waits on the BufferSyncAspect object monitor; and if no other thread ever accesses the buffer, the second thread waits for ever—deadlock!

2.2.2 No Correct Sequential Instrumentation
One would expect the two aspects written in AspectWerkz (Listing 4) and AspectJ (Listing 5) to interact as if they were two aspects written in a single aspect extension (e.g., the future AspectJ 5 platform). On the one hand, the AspectJ auditor should log all method calls within the AWLogger aspect. On the other hand, the AspectWerkz logger should log all method calls within AJAuditor. (And both should log all method calls in the base program as well.)

However, applying the AspectJ and AspectWerkz instrumentation mechanisms sequentially, in any order, produces an unexpected result. The mechanism that is run first may not be able to interpret the second extension’s aspect program. Specifically, the AspectWerkz mechanism does not understand AspectJ’s syntax. It can be applied to the bounded buffer code but not to the AJAuditor aspect. Thus, when AspectWerkz is run first, some expected log messages will be missing.

The mechanism that is run last logs method calls that are not supposed to be logged. For example, when AspectWerkz is run second, the following unexpected log message is generated by the AWLogger aspect:

```
AW:public void AJAuditor.ajc$afterReturning$-JAAuditor$2$ba1fbd8a(org.aspectj.lang.JoinPoint)
```

\(^5\)Note that calls to wait and notifyAll cannot be avoided.\(^6\)Assuming that the first thread still owns the lock.

3. PROBLEM INSTANCE
We now return to MyBase, HisExt, and HerExt in order to analyze the problem and illustrate our approach concretely. After a brief introduction to the syntax, we informally explain MyBase, HisExt, and HerExt through a programming example. The code fragments are actual running code in our implementation, and their semantics is formally presented in Section 5.

3.1 Syntax

3.1.1 MyBase Syntax
The syntax of MyBase is given in Figure 1. MyBase is a procedural language. Procedures are mutually-recursive with call-by-value semantics. The set of procedures is immutable at run-time. Expressed values are either booleans or numbers (but not procedures). The execution of a program starts by evaluating the body of a procedure named `main`.

3.1.2 HisExt Syntax
The syntax for HisExt is given in Figure 2. HisExt is a simple AspectJ-like aspect extension to MyBase. HisExt allows one to impose advice around procedure calls and procedure executions. Advice code is declared in a manner similar to procedures. Like in AspectJ, the set of advice is immutable at run-time. Each advice has two parts: a pointcut designer and an advice body expression. Atomic pointcuts are `pcall-pcd`, `pexecution-pcd`, `cflow-pcd`, and `args-pcd`. The and-pcd and or-pcd allows one to combine several pointcuts under conjunction and disjunction, respectively. Unlike AspectJ, `around` is the only advice kind in HisExt. There is no support for patterns in pointcut designators.

HisExt introduces a new `proceed-exp` expression. The notation:

```
|proceed-exp| Exps := ... |proceed-exp
```

redefines Exps within the aspect extension syntax only, without propagating the change to the syntax of MyBase expressions. In particular, `proceed-exp` expressions are valid only within a HisExt advice-body expression.

3.1.3 HerExt Syntax
HerExt allows one to declare a set of exception handlers in MyBase for catching and handling division by zero before an exception occurs. Advice code in HerExt specifies an exception handler expression. A guard clause allows one to specify a dynamic scope for the handler. HerExt introduces a new expression, namely `raise-exp`, which is allowed within a handler. It passes the exception handling to the next handler (in a manner, similar to `proceed-exp` of HisExt). The syntax of the language is given in Figure 3.

The semantics for HerExt is straightforward. Whenever the second argument to the division primitive evaluates to zero, the advice handler (if one exits) is invoked. The handler is evaluated and the result value substitutes the offending zero in the second argument to the division primitive, and the program execution resumes.

Listing 8 shows an aspect we can write in HerExt. This aspect resumes the execution with the value of `Precision(1)` whenever the second argument of a division primitive evaluates to 0 within the control flow of the `SORT` procedure.
Program ::= Declaration
Declaration ::= "program" "procedure" PName "Id" "f" Exps
Procedure ::= "procedure" PName "Id" "f" Exps
Exps ::= lit-exp | true-exp | false-exp | var-exp | app-exp | begin-exp | if-exp | assign-exp | let-exp | primapp-exp
lit-exp ::= Number
true-exp ::= "true"
false-exp ::= "false"
var-exp ::= Id
app-exp ::= "call" PName "Id" "f" Exps
begin-exp ::= "f" Exps "procedure call"
if-exp ::= "if" Exps "then" Exps "else" Exps
assign-exp ::= "let" Id "=" Exps
let-exp ::= "let" Id "f" Exps "in" Exps
primapp-exp ::= Prim "call" Exps

Fig. 1: MyBase syntax

AOP1-Program ::= AOP1-Declaration
AOP1-Declaration ::= "aop1" "procedure" PName "Id"
Advice ::= "around" "f" Pointcut Exps
Pointcut ::= call-pcd | exec-pcd | cflow-pcd | args-pcd | and-pcd | or-pcd
call-pcd ::= "pcall" PName "call-pcd"
exec-pcd ::= "pexecution" PName "exec-pcd"
cflow-pcd ::= "cflow" PName "cflow-pcd"
args-pcd ::= "args" Id "args-pcd"
and-pcd ::= "and" PName "and-pcd"
or-pcd ::= "or" PName "or-pcd"
Exps ::= Exps | "proceed-exp"
proceed-exp ::= "proceed"

Fig. 2: HisExt1 syntax

AOP2-Program ::= AOP2-Declaration
AOP2-Declaration ::= "aop2" "procedure" PName "Id"
Handler ::= "guard" PName "resume" Exps
Exps ::= Exps | "raise-exp"
raise-exp ::= "raise"

Fig. 3: HerExt2 syntax
3.2 A Programming Example

The semantics for the base procedural language MyBase and the aspect extensions HisExt, and HerExt, are implemented as interpreters [18]. The example presented here is a simple executable arithmetic program in MyBase for computing the square root of a given number. While simple, the example is representative in terms of illustrating the complexity of achieving collaboration among aspect extensions, and its semantics serves as a proof of concept.

The procedure SQRT in Listing 6 implements in MyBase a simple approximation algorithm using a sequence generated by a recurrence relation:

\[ a_0 = \text{approximation}; \text{repeat} \quad a_n = f(a_{n-1}) \text{ until} \quad \text{precise} \]

By default, it sets \( a_0 = 0 \), and calls SqrtIter to generate the recurrence sequence:

\[ a_n = a_{n-1} + \epsilon \]

until \( (a_n)^2 > x \). The procedure Improve generates the next element in the sequence; IsPreciseEnough? checks the termination condition; and the value \( \epsilon = \epsilon(x) \) is computed as a function of \( x \) by the procedure Precision.

The resulted computation of \( \sqrt{x} \) is inaccurate and extremely inefficient. However, it serves our purpose well. We will non-intrusively improve its efficiency using an aspect in HisExt. We will correct its behavior for the singular point \( x = 0 \) using HerExt.

The code in Listing 7, written in HisExt, advises the base code for drastically improving its efficiency and accuracy. Four pieces of advice are used. The first around advice (lines 202–204) intercepts executions of the procedure Improve and instead applies Newton’s method:

\[ a_{n+1} = \frac{1}{2} \left( a_n + \frac{x}{a_n} \right) \]

The second around advice (lines 205–207) intercepts IsPreciseEnough? executions and checks instead whether or not

\[ |(a_n)^2 - x| < \epsilon \]

where \( \epsilon = \frac{1}{1000} \) is set in the third around advice (lines 208–211). The successive approximations now converge quadratically.

Running main and calling

```
call SQRT(5)
```

returns 7

```
(num-val 161/72)
```

meaning \( \frac{161}{72} = 2.2361111 = \sqrt{5.0001929} \approx \sqrt{5} \).

The improved program works well for all non-negative inputs to SQRT, except for when the radicand is 0. In this case, Improve is called with the first argument \( a_0 \) set to 0. The execution of Improve triggers the advice around Improve execution which divides \( x \) by \( a_n \). Since the value of \( a_n \) is 0 an exception occurs.

3.3 Third-party Composition

The main point of this example is that HisExt, and HerExt are subject to third-party composition with MyBase and work collaboratively:

- **Units of independent production.** HisExt and HerExt are independently constructed.

- **Units of composition.** MyBase, HisExt, and HerExt are units of composition. MyBase can be used by itself (running only Listing 6). MyBase can be used with HisExt alone (omitting Listing 8). MyBase can be used with HerExt alone (omitting Listing 7). MyBase can be used with both HisExt and HerExt.

The result shown is the actual value returned by the Scheme [37] implementation.

• **Units of collaboration.** When HisExt\(_1\) and HerExt\(_2\) are both used they collaborate. In the absence of HerExt\(_2\), calling

\[
\text{call SQRT}(0)
\]

results in

\text{Error in } /: \text{undefined for } 0.

However, when HerExt\(_2\) with the advice code in Listing 8 are present, the correct value 0 is returned. The violating primitive division application is introduced by the advice of HisExt\(_1\), yet intercepted by the advice of HerExt\(_2\). This desired behavior is non-trivial because HisExt\(_1\) was constructed without any prior knowledge of HerExt\(_2\).

3.4 Analysis

In order to achieve a correct collaboration:

• The aspectual effect of all extension programs needs to be exposed to all the collaborating aspect mechanisms.

• Each individual aspect mechanism must hide its implementation from other aspect mechanisms.

3.4.1 Effect Exposure

In the context of multiple distinct aspect mechanisms, certain elements of the aspect program should be exposed to all collaborating aspect mechanisms. We call these elements the *aspectual effect*. The aspectual effect of an aspect program generally specifies the implementation of a crosscutting concern. We assume that the aspectual effect is expressed in the base language.

In our example, the aspectual effect of an aspect \(_1 \in \text{HisExt}\(_1\)\) is specified by advice-body expressions; the aspectual effect of an aspect \(_2 \in \text{HerExt}\(_2\)\) is specified by handler expressions. When HisExt\(_1\) and HerExt\(_2\) are composed together, their mechanisms must reflect each other’s effect. Specifically, HisExt\(_1\) aspects must be able to advise procedure calls made from the HerExt\(_2\) handler expressions; and HerExt\(_2\) handlers must be able to intercept exceptions introduced by the HisExt\(_1\) pieces of advice.

3.4.2 Implementation Hiding

An aspect extension extends the base language with new functionality. For example, HisExt\(_1\) adds advice binding, and HerExt\(_2\) adds exception handling to the base language. An aspect mechanism that implements the new functionality must hide its internal operations from the other aspect mechanisms. In our example, pointcut matching and advice selection operations of the HisExt\(_1\) mechanism must be hidden from the HerExt\(_2\) mechanism. Conversely, testing whether the second division primitive argument evaluates to zero and the exception handler selection of HerExt\(_2\) should be invisible to the HisExt\(_1\) mechanism.

4. **OUR APPROACH**

Now that we have illustrated a desired behavior, we explain our solution to the aspect mechanism composition problem in general.

4.1 Aspect Mechanisms as Mixins

The primary idea is to view an aspect mechanism that extends a base mechanism as a *mixin* [11] that is applied to the base mechanism description. A description of a mechanism is an encoding of its implementation (e.g., a configuration of an abstract machine or its semantics). An *aspect mixin mechanism* transforms some of the base mechanism description and introduces some additional description.\(^8\)

By keeping a clean separation between the descriptions of the base aspect mechanisms, the aspect mixin mechanism may be composed with other mechanisms that extend the same base language. The particular composition strategy may differ. In the next section, we show a concrete instance of this general approach.

4.2 Solution Instance

We illustrate the approach specifically for expression evaluation semantics. In our solution, the base mechanism \(B\) and the aspect mechanisms \(M_1, \ldots, M_n\) compose into an AOP interpreter \(A\) (Figure 4). When the set of aspect mechanisms is empty, \(A\) behaves as a base interpreter (Figure 4(a)).

Each aspect mechanism \(M_i\) is *designed* as a wrapper around the base mechanism (Figure 4(b)). In the composition, \(M_i\) overrides the base functionality gracefully: the mechanism delegates all base operations to \(B\); it implements only its respective aspectual functionality.

To build a multi mechanism \(A\), the aspect mechanisms are subject to third-party composition (Figure 4(c)). The composed mechanisms are organized in a chain-of-responsibility [19], pipe-and-filter architecture [38]. Each aspect mechanism performs some part of the evaluation and forwards other parts of the evaluation to the next mechanism using delegation semantics (“super”-like calls) [6]. If an expression is delegated by all mechanisms then it is eventually evaluated in \(B\). All the mechanisms defer to \(A\) for the evaluation of recursive and other “self”-calls.

\(^8\)We generally assume that a granularity requirement of an aspect mechanism can always be satisfied by either taking the most fine-grained description form (e.g., small-step operational semantics), or refining the current description (e.g., via annotations).
A subtlety in designing a collaborative aspect mechanism is deciding what to hide, what to delegate, and what to expose. A mechanism may hide its effect by reducing an expression internally. A mechanism may expose its effect by evaluating expressions in \( A \). The latter allows what is known as “weaving.” The exposed expressions are then evaluated collaboratively by all the mechanisms. As a result, an effect of an aspect mechanism is made visible to all the other mechanisms. Hence, the mechanisms reflect one another’s effect. Overall, a mechanism is considered a collaborative unit provided it properly hides, delegates, or exposes the evaluation.

**Notation.** The following notations are pertinent. We express functions in Curried form. The Curried function definition

\[
\text{fn } \text{pat}_1 \text{ pat}_2 \ldots \text{ pat}_n \Rightarrow \exp
\]

is the same as the lambda expression \( \lambda \text{pat}_1. \lambda \text{pat}_2 \ldots. \lambda \text{pat}_n. \exp \). Correspondingly, we write a list of function arguments with no parentheses or commas to express a function application that takes the first argument as its single parameter, which could be a tuple, constructs and returns a new function, which then takes the next argument as its single parameter, and so on. In function types, ‘\(-\)’ associates to the right.

We use the form \( (\text{id as pat}) \) in a formal argument to bind an identifier \( \text{id} \) to a value and match the value with a pattern \( \text{pat} \). Variables in the pattern bind to their corresponding values. We use \( \text{val pat} = \text{val} \) to split apart a value. The symbol ‘\(-\)’ stands for an anonymous variable (don’t care). The symbol ‘\(o\)’ denotes an empty mapping and ‘\([]\)’ denotes an empty list.

### 4.2.1 Expressions

The Base grammar introduces a set of expression productions; \( \text{Exp}_0 \) is the set of base expressions whose pattern matches one of these productions. Each of the extensions \( \text{Ext}_1, \text{Ext}_2, \ldots, \text{Ext}_n \) may extend \( \text{Base} \) with its own respective set of additional expressions \( \text{Exp}_1, \text{Exp}_2, \ldots, \text{Exp}_n \). The set \( \text{Exp}_A \) of AOP expressions is hence a union of pairwise disjointed expression sets defined by:

\[
\text{Exp}_A = \text{Exp}_0 + \text{Exp}_1 + \text{Exp}_2 + \cdots + \text{Exp}_n
\]

Note that in an extended grammar, an expression in \( \text{Exp}_0 \) may contain subexpressions not in \( \text{Exp}_0 \). For example, in the case of \( \text{MyBase} \) and \( \text{HisExt}_1 \), the expression

\[
* (2, \text{proceed})
\]

is a base expression (because its pattern matches the production \( \text{primapp}=\text{exp} \) in \( \text{MyBase} \)) but \( \text{proceed} \) is not.

### 4.2.2 Overall Semantics

Let \( A[\exp] \) denote the meaning of an AOP expression \( \exp \). Our goal is to be able to build the multi mechanism \( A \) by composing the base mechanism \( B \) and the mutually independent aspect mechanisms \( \mu_1, \ldots, \mu_n \). We use the term AOP configuration to denote the state of a multi mechanism \( A \). An AOP configuration \( cf_g \in Cfg_A \) is a vector of configurations of the composed mechanisms:

\[
C_{\mu_i} \times C_{\mu_1} \times C_{\mu_2} \times \cdots \times C_{\mu_n}
\]

We assume that \( \text{Exp}_i \cap \text{Exp}_j = \emptyset \) for all \( 0 \leq i < j \leq n \). where \( \text{Cfg}_i \) denotes a domain of the base mechanism states, and \( \text{Cfg}_i, 1 \leq i \leq n \), denotes a domain of the aspect mechanism \( \mu_i \) states. For example, a \( \text{MyBase} \) mechanism configuration comprises a procedure environment, a variable environment, and a store. A \( \text{HisExt}_1 \) mechanism configuration comprises a list of advice, a “current” join point, and a “current” proceed computation.

The effect of evaluating an expression \( \exp \in \text{Exp}_A \) is to change the AOP configuration. The meaning of an expression \( \exp \in \text{Exp}_A \), denoted \( A[\exp] \), is defined to be a partial function on configurations:

\[
A : \text{Exp}_A \to (\text{Cfg}_A \leftarrow \text{Cfg}_A)
\]

We denote by \( \text{Cont}_A \) the set of partial functions on \( \text{Cfg}_A \).

#### 4.2.3 Design Guidelines for the Base Mechanism

\( B \) provides semantics for expressions in \( \text{Base} \). The meaning of an expression \( \exp \in \text{Exp}_0 \) in \( \text{Base} \), denoted \( B[\exp] \), is expected to be defined as:

\[
B : \text{Exp}_0 \to \text{Cont}_A
\]

The semantical function \( B \) should adhere to the following design principles:

- All sub-reductions within a \( B \)-reduction are reduced by calling \( A \) instead of \( B \).
- \( B \) only accesses and updates the head \( \text{Cfg}_0 \)-element of the \( cf_g \in \text{Cfg}_A \) configuration list, and carries the tail through the computation.

Note that the fact that \( \text{Cfg}_0 \) does not mean that \( A \) or \( n \) are known at the time of writing \( B \). At the time of writing the base mechanism, \( A \) is assumed to delegate everything to \( B \):

\[
A[\exp] = \begin{cases} \emptyset & \text{if } \exp \in \text{Exp}_0 \\ \exp & \text{otherwise} \end{cases}
\]

where \( \emptyset \) stands for “undefined.” Let \( \tilde{B} : \text{Exp}_0 \rightarrow \text{Cfg}_0 \rightarrow \text{Cfg}_0 \) denote the evaluation semantics for \( \text{Base} \) with its standard signature. \( B \) has a different signature than \( \tilde{B} \) but the same behavior as \( \tilde{B} \).

\[
\forall \exp \in \text{Exp}_0 \forall cf_g = \text{Cfg}_0 . \forall cf_g^* \in \text{Cfg}_A : \tilde{B}[\exp] cf_g = \begin{cases} cf_g^* & \text{if } cf_g = cf_g^* \\ \tilde{B}[\exp] cf_g & \text{if } \text{dom } h \end{cases}
\]

#### 4.2.4 Design Guidelines for an Aspect Mechanism

We construct the aspect mechanism \( \mu_i \), for an aspect extension \( \text{Ext}_i \), as the override combination\(^{\text{10}}\) of a semantics transformer \( T_i \) and a semantical function \( \tilde{E}_i \):

\[
\text{val } \mu_i = \begin{cases} \text{eval} & (T_i \text{ eval}) \oplus \tilde{E}_i \end{cases}
\]

Semantics for the \( \text{Ext}_i \)’s newly introduced expressions \( \text{Exp}_i \) is defined by:

\[
\tilde{E}_i : \text{Exp}_i \to \text{Cont}_A
\]

\(^{\text{10}}\)For two partial functions \( g \) and \( h \), their override combination \( g \oplus h \) (\( h \) overrides \( g \)), is defined by:

\[
(g \oplus h)(x) = \begin{cases} h(x) & x \in \text{dom } h \\ g(x) & \text{otherwise} \end{cases}
\]
The introduction of Ext into the base language also requires a change to the evaluation semantics for a non-empty subset of the existing base language expressions \( \text{Exp}_0 \subseteq \text{Exp}_b \). We define this part of the semantics for Ext, as a language semantics transformer:

\[
T_i : \langle \text{Exp}_b \rightarrow \text{Cont}_A \rangle \rightarrow \langle \text{Exp}_0 \rightarrow \text{Cont}_A \rangle
\]

The semantics transformer \( T_i \) should adhere to the following design principles:

- \( T_i \) defines the semantics for Ext, and nothing more. Let \( B' \) denote a semantical function with the same signature as \( B \) or an extended signature.\(^{12} \) \( T_i(B') \) delegates the evaluation to \( B' \) whenever the base language semantics is required.
- \( T_i(B') \) accesses only the \( \text{Cfg}_{b,0} \) and \( \text{Cfg}_{b,1} \)-elements in a \( \text{cfg} \in \text{Cfg}_{A,0} \) configuration, while the rest are carried through the computation.

Note that allowing the aspect mechanism access to the \( \text{Cfg}_{b,0} \) element is needed for modeling interesting cases of aspect mechanism interactions.

### 4.2.5 Third-party Construction of an AOP Language

Let \( K = \{ k_i \}_{i=1}^l \) be an ordered index set, and let \( m_{k_1}, \ldots, m_{k_l} \) denote the \( l \leq n \) aspect mechanisms to be composed. Let \( B \) denote the Base mechanism, and let \( A^K \) denote the multi mechanism being constructed.

We construct the multi mechanism \( A^K \) as the composition:

\[
A^K = A_l \oplus (B; m_{k_1}, \ldots, m_{k_l})
\]

where the composition semantics for \( \oplus \) is defined as follows. The meaning of \( \text{exp} \in \text{Exp}_A \), denoted \( A_{[\text{exp}]} \), is given by the recurrence relation:

\[
A_0 = B
A_i = A_{i-1} \oplus (m_{k_i} A_{i-1})
\]

By construction,

\[
A_l : \langle \text{Exp}_b + \text{Exp}_{k_1} + \cdots + \text{Exp}_{k_l} \rangle \rightarrow \text{Cont}_A
\]

is of the right signature and obeys the composition principle. To illustrate the construction, we conclude by elaborating the first three instances:

- For \( l = 0 \), we have that \( \text{Exp}_A = \text{Exp}_0 \), and the meaning of \( \text{exp} \in \text{Exp}_A \) is the same as the meaning of \( \text{exp} \) in Base:

\[
A^0 : \text{Exp}_0 \rightarrow \text{Cont}_A
A^0[\text{exp}] = \text{eval}_0[\text{exp}]
\]

- For \( l = 1 \) and the singleton index set \( \{ i \} \) for some \( 1 \leq i \leq n \), we have that \( \text{Exp}_A = \text{Exp}_0 + \text{Exp}_i \). The meaning of \( \text{exp} \in \text{Exp}_A \) is

\[
A^1 : \langle \text{Exp}_0 + \text{Exp}_i \rangle \rightarrow \text{Cont}_A
\]

\[\text{Eval}_0, \text{Eval}_i\]

We construct:

\[
A^{(i)} = B \oplus (T_i B) \oplus E_i
\]

\[
A^{(i)}[\text{exp}] = \begin{cases} E_i[\text{exp}] & \text{exp} \in \text{Exp}_i \\ (T_i B)[\text{exp}] & \text{exp} \in \text{Exp}_0 \\ B[\text{exp}] & \text{otherwise} \end{cases}
\]

- For \( l = 2 \) and the ordered index set \( \{ i, j \} \) for some \( 1 \leq i, j \leq n \), we have that \( \text{Exp}_A = \text{Exp}_0 + \text{Exp}_i + \text{Exp}_j \) (Figure 5). The meaning of \( \text{exp} \in \text{Exp}_A \) is

\[
A^{(i,j)} : \langle \text{Exp}_0 + \text{Exp}_i + \text{Exp}_j \rangle \rightarrow \text{Cont}_A
\]

We construct:

\[
A^{(i,j)} = A^{(i)} \oplus (T_j A^{(i)}) \oplus E_j
\]

\[
A^{(i,j)}[\text{exp}] = \begin{cases} E_j[\text{exp}] & \text{exp} \in \text{Exp}_j \\ E_i[\text{exp}] & \text{exp} \in \text{Exp}_i \\ (T_j B)[\text{exp}] & \text{exp} \in \text{Exp}_0 - \text{Exp}_0 \\ (T_i B)[\text{exp}] & \text{exp} \in \text{Exp}_0 - \text{Exp}_0 \\ (T_j (T_i B))[\text{exp}] & \text{exp} \in \text{Exp}_0 \cap \text{Exp}_0 \\ B[\text{exp}] & \text{otherwise} \end{cases}
\]

## 5. IMPLEMENTATION

As a proof of concept we have implemented MyBase, HisExt1, and HerExt2 for the example presented in Section 3. This section provides the implementation details more formally to the so-inclined reader.

### 5.1 Base Mechanism Implementation

The domain \( \text{Exp}_A \) of AOP expressions includes MyBase, HisExt1, and HerExt2 expressions. We define \( \text{Exp}_b \) by extending the MyBase expression grammar \( \text{Exp}_b \) (Figure 1) with a set of annotated expression \( \text{annotated-exp} \) (Figure 6).

\[
\text{Exp}_b = \text{Exp}_b + \text{annotated-exp}
\]

Annotated expressions extend the interface of the base mechanism to satisfy granularity needs of the HisExt1 and HerExt2 mechanisms. In the extended grammar, a complex expression (Figure 7) includes annotated expressions as subexpressions.

The base configuration domain \( \text{Cfg}_{b,0} \) consists of a procedure environment domain \( \text{Env}_P \), a variable environment domain \( \text{Env}_V \), and
The evaluation semantics $B$ (Figure 9) for $\text{Exp}_i$ expressions satisfies the design principles for the base mechanisms: (1) all expression evaluations in $B$ are exposed to $A$ (highlighted in the figure); (2) it accesses and updates only the $\text{Cf}_{g_0}$-element of the configuration; (3) the other configurations are carried through the computation.

5.2 Aspect Mechanism Implementation

The semantics for $\text{Ext}_i$ is specified using three constructor functions:

- **build-$E_i$**, constructs an evaluator for $\text{Exp}_i$ expressions:

  $\text{build}-E_i : \text{Int} \rightarrow (\text{Exp}_i \rightarrow \text{Cont}_A)$

- **build-$T_i$**, constructs the semantics transformer for the $\text{Ext}_i$:

  $\text{build}-T_i : \text{Int} \rightarrow ((\text{Exp}_0 \rightarrow \text{Cont}_A) \rightarrow (\text{Exp}_0 \rightarrow \text{Cont}_A))$

Intuitively, the aspect mechanisms are implemented as mixins to the base mechanism (Figure 10). The $\text{HisExt}_1$ mechanism $\mu_1$ redefines the semantics for $\text{app-exp}$ and $\text{procbody-exp}$ and introduces semantics for $\text{proceed-exp}$. The $\text{HerExt}_2$ mechanism $\nu_2$ redefines $\text{primarg-exp}$ and $\text{procbody-exp}$ and introduces semantics for $\text{raise-exp}$.
\[ \begin{align*}
\text{exp} & \in \text{Exp}_{\text{adv}} \quad \Rightarrow \quad \text{exp} = \text{Exp}_0 + \text{Exp}_1 & \text{Advice exps} \\
\text{cfg}_1 & \in \text{Cfg}_1 \quad \Rightarrow \quad \text{cfg}_1 = \text{Adv} \times \text{JP} \times \text{Cont}_A & \text{Configuration} \\
\text{adv} & \in \text{Adv} \quad \Rightarrow \quad \text{adv} = \text{PCD} \times \text{Exp}_{\text{adv}} & \text{Advice} \\
\text{jp} & \in \text{JP} \quad \Rightarrow \quad \text{jp} = \{\text{call}, \text{exec}\} \times \text{PNm} & \text{Join points} \\
\text{pcd} & \in \text{PCD} \quad \Rightarrow \quad \text{pcd} = \text{Var} \times \text{Val} \times \text{JP + Unit} & \text{Pointcuts} \\
\text{effect} & \in \text{Effect} \quad \Rightarrow \quad \text{effect} = \text{Bnd} \times \text{Exp}_{\text{adv}} & \text{Effects} \\
\text{bnd} & \in \text{Bnd} \quad \Rightarrow \quad \text{bnd} = \text{Var} \times \text{Val} & \text{Binding}
\end{align*} \]

Figure 11: HisExt\(_1\) domains

5.2.1 HisExt\(_1\) Mechanism

The aspect mechanism \(\mathcal{M}_1\) transforms the semantics for procedure calls and executions, and supplies semantics for \(\text{Exp}_1\)’s new proceed expression:

\[ \begin{align*}
\text{Exp}_0^1 &= \{\text{app-exp, probody-exp}\} \\
\text{Exp}_1 &= \{\text{proceed-exp}\}
\end{align*} \]

A configuration \(\text{cfg}_1 \in \text{Cfg}_1\) for HisExt\(_1\) (Figure 11) comprises a set of advice, a “current” join point, and a “current” proceed continuation. An advice \(\text{adv} \in \text{Adv}\) is derived directly from HisExt\(_1\)’s syntax. A join point \(\text{jp} \in \text{JP}\) is an abstraction of the procedure call stack. It stores the name and formal and actual arguments of a corresponding procedure. The third element provides a meaning for proceed expressions. The effect and binding domains are internal to the mechanism. An effect carries a set of bindings and an advice body expression. The bindings provide an appropriate variable environment for evaluating the advice body expression.

The interesting part of the aspect mechanism \(\mathcal{M}_1\) implementation is given by \texttt{build-}\(\text{T}_1\) (Figure 12). \texttt{build-}\(\text{T}_1\) defines a transformer of the semantics for procedure calls and procedure executions. The new semantics creates a joint point, matches it against an advice list, and applies selected advice effects in \texttt{app-eff}. The function ensures that the mechanism’s configuration properly reflects a “current” join point by setting it before and after an effect application.

\texttt{app-eff} has two general behaviors. If the effect list is empty then the expression evaluation is \texttt{[delegated]}. Otherwise, the function \texttt{expose} the effect by evaluating the advice expression \(\text{exp}_{\text{adv}}\) in \(\mathcal{A}\). \(\text{exp}_{\text{adv}}\) is evaluated in a properly constructed variable environment \(\text{env}_{\text{v}}\) and a proceed continuation \(\text{procd}\).

\texttt{app-eff} ensures that the mechanism configuration always stores a proper proceed continuation in the same manner as \texttt{build-}\(\text{T}_1\) reflects a “current” join point. This makes \texttt{build-}\(\text{T}_1\) straightforward (Figure 13). The meaning of a \texttt{proceed-exp} expression is given by the proceed continuation obtained from the configuration. The continuation then runs \texttt{app-eff} on the rest of the effect list. In other words, a \text{proceed-exp} expression either evaluates the next advice in \(\mathcal{A}\) or delegates the evaluation to \texttt{eval} if there is no advice left.

Due to space considerations, we omit the HisExt\(_1\) functions \texttt{match-jp}, \texttt{build-jp} and \texttt{build-adv-env}, which do not affect the mechanism composition semantics.

\[\begin{align*}
\text{app-eff} : \text{Int} & \rightarrow \text{Effect}^* \rightarrow \text{Eval}_0 \rightarrow \text{Eval}_0 \\
& = \text{fn } i \quad \text{eval exp} \ cfg \Rightarrow \text{eval exp} \ cfg_0 \\
& \quad | \text{fn } i \quad \text{exp}_{\text{adv}}, \text{exp}_{\text{adv}}) : \text{effect} \Rightarrow \text{eval} \\
& \quad \text{fn } \text{exp} \ \text{env}, \text{env}, \text{sto} : \text{cfg} \Rightarrow \\
& \quad \text{let} \\
& \quad \text{val} \ \text{adve}, \text{jp}, \text{procd} = \text{pi}(\text{cfg}^*) \\
& \quad \text{val} \ \text{procd} : \text{Cont}_A \\
& \quad = \text{fn } (? \text{env, sto}) : \text{cfg}^* \\
& \quad \text{app-eff } i \ \text{effect} \Rightarrow \text{eval exp} \ \text{env}, \text{env}, \text{sto} : \text{cfg}^* \\
& \quad \text{val} \ \text{env}, \text{sto} = \text{build-adv-env} \ \text{bind}_{\text{adv}}, \text{sto} \\
& \quad \text{val} \ \text{cfg}^* = \text{cfg} \ [\text{i} \mapsto \ {\text{adve}}, \text{jp}, \text{procd}] \\
& \quad \text{val} \ \text{cfg}^*_0 : \text{cfg}^* = \text{app-eff } i \ \text{effect} \Rightarrow \text{eval exp} \ \text{cfg}^*_0 : \text{cfg}^*_0 \\
& \quad \text{in} \\
& \quad \text{cfg}^*_0 : \text{cfg}^*_0 [\text{i} \mapsto \ {\text{adve}}, \text{jp}, \text{procd}] \\
& \quad \text{end} \\
& \quad \text{end}
\end{align*}\]

Figure 12: \texttt{build-}\(\text{T}_1\)

\[\begin{align*}
\text{val} \ \text{build-E}_1 : \text{Int} & \rightarrow \text{Exp}_1 \rightarrow \text{Cont}_A \\
& = \text{fn } i \quad \text{proceed-exp} () \ (\text{cfg as } \_ : \text{cfg}^*) \Rightarrow \\
& \quad \text{let} \\
& \quad \text{val} \ \text{pi} (\_ \text{procd}) = \text{pi}(\text{cfg}^*) \\
& \quad \text{in} \\
& \quad \text{procd} \ \text{cfg} \\
& \quad \text{end} \\
& \quad \text{end}
\end{align*}\]

Figure 13: \texttt{build-}\(\text{E}_1\)

5.2.2 HerExt\(_2\) Mechanism

The \(\mathcal{M}_2\) mechanism for HerExt\(_2\) transforms the semantics for a primitive argument and procedure execution expressions, and supplies semantics for \(\text{Exp}_2\)’s new raise expression:

\[\begin{align*}
\text{Exp}_0^2 &= \{\text{primarg-exp, probody-exp}\} \\
\text{Exp}_2 &= \{\text{raise-exp}\}
\end{align*}\]

A configuration \(\text{cfg}_2 \in \text{Cfg}_2\) (Figure 14) stores a list of handlers, a stack of currently executing procedures (a list of procedure names), and a “current” raise continuation. A handler \texttt{hnd} in \texttt{Handler} is derived from the syntax of HerExt\(_2\). It contains a name of a guarded procedure and a handler expression. A handler expression may contain a \texttt{raise-exp} expression.
The new semantics for primarg-exp enables the invocation of a handler in an exceptional situation when the second argument of a division primitive evaluates to zero. In this case, build-T2 (Figure 15) selects a list of handler expressions using match-handler and invokes them using app-handler. If no exception occurs, the original semantics is used.

The mechanism reflects the execution stack in its configuration by transforming the semantics for procbody-exp expressions. The new semantics simply pushes the stack before and pops it after applying eval.

app-handler produces a configuration transformer from a list of handler expressions. If the list is empty then the transformer is the identity function. Otherwise, the configuration is constructed by evaluating in A the first handler expression. The function also constructs and reflects a raise continuation in the mechanism configuration. The continuation simply applies app-handler to the rest of the handlers.

The build-E2 function (Figure 16) is similar to build-E1. The meaning of a raise-exp expression is provided by the raise continuation drawn from the configuration.

Due to space considerations, we omit the match-handler function of HerExt2. This function bars no affect on the mechanism composition semantics.

5.3 Constructing an AOP Language
We construct the semantical function for the composed AOP language as follows:

\[ A = \{ B, m_1, m_2 \} \]

where

\[ m_1 = \text{build-} m_1 1 \]

and

\[ m_2 = \text{build-} m_2 2 \]

The meaning of a program

\[ p = \langle \text{base, aspect}_1, \text{aspect}_2 \rangle \]

in the composed AOP language is defined as:

\[ \mathcal{W}[p] = A \exp_{\text{main}} \langle \text{cfg}_0, \text{cfg}_1, \text{cfg}_2 \rangle \]

such that

\[ \exp_{\text{main}} = \langle \text{app-exp} \langle \text{main}, [] \rangle \rangle \]

\[ \text{cfg}_0 = \langle \text{env}_p, \ldots, \ldots \rangle \quad \text{env}_p = \mathcal{D}_0[\text{base}] \]

\[ \text{cfg}_1 = \langle \text{adv}^*, \ldots, \ldots \rangle \quad \text{adv}^* = \mathcal{D}_1[\text{aspect}_1] \]

\[ \text{cfg}_2 = \langle \text{hnd}^*, [], \ldots \rangle \quad \text{hnd}^* = \mathcal{D}_2[\text{aspect}_2] \]
6. DISCUSSION AND FUTURE WORK
Our study of constructing an AOP language with multiple aspect extensions opens interesting research questions.

6.1 Alternative Collaboration Semantics
The co-existence of multiple aspect extensions raises a question concerning the desired policy of collaboration. The solution instance we presented in Section 4.2 defines the combinator \( \boxplus \) to “wrap” aspect mechanisms around the original meaning and around each other. We call this a composition with wrapping semantics.

Composition with wrapping semantics allows to compose arbitrary aspect mechanisms as long as the mechanisms can be defined as semantics transformers. However, wrapping semantics limits the ability of the multi mechanism to observe and affect the program execution. In this section we elaborate on the restrictions, and discuss how alternative solution instances can be constructed.

6.1.1 Observed Execution
Wrapping semantics grants the aspect mechanism with complete control over the original meaning and with the option to override the semantics. For example, the HisExt\( _1 \) mechanism might disable the original semantics of \( \text{app-exp} \) and \( \text{procbody-exp} \) expressions. A mechanism can either delegate the expression evaluation to the next mechanism or evaluate the expression itself. In the latter case, the evaluated expression is “filtered” out (hidden) from the aspect mechanisms downstream. However, when delegating is semantically not the right thing to do, hiding is unavoidable. For example, the HisExt\( _1 \) mechanism must filter out procedure calls and execution that are advised with no \( \text{proceed} \).

Collaboration with wrapping semantics is therefore sensitive to the order of composition. The program example in Listing 9 illustrates a collaboration of HisExt\( _1 \) and HerExt\( _2 \) with wrapping semantics.

Listing 9: Collaboration with wrapping semantics

```
program { procedure main() { 1 } }
app1 { around (): pexecution(main) {/(1,0)} }
app2 { guard cflow main resume with 2 }
```

If the AOP language is constructed as

\[ A = \boxplus \{ B, M_2, M_1 \} \]

\( M_1 \) applies first and replaces the \( \text{procbody-exp} \) of \( \text{main} \) with the advice body expression. Consequently, the execution of \( \text{main} \) is not reflected in \( M_2 \)'s execution stack and \( M_2 \) would not guard the division. The program would therefore throw a divide-by-zero exception. On the other hand, if the language is constructed as

\[ A = \boxplus \{ B, M_1, M_2 \} \]

the exception is caught.

Generally, with wrapping semantics the various mechanisms observe a program execution differently. Only the first mechanism in \( A \) has a complete view of the execution. Downstream mechanisms view less of the execution than upstream ones.

Alternatively, one can provide a collaboration semantics where all the mechanisms share a complete, coherent view of the execution. This can be achieved by decoupling the reification and reflection processes in a mechanism. With such a semantics, every expression evaluated in \( A \) is reified by all the mechanisms. The evaluation semantics is then constructed by all the mechanisms collaboratively with respect to the ordering. Given this alternative semantics, the program example in Listing 9 would produce no exception independently of the ordering of \( M_1 \) and \( M_2 \).

6.1.2 Complex Compositions
Wrapping semantics does not support complex compositions of aspectual effects. The aspectual effects in different extensions “wrap” around each other when they apply to the same join point. The resulted behavior is similar to the application of multiple \( \text{around} \) advice pieces in AspectJ. Unfortunately, this behavior is not always desirable. For example, a reasonable composition of AspectJ and AspectWerkz might require that, at each join point, \( \text{before} \) advice in both AspectJ and AspectWerkz aspects, are executed before any \( \text{around} \) advice, and finally followed by \( \text{after} \) advice. However, such an AspectJ/AspectWerkz composition is not achievable with wrapping semantics.

Our approach is not limited to the pipe-and-filter composition architecture. More complex composition semantics can be provided by imposing additional requirements on the aspect mechanism design. For example, one possibility is to specify types of aspectual effect that a mechanism can produce. With such a semantics, the overall aspectual effect can be constructed from aspectual effects of the collaborating mechanisms with regard to those effect types.

6.2 Other Mechanism Descriptions
Our choice of the mechanism’s description style restricts access to the context data. Specifically, a mechanism can only access elements of the current or parent expression, environment, and stores. While this data can be sufficient for implementing the HisExt\( _1 \) and HerExt\( _2 \) aspect extensions for MyBase, real-world aspect extensions may generally require more information. For example, AspectJ needs access to callee and caller objects to construct a method call join point. Instantiating the approach for a description style that uses an explicit representation of the evaluation context (e.g., using a CEKS machine [14, 15]) would produce a more general solution.

In our solution we used annotated expressions to meet the granularity requirement of HisExt\( _1 \) and HerExt\( _2 \). The same result can be achieved by using small-step operational semantics for describing the mechanisms. In that case, each aspect mechanism would transform and extend certain base operational semantics rules.

7. RELATED WORK
7.1 Composing Aspect Extensions
Several authors point out the expressiveness drawback in using a single general-purpose AOP language, and emphasize the usefulness of combining modular domain-specific aspect extensions [12, 13, 21, 49, 39, 31]. However, the problem of composition has not received a thorough study.

7.1.1 XAspects
Shonle et al. [39] present a framework for aspect compilation that allows to combine multiple domain-specific aspect extensions. The framework’s composition semantics is to reduce all extensions to a single general-purpose aspect extension (AspectJ). Specifically, given a set of programs written in different aspect extensions, XAspects produces a single program in AspectJ. An aspect extension program is translated to one or more AspectJ aspects. In XAspects,
collaboration between the aspect extensions is realized as a collaboration between the translated AspectJ’s aspects.

The XAspects framework uses a translation-based approach. Specifically, XAspects translates programs in domain-specific aspect extensions to AspectJ. Unfortunately, in the presence of other aspects, this approach does not preserve the behavior of the domain-specific aspects, and therefore the XAspects approach does not guarantee a correct result.

Moreover, extensions in XAspects must be reducible to AspectJ. Since only a subset of aspect extensions is expressible in AspectJ, XAspects does not achieve composition in general. Our approach to composition and collaboration is not based on translation. In comparison to XAspects our approach is more general.

7.1.2 Concern Manipulation Environment

IBM’s Concern Manipulation Environment provides developers with an extensible platform for concern separation: “The CME provides a common platform in which different AOSD tools can interoperate and integrate” [22]. CME would be a natural environment for a large scale application of our approach.

7.2 AOP Semantics

Existing works in AOP semantics explain existing aspect extensions and model AOP in general. We base some of our work on these studies. Unfortunately, they do not address the problem of aspect mechanism composition directly.

7.2.1 Semantics for Existing AOP Languages

Wand et al.’s [50] semantics for advice and dynamic join points explains a simplified dynamic AspectJ. They provide denotational semantics for a small procedural language, similar to ours. The language embodies key features of dynamic join points, pointcuts and advice. Their work, however, does not separate the AOP semantics from the base. Nevertheless, advice weaving is defined there as a procedure transformer. This is a special case of a language semantics transformer as we choose to define an aspect mechanism.

Method-Call Interception [28] is another semantical model that provides semantics for advising method calls. Similar to the previously discussed work, it highlights a very specific piece of AOP expressiveness (similar to AspectJ).

7.2.2 Semantical Models of AOP

Several studies of AOP semantics provide a general model of AOP functionality. Walker et al. [45] defines aspects through explicitly labeled program points and first-class dynamic advice. Jagadeesan et al. [24] uses similar abstractions (pointcuts and advice). Clifton et al. [8, 9] provides parameterized aspect calculus for modeling AOP semantics. In their model, AOP functionality can be applied to any reduction step in a base language semantics. This is similar to the definition of an aspect mechanism we use.

In comparison to our semantics, these models define AOP functionality using low-level language semantics abstractions. Using these more formal approaches for describing our method is left for future work.

7.2.3 Modular Semantics for AOP

We define an aspect mechanism separately from the base language and require it to specify only the AOP transformation functionality. This approach leads to the construction of modular AOP semantics. Exploring the application of other approaches for modular language semantics (e.g., modular SOS [32] and monad-based denotational semantics) to describing aspect mechanism is another area for further research.

8. CONCLUSION

In this paper, we address the open problem of integrating and using a base language Base with a set of third-party aspect extensions Ext_1, . . . , Ext_n for that language. We present a semantical framework in which independently developed, dynamic aspect mechanisms can be subject to third-party composition and work collaboratively.

We instantiate our approach for aspect mechanisms defined as expression evaluation transformers. The mechanisms can be composed like mixin layers [40, 35, 36] in a pipe-and-filter architecture with delegation semantics. Each mechanism collaborates by delegating or exposing the evaluation of expressions. The base mechanism serves as a terminator and does not delegate the evaluation further.

We applied our approach in the implementation of a concrete base language MyBase and two concrete aspect extensions to that language, HisExt_1 and HerExt_2. The implementation illustrates the construction steps. It provides semantics for third-party composition of aspect mechanisms.

The semantics for HisExt_1 resembles that for AspectJ. Indeed, our approach can be applied to implementing the pointcut and advice mechanism of AspectJ as an aspect extension to Java. This would provide a practical way to compose AspectJ with new domain-specific aspect extensions as they become available.

Acknowledgment

We thank Gene Cooperman, Darren Ng, Mitchell Wand, and the anonymous reviewers for their feedback and comments.

9. REFERENCES


A.2 Modeling aspect mechanisms: A top-down approach.
Modeling Aspect Mechanisms: A Top-Down Approach

Sergei Kojarski†  David H. Lorenz
Department of Computer Science, University of Virginia
Charlottesville, Virginia 22904-4740, USA
{kajarski,lorenz}@cs.virginia.edu

ABSTRACT
A plethora of aspect mechanisms exist today. All of these diverse mechanisms integrate concerns into artifacts that exhibit crosscutting structure. What we lack and need is a characterization of the design space that these aspect mechanisms inhabit and a model description of their weaving processes. A good design space representation provides a common framework for understanding and evaluating existing mechanisms. A well-understood model of the weaving process can guide the implementor of new aspect mechanisms. It can guide the designer when mechanisms implementing new kinds of weaving are needed. It can also help teach aspect-oriented programming (AOP). In this paper we present and evaluate such a model of the design space for aspect mechanisms and their weaving processes. We model weaving, at an abstract level, as a concern integration process. We derive a weaving process model (WPM) top-down, differentiating a reactive from a nonreactive process. The model provides an in-depth explanation of the key subprocesses used by existing aspect mechanisms.

Categories and Subject Descriptors
D.2.10 [Software Engineering]: Design; D.1.5 [Programming Techniques]: Aspect-oriented Programming; D.3.2 [Programming Languages]: Language Classifications

General Terms
Design, Languages

Keywords
AOP, AspectJ, DFD, Hyper/J, Open Classes, aspect mechanism, crosscutting concerns, definition, nonreactive, reactive, taxonomy, top-down classification, weaving process model (WPM).

1. INTRODUCTION
Concern integration is an essential process in aspect-oriented programming (AOP). Kiczales et al. [6] explicitly express the goal of AOP in terms of combining separate crosscutting concerns:

“The goal of Aspect-Oriented Programming is to make it possible to deal with cross-cutting aspects of a system’s behavior as separately as possible. We want to allow programmers to first express each of a system’s aspects of concern in a separate and natural form, and then automatically combine those separate descriptions into a final executable form using a tool called an Aspect Weaver."

The process of concern integration is called weaving, and the implementation of an “aspect weaver”—an aspect mechanism. This paper presents a top-down model of an aspect mechanism and its internal weaving process.

1.1 Evolution of AOP Models
A model primarily reflects understanding. We identify the following evolutionary stages in the common understanding of AOP:

1. \((A \times B \rightarrow B)\) In early AOP models, an aspect mechanism was explained in terms of compilation semantics. This led to thinking about AOP in terms of code instrumentation, where an aspect program \((p_A \in A)\) is woven into a base program \((p_B, p_B' \in B)\). We refer to this initial understanding of AOP as the ABB model (Figure 1).

2. \((A \times B \rightarrow X)\) Later, code instrumentation was found to be misleading and confusing in explaining the essence of AOP. When AOP was better understood, the compilation semantics was replaced with dynamic weaving semantics. The main insight was that crosscutting is an emerging property found in

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE ’06, May 20–28, 2006, Shanghai, China.
Copyright 2006 ACM 1-59593-085-X/06/0005 ...$5.00.
the composed program \((px \in X)\) rather than a pre-existing property in either \(pA \in A\) or \(pB \in B\). We refer to this observation as the ABX model (Figure 2), named to correspond with Masuhara and Kiczales’ ABX model of crosscutting [14].

3. \((C \times R \rightarrow X)\) More recently, the syntactical distinction between aspect \((A)\) and base \((B)\) has begun to diminish. In modern AOP languages, such as AspectWerkz [1] and Classpects [17], there is no essential distinction between aspects and classes (as presumably will also be the case in AspectJ 5). The traditional, syntactical dichotomy into \(pA \in A\) and \(pB \in B\) is being slowly replaced with a semantical distinction between concerns \(pC \in C = A \cup B \setminus R\), expressed in either \(A\) or \(B\), and integration rules \(pR \in R\), expressed in code, annotations, XML, or some other form. We introduce and refer to this model as CRX (Figure 3).

Interestingly, the CRX model is very close to the original Hyper/J model [16], taking this evolutionary path a full circle. The introduction of the CRX model as a general AOP model that explains both Hyper/J-like and AspectJ-like aspect mechanisms is a contribution of this paper and a direct result of a top-down approach.

1.2 Problems with Current Models

Models of aspect mechanisms have focused either on a selected mechanism, or on a bottom-up generalization of several mechanisms. In a bottom-up abstraction, a model is built by identifying and generalizing common patterns. Since each aspect mechanism is useful precisely because it does some crosscutting thing very different, abstracting over several aspect mechanisms is difficult and often results in a fine-grained model which would likely need to be further extended to fit future aspect mechanisms.

We distinguish between two general approaches to modeling aspect mechanisms, namely, semantical and conceptual (Table 1, two left columns). Existing semantical models either explain a specific AspectJ-like Pointcut and Advice (PA) mechanism or generalize a PA model. Lämmel [8] explains a join-point-and-advice mechanism named Method-Call Interception (MCI). Wand et al. [22, 15] explain an aspect mechanism for AspectJ. Walker et al. [21] define aspects through explicitly labeled program points and first-class dynamic advice. Jagadeesan et al. [5] use PA to define AOP functionality. Although very precise, all of these semantical models do not generalize over non-PA aspect mechanisms.

An example of a conceptual model is presented by the work of Masuhara and Kiczales [14] on modeling crosscutting in aspect-oriented mechanisms. Their aspect sand-box (ASB) [14, 20] framework generalizes over four mechanisms: Pointcut and Advice (PA), Open Classes (OC), Traversal (TRV), and Compositor (CMP). PA and OC are implemented by AspectJ; TRV is found in Demeter; and CMP is realized by Hyper/J.

ASB belongs to the category of ABX models. Generally, ASB represents each aspect mechanism as a weaver that combines an aspect program and a base program into a result computation (or program) \(X\). In ASB, a mechanism realizes a function with the signature:

\[
A \times B \times META \rightarrow X
\]

where \(META\) stands for composition rules existing only in CMP. The weaver is said to model a weaving process, which is defined informally as [14]:

"taking two programs and coordinating their coming together into a single combined computation."

The weaver’s semantics is presented by an 11-tuple structure:

\[
(X, X_{JP}, A, A_{ID}, A_{EFF}, A_{MOD}, B, B_{ID}, B_{EFF}, B_{MOD}, META)\]

where \(A\) and \(B\) denote the languages of the input programs, \(X\) is the result domain of the weaving process, and \(X_{JP}\) are join points in \(X\). The elements \(A_{ID}, A_{EFF}\) and \(B_{ID}, B_{EFF}\) are the elements of \(X_{JP}\) using \(A_{ID}\) and \(B_{ID}\) and contribute their \(A_{EFF}\) and \(B_{EFF}\) effects, respectively, to the semantics of the corresponding \(X\) computations (program declarations). The remaining two elements, \(A_{MOD}\) and \(B_{MOD}\), refer to structures of modularity in the input languages.

The introduction of a result domain \(X\) (that also coincides with our CRX model) is the main contribution of ASB. In ASB, join points exist only in \(X\). An aspect mechanism combines semantics contributed by \(A\) and \(B\) at join points in \(X\) using the common frame of reference. \(A\) and \(B\) do not crosscut each other directly, but only with respect to \(X\).

The essential shortcomings of ASB are:

- **Syntactical dichotomy.** ASB defines aspect mechanisms over an AOP language of a fixed syntactical form. Specifically, an AOP program is required to consist of an aspect program \(pA \in A\) and base program \(pB \in B\), both specifying concerns. When the syntax of the AOP language is

\[1\] For consistency with the other models and to avoid confusion we use \(A\) to denote the aspect language and \(B\) to denote the base language, whereas Masuhara and Kiczales originally used \(A\) for the base language and \(B\) for the aspect language.

\[2\] This is a criticism of the term aspect-oriented being used today to describe a broad array of diverse programming mechanisms, rather than a criticism of bottom-up generalization per se.

\[3\] For all of the other mechanisms, \(META\) is left out of the model.
different, the model includes it as a special case. Most noticeable, the META component was added so that ASB can also model HyperJ.

The definition of an aspect mechanism in ASB does not generalize neatly over languages with similar semantics but a different syntactical structure. For example, Classpects [17] has no aspects, just classes. AspectWerkz [1] has no advice, just methods. In AspectWerkz, a method annotated as advice can be invoked explicitly as a method or implicitly as advice. This duality is difficult to explain in an ABX model due to its firm syntactical distinction between aspect $p_a \in A$ and base $(p_B \in B)$ programs. Furthermore, the annotations in AspectWerkz can be extracted and placed in a separate XML file. These XML integration rules are not in $A$, $B$, or $X$.

- AspectJ-based abstraction. ASB selects $X_{JP}$ as the central abstraction found in all aspect mechanisms. In ASB terms, the common frame of reference is $X_{JP}$ and the programs connect at the join points. While the concept of a join point is essential in PA, it is found in only a subset of existing aspect mechanisms. Specifically, OC may be explained with or without join points (Section 5); it is unclear that join points are the right abstraction for TRV [9]; and CMP does not involve join points in $X$ at all [14, 20].

- Coarse-grained process. In ASB, a join point in $X_{JP}$ describes a computation in $X$, and the computation in $X$ is constructed using the join point in $X_{JP}$. Obviously, an aspect mechanism must construct the join point prior to the construction of the corresponding computation it describes. ASB does not explain how this cyclic dependency is resolved, and generally lacks an explicit weaving process model.

- Over generalization. ASB takes upon itself to abstract over four mechanisms that do not necessarily share properties that can be reasonably generalized. PA and CMP are oblivious [4] and provide integration mechanisms. TRV, on the other hand, is not oblivious and provides an adaptation mechanism [9]. Yet, ASB generalizes over all of them [20]. To reconcile the difference between these four mechanisms, ASB provides a fine-enough grained structure. ASB’s bottom-up generalization underscores the similarities between various aspect mechanisms. What is lost in the generalization, however, is the ability to also understand their differences.

### 1.3 Contribution

This paper presents an abstraction in the opposite direction. Using a top-down approach, we derive an abstract, yet precise, architectural model of the design space for aspect mechanisms. The model is build through analysis and design. The analysis phase characterizes the external behavior of an aspect mechanism. The design phase models the mechanism’s internal behavior.

The novelty of our model lies in its semantical rather than syntactical generalization. We define the aspect mechanism’s domains over a semantical representation of an AOP program, which can be constructed from various syntactical forms. Our model explains the essential differences between various aspect mechanisms. It categorizes aspect mechanisms according to syntactical operations.

Our CRX model emphasizes integration. The integration logic is defined by integration rules, separate from concern code defined by aspect or base programs. The CRX model provides a natural abstraction for integration-based weaving processes. It captures the similarities as well as the differences between aspect mechanisms on various levels of abstraction. The top-down specialization respects individual features of different aspect mechanisms. For example, at the top level, the HyperJ and AspectJ mechanisms are similar; and a deeper level reveals how different they are.

### 1.4 Outline

In Section 2 we analyze the weaving problem and present a general CRX weaving process model (WPM). The problem and the model are defined in abstract terms. They are constructed top-down, independent from any particular aspect mechanism. The top-down construction introduces different categories of a weaving process. In Sections 3 and 4 we design a nonreactive and a reactive specialization of CRX, respectively. We use concrete aspect mechanisms to exemplify each category. In Section 5 we explain the duality of AspectJ’s OC with respect to these categories. In Section 6 we compare the CRX model to the ABX model, in terms of providing a deeper explanation of AOP. Section 7 discusses how our classification of weaving processes is different from existing categorizations of AOP languages. Finally, Section 8 discusses how the model can help in constructing new aspect mechanisms.

## 2. ANALYSIS

This section explains the fundamentals of the concern integration process at an abstract level. We begin by articulating what a weaving problem is, and what is a solution, termed a weaving plan, to a weaving problem. We then present a high-level model of a weaving process that explains the external behavior of an aspect mechanism.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Other Models</th>
<th>Our Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design level</td>
<td>Semantical</td>
<td>Conceptual</td>
</tr>
<tr>
<td>Generalization method</td>
<td>Implementation</td>
<td>Analysis</td>
</tr>
<tr>
<td>Representative example</td>
<td>MCI [8]</td>
<td>ASB [14, 20]</td>
</tr>
<tr>
<td>Mechanisms modeled</td>
<td>PA</td>
<td>PA, OC, CMP, TRV</td>
</tr>
<tr>
<td>Unifying concept</td>
<td>-</td>
<td>Join points, JPM</td>
</tr>
<tr>
<td>Dichotomy</td>
<td>Semantical</td>
<td>Syntactical</td>
</tr>
<tr>
<td>Organizational structure</td>
<td>-</td>
<td>Flat</td>
</tr>
<tr>
<td>Resolution</td>
<td>Fine-grained</td>
<td>Coarse-grained</td>
</tr>
<tr>
<td>Abstract</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Precise</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Uniform</td>
<td>-</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 1: Comparison of approaches to modeling aspect mechanisms
Abstractly, a program, $n$, is a set of domain of concern elements. A modularized concern program $P$ is partitioned into pairwise-disjointed concern modules. For example, $P$ can be "computed" they need to be woven together to produce a "computable" composed element. That is the job of an aspect mechanism.

Per program, the problem of concern integration is to map the modularized program $P$, in which concerns are separated, to a composed program $\varphi$, in which the concerns are woven (Figure 4). We denote the mapping by:

$$P \xrightarrow{\text{weave}} \varphi$$

Abstractly, a program is either code or computation. A concern program,

$$P = \{c_1, \ldots, c_n\}$$

is a set of $n$ program elements, $c_j \in C, j = 1, \ldots, n$, where $C$ is a domain of concern elements. A modularized concern program $P$ is partitioned into pairwise-disjointed concern modules. For example, in Figure 4, $P = M_1 \cup M_2 \cup M_3$.

A composed program,

$$\varphi = \{x_1, \ldots, x_m\}$$

is a set of $m$ program elements, $x_i \in X, i = 1, \ldots, m$, where $X$ is a domain of composed elements and where the crosscutting occurs.

### 2.2 Weaving Plan

A solution to a concern integration problem is a weaving plan. A weaving plan establishes a mapping between elements of the concern program $P$ and elements of the composed program $\varphi$. A weaving plan is specified by a set of integration rules,

$$\varphi = \{r_1, \ldots, r_k\}$$

where $R$ is a domain of integration rules, and $r_i \in \mathcal{R}, i = 1, \ldots, k$.

The meaning of a weaving plan is a set of pairs,

$$[\varphi] = \{(\text{match}_i, \text{mix}_i) : x_i \in \varphi\}$$

where for every element $x_i$ in $\varphi$, $\text{match}_i \in 2^P$ is a subset of program elements, $\text{mix}_i : 2^P \rightarrow \varphi$ is a constructor, and the weaving plan satisfies the condition that $x_i \in \varphi$ is constructed by applying $\text{mix}_i$ to $\text{match}_i$. We denote this mapping by:

$$\text{match}_i \xrightarrow{\text{mix}_i} x_i$$

### 2.3 Weaving Process

We model weaving as a concern integration process. An aspect mechanism constructs $\varphi$ by integrating concerns in $P$ according to a plan $\varphi$. In our model, the aspect mechanism implements a CRX weaving process (Figure 5). At each step of the process, the mechanism accesses elements of $P$ (the $C$-register), elements of $\varphi$ (the $R$-register), and elements of the about-to-be-composed program $\varphi$ (the $X$-register). The mechanism produces new elements of the composed program $\varphi$, and writes them to the $X$-register.

A weaving process is said to be nonreactive (Section 3) if does not depend on the state of the $X$-register. A weaving process is said to be reactive (Section 4) if it requires read-access to the $X$-register in order to produce the composed elements.

The top level CRX process is the most general model of an aspect mechanism. In the next two sections we describe two specializations of weaving process. We explain the semantics of these models by mapping them to well known aspect mechanisms.

### 2.4 Notation

We describe the weaving process using a sequence of data flow diagrams (DFDs) [24, 18, 23]. A DFD is a graph that models the data flow communication among processes and data stores. Circles represent transformations. Pairs of parallel lines represent data stores. Arrows represent data. Collectively, a sequence of DFDs specifies the behavior of an aspect mechanism in terms of its internal processes that transform data and the type of data being transformed. At the top level, a context diagram describes the overall system and its data flow interfaces with its environment. The context diagram is then decomposed into level-1 processes, which are then decomposed into level-2 processes, and so on.

We use the standard DFD convention for labeling processes. A context level DFD is numbered $n.0$. Level-1 processes are labeled $n.1$, $n.2$, etc. Successive nested levels follow a similar numbering convention. This convention provides for easy tractability. We use the leading number $n$ to relate a sequence of diagrams to the section where they are discussed. The first time a process is mentioned in the text, we indicate its label in parentheses.

Note that a DFD does not specify when communications take place. In particular the labels are just identification tags and do not imply any specific processing order.

### 3. DESIGN OF A NONREACTIVE WPM

At each step of the weaving process, a nonreactive aspect mechanism computes an element (or several elements) of the composed program $\varphi$ only from elements of $P$ and $\varphi$. In other words, the weaving plan of a nonreactive aspect mechanism does not consult the state of the $X$-register in order to specify composed elements. In this section we informally introduce the inner working of a nonreactive aspect mechanism through an overview of the weaving process in Hyper/J.

#### 3.1 A Nonreactive Aspect Mechanism

The CMP mechanism [16] of Hyper/J [19] is essentially a Java program transformer. In Hyper/J’s terms, the input program $P$ is...
called a hyperspace, the concerns are called hyperslices (a hyperspace is a set of hyperslices corresponding to, e.g., $M_1, M_2, M_3$ in Figure 4), and the composed program $\varnothing$ is called a hypermodule. The composition logic, called composition rules, is the weaving plan $\varnothing$ (Figure 6). $P$ keeps the concerns separate, but does not specify a coherent behavior. CMP generates $\varnothing$ with the desired behavior by integrating the concerns together according to the composition rules $\varnothing$.

The composition rules are specified in terms of unit trees. A unit tree is an abstraction of a program’s abstract syntax tree (AST), where units represent nodes. In general, the rules define each hypermodule unit as a composition of multiple hyperspace units.

We explain the COMPOSE (3.0) process through a coding example. Consider two classes describing a person, one from a personal perspective (Listing 1); another from a tax perspective (Listing 2). We can use Hyper/J to produce an integrated view. We start by specifying a hyperspace and a mapping of the classes to hyperslices (hyperspace concerns, Listing 3). The Person hyperspace defines two hyperslices, namely, PersonalView and TaxView. The former consists of units found in the personal Java package, and the latter comprises units of the taxes package.

The composition rules (Listing 4) define the Result hypermodule as a composition of the PersonalView and TaxView hyperslices under the mergeByName composition strategy. The strategy specifies that same-name-same-type hyperspace units (e.g., the class units PersonalView.Person and TaxView.Person or the method units PersonalView.Person.getName and TaxView.Person.getName) should be merged in the hypermodule.

Generally, the meaning of a composition rule depends on the structure of the input hyperspace. In our example, the result of the composition rules depends entirely on the hyperspace structure.

The semantics for Hyper/J’s composition rules [16] define a two-
4. DESIGN OF A REACTIVE WPM

In this section we model a reactive CRX process. We present the model through an explanation of AspectJ’s PA mechanism.

4.1 A Reactive Aspect Mechanism

The context level DFD in Figure 8 depicts a reactive weaving process of a PA mechanism. PA realizes a program execution process. Given a concern program $P$, a weaving plan $\varphi$, and a computation trace $\mathcal{P}$, WEAVE (4.0) executes the program by constructing, transforming, and running computations.

$P$ and $\varphi$ are constructed from an input program string. $P$ consists of all the methods and advice-body expressions found in either the base (e.g., Java classes) or the aspect (e.g., AspectJ aspects) parts of the input program. $\varphi$ is constructed from three sources:

1. pointcut designators;
2. pointcut-to-advice mapping; and
3. advice-to-type mapping.

The first two elements provide collaboratively a mapping from join points to advice-body expressions. The last element provides a mapping from an advice-body expression to its advice type (before, after, or around), which is necessary for a proper weaving.

$\varphi$ is a vector of computations that constitutes the composed program. WEAVE executes the program $\varphi$ by constructing computations and running them. At each step of the process, WEAVE reads in the most recent computation in the trace, and executes it. The execution produces either a null-computation that wraps around a value or a set of sub-computations that extend $\varphi$.

Figure 9 is an explosion of WEAVE (4.0). ADVISE (4.1) and EVALUATE (4.2) are level-1 subprocesses. EVALUATE realizes an expression evaluator (a Java interpreter). This process executes the most recent $\varphi$ computation, and generally produces one or more join point computations $x_{jp}$. The ADVISE process intercepts a join point computation $x_{jp}$, and transforms it by wrapping it with advice computations. The computation produced by the ADVISE process replaces $x_{jp}$ in the program execution.

The PA mechanism represents a join point computation $x_{jp}$ as a join point description $desc_{jp}$. The description provides the type of the join point computation (e.g., method call, method execution, constructor execution), values drawn from its evaluation context, and related static and lexical data. The join point description is a means by which EVALUATE passes information about the program state to ADVISE. The ADVISE process then uses this information for selecting the appropriate advice.

A further breakdown of the PA’s weaving process is depicted in Figure 10. Four subprocesses constitute a reactive weaving process. MATCH (4.1.1) and MIX (4.1.2) realize advice selection and advice weaving, respectively. MATCH selects pieces of advice by applying the $jp$—advice mapping to $desc_{jp}$. The process returns
the selected advice-body expressions as advice computations \( \pi_{adv} \). MIX combines the advice computations with respect to their types with a join point computation \( \pi_{jp} \) obtained from \textsc{compute} (4.2.1). The computation composed by MIX extends the execution trace \( \varphi \).

The \textsc{control} (4.2.2) process runs the most recent computation in \( \varphi \). In general, the computation is built from a complex expression. Running the computation requires evaluation of the expression’s subterms. \textsc{control} represents subterms as instructions, and delegates their evaluation to \textsc{compute}. Since the process runs MIX-computations, the instruction set may include both base language (Java) instructions (e.g., method invocation instruction), and aspect-specific (AspectJ) instructions (e.g., advice execution instruction). The \textsc{compute} process takes an instruction \( \text{instr} \) and produces a join point computation \( \pi_{jp} \). The process also constructs a join point \( \text{desc}_{jp} \) that describes the computation.

5. THE DUALITY OF OPEN CLASSES

Open Classes (OC) is an aspect mechanism found in AspectJ that allows aspects to change the structure of a concern program. The mechanism realizes the semantics of inter-type declarations. An inter-type declaration construct associates a target Java type (what to change) with an OC effect (how to change it).

There are two kinds of OC declaration effects: member and super type. A member declaration effect introduces a new member into the target Java type. For example, the following inter-type declaration introduces an observers field into the Point class:

```java
public Vector Point.observers;
```

A super type declaration effect transforms the inheritance relationship of the target type. It is specified by a \texttt{declare parents} construct. For example, the following declaration resets Point’s superclass to be \texttt{Observable}:

```java
declare parents: Point extends Observable;
```

5.1 A Nonreactive OC Mechanism

We can explain the OC mechanism in terms of a nonreactive CRX weaving process (Figure 11). The concern program \( P \) contains OC effects and base program types (including aspect classes); the weaving plan \( \varphi \) maps target types to the effects; and \( \varphi \) is a Java program where the types and the effects are combined.

![Figure 11: Nonreactive OC Mechanism](image)

The \textsc{weave} (5.1) process iterates over \( P \) types. For each type, \textsc{weave} selects the relevant OC effects using the \( \varphi \) mappings. Then it transforms the type by applying the effects; and finally it commits the transformation result to the composed program.

5.2 A Reactive OC Mechanism

Interestingly, the OC mechanism can also be explained in terms of a static reactive CRX weaving process. A reactive OC mechanism iteratively constructs an AST of a composed program. For clarity, we focus on a simplified representation of an AST that has four types of nodes, namely (from parent to children), root, type name, type declaration, and type member (Figure 12).

The DFD diagram in Figure 13 depicts an (alternative) reactive model for OC. The model combines three processes: MATCH (5.2), MIX (5.3), and SELECT (5.4). Initially, the composed program \( \varphi \) contains only a single root node marked as open. At each step, the mechanism selects an open parent node in \( \varphi \), populating its children with AST nodes drawn from \( P \). That is, SELECT selects the parent from \( \varphi \) and the parent’s “base” child nodes \( child_0 \) from the AST of the concern program \( P \). MATCH selects OC effects associated with parent, and translates them to \( child_0 \) nodes.\(^3\) The MIX process integrates the base and the “advice” children together, and commits the result to the composed program. The child nodes are marked open, with the exception of type member nodes, which are marked closed. Weaving continues until all \( \varphi \) nodes are closed.

A reactive OC resembles PA. An open node parent corresponds to a join point \( desc_{jp} \), \( child_0 \) correspond to a join point computation \( \pi_{jp} \), and \( child_A \) correspond to advice computations \( \pi_{adv} \).

The nonreactive and reactive alternative OC models illustrate different understandings of this mechanism. OC neither changes type names nor adds or removes types. The concern program types always end up in the composed program under the same name. This allows one to interpret the OC weaving plan mappings in two ways: as a mapping between concern types and effects, or as a mapping between composed types and effects. The first interpretation implies a nonreactive weaving process; the second, a reactive one.

\(^3\)\textit{child}_A^* \) are parent’s children defined by inter-type declarations. For example, \texttt{Vector Point.observers} defines the \texttt{observers} field as a child of the \texttt{Point} class declaration.
6. BEYOND CONCEPTUAL MODELS

The CRX model affords deeper understanding of an aspect mechanism and its internal weaving process.

6.1 Abstract Domains

In CRX, the abstract AOP language representation overcomes the syntax dependency problem found in ABX. CRX defines the mechanism in terms of concern elements (domain $\mathcal{C}$) and integration rules (domain $\mathcal{R}$). These domains abstract away from any concrete representation. A program in a concrete AOP language syntax, e.g., in AspectJ, can be preprocessed, e.g., by a READ (6,0) process (Figure 14) that reads $pA$ and $pB$ in and writes $P$ and $q$ out, which in turn can be used by $\text{WEAVE}$ (4,0, Figure 8). Consequently, the CRX weaving process model generalizes over asymmetric (e.g., AspectJ) and symmetric (e.g., Hyper/J, AspectWerkz, and Classpects) AOP languages uniformly.

![Figure 14: Syntactical versus semantical](image)

6.2 WPM as a Unifying Concept

The CRX process model reveals that join points are only used in reactive mechanisms. Nonreactive mechanisms do not use join points because they do not read from the composed program register $X$. In other words, the existence of a join point abstraction can be used to distinguish rather than unify various aspect mechanisms, e.g., PA is a join point-based mechanism, CMP is not.

A CRX weaving process model was constructed top-down using abstract terms. The abstract terminology is useful for comparing different weaving processes. Zooming out highlights the similarities. Zooming in exposes differences. For example, at the top level both CMP and PA realize the same CRX weaving process. However, they fall into two different categories. CMP is a nonreactive, whereas PA is a reactive mechanism.

6.3 Cyclic Dependency in AspectJ

AspectJ has a cyclic dependency between a computation in $X$ and the join point in $X_{jp}$ that describes that computation. The CRX weaving model (recall Figure 10) explains how the cyclic dependency is resolved in the PA weaving process. Specifically, the join point $\text{desc}_{jp}$ describes the join point computation $x_{jp}$ rather than the composed program computation. Thus, the join point is created before the composed computation is constructed.

6.4 Advising Advice Execution

ASB’s view that PA selects and merges base and aspect program elements at each join point does not explain weaving at advice execution join points. Clearly, at advice execution join points, no base element should be selected. The role of base is played by a piece of advised advice. ASB does not generalize over advice execution join points because it explains advised elements strictly as base.

In contrast, the CRX weaving model of PA (Figure 10) distinguishes between a join point computation (an advised computation), and an advice computation. The join point computation ($x_{jp}$) is produced by the evaluator’s COMPUTE process as a result of running $q$ computations. Since $q$ generally contains previously woven advice computations, the join point computation might be an advice execution computation. Thus, the CRX weaving process model also explains how advice executions are advised.

6.5 Untangling Evaluate and Advise

The reactive CRX model of PA explicitly separates EVALUATE (interpreting process) from ADVISE (advising process). EVALUATE selects and computes the meaning of advised elements (e.g., methods). ADVISE selects advice, computes their meaning, and weaves them. In contrast, ASB modifies the interpreter to include aspectual operations. For example, in ASB’s PA mechanism, calls to lookup-advice are scattered throughout the interpreter and tangled with calls to lookup-method.

7. TOP-DOWN CLASSIFICATION

Our top-down analysis yields a process-oriented classification of aspect mechanisms. At the top level, we distinguish between a reactive and a nonreactive mechanism. In this section we compare our classification to existing categorizations of AOP languages. We show that our classification is different, and explain its usefulness.

7.1 Symmetric versus Asymmetric Languages

Today, existing AOP languages are sometimes categorized using the syntactic property of symmetry. Under this view, AspectJ is asymmetric—an AspectJ program consists of a base Java program and a set of aspect definitions. Hyper/J is said to be symmetric—all concerns are written in Java.

Unfortunately, this view does not provide much insight into the essential difference between AOP languages. It fails to explain how Hyper/J differs from AspectJ on a semantical level. Worse yet, under the symmetry-based view, “symmetrical” AspectJ-like languages (e.g., AspectWerkz and Classpects [17]) fall into the category of Hyper/J, instead of sharing a category with AspectJ.

Our approach abstracts away from the syntax of a language, and focuses instead on the weaving process that realizes its semantics. We represent an input AOP program in a syntax-independent manner, through sets of concern elements and integration rules.

We explain the working of an aspect mechanism according to its semantical operation. The reactive PA weaving process (Figure 10) explains AspectWerkz and Classpects in exactly the same manner as it explains AspectJ. Our classification identifies that AspectWerkz, Classpects, and AspectJ are all reactive aspect mechanisms, and contrasts them with the nonreactive Hyper/J CMP.

7.2 Static versus Dynamic Languages

Another classification approach categorizes AOP languages as static or dynamic. A static language expresses a weaving process over an abstract syntax tree (AST). For example, Hyper/J dictates the construction of a hypermodule unit tree from a hyperspace unit tree, where units represent AST nodes. Hyper/J is often referred to as a static AOP language. A dynamic AOP language allows the expression of concern integration logic over computations, using run-time abstractions. For example, advice weaving in AspectJ is defined in terms of join points in the program execution. AspectJ is often said to be a dynamic AOP language.

In our approach the weaving problem and the general CRX weaving process model are abstracted in terms of concern and composed elements. Identifying the composed and concern elements as AST nodes yields a weaving model for static languages. Defining the elements to be computations yields a weaving model for dynamic languages. Although it is tempting to think that nonreactive aspect
mechanisms are always static, and reactive aspect mechanisms are always dynamic, it is not so. Specifically, the static OC mechanism can realize a reactive weaving process. A reactive–nonreactive classification is, therefore, fundamentally different than a static–dynamic one.

7.3 Static versus Dynamic Semantics
Aside from the static–dynamic classification of AOP languages, there is an analogous classification of AOP language semantics. Semantics for dynamic AOP language can be specified in terms of code transformation. It is also possible for static languages or mechanisms to be given dynamic semantics.

In our approach, a weaving process always maps to a specific semantics, either static or dynamic. Alternative semantics for the same AOP language are normally realized by different weaving processes. For example, compilation semantics for AspectJ can be realized by a static nonreactive mechanism; dynamic semantics for AspectJ can be realized by a dynamic reactive mechanism.

8. TOP-DOWN CONSTRUCTION
A property of our top-down approach for modeling a weaving process is the ability to systematically construct new aspect mechanisms. The mechanisms can be derived in a step-wise refinement of the general CRX weaving process model (Figure 5). For example, the reactive PA mechanism model (Figure 10), which refines the general CRX weaving process model, can be further specialized to produce a concrete weaving process specification.

8.1 Component-Based Design
In our approach an aspect mechanism is modeled by a set of collaborating subprocesses. A component-based design [12] for an aspect mechanism would support a decomposition of an aspect mechanisms into distinct processes. In such a component-based design, it would be possible to replace and reuse subprocesses, thus facilitating aspect mechanism evolution. For example, the reactive PA weaving process model decouples the interpreter functionality from the aspectual advice selection and weaving processes [13]. The interpreter operations are realized by CONTROL (4.2.2) and COMPURE (4.2.1). MATCH (4.1.1) and MIX (4.1.2) realize the match and mix aspectual functionality.

Decomposing the aspect mechanism into subcomponents affords replacement of one component at a time. Consider two reasonable enhancements to AspectJ: extending the pointcut designators with new regular expressions, and adding new types of advice. Each enhancement requires replacement of exactly one component. Extending the original MATCH component to handle new regular expressions would enable the pointcut designators enhancement. To extend the advice type set, the MIX component should be upgraded.

8.2 New AOP Functionality
Most (if not all) the weaving processes that exist to date construct the composed program by extending it with new elements. For example, the advice weaving process in PA simply extends, at each step, the composed program’s computation trace with new computations. AspectJ never transforms computations within the composed program (i.e., an already executed or currently active computations). Similarly, the weaving processes of CMP and OC never change elements of the composed program.

The general CRX weaving process model (Figure 5) lends itself toward inventing novel weaving functionality beyond current practices. The model defines an aspect mechanism as a composed program transformer. At each step of the weaving process, the mechanism transforms the composed program $\phi$. Under this view, the process may include steps that actually modify or replace elements in $\phi$. For example, one can imagine a dynamic reactive process that would update currently active computations (e.g., transform the current continuation). Another example would be a static reactive process that can extend $\phi$ with new AST nodes, and transform previously composed AST nodes.

9. CONCLUSION
This paper characterizes the design space for aspect mechanisms. An aspect mechanism is an implementation of an aspect weaver. We model weaving as the process of concern integration, and define the weaving process in an abstract way, independent of any specific aspect mechanism. The abstract model describes the design space, and concrete aspect mechanisms are derived top-down from the abstract model. We distinguish between reactive and non-reactive mechanisms. This taxonomy provides a common framework for comparing and contrasting different aspect mechanisms.

We offer a new CRX model as the next evolutionary step in modeling aspect mechanisms. In CRX, integration rules in $\mathbb{R}$ govern the process of integrating concerns in $\mathbb{C}$ into crosscutting structures in $\mathbb{X}$. We formalize the CRX model as a weaving problem over corresponding abstract domains $\mathbb{R}$, $\mathbb{C}$, and $\mathbb{X}$. A CRX weaving process executes a solution to a weaving problem.

The weaving processes are classified as either reactive or non-reactive, instead of the more classic static–dynamic or symmetric–asymmetric classifications of AOP languages. We design a non-reactive CRX process for the static CMP aspect mechanism of the symmetric Hyper/J language. We design a reactive CRX process for the dynamic PA aspect mechanism of the asymmetric AspectJ language. We also design both a nonreactive and a reactive CRX processes for the static OC aspect mechanism of AspectJ.

Our top-down model overcomes many limitations in other models in terms of abstraction, precision, and uniformity. The model was not generalized from a particular set of aspect mechanisms, yet existing mechanisms can be mapped, analyzed, and explained in terms of this model. In contrast, other models are generally constructed bottom-up by analyzing existing aspect mechanisms. The generality of these bottom-up models is illustrated by expressing in terms of the model only the mechanisms used to derive them.

The design space for CRX aspect mechanisms can help not only classify existing mechanisms but also develop new ones. It can guide the implementor of new aspect mechanisms in existing classes. It can also guide the designer when mechanisms implementing new kinds of weaving are needed. We have used the CRX model in researching third-party composition of aspect mechanisms. The model was particularly useful in resolving sequential black-box composition of heterogeneous aspect mechanisms [7] and parallel glass-box composition of homogeneous aspect mechanisms [11]. Future work may include extending the CRX model to also describe a multi-mechanism weaving process.

We believe that the CRX model is a good way to explain and teach AOP [10]. Data flow analysis is effective for conceptual level modeling, and a DFD provides an easy way to understand graphical representation of a system [24]. Of course, a data flow description of the weaving process is just a step toward a complete definition or a blueprint of an aspect mechanism. Documenting the weaving process also in terms of state transitions is another logical follow-up to this work.

Acknowledgment
We are grateful to Kevin Sullivan for his helpful suggestions on this paper. We also thank Gene Cooperman, Westley Weimer, and the anonymous reviewers for their insightful comments.
10. REFERENCES


A.3 Identifying feature interactions in aspect-oriented frameworks.
Identifying Feature Interactions in Multi-Language Aspect-Oriented Frameworks*

Sergei Kojarski† David H. Lorenz
Department of Computer Science, University of Virginia
Charlottesville, Virginia 22904-4740, USA
{kkojarski,lorenz}@cs.virginia.edu

Abstract

The simultaneous use of multiple aspect languages has the potential of becoming a significant one, as new aspect-oriented frameworks are developed and existing ones expand to incorporate features of others. A key challenge in combining multiple aspect-oriented languages is identifying and resolving adverse feature interactions. These interactions occur due to the incompatible and inconsistent treatment of aspects, join points, and advice across different languages. In this paper, we analyze the root cause of this feature interaction problem. We classify common features of aspect languages, describe how these features may interact when using different aspect languages in tandem, and concretely illustrate how these interactions may be resolved. Our work allows AOP users and tool developers to reason about the occurrence of such adverse and unexpected feature interactions, and to apply several patterns for resolving these problems.

1. Introduction

The inherent goal of software design is to create sound, implementable artifacts that are also easily maintainable, quickly evolvable, and clearly readable. These characteristics emerge naturally in design artifacts that keep system concerns separate. Aspect-oriented programming (AOP) [6] provides support for separation of otherwise crosscutting concerns. As such, aspect-oriented software engineering has the potential of becoming an effective software design and development approach.

The available AOP languages vary in their potential effectiveness [18]. General purpose AOP languages, e.g., ASPECTJ [5] or ASPECTWERKZ [1], express imperatively a wide range of crosscutting concerns, but do so at the price of complex aspect descriptions. In comparison, domain specific AOP languages, e.g., COOL [10], express domain concerns declaratively, but lack the expressiveness to tackle all cases of crosscutting.

The ability to concurrently use several AOP languages, whether general purpose or domain specific, can improve the overall effectiveness of AOP. Indeed, a current trend in language and tool support for aspect-oriented software development (AOSD) [3] is to provide “multi-language” aspect-oriented frameworks which seamlessly integrate several AOP languages.† Some development environments, such as IBM’s CME, even strive to provide developers with “a common platform in which different AOSD tools can interoperate and integrate” [4].

Unfortunately, there is no methodology to facilitate the construction of multi-extension aspect-oriented frameworks. Concrete integrations are generally built ad-hoc. We examined the level of integration support and tested the quality of the integrated product in three state-of-the-art bodies of code:

- The latest version and compiler for ASPECTJ/5 [2], which is a composition of ASPECTJ 1.2 [5] and ASPECTWERKZ [1], but does not support COOL [10].
- Reflex [17], which is a multi-extension AOP kernel [16] based on an intermediate reflective representation for implementing AOP extensions and resolving aspect interactions. It includes a plugin for a subset of ASPECTJ [14], but currently not for ASPECTWERKZ nor for COOL.
- XAspects [15], which is a framework for composing aspects written in different AOP languages by translating them to ASPECTJ. It includes support for ASPECTJ and COOL, but currently not for ASPECTWERKZ.

---

*This work was supported in part by NSF’s Science of Design program under grants numbered CCF-0438971 and CCF-0609612.
†Sergei Kojarski is a PhD candidate at Northeastern University and a visiting graduate student at University of Virginia.

†The term multi-language is a misnomer since it really refers to multiple aspect extensions to the same base language.
We found that, from a user perspective, the available compositions exhibit obscure, unexpected, and even arguably incorrect behavior. We later show that Reflex handles advice erroneously (Listing 6); ASPECTJ/5 exhibits unexpected behavior [12]; and programs pre-processed in XAspects may behave incorrectly [7]. From a tool developer perspective, implementing such integrations in the respective frameworks is difficult, sometimes impossible. For example, the translation approach employed by Reflex and XAspects does not support a reasonable composition of ASPECTJ and COOL. Elsewhere [7] we show an example illustrating how translated aspect code might fail.

Based on these observations, we conclude that the design and development of an aspect-oriented framework that integrates several aspect extensions is an aspect-oriented software engineering challenge in and of itself. In this paper, we perform an analysis of the aspect extension composition problem. Our analysis reveals that the reason why implementing a multi-extension composition is so difficult is a fundamental feature interaction problem.

This paper is the first to analyze this feature interaction problem. We note that the aspect extension composition problem [7] is fundamentally different from the aspect composition problem [11]. The former concerns extensions, while the latter concerns aspects. The problem of composing extensions is the problem of defining the semantics of a new multi-extension language. In contrast, the problem of composing aspects is the problem of specifying the behavior of an AOP program under the pre-defined semantics of the AOP language.

In this work we focus on a dominant family of aspect extensions known as join point and advice (J&A). The J&A family is a practical space to investigate. It hosts many of the existing AOP languages, including ASPECTJ [5] ASPECTWERKZ [1], and COOL [10]. We use the following font, shape, and color convention. Code is displayed in Typewriter font, with keywords written in Bold series: black is used for JAVA; blue for COOL; red for ASPECTJ. Join point and advice types are written in Sans serif font; computations in Stanted shape.

2. Illustration of the Problem

Consider a composition of ASPECTJ and COOL. COOL is a domain specific aspect extension to Java for modularization of synchronization concerns. Its syntax is quite different from the syntax of ASPECTJ. COOL does not have pointcuts. Join points in COOL are implicit and are not reflected in the syntax. Advice in COOL is defined using multiple separate terms and expressions.

Aspects in COOL (called coordinators) synchronize Java methods. A synchronization policy for a method is expressed using several expressions. The expressions selfex, mutex, and requires define pre-conditions on a method invocation. A thread may execute the method only if the pre-conditions are met. Otherwise, the thread suspends and waits on the method. The on_entry and on_exit expressions are executed immediately before and immediately after a method execution.

For example, BoundedStack (Listing 1) is a coordinator in COOL that advises a Java class (with the same name) whose code is not shown. The coordinator synchronizes thread access to the methods push, pop, and top of that class. The selfex declaration (line 5) specifies that neither push nor pop may be executed by more than one thread at a time. The mutex declaration (line 6) prohibits concurrent execution of push, pop, and top. Specifically, while a thread executes one of the three methods (e.g., push), no other thread may enter either one of the other two (e.g., pop or top). A requires expression specifies a precondition over the condition fields (line 2). For example, requires !full (line 8) guards the push method from being executed when the stack is full.

The on_entry and on_exit expressions update the aspect fields immediately before and after a method execution, respectively. The expressions may read instance variables of the coordinated Java object. For example, the on_exit expression (lines 10-13) updates the empty and full coordinator variables, and reads from the size and capacity fields of the coordinated BoundedStack Java object.

The problem of composing ASPECTJ and COOL raises many questions including, e.g.:

---

Listing 1: BoundedStack.coool

```java
coordinator BoundedStack {
  condition empty = true, full=false;
  int reads=0, writes=0;
  selfex push, pop;
  mutex (push, pop, top);

  push: requires !full;
  on_entry {writes++;}
  on_exit {
    empty = false;
    if (size == capacity) full = true;
  }

  pop: requires !empty;
  on_entry {reads++;}
  on_exit {
    full = false;
    if (size == 0) empty = true;
  }

  top: requires !empty;
  on_entry {reads++;}
}
```
• What are the join points in COOL, and how do they correspond to join points in ASPECTJ?
• Should the execution of an advice in COOL generate an advice execution join point in ASPECTJ?
• Should aspects in ASPECTJ be permitted to advise the expressions requires, on_exit, and on_entry, and if so, what precisely are the rules of engagement?
• Should the execution of mutex and selfex pre-conditions be advisable by aspects in ASPECTJ, and if so, what join points do such executions generate?
• Should COOL synchronize advice in ASPECTJ that apply to the same method, or just synchronize the body of the method?

There is no simple way to identify and answer these questions for the composition of COOL and ASPECTJ. Even if one knows how to put COOL and ASPECTJ together, it is difficult to evaluate whether the composition behaves as expected.

3. Analysis of the Problem

From a feature interaction perspective, an aspect extension introduces features that incrementally extend the base language specification. The problem of specifying the behavior of a multi-extension composition is the problem of identifying and resolving the interactions between features of the composed extensions.

Each aspect extension to the base language is specified in the context of a single-extension language. In a multi-extension language, however, a program contains aspects written in various aspect extensions. Aspects written in one extension generally advise not only code written in that extension and in the base language, but also code written in the foreign extensions. Moreover, aspects written in different extensions may collaboratively affect the same point in the execution (join point). These behaviors are not described in the specification of any of the composed extensions, but rather emerge in their composition.

Our focus in this paper is on the family of extensions whose high-level behavior evolves around join points and advice (J&A). We identify the abstract features that are common to all J&A extensions [9], and we refer to the feature model (the set of abstract features) as an abstract extension. A virtual composition of such abstract extensions provides us with a conceptual framework for reasoning about concrete compositions of concrete extensions. We refer to the interactions between abstract features in a virtual composition as feature interaction patterns.

We distinguish between three related definitions commonly associated with the term join point. A join point computation is a “point” in the execution. A join point type is an extension-specific data structure for describing a computation. A join point instance is an internal description of a particular computation.

3.1. Join Point Features

We identify two groups of interaction-related abstract features. The first group includes join point features that characterize the ability to observe a program execution. The features are: granularity, type, visibility, and history.

Join Point Granularity. The join point granularity feature specifies what kinds of computations may be intercepted by the extension. The granularity is the consolidation of base language computations (base granularity) and extension-specific computations (aspect granularity). In ASPECTJ, for example, the base granularity includes certain computations in Java (method call, method execution, etc.); the aspect granularity includes only advice execution computations. In COOL, the base granularity includes only method invocation computations in Java; and the aspect granularity is empty.

Join Point Type. The join point type feature defines the extension-specific data structure for describing computations. In ASPECTJ this structure includes the kind of computation (e.g., method call) and its dynamic, static, and lexical contexts (e.g., this and target objects, signature of the target method, and location in the source file). In COOL, the description contains only the signature of the target method.

Join Point Visibility. The join point visibility feature maps join point computations to join point instances. The feature classifies computations as either visible or invisible; and instantiates join points for the visible ones. The join point visibility feature of ASPECTJ, for example, filters out all the computations within the lexical scope of an if pointcut expression; and constructs join points for all the rest. All method call, get, and set computations within the if pointcut expression are invisible, although subcomputations in their control flow (e.g., executions of methods that are called from the pointcut expression) are visible.

Join Point History and Genealogy. The join point history and genealogy feature stores join point instances, and establishes their relation to provide a genealogy of join points. Intuitively, the feature builds an extension-specific representation of the program execution. For example, ASPECTJ represents a program execution as a join point stack. The stack defines a dynamic context relation over
join points: join points pushed on the top of the stack are said to be in the dynamic context of those below them. An enclosing join point relation associates each join point with its immediate parent on the stack.

3.2. Advice Features

The second group includes advice features that shape the capability to modify a program. The features are: advice type, join point advisability, advice ordering, and advice execution.

Advice Type. The advice type feature defines the sorts of advice weaving that an aspect extension supports. For example, ASPECTJ and ASPECTWERKZ support before, after, and around advice. Identifying the advice types in COOL, on the other hand, is more of a challenge. COOL defines two blocks of operations. The first block is weaved before a method invocation. It comprises operations defined by mutex, selfex, requires, and on_entry expressions. We refer to this as the lock block. The second block is weaved after a method invocation and includes on_exit expressions. We refer to this as the unlock block. The semantics of COOL specify that both the lock and unlock blocks are synchronized. A synchronized block is never executed concurrently with another synchronized block. Therefore, it is reasonable to identify lock and unlock as advice types in COOL.

Join Point Advisability. The join point advisability feature associates advice with a join point. The association logic is normally based on the join point history. For example, in ASPECTJ an advice is selected if its pointcut matches the join point stack. Besides the core advice selection logic, the feature may define advising constraints for various types of join points, or even for specific join points. For example, the join point advisability feature of ASPECTJ restricts the advisability of handler join points to before advice only; and filters around advice at initialization and preinitialization join points.

The join point visibility and join point advisability features are not interchangeable. In general, filtering a join point is not equivalent to filtering all the advice for that join point. In the first case, the join point imposes no effect on the program execution. In the second case, the join point is reflected in the join point history, and can potentially affect the execution at a future join point. For example, if ASPECTJ were to filter join points within if pointcut expressions by disallowing advice, then these join points could still be accessed via a cflow pointcut designator.

Advice Ordering. The advice ordering feature prescribes the semantics for sorting advice that is selected at a join point into a specific execution order. For example, ASPECTJ orders pieces of advice according to their type, their lexical location in the aspect definition, and a precedence relationship over aspects.

Advice Execution. The advice execution feature controls how an extension observes the execution of its own advice. Specifically, the feature determines how advice computations are built. Once built, the computations can be recursively intercepted and transformed by the extension, thus allowing aspects to advise each other. For example, the advice execution feature of ASPECTJ executes an advice as an advice execution computation. The computation is then intercepted by the join point visibility feature and advised by ASPECTJ as an advice execution join point. On the other hand, the advice execution feature of COOL is empty, because a COOL coordinator can only advise Java methods.

4. Patterns of Interaction

We analyze the feature interaction problem by focusing on a composition of abstract extensions. Our analysis yields seven patterns of feature interaction. We present each pattern by listing the interacting abstract features and discussing how the interaction can be resolved.

Emergent Join Point Granularity and Type. The emergent join point granularity feature normalizes and consolidates the individual granularities to enable a common reference for mutual interaction. Unfortunately, this process is generally ambiguous. The problem occurs because the join point granularity features of the individual extensions are given in different terms and at different levels of detail; and they may be normalized in more than one way. This ambiguity must be resolved in the composition.

The ASPECTJ granularity feature includes method execution and method call computations. The COOL granularity feature is less detailed and only includes method invocation computations. Thus, there are at least three ways to normalize the granularities of COOL and ASPECTJ. A method invocation can be equated with a method call, or a method execution, or it can be mapped to a computation between the call and the execution (e.g., nested within a method dispatch computation, but around a method execution computation).

Each of the alternative normalizations specifies a unique composition behavior. Equating method invocation with method call join point computations forces coordinators in COOL to synchronize ASPECTJ advice at method execution join points. It also allows the composition to choose whether or not the coordinators synchronize ASPECTJ advice at method call join points (by ordering the advice of COOL and ASPECTJ). In contrast, equating method invo-
cation with method execution join point computations prevents the COOL coordinators from synchronizing AspectJ advice at method join points, and allows the composition to choose whether or not the coordinators synchronize AspectJ advice at method execution join points. Finally, mapping method invocation to a computation in-between AspectJ’s method call and method execution prevents COOL from synchronizing AspectJ advice at method call join points, and forces the coordinators in COOL to synchronize AspectJ advice at method execution join points.

Advice Execution Interaction. An aspect extension introduces extension-specific terms and expressions. In a multi-extension AOP language, these extension-specific terms and expressions can be advised by foreign aspects. However, it is unspecified how these terms and expressions are observed by the foreign extensions.

Consider deploying the BoundedStack (Listing 1) coordinator in a multi-extension composition of COOL and AspectJ. An execution of the coordinator locks and unlocks a target Java method, reads and writes to the empty, full, reads, and writes instance variables of the coordinator, and reads from the size and capacity fields of the coordinated BoundedStack Java object. In COOL these operations are not join point computations because the granularity of COOL is limited to method invocations only. The composition of COOL and AspectJ has a finer join point granularity that might intersect with COOL coordinators. This poses the question: what join point computations in the emergent granularity does the BoundedStack coordinator contain?

The problem can be attributed to the interaction between the advice execution feature of an aspect extension and the emergent granularity feature of the multi-extension composition. The advice execution feature controls how the extension observes execution of its own aspects in the extension’s granularity. In a multi-extension composition, the aspect can generally be advised by foreign extensions at the emergent granularity level. Because the emergent granularity is generally finer than the extension’s granularity, it is undefined what join point computations in the emergent granularity the aspect contains.

To resolve this interaction, a composition designer must refine the advice execution feature to build join point computations in the emergent granularity domain. In general, however, the feature can be refined in multiple alternative ways. One point of variability is the inherent difference between base language and aspect language terms and concepts. In our example, the composition designer should decide whether COOL coordinators behave like Java classes; whether static initializations of the coordinator classes generate static initialization computations; whether instantiations of the coordinator objects generate preinitialization, constructor execution, and initialization computations; whether COOL condition fields can be treated as boolean Java fields; what join point computations are found in on_entry, on_exit, and requires expressions; and whether or not other COOL constructs (e.g., mutex) produce Java join point computations.

Another point of variability is dealing with join point computations that are specific to foreign extensions (e.g., advice execution). Consider deploying the BoundedStack coordinator together with the AJAdviceLogger (Listing 2) aspect. The AJAdviceLogger aspect logs all advice execution join points in the program. The composition specification should define how, if at all, AJAdviceLogger logs executions of the BoundedStack coordinator.

One option is to execute advice in COOL as advice execution computations. Another option is to evaluate only on_entry and on_exit as advice execution computations. Yet another option is to have no advice execution computations within COOL coordinators whatsoever.

Join Point Visibility Interaction. The join point visibility feature of an aspect extension maps join point computations to extension-specific join point instances. In a multi-extension composition the mapping is undefined over join point computations that are produced by foreign aspects.

Consider now the deployment of BoundedStack (Listing 1) together with AJAccessLogger (Listing 3) in a composition of COOL and AspectJ. BoundedStack accesses the fields of the coordinator and the coordinated object, and AJAccessLogger logs access to instance variables. Even if we assume that the advice execution feature of COOL is defined to generate field access (get and set) join point computations in executions of requires, on_entry, and on_exit expressions, it is still unspecified what join points AJAccessLogger logs in the BoundedStack coordinator.

<table>
<thead>
<tr>
<th>Listing 1: AJAccessLogger.java</th>
</tr>
</thead>
<tbody>
<tr>
<td>```java</td>
</tr>
<tr>
<td>public aspect AJAccessLogger {</td>
</tr>
<tr>
<td>before():on_entry() {</td>
</tr>
<tr>
<td>System.out.println(&quot;Access:&quot;+thisJoinPoint);</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>```</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 2: AJAdviceLogger.java</th>
</tr>
</thead>
<tbody>
<tr>
<td>```java</td>
</tr>
<tr>
<td>public aspect AJAdviceLogger {</td>
</tr>
<tr>
<td>before():on_entry() {</td>
</tr>
<tr>
<td>System.out.println(&quot;Advice:&quot;+thisJoinPoint);</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>```</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 3: BoundedStack.java</th>
</tr>
</thead>
<tbody>
<tr>
<td>```java</td>
</tr>
<tr>
<td>public aspect BoundedStack {</td>
</tr>
<tr>
<td>before():on_entry() { system.out.println(&quot;Size:&quot;+size);</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
| ```
The problem can be attributed to the interaction between the join point visibility feature of AspectJ and the advice execution feature of COOL. The interaction is resolved by extending the join point visibility feature to classify and instantiate join points in coordinators. In general, the feature can be extended in many possible ways. In our example, the join point visibility feature can be defined to construct AspectJ join points for all join point computations in COOL coordinators; it can be defined to ignore get and set computations that access the condition fields of a coordinator; or it can be defined to ignore all COOL computations. In the first case, AJAccessLogger would log any set and get field access within the BoundedStack coordinator. In the second case, the aspect would not log set and get access to the full and empty fields. In the third case, the aspect would not advise the coordinator at all.

Join Point History Interaction. In a single extension, the join point history and genealogy feature establishes and maintains a relation over the join points found in the aspect and base programs. In a multi-extension composition the feature is unspecified over join points that the extension observes in foreign aspects. We call these foreign join points.

Consider the AJStackExecScope aspect (Listing 4). The dynamic context relation in AspectJ provides the semantics to the cflow pointcut. AJStackExecScope’s before advice logs all join points in the dynamic context (control flow) of the execution of a BoundedStack method (except for the join points in the control flow of the advice).2

Next, consider the AJStackLogger aspect (Listing 5). In AspectJ, advice is executed in the dynamic context of the join point it advises. AJStackLogger advises executions of BoundedStack methods. If AJStackExecScope and AJStackLogger are deployed together then the former will always advise join points inside the latter.

In a composition of AspectJ and COOL, however, the dynamic context relation of AspectJ is unspecified over join points found in coordinators. Let assume that aspects advise get and set join points in coordinators. The BoundedStack (Listing 1) coordinator runs COOL expressions on every execution of pop, push, and top. When the coordinator and the AJStackExecScope aspect are deployed simultaneously, the join point history and genealogy feature of AspectJ can treat the BoundedStack coordinator in one of several ways.

One option is to run the requires, on_entry, and on_exit expressions in the dynamic context of AspectJ’s method execution join points. In that case, AJStackExecScope logs join points in the BoundedStack aspect. An alternative is to run the expressions outside the dynamic context of the method execution join points. In that case, the AJStackExecScope aspect does not advise the BoundedStack coordinator.

Join Point Advisability Interaction. The join point advisability feature selects advice at a join point. In a composition of multiple extensions, however, the feature is undefined over foreign join points.

The construction of foreign join points is resolved in the join point visibility interaction. The join point advisability interaction is therefore an interaction between the join point advisability and join point visibility features of the aspect extension. The interaction is resolved by extending the join point advisability feature to select advice at the foreign join points.

For example, assume that the join point visibility feature of AspectJ constructs get and set join points for every access to any field of a COOL coordinator or a coordinated object (from within the coordinator). The interaction between the join point advisability and join point visibility features of AspectJ then raises the question of how AspectJ advises these join points.

The interaction can be resolved in at least three reasonable ways. First, AspectJ may treat these join points as regular field access without constraining the kind of advice. A second option is to allow AspectJ aspects to advise only a field access that targets a field of a coordinated object (e.g., access to size and capacity), while hiding any access to an internal field of a coordinator (e.g., empty, full, reads, writes). This solution respects the differences between COOL and Java objects, while allowing AspectJ aspects to advise any access to a Java field. A third option is to allow AspectJ aspects to advise an access to a field of a coordinator with before advice only. In this manner, the internal synchronization logic of COOL coordinators cannot be overridden by AspectJ, while AspectJ aspects can still monitor the coordinator.

---

Listing 4: AJStackExecScope.java
```java
public aspect AJStackExecScope {
  before():cflow(execution:+ BoundedStack.*{}))
  & & :cflow(within(AJStackExecScope)) {
    System.out.println("Cflow:"+thisJoinPoint);
  }
}
```

Listing 5: AJStackLogger.java
```java
public aspect AJStackLogger {
  before():execution(+ BoundedStack..{})) {
    System.out.println("Method:"+thisJoinPoint);
  }
}
```
Emergent Advice Type. The advice type feature of the composition normalizes the set of advice types defined by the individual extensions. However, the normalization of advice types is generally ambiguous. Different normalization yield different composition specifications. This ambiguity constitutes the unspecified behavior found in the composition.

The normalization of COOL and ASPECTJ advice types is a good example. The first challenge is to identify correctly the COOL advice types (recall Section 3.2). Once the advice types are identified, the problem of normalizing advice types in the composition of COOL and ASPECTJ is the problem of matching the lock and unlock advice types of COOL with the before, after, and around advice types of ASPECTJ. One option is to equate lock and unlock with before and after, respectively. Another option is to equate a pair of lock and unlock advice blocks with a single around advice. A third option is to consider COOL and ASPECTJ advice types distinct, and allow COOL advice to dominate over ASPECTJ advice (i.e., always run lock advice before ASPECTJ’s advice and unlock advice after ASPECTJ’s advice). This ambiguity illustrates the general problem of advice type feature interaction in a composition of multiple aspect extensions.

Emergent Advice Ordering. A single join point computation can generally match advice written in several extensions. The advice ordering feature of the composition (not to be confused with the advice ordering feature of an individual extension) defines the ordering of selected multi-extension advice.

The ordering of multi-extension advice is unknown at the level of the individual extensions. Rather, this feature emerges in the combination of the various advice ordering features and is thus unspecified.

Consider a composition of ASPECTJ and ASPECTWerkz. For this composition, the advice ordering feature interaction problem has many alternative solutions that exhibit considerably different behavior. For example, one option is (a) to execute before advice of both extensions first; then to wrap their around advice over each other; and finally execute the remaining after advice. Another option is (b) to run ASPECTWerkz’s advice only when ASPECTJ proceeds to the base program. In this case, ASPECTWerkz advice are nested within the execution of ASPECTJ around advice.

Adding an extension to the composition generally increases a number of alternative solutions. For example, consider a composition of ASPECTJ, ASPECTWerkz and COOL. Assuming that COOL method invocation join points are equated with method execution join points, and COOL lock and unlock advice types are equated with before and after advice types of ASPECTJ/ASPECTWerkz, each ordering of ASPECTJ and ASPECTWerkz advice corresponds to multiple combinations of ASPECTJ, ASPECTWerkz, and COOL advice. For example, if ASPECTJ and ASPECTWerkz advice are ordered as suggested by option (a), the lock and unlock advice may be ordered to (i) synchronize all ASPECTJ/ASPECTWerkz advice; (ii) synchronize only around advice; (iii) synchronize only Java methods, and so on.

5. Assessment

In this section we assess the benefit of our analysis and conceptual framework by using the patterns to identify adverse feature interactions and suggest ways to resolve them in the composition of ASPECTJ, ASPECTWerkz and COOL. We also use the patterns to identify and explain the unexpected behavior we observed in ASPECTJ/5, Reflex, and XAspects.

5.1. Specifying New Compositions

Consider a composition of three extensions: ASPECTJ, ASPECTWerkz and COOL. For this composition, the feature interaction patterns generate a set of 24 interactions to be analyzed. Concretely, the three emergent patterns generate one feature interaction each, the advice execution pattern yields three interactions, and each of the other three patterns generates six feature interactions, e.g., interactions between the join point visibility feature of COOL/ASPECTJ/ASPECTWerkz and the advice execution features of ASPECTJ and ASPECTWerkz (COOL and ASPECTWerkz, ASPECTJ and COOL). The feature interactions pose questions, including:

- What join point computations are observed by ASPECTJ (ASPECTWerkz) within the coordinators?
- How does ASPECTJ (ASPECTWerkz) define the dynamic context relation over coordinator’s join points?
- How does ASPECTJ (ASPECTWerkz) advise join points that originate from the coordinators?

Although the questions are initially expressed in abstract terms, they are easily restated in concrete terms of COOL, ASPECTJ, and ASPECTWerkz. Each question expands into several extension-specific questions. Due to space consideration, we list and answer only the most interesting ones:

What is the granularity of the composition? The composition has the same granularity as ASPECTJ. Corresponding join point types of ASPECTJ and ASPECTWerkz are equated. Method invocation computations of COOL are equated with method execution computations of ASPECTJ.
What join point computations are specified by a COOL coordinator? The coordinator advises a target method with `lock` and `unlock` advice. An execution of a COOL advice yields an advice execution computation. Within the advice, COOL exposes `get` and `set` computations as defined by `requires`, `on_entry`, and `on_exit` expressions.

What join points does ASPECTJ (ASPECTWerkz) instantiate for a coordinator’s computation? An advice execution computation is mapped to an advice execution join point. A `get` (`set`) computation is represented by a `get` (`set`) join point.

How does ASPECTJ (ASPECTWerkz) advise COOL join points? ASPECTJ (ASPECTWerkz) may only impose before advice over `set` and `get` join points that target fields of the coordinator object. This decision protects the internal thread synchronization logic of the coordinator, while allowing ASPECTJ (ASPECTWerkz) aspects to inspect its execution. An access to a field of the coordinated object is advised in the usual way. The advice execution join points (that represent executions of `lock` and `unlock` advice) can be advised by before and after advice only. This decision protects COOL advice from being overridden by ASPECTJ (ASPECTWerkz) aspects.

What join points are observed by COOL in ASPECTJ (ASPECTWerkz) aspects? COOL treats the ASPECTJ (ASPECTWerkz) aspects as Java classes. It builds method invocation join points for method execution computations that originate from the aspects. Moreover, implicit executions of ASPECTWerkz advice or pointcut methods are also observed by COOL as method invocations.

What join points are observed by ASPECTJ aspects in ASPECTWerkz aspects? In ASPECTWerkz an advice is an annotated method with a dual purpose (and a pointcut can also be defined using a method). The method may be executed implicitly when playing the role of an advice (or a pointcut). It may also be invoked explicitly as a regular Java method. Implicit executions of the advice methods of ASPECTWerkz are exposed as advice execution join points in ASPECTJ. Implicit executions of the pointcut methods are hidden from ASPECTJ, including join points that lexically reside within the pointcut methods. Explicit executions of the pointcut and advice methods are treated as regular Java method executions.

The rest of the feature interactions between ASPECTJ, ASPECTWerkz, and COOL are omitted. A complete list of questions and answers fully specifies the composition. The specification we have presented defines only one possible composition of ASPECTJ, ASPECTWerkz and COOL.

5.2. Analyzing Existing Compositions

Our analysis is also useful for identifying and explaining adverse feature interactions in existing compositions of aspect extensions. Applying the analysis to ASPECTJ/5, Reflex, and XAspects reveals that they fail to adequately resolve feature interactions in multi-extension compositions. All three systems exhibit an unexpected behavior of aspects “misadvising” foreign aspects. ASPECTJ/5 fails to address a particular instance of the advice execution feature interaction; and Reflex and XAspects fail to address the feature interaction problem in general.

ASPECTJ/5. ASPECTJ/5 [2] is a merger of ASPECTJ and ASPECTWerkz. ASPECTJ and ASPECTWerkz are very similar semantically, diverging mainly at the syntactical level. ASPECTJ differentiates syntactically between a method, an advice, and a pointcut; whereas ASPECTWerkz does not.

We tested ASPECTJ/5 using a set of ASPECTJ and ASPECTWerkz aspects that address all the interactions that are generated from the interaction patterns. Running these aspects in ASPECTJ/5 revealed that:

- Explicit executions of either pointcut or advice methods in ASPECTWerkz aspects do not generate method execution join points;
- Advice execution join points are generated for both implicit and explicit executions of the advice methods;
- No join points are generated within the body of a pointcut method, even if it is executed explicitly.

We attribute this behavior of ASPECTJ/5 to a failure to identify and address the interactions between the advice execution feature of ASPECTWerkz and the emergent granularity feature of ASPECTJ/5. In our minds, ASPECTJ/5 generates unexpected advice execution join points without any actual advice execution; unexpectedly hides method execution join points at explicit executions of pointcut and advice methods; and unexpectedly hides all join points within pointcut methods at their explicit execution.

Reflex. Tanter et al. [16] present a framework for resolving aspect interactions. The Reflex framework is also used to address the aspect extension composition problem, by reducing the problem of composing multi-extension aspects to the problem of composing multiple aspects in Reflex. The framework applies a translation approach: it takes as input an aspect program written in a given language and outputs an aspect program in Reflex [17].

A designer of a multi-extension composition can use Reflex to specify composition rules that resolve aspect-level interactions in the translated program. Concretely, the designer can resolve “aspect interactions [that] occur when
several aspects affect the same program point” [17], namely ordering and nesting of aspects, mutual exclusion (an aspect should not apply whenever another aspect applies), and implicit cut (an aspect should apply whenever another aspect applies).

However, while the framework provides adequate support for resolving aspect-level interactions in a specific Reflex program, it does not provide any means for resolving extension-level interactions in the multi-extension AOP language. Most notably, the lack of support for resolving advice execution interactions renders the Reflex framework inapplicable for composing aspect extensions in general.

The source of the problem is that Reflex fills in the unspecified behavior in an ad-hoc manner. In particular, the translated aspects in Reflex may misadvise each other. When the framework translates an aspect from a source extension to Reflex, it introduces implementation-specific operations into the target Reflex aspect (realized in part as a metaobject class). The implementation-specific operations are not explicit in the code of the source aspect; they realize behavior of terms, expressions, and concepts that are specific to the source extension. For example, means of synchronizing threads that are implicit in a COOL coordinator might be translated into explicit wait and notifyAll method calls in a target Reflex aspect.

The implementation-specific operations introduce unexpected join points into a translated Reflex program. The program aspects cannot distinguish the bad (unexpected) from the good (expected) join points, and can erroneously advise the bad points. Advice executions at bad join points are not only unexpected to an aspect programmer, but may also render the program incorrect. For example, the empty advice in the AJNothing aspect (Listing 6) is supposed to do nothing but results in an infinite loop when weaved by Reflex. The source of the problem is that Reflex translates the before advice to a method in the target metaobject class. The aspect then erroneously advises executions of this method as method execution join points. This simple failure illustrates an inability of Reflex to deal with complex extension-level interactions in a multi-extension composition.

Listing 6: AJNothing.java

```java
aspect AJNothing {
  before():execution(* *(..)) {} 
}
```

XAsects. Shonle et al. [15] present a framework for compiling aspects written in multiple domain-specific aspect extensions. XAspects uses a translation approach. It reduces all extensions to ASPECTJ. Given a set of programs written in different extensions, XAspects produces a single program in ASPECTJ.

Similar to Reflex, the XAspects framework does not provide adequate support for resolving feature interactions in the multi-extension AOP language. We analyzed the support for COOL and ASPECTJ in XAspects. We found that the major problem of the composition lies in a failure to resolve the interaction between the advice execution feature of COOL and the emergent granularity feature of the composition. Because of that interaction, COOL coordinators expose unexpected join point computations that can be erroneously advised by ASPECTJ aspects. In some multi-extension programs this interaction causes an unexpected deadlock [7]. Most of the other interactions are also not addressed in XAspects.

6. Other Related Work

As we have covered ASPECTJ/5, Reflex, and XAspects, we complete the discussion of related work with two studies that focus on the problem of composing aspect extensions and on the problem of composing aspects.

Disciplined Aspect Extension Composition. Pluggable AOP [7] is a study of the problem of undefined semantics for aspect extension compositions. The work presents a framework for third-party composition of arbitrary dynamic aspect mechanisms into an AOP interpreter. In the Pluggable AOP framework, the aspect mechanisms collaborate by hiding, delegating, and exposing expression evaluations.

Unlike other composition frameworks, Pluggable AOP resolves advice execution interactions in the composition. Specifically, the framework establishes a common granularity for the multi-extension AOP language, and provides design guidelines for resolving the advice execution interactions [13]. The framework, however, provides very restricted support for resolving advice ordering (other than ordering the mechanisms), and does not address other interactions [8].

In contrast, this paper studies feature interactions that occur in the composition. The results we present here are relevant for any composition of aspect extensions, not just for third-party compositions. Specifying the composition semantics is inherently complex. Our analysis simplifies some of that complexity by identifying what must be specified.

Disciplined Aspect Composition. Lopez-Herrejon et al. [11] study the problem of undefined semantics for aspect compositions in ASPECTJ. Although the problem of undefined semantics in a single-extension AOP language is orthogonal to our study, their algebraic model can be used to specify multi-extension aspect composition more flexibly.
7. Conclusion

Our work analyzes the feature interaction problem in the composition of aspect extensions. This problem is the reason why the task of composing multiple extensions is so complex, error prone, and unintuitive. Failing to address feature interactions normally results in unexpected behavior in the multi-language aspect-oriented framework.

One contribution of this paper is a road map for resolving the feature interaction problem in multi-language aspect-oriented frameworks. We analyze existing aspect extensions and derive a set of abstract features and their patterns of interaction. The value of our analysis is in providing users and tool developers with an appropriate abstraction for identifying feature interactions in a large set of compositions.

Our analysis is useful for specifying the behavior of new compositions of existing aspect extensions. We identify and resolve potential feature interaction problems in a hypothetical composition of ASPECTJ, ASPECTWERKZ, and COOL.

Another contribution of this paper is the understanding of the space of compositions. Our analysis is useful for verifying that feature interactions in existing frameworks are resolved properly. We identify and illustrate feature interaction problems in ASPECTJ/5, Reflex, and XAspects.

Finally, the set of features we identify contributes a vocabulary for documenting, understanding, and communicating designs of multi-language AOP frameworks.

Acknowledgment We thank Kim Hazelwood and the anonymous reviewers for their helpful comments.

References

A.4 AWESOME: An aspect co-weaving system for composing multiple aspect-oriented extensions.
AWESOME: An Aspect Co-Weaving System for Composing Multiple Aspect-Oriented Extensions*

Sergei Kojarski
College of Computer and Information Science
Northeastern University
360 Huntington Ave.,
Boston, Massachusetts 02115, USA
kojarski@ccs.neu.edu

David H. Lorenz
Dept. of Mathematics and Computer Science
The Open University of Israel
108 Ravutski St., P.O. Box 808,
Raanana 43107, Israel
lorenz@openu.ac.il

Abstract
Domain specific aspect-oriented language extensions offer unique capabilities to deal with a variety of crosscutting concerns. Ideally, one should be able to use several of these extensions together in a single program. Unfortunately, each extension generally implements its own specialized weaver and the different weavers are incompatible. Even if the weavers were compatible, combining them is a difficult problem to solve in general, because each extension defines its own language with new semantics. In this paper we present a practical composition framework, named AWESOME, for constructing a multi-extension weaver by plugging together independently developed aspect mechanisms. The framework has a component-based and aspect-oriented architecture that facilitates the development and integration of aspect weavers. To be scalable, the framework provides a default resolution of feature interactions in the composition. To be general, the framework provides means for customizing the composition behavior. Furthermore, to be practically useful, there is no framework-associated overhead on the runtime performance of compiled aspect programs. To illustrate the AWESOME framework concretely, we demonstrate the construction of a weaver for a multi-extension AOP language that combines COOL and AspectJ. However, the composition method is not exclusive to COOL and AspectJ—it can be applied to combine any comparable reactive aspect mechanisms.

Categories and Subject Descriptors D.1.5 [Programming Techniques]: Aspect-oriented Programming; D.2.12 [Software Engineering]: Interoperability; D.3.4 [Programming Languages]: Processors

General Terms Design, Languages

Keywords AOP, aspect extension, aspect mechanism, aspect weaver, composition, DSL, framework, pluggability.

1. Introduction
Aspect-oriented programming (AOP) [21] is celebrating a decade of research and development and industry adoption. New language features are continually being proposed. These features need not only be implemented and evaluated, but also tested to work with existing AOP languages. Facilitating the construction of new aspect extensions that incorporate new features and that can be composed with other extensions is thus important for making research advances accessible for experimentation and use in realistic settings.

Supporting the composition and use of newly developed aspect extensions together with established mainstream extensions can leverage and broaden their respective impact. The ability to program in a multi-extension AOP language can also help compare features. It can eliminate tradeoffs associated with choosing an extension with the most appropriate features. Most importantly, it can help realize the vision of domain specific aspect languages (aspect DSLs).

Unfortunately, despite the availability of extensible aspect weavers (e.g., [4, 5]) and extension composition frameworks (e.g., [37, 36, 22]), implementing industry-quality weavers that are composable remains a complex and costly task. For example, abc [4, 5] is more extensible than ajc [17], but does not support composition with foreign extensions. Reflex [37] and XAspects [36] support composition, but ignore foreign advising [28]: they lack the customizability necessary for preventing aspects from “misadvising” foreign aspects. Pluggable AOP [22] resolves the flawed foreign advising behavior found in Reflex and XAspects. However,
Pluggable AOP is impractical for combining “real world” AOP languages. It supports the customization of individual extensions, but provides only limited customization of the composition semantics. Also, Pluggable AOP composes extensions by constructing an interpreter, which is inefficient, and thus deemed inappropriate for industrial use.

Today, potentially useful aspect extensions are not readily available because either it is too difficult to implement them or they are implemented but cannot be used together with AspectJ [20]. The unattended need for combining COOL [27] and AspectJ is a representative example. Software engineering studies [32, 33, 40, 41] that compared COOL and AspectJ have concluded that COOL code is easier to understand and debug than Java or AspectJ code. Yet, COOL is not widely used; AspectJ programmers cannot embed COOL code in their AspectJ programs.

1.1 The Composition Problem

There are two main impediments to overcome. The first is the composition specification problem [24]: given a set of $n$ extensions, identify and resolve feature interactions in their composition. For example, COOL extends Java with a method synchronization mechanism; AspectJ extends Java with an advice binding mechanism. In a composition of COOL and AspectJ, coordinators and aspects may interact in unexpected ways.¹ These interactions need to be identified and resolved.

The specification problem is inherently complex and difficult, and its resolution is outside the scope of this paper. It is complex because for a choice of $n$ extensions there are $O(n^2)$ pairwise interactions to specify.² It is difficult because several reasonable resolutions exist for each interaction [24].

The second impediment, which is the main focus of this paper, is the composition implementation problem: design a composition framework such that, given $n$ aspect weavers (Figure 1) and a composition specification (Figure 2), plugging them into the framework implements the composition under the specification.

The implementation challenge is to design a framework that has the following characteristics:

¹ A coordinator in COOL is the equivalent of an aspect in AspectJ.
² In the worst-case, there might be $O(2^n)$ combinations to consider.

Minimum performance overhead. To be practically useful, the framework should construct weavers without inflicting performance degradation in the runtime of compiled aspect programs.

Maximum code reuse. The framework should provide libraries and abstractions that support rapid development of individual weavers. Components that are common to all AspectJ-compatible weavers should be reused whenever possible.³ New weavers should implement only the necessary extension-specific operations. Avoiding unnecessary code repetitions in constructing different weavers also improves their reliability.

Auto-configuration. The framework should support automatic composition of multiple weavers into a multi-weaver that exhibits a reasonable default behavior, thus avoiding whenever possible the tedious task of resolving all the interactions explicitly.

Manual override. Although the default multi-weaver behavior is likely to be appropriate for most cases, a composition specification might require to resolve some of the feature interactions differently. The framework must allow the language designer to override parts of the default configuration in order to comply with the specification.

1.2 Contribution

The main contribution of this paper is an Aspect co-WEaving System for composing Multiple Extensions (AWESOME). The AWESOME framework is:

- Composable: enables third-party composition of aspect weavers into a multi-weaver with a reasonable default behavior.
- Customizable: provides means for customizing the behavior of the constructed multi-weaver to cater for the composition specification; and
- Efficient: employs a compile-time weaving scheme.

In comparison to other frameworks (Table 1), AWESOME is the only one to provide a flexible customization mechanism. The compile-time weaving scheme used by AWE-

³ By AspectJ-compatible we refer to a reactive join point and advice aspect extension [23] that can be reduced to AspectJ, e.g., COOL, AspectWerkz [6], CaesarJ [3].
Reex provides support for resolving interactions between aspects, but not between aspect extensions. The quality of the code woven in AWESOME is comparable to that produced by standard aspect compilers.

AWESOME is also the only composition framework to be demonstrated and tested on real-world languages. We demonstrate the construction of an AWESOME weaver for a multi-extension AOP language, named COOLAJ, that combines COOL and AspectJ. Although we are not the first to pursue a combination of COOL and AspectJ, we are the first to do so systematically by: (a) giving a specification of COOLAJ; (b) constructing a weaver for COOLAJ; and (c) evaluating the weaver by testing its behavior against the COOLAJ specification. We also compare the woven code to code produced by ajc [17] and other weaving algorithms.

Another contribution is the analysis of the extension composition problem: a set of requirements for an aspect extension composition framework: and a reasonable default resolution of the feature interaction problem in a composition of aspect extensions. To concretely illustrate the composition problem and its solution, we include specific implementation details for the composition of COOL and AspectJ into COOLAJ. While a specification and a weaver for COOLAJ is a novel and useful contribution in and of itself, the composition approach is not specific to COOL and AspectJ. It generalizes to a large category of (reactive) aspect mechanisms [23].

Outline

By way of background, we first explain how individual weavers work. In Section 2 we review the working of the ajc weaver for AspectJ, and in Section 3 we describe the working of a weaver for COOL. The objective of this overview is to provide the reader with a familiarity with the internal components of a weaver and to introduce the necessary terminology. In Section 4 we analyze the foreign- and co-advising interactions, and we formulate the requirements for the co-weaving system. A novel aspect-oriented architecture for the framework is presented in Section 5. In Section 6 we refactor the implementation of the AspectJ and the COOL weavers to reflect this new AOP design. In Section 7, as a case study, we informally specify and describe the implementation of a multi-weaver for COOLAJ. In Section 8 we evaluate our AWESOME system and demonstrate its pluggability, correctness, and efficiency.

2. An Aspect Compiler

An aspect compiler compiles aspect programs into an executable. We begin by describing the high-level architecture of an aspect compiler. For concreteness, we describe the ajc compiler for AspectJ.

In general, an aspect compiler has a front-end and a back-end. The front-end translates aspects to (annotated) classes in the base language. For example, the front-end of ajc translates aspects written in AspectJ (.java and .aj files) to annotated classes in Java. The translation process in ajc is mostly straightforward: an aspect is translated to a Java class with the same name; an advice declaration is transformed into a method declaration with the same body. The compiled advice method is also annotated with attributes that store its aspect-specific data (e.g., pointcut declarations). The annotations distinguish aspect classes from other Java classes, and provide pointcut designators for advice methods.

The back-end implements the semantics of the aspect extension. The semantics define the meaning of advice weaving in terms of computations. A computation in this context is a block of program execution, e.g., a method execution. It encapsulates a sequence of operations that define a behavior and a dynamic context that includes all arguments and other values accessible by the computation. An advice is a computation transformer [17, 22, 30, 31, 42]. It takes a computation and produces a transformed computation that runs the advice body before, after, or instead of the original computation.

While the extension’s semantics define weaving in terms of dynamic runtime abstractions, the weaver implements the semantics statically by transforming the base and aspect classes. The weaver transforms a computation by transforming its shadow, a body of code that defines a computation’s

---

4 Reflex provides support for resolving interactions between aspects, but not between aspect extensions.

5 The reflection-based weaving scheme of Reflex may degrade the runtime performance of the woven program.

6 More precisely, the target classes are expressed in the Java Virtual Machine (JVM) bytecode language.
behavior. For example, the ajc weaver transforms Java bytecode. At the bytecode level, an advisable shadow maps to a continuous block of instructions with a well-defined begin and end.

The weaver implements an abstract weaving process that comprises four subprocesses, namely reify, match, order, and mix (Listing 1). These (sub)processes are found in all reactive aspect mechanisms [23]—mechanisms that can be semantically modeled as a closed-loop feedback control system—including Cool and AspectJ.

The reify process takes as input a class file and constructs a weaver-specific representation of the class. For example, the AspectJ weaver represents a class as a set of computation shadows. Its reify process examines the input Java class cf, and identifies all the shadows that can possibly be advised. Each shadow references a list of instructions embedded in one of cf’s methods (the body of the shadow), and provides static and lexical descriptions of these instructions (the static context of the shadow).

The match process associates elements of the program representation (shadows) with pieces of advice. In AspectJ, the weaver selects the set of advice by matching the description of the shadow (the static context) against the static part of the advice pointcuts.

The order process sorts and orders all pieces of advice that match the same shadow into a correct application order. The ajc weaver orders the pieces of advice according to the rules defined by the AspectJ language semantics.

The mix process transforms an actual body of a shadow by introducing code of advice (or calls to advice) that match this shadow. The AspectJ weaver transforms the shadow’s instruction list by sequentially introducing calls to the advice methods before, after, or instead of the original code. The advice pieces are woven in by sequentially transforming the body of the shadow. An advice then injects new code inside the body of the shadow, immediately before, immediately after, or instead of the original code. The transformation considers instructions that were woven earlier as if they were a part of the original shadow. This way the advice pieces “wrap” around each other in the transformed shadow.

The four processes provide a high-level description of the advice weaving semantics. A concrete weaver may also realize other kinds of transformations. For example, the ajc weaver implements intertype declarations and advice weaving in two separate steps (Listing 2). First, the weaver extends and transforms the class cf by applying the intertype declarations (the call to applyIntroductions in Listing 2). Once the declarations are applied, the weaver calls weaveClass, which implements the advice weaving behavior. The additional transformations are normally static in nature, and do not interfere with the dynamic advice weaving behavior.

3. A Compiler for COOL

The architecture of a compiler for COOL is similar to that of ajc. The front-end translates coordinators in COOL to classes in Java. The back-end instruments the program with calls to methods of coordinator classes. We explain by example the basics of the COOL language and the internal workings of its compiler. Consider the implementation of a bounded stack in Java (Listing 3). Stack defines two public methods: push and pop. An attempt to pop objects off an empty stack or push objects onto a full stack throws an exception.7

COOL relieves the implementor of Stack from dealing with multi-threading. A separate Stack8 coordinator (Listing 4) imposes the synchronization logic over push and pop in an aspect-oriented manner. The Stack methods are not synchronized. But in the presence of the Stack coordinator, the stack object operates correctly even when multiple client threads execute methods simultaneously.

The synchronization policy is expressed in COOL using declarations (mutex, selfex.condition), expressions (requires), and statements (on_exit, on_entry). The selfex declaration (line 402) specifies that neither push nor pop may be executed by more than one thread at a time. The mutex declaration (line 403) prohibits push and pop from being executed concurrently. The requires expressions (lines 406 and 412) further guard push and pop executions. If the guard is false, a thread suspends, even if the mutex and selfex conditions are satisfied. The execution resumes when the guard becomes true. full and empty are condition boolean variables (line 405).

The on_entry and on_exit blocks update the aspect state immediately before and immediately after the execution of an advised method body, respectively. They are used in this example to track the number of elements in the stack.

---

7 java.lang.ArrayIndexOutOfBoundsException
8 In COOL, the names of the coordinator and the coordinated class must be the same.
Java expressions within COOL statements have read and write access to the coordinator’s fields. In addition, expressions may inspect instance variables of the coordinatee, e.g., access the buf field of the Stack object (line 410). Note that the coordinator’s expressions may access not only public but also package-protected and even private fields of the coordinatee object.

3.1 Front-end Translation

The COOL front-end translates a coordinator in COOL to a coordinator class in Java. The name of the class is obtained by appending “Coord” to the name of the aspect, e.g., StackCoord.

StackCoord (Listing 5) implements the synchronization logic via special synchronized methods and instance variables. The class provides a pair of lock_ and unlock_ methods and an instance variable for every method that is advised by the coordinator. Specifically, the synchronization for the Stack.push method is realized by lock_push and unlock_push. Similarly, the synchronization logic for Stack.pop is realized by lock_pop and unlock_pop. At any point of the execution, the pushState (popState) instance variable stores all threads that are currently executing the push (pop) method on the coordinated object. The coordinator class also includes all fields of its coordinator.

The lock_ methods implement the semantics for mutex, selfex, and requires, and run on_entry blocks. A while loop
in the lock_push method suspends the execution of the current thread (by invoking wait on the coordinator object) so long as either one of the requires, selfex, or mutex conditions is in violation. The requires condition is checked by the !(full) expression (line 504). selfex fails if push is run by another thread (line 505); and mutex fails if pop is run in parallel (line 506). If all the guard conditions are satisfied, the thread executes all the existing on_entry statements, and locks the coordinated push method by adding its Thread object to the pushState list (line 509).

The unlock methods unlock the coordinated method and run the on_exit statements. Specifically, unlock_push unlocks the coordinated push method by removing the current Thread object from the pushState list (line 513). It then executes the on_exit statement (lines 515–518) and notifies the other threads waiting on the lock that the coordinated method is free (line 519). Note that accesses to the coordinated object fields (instance variables) are translated into method calls on the coordinated object. Specifically, access to the buf field of the coordinated object is translated into a _buf() method call (line 517). This is the way in which the coordinator class gains access to protected or private fields of the coordinated class. The method is generated in the coordinated class by the COOL weaver, and simply returns the value of the corresponding field.

3.2 Back-end Weaving

The COOL weaver applies four kinds of transformations to a coordinated class, namely method transformation, constructor transformation, field introduction, and method introduction. When applied to the non-synchronized Stack (Listing 3), these transformations yield a synchronized stack (Listing 6). The weaver associates a coordinator with a coordinatee by introducing a _coord field in the coordinated class (line 617), and adding an initialization statement in the constructor (line 604). The weaver also introduces public getter methods (_buf()) for protected and private fields of the coordinated class that need to be accessed by the coordinator.

The weaver transforms the coordinated methods by introducing calls to the coordinator’s lock_ and unlock_ methods before and after the original body. To ensure invocation of the unlock_ method, the weaver also introduces a try-finally block around the original body.

In sum, the COOL weaver realizes the COOLWeaver algorithm (Listing 7). Given a class file cf to be transformed, the weaver searches for its coordinator (findAspect, line 702). If found, the weaver introduces a coordinator field (addCoordField, line 704), transforms the constructors to initialize that field (transformConstructor, line 705), and generates getter methods for protected and private cf fields that are read by the coordinator (addGetterMethods, line 706).

---

Listing 6. A synchronized bounded stack

```java
public class Stack {
    public Stack(int capacity) {
        buf = new Object[capacity];
        _coord = new StackCoord();
    }

    public void push(Object obj) {
        _coord.lock_push(this);
        try {
            buf[ind] = obj;
            ind++;
        } finally { _coord.unlock_push(this); }
    }

    public Object pop() { /*omitted*/
        if (full) return null;
        return buf;
    }

    private Object[] _buf() { return buf; }

    private StackCoord _coord;
    private int ind = 0;
    private StackCoord() {
        ind = 0;
    }
}
```

Listing 7. The COOL weaver

```java
public void COOLWeaver(ClassFile cf) {
    ClassFile coordAspect = findAspect(cf);
    if (coordAspect!=null) {
        addCoordField(cf, coordAspect);
        transformConstructor(cf, coordAspect);
        addGetterMethods(cf, coordAspect);
        weaveClass(cf);
    }
}
```

The COOL weaver applies four kinds of transformations to a coordinated class, namely method transformation, constructor transformation, field introduction, and method introduction. When applied to the non-synchronized Stack (Listing 3), these transformations yield a synchronized stack (Listing 6). The weaver associates a coordinator with a coordinatee by introducing a _coord field in the coordinated class (line 617), and adding an initialization statement in the constructor (line 604). The weaver also introduces public getter methods (_buf()) for protected and private fields of the coordinated class that need to be accessed by the coordinator.

The weaver transforms the coordinated methods by introducing calls to the coordinator’s lock_ and unlock_ methods before and after the original body. To ensure invocation of the unlock_ method, the weaver also introduces a try-finally block around the original body.

In sum, the COOL weaver realizes the COOLWeaver algorithm (Listing 7). Given a class file cf to be transformed, the weaver searches for its coordinator (findAspect, line 702). If found, the weaver introduces a coordinator field (addCoordField, line 704), transforms the constructors to initialize that field (transformConstructor, line 705), and generates getter methods for protected and private cf fields that are read by the coordinator (addGetterMethods, line 706).
Then, the weaver synchronizes the methods of cf by imposing locking and unlocking advice before and after their bodies, respectively. Advice weaving in COOL follows the same four-process model as in AspectJ (call to `weaveClass`, line 707). In terms of this four-process model, a shadow in COOL is a method of cf, and the advice are the lock_ and unlock_ methods of the coordinator class.

The reify process of the COOL weaver represents an input class file as a set of methods (the reify method, line 711). The match process uses the signature of a yet-to-be-coordinated method to select a pair of lock_ and unlock_ advice methods (the match method, lines 713–722). For every coordinated method, the weaver finds the corresponding lock_ and unlock_ methods in the coordinator class (findLock, line 716; findUnlock, line 718). The order process of the weaver is empty (the order method, lines 724–726). Lastly, the mix process (mix, lines 728–733) introduces a call to the lock_ method before the method body (addCallBefore, line 730), and a call to unlock_ after the method body (addCallAfter, line 731).

Listing 8. A logger aspect in AspectJ

```java
public aspect Logger {
  pointcut scope(): !cf\(\text{within}(\text{Logger});

  before(): scope() {
    System.out.println("before " + thisJoinPoint);
  }

  Object around(): scope() {
    System.out.println("around" + thisJoinPoint);
    return proceed();
  }

  after(): scope() {
    System.out.println("after" + thisJoinPoint);
  }
}
```

4. Analysis and Design

Now that we have reviewed the working of a weaver, we move on to the main focus of this paper: the problem of composing aspect weavers. Our goal is to build a weaver composition framework for implementing a multi-extension AOP language. Given a set of aspect weavers and a composition specification, the framework should construct an appropriate multi-weaver. In the previous sections we discussed the four-process model of an abstract weaver. In this section we examine the composition specification; and then we derive the design requirements for the composition implementation.

The specification needs to resolve the feature interactions between the composed extensions. There are two main kinds of interactions that the specification should address, namely foreign advising and co-advising [28]. We reason about the specification by analyzing these interactions, using the composition of COOL and AspectJ as a running example.

4.1 Foreign Advising

A multi-extension AOP language is an AOP language that combines multiple aspect extensions. We call a program in this language a multi-extension program. In a multi-extension program, an aspect can generally interact with foreign aspects by advising join points in their execution. The foreign advising interaction determines how aspects in one extension advise foreign aspects in other extensions. Particularly, in a composition of COOL and AspectJ the foreign advising interaction controls the weaving of AspectJ advice into foreign COOL coordinators, and the weaving of COOL advice into foreign AspectJ aspects. For example, consider running a Logger aspect in AspectJ (Listing 8) together with our Stack coordinator in COOL (Listing 4). Logger logs all join points in a program execution, including join points within executions of the Stack coordinator. A resolution of the foreign advising interaction must determine what join points Logger advises within the Stack coordinator, and how. But neither the AspectJ nor the COOL specification define how AspectJ aspects advise COOL coordinators.

Foreign advising is not solvable by merely using a weaver for COOL (AspectJ) to weave the foreign aspects (coordinators), because the one language does not recognize the syntax or semantics of the other. Even though the weavers for COOL and AspectJ may both use Java classes as their intermediate representation, applying the COOL (AspectJ) weaver to the Java representation of foreign aspects (coordinators) will not do the job either. This is because the classes embed synthetic code that is generated during the translation to the intermediate representations, e.g., calls to wait and notifyAll in the coordinator class StackCoord (Listing 5). This synthetic code is specific to a particular implementation of the foreign compiler. The code cannot be traced back to the original source aspect, and exposing it to a foreign weaver may result in the latter advising it, thereby causing unexpected behavior in the program.

In terms of the abstract weaving process, foreign advising is a problem of representing foreign aspects correctly, and is the responsibility of the reify process. For example, the incorrect behavior observed in translation-based composition frameworks (e.g., XAspects and Reflex) is a result of the reify process of the framework’s weaver failing to provide a correct representation of the foreign aspect classes. Consequently, the weaver erroneously includes shadows also for implementation-specific operations that are introduced by the front-end translator into the intermediate aspect classes
4.2 Co-advising

In a multi-extension program, a join point can generally be advised by several aspects that are written in different extensions. We refer to this behavior as co-advising. The co-advising interaction controls the collaborative application of multi-extension advice at the same join point, which is undefined at the level of the individual extensions.

In a composition of COOL and AspectJ, the co-advising interaction coordinates the weaving of COOL and AspectJ advice into the same program element. For example, consider again running the Logger aspect in AspectJ (Listing 8) together with the Stack coordinator in COOL (Listing 4) and the Stack class in Java (Listing 3). The Logger and the Stack coordinator collaboratively advise executions of the Stack methods. A resolution of the co-advising interaction of the composition must determine in what order the pieces of advice of the aspect and the coordinator execute.

In terms of the abstract weaving process, co-advising is a problem of coordinating the match, order, and mix processes of the composed weavers. This problem cannot be resolved just by a sequential application of the individual weavers. If weavers were scheduled to run one after the other sequentially, then (at the same join point) advice that is applied later would always “wraps” around advice that is applied earlier. This would result in a very restrictive behavior that does not support the flexible ordering needed in general for co-advising. Moreover, a weaver may erroneously advise advice binding operations (e.g., calls to advice or coordinator methods) that were introduced into the shadow by previously applied weavers.

4.3 Resolving Feature Interactions

Our analysis clarifies why the feature interaction problem is so complex and difficult. First, in a composition of multiple extensions, the foreign advising interaction generally occurs between every pair of composed extensions. Furthermore, because the interaction involves features that are unique to the interacting parties, the behavior is specific to the composed extensions. For example, a foreign advising interaction between COOL and AspectJ involves terms, expressions, and constructs that are unique to COOL, e.g., requires, on_entry, and on_exit.

Moreover, there is no single correct way to resolve these interactions. For example, a foreign advising interaction between COOL coordinators and AspectJ aspects may allow the aspects to only advise access to fields of coordinated objects (that are made from within the coordinators), e.g., access to the field buf of the Stack class from within the Stack coordinator (Listing 4, line 410). Alternatively, the interaction can be resolved by letting the aspects advise all field access operations within the requires, on_entry, and on_exit expressions of a coordinator. It can also be resolved to allow the aspects to advise all field access plus (un)lock COOL computations (e.g., as advice-execution join points). Each of the three options can be advocated [24], and many more reasonable options exist.

4.4 System Design Requirements

Next, we use the terminology and abstractions from the analysis to formulate three design requirements for the composition implementation. The requirements are: decoupling, composability, and customizability.

4.4.1 Decoupling

The composition framework should decouple abstractions that are common to all AspectJ-compatible weavers from abstractions that are weaver-specific. This reduces the responsibility of the individual weaver to implementing only the extension-specific weaving operations. By reusing the framework’s abstractions as much as possible, the development of new weavers is drastically simplified.

4.4.2 Composability

The framework should support the composition of multiple aspect weavers into a default multi-weaver that resolves interactions automatically in a well-defined and reasonable way. This requirement targets the scalability problem that is inherent to the feature interaction problem. It enables an extendable and scalable solution to the problem of composing weavers.

We define the default multi-weaver behavior according to the following principles:

1. Preserving behavior of individual weavers: a default multi-weaver preserves the behavior of the individual composed weavers as observed when weaving their respective single-extension programs. For example, a multi-weaver for a composition of COOL and AspectJ would weave pure AspectJ (COOL) programs in exactly the same manner as a stand-alone AspectJ (COOL) weaver would have.

2. Default foreign advising. Syntactically, an aspect is a mixture of Java code and extension-specific terms. The default foreign advising behavior allows an aspect to advise a foreign aspect by advising Java statements within its source code, and only those statements. For example, a COOL coordinator embeds Java expressions within requires, on_exit, and on_entry constructs. In a default composition of COOL and AspectJ, AspectJ aspects advise COOL coordinators by advising only their Java expressions. Under this behavior, the Logger aspect (Listing 8) advises all field access join points within requires, on_exit, and on_entry expressions of the Stack coordinator (Listing 4).

3. Default co-advising. The co-advising behavior controls matching and ordering of multi-extension advice at a join point. The default matching policy is to unify the individual matching results of the composed extensions. The selected multi-extension advice include all pieces of
advice that match the join point under the semantics of their extensions. Each individual aspect weaver selects advice only from its own aspects, and does not interfere with the matching in foreign weavers.

The default ordering policy relies on the advice types being similar in all AspectJ-compatible extensions. An aspect advises a program by transforming computations at certain join points. We identify three types of transformations: to add advice before a join point computation, to add the advice after the computation, and to introduce the advice instead of (around) the computation. When the multi-weaver selects multi-extension advice at a join point, the default multi-extension ordering behavior is to run multi-extension before advice first, then multi-extension around advice, and finally the multi-extension after advice. For example, a default multi-weaver for a composition of COOL and AspectJ would order lock and unlock (COOL advice) to execute before and after AspectJ’s around advice, respectively. The multi-weaver also preserves a partial order of same-extension advice within the selected multi-extension advice.

4.4.3 Customizability

Although the default multi-weaver implements a reasonable behavior, a composition specification may define special foreign advising and co-advising behavior. For example, a foreign advising specification for a composition of COOL and AspectJ might choose to allow AspectJ aspects to advise COOL’s (un)lock computations as advice-execution join points. Hence, the framework must also allow the language designer to configure the multi-weaver to comply with the composition-specific foreign advising and co-advising specifications.

5. Aspect-Oriented Architecture

This section introduces a practical component-based and aspect-oriented architecture that facilitates the development of aspect weavers, and supports the integration of independently developed aspect weavers into a multi-weaver. We introduce the architecture in three steps. First, we explain the design decisions for decoupling extension-specific from common components. We refer to the extension-specific components as the aspect mechanism, and to the common components as the platform. The platform is implemented once; it provides facilities that, if reused, significantly ease the development of new weavers. Second, we present the design principles and decisions that allow the multiple aspect mechanisms and the platform to be automatically composed into a default multi-weaver. Third, we present a solution to the multi-weaver customizability problem. The architecture provides support for configuring the default multi-weaver to comply with a specialized composition specification.

5.1 Decoupling

We use the four-process weaver model [23] to identify the extension-specific and the common weaver components.

The reify process of a weaver constructs shadows for the base language classes and the extension aspects. As a part of its functionality, the process realizes a base representation function, i.e., a function that builds shadows for base program classes. Because the weaver is AspectJ-compatible, it can be realized using the base shadow domain and the base representation function of AspectJ. 10 Thus, the base shadow domain and the base representation function of AspectJ can be shared by all AspectJ-compatible weavers.

If a weaver’s extension does not allow an aspect to advise aspects (e.g., a coordinator in COOL cannot advise other coordinators), then the common base representation function realizes the weaver’s reify process in full. However, in the more general case (e.g., in AspectJ, an aspect can advise itself, as well as other aspects), a weaver needs to realize an extension-specific representation function, i.e., a function that builds shadows for aspects. The reify process of the weaver is thus a composition of the common base representation function and the specific representation function for aspects.

The match and order processes of a weaver are extension-specific. An individual weaver matches advice in its own aspects; and orders only extension-specific advice. The mix process weaves the ordered pieces of advice by transforming the shadow. Since an advice defines a shadow transformer function, mix can be modeled as a common extension-independent process that iteratively applies the advice transformers to the shadow.

Figure 3 depicts the design of an AspectJ weaver as a composition of common and AspectJ-specific components.

---

10 The base shadow domain of AspectJ includes all shadows except for advice-execution.
There are two main architectural parts: (a) a platform that provides common facilities; (b) a mechanism that implements extension-specific behavior. The platform’s behavior is realized by the Platform class. The mechanism is realized by an aspect that implements the Mechanism interface. The dashed arc in the figure denotes advising.

### 5.1.1 Platform
The platform provides the base shadow domain. Its methods `reify`, `mix`, and `weaveClass` implement the common weaver’s operations: `reify` uses base shadows to represent base classes; `mix` weaves advice at each shadow; and `weaveClass` implements the high-level weaving algorithm (Listing 1).

### 5.1.2 Mechanism
Each mechanism is realized as an aspect that implements the Mechanism interface by realizing the extension-specific matching and ordering processes via the implementation of the methods `match` and `order`, respectively. If a mechanism’s extension allows an aspect to advise other aspects, then the mechanism realizes the extension-specific representation function as an `around` advice that refines executions of the platform’s `reify` method. The methods and the advice are implemented with the following conception: the mechanism uses base shadows as a representation domain for base classes; the advice uses base and extension-specific shadows as a representation domain for aspects; and the advice defer to the platform the representation of base classes.

### 5.2 Composability
The architecture supports the compositions of multiple aspect mechanisms into a multi-weaver. Figure 4 illustrates the extended architecture by showing a default multi-weaver for a composition of COOL and AspectJ. In the extended architecture, the multi-weaver is realized by the platform. The platform mediates between the composed mechanisms, and manages their collaborative application. Furthermore, the Mechanism interface is replaced with the abstract Mechanism aspect, which defines the abstract methods `match` and `order`. In addition, the aspect advises the Platform’s constructor to register the mechanisms with the platform.\(^\text{12}\)

At an abstract level, the multi-weaver implements the same high-level weaving process as a stand-alone weaver (Listing 1). The four subprocesses of the multi-weaver are built by integrating and unifying the corresponding processes of the individual weavers. The `reify` process of the multi-weaver represents base classes, and aspects that are written in different extensions. The `match` and `order` processes of the multi-weaver select and order multi-extension advice, respectively. In the extended architecture, Platform provides the methods `match` and `order` to realize these processes. The `mix` process weaves the ordered multi-extension advice by transforming the shadow.

We enable composability of aspect mechanisms by introducing additional design principles:

### 5.2.1 Mandatory Aspect Representation
To enable a default foreign advising behavior, an aspect mechanism must realize an extension-specific aspect representation function, even if the function is not normally required for its own stand-alone operation. This policy ensures that a multi-weaver builds shadows for all aspects in a multi-extension program, thus letting an aspect observe and advise Java shadows in any foreign aspect. For example, a stand-alone weaver for COOL does not advise coordinators, and thus does not need to represent them. A multi-weaver for a composition of COOL and AspectJ, in contrast, uses the COOL representation function for exposing the coordinators to AspectJ aspects. The aspects can then advise Java shadows within the coordinators.

Intuitively, the aspect representation function provides the most fine-grained representation of an aspect that includes all base shadows for its Java fragments, and ded-

\(^{11}\) Extension-specific shadows represent constructs, declarations, and expressions that are specific to the extension, e.g., the advice-execution shadow in AspectJ.

\(^{12}\) This behavior could have been realized in an object-oriented manner, but AOP enables a more elegant design.
icated extension-specific shadows for all the extension-specific computations. For example, a function for representing coordinators must build shadows for all Java fragments within a coordinator’s code (e.g., Java expressions within a requires statement), and on_entry, on_exit, requires, lock, and unlock shadows for all the respective computations.

5.2.2 Parallel Matching of Multi-extension Advice

To enable a default multi-extension advice matching behavior, the match method of the platform should run the match methods of the composed mechanisms in parallel. The multi-extension advice selected at a shadow is then a list of extension-specific advice sets that are produced by the composed aspect mechanisms.

5.2.3 Uniform Advice Types

To enable a default multi-extension advice ordering behavior, an aspect mechanism should partition advice into three ordered sets, namely before, around, and after. In terms of the architecture, the order method of a mechanism returns a list of three advice arrays, the first contains before advice, the second contains around advice, and the third contains after advice. The platform’s order method then runs the order methods of the composed mechanisms in parallel, and linearizes their results into a single advice vector in accordance with the default multi-extension ordering policy.

A composition of the aspect mechanisms with the platform produces a multi-weaver with a default behavior. The aspect representation principle enables a default foreign advising behavior, and the other principles enable a default co-advising behavior. The multi-weaver uses as its common shadow domain the union of the common base shadow domain and all the extension-specific shadow domains. It represents the multi-extension aspects and the base classes as appropriate for the composed mechanisms. It has a well-defined multi-extension advice matching and weaving behavior; and it uses the order method of the platform for ordering advice.

5.3 Customizability

Of course, the default behavior of the multi-weaver may differ from the actually desired one. The specification may dictate foreign advising and co-advising rules that involve several extensions. For example, the specification may require an aspect in AspectJ to advise executions of lock and unlock in COOL as advice-execution join points. Generally, the foreign advising rules alter the semantics of the individual extension for advising foreign aspects. The co-advising rules specify a collaborative behavior for multi-extension aspects that advise the same join point. These rules are composition-specific and thus cannot be defined on the level of an individual extension.

To this end, the architecture provides a Config aspect that customizes the behavior of the default multi-weaver (Figure 5). The configuration aspect implements the composition-specific foreign advising behavior by extending and overriding the match and order methods of the aspect mechanisms. The aspect specializes the co-advising behavior by advising the match and order methods of the platform.

The architecture thus supports the construction of a multi-weaver with a customized behavior. The multi-weaver reify method recognizes and represents properly aspects of all the composed extensions using a common shadow domain. The adapted match and order methods of the individual mechanisms select and order extension-specific advice according to the foreign advising specification. The customized multi-weaver match and order methods select and order the multi-extension advice in accordance with the co-advising specification.

In sum, the architecture (Figure 5) establishes the fundamental principles for designing composable aspect mechanisms. In Section 6 we apply these principles to build a concrete co-weaving system for composing multiple extensions (AWESOME). Using AWESOME, we then implement an aspect mechanism for COOL, another for AspectJ, and then combine the two to produce an AWESOME weaver for the AOP language COOLAJ, which is described in Section 7.
6. Implementation by Refactoring AspectJ

As a proof of concept, we realized the weaving system and the mechanisms by refactoring the ajc compiler and the COOL weaver. In the ajc code, shared and AspectJ-specific operations are intertwined. Through refactoring we untangled and separated these two kinds of operations, moving the ones in common to the Platform and modularizing the rest in the AspectJ mechanism. The weaver for COOL, on the other hand, uses methods to represent Java classes, and does not have a representation for coordinator classes. In our system, we use AspectJ shadows (with the exception of advice-execution shadows) as a base shadow domain. The refactoring here involved the use of method-execution shadows for advice matching and weaving; and providing a shadow representation for the coordinator classes.

6.1 Implementing a Platform

The platform is realized by the Platform class. A list of plugged aspect mechanisms is stored in the mechanisms instance variable. weaveClass is a TEMPLATE METHOD [15] that implements the weaving process (Listing 1) using reify, match, order, and mix.

```java
public abstract Advice[][] match(Shadow shadow) {
    Advice[][] result = new Advice[mechanisms.length][0];
    for (int i=0;i<mechanisms.length;i++)
        result[i] = mechanisms[i].match(shadow);
    return result;
}
```

The `match` method selects advice by calling its `match` counterparts in the individual mechanisms:

```java
public Advice[][] match(Shadow shadow) {
    Advice[][] result = new Advice[mechanisms.length][0];
    for (int i=0;i<mechanisms.length;i++)
        result[i] = mechanisms[i].match(shadow);
    return result;
}
```

The `order` method calls the `order` methods of the individual mechanisms, and then linearizes the multi-extension advice:

```java
public Advice[] order(Shadow shadow, Advice[] multiAdvs) {
    Advice[] bfAdv = new Advice[0];
    Advice[] ardAdv = new Advice[0];
    Advice[] afAdv = new Advice[0];
    for (int i=0;i<mechanisms.length;i++) {
        Advice[] mechAdvs = mechanisms[i].order(shadow, multiAdvs);
        bfAdv = addAll(bfAdv, mechAdvs[0]);
        ardAdv = addAll(ardAdv, mechAdvs[1]);
        afAdv = addAll(afAdv, mechAdvs[2]);
    }
    return addAll(ardAdv, addAll(bfAdv, afAdv));
}
```

where `addAll` is an auxiliary method that takes two argument arrays, and concatenates them by appending the second one to the first one. The `order` method schedules `around` advice to be woven first, so that `before` and `after` advice “wrap” any `around` advice at the same shadow. Note that weaving order is not the same as execution order.

Finally, the `mix` method sequentially applies advice transformers to the shadow:

```java
void mix(Shadow shadow, Advice[] advs) {
    for(Advice a:advs) a.transform(shadow);
}
```

The transform method integrates the advice instructions into the shadow.

6.2 An Abstract Aspect Mechanism

We implemented the mechanisms for AspectJ and COOL as aspects that extend the abstract SINGLETON [15] aspect, named Mechanism:

```java
public abstract aspect Mechanism {
    after(Platform mw):
        initialization(Platform.new(..)) && this(mw) {
            mw.mechanisms = addAll(mw.mechanisms, new Mechanism[] {this});
        }
    public abstract Advice[] match(Shadow shadow);
    public abstract Advice[][] order(Shadow shadow, Advice[] advs);
}
```

The `after` advice ensures that aspect mechanism instances are created and plugged into the platform as soon as the platform is instantiated. The concrete mechanisms (AJMechanism and COOLMechanism) provide an implementation for `match` and `order` and override the `Platform.reify` method by advising it with `around` advice.

6.3 Implementing an AspectJ Mechanism

The AJMechanism aspect advises the representation `reify` method of the platform. If the argument class is an AspectJ aspect, then the advice provides a shadow representation for it; otherwise, the advice proceeds:

```java
Shadow[] around(ClassFile cf): args(cf) && execution(Shadow[] Platform.reify(ClassFile)) {
    return isAJAspect(cf) ? reifyAspect(cf) : proceed(cf);
}
```

where isAJAspect determines whether or not the argument class represents an aspect; and reifyAspect constructs shadow representation of the aspect class. The aspect representation includes AspectJ-specific advice-execution shadows.

The `before` advice to the `weaveClass` method introduces to the multi-weaver an intertype declaration mechanism:
before(ClassFile cf): args(cf) 
   execution(void Platform.weaveClass(ClassFile)) {
      applyIntroductions(cf);
   }

The match and the order methods are copied from the original code. We omit them, as well as isAJAspect and reifyAspect, due to space considerations.

6.4 Implementing a COOL Mechanism

The refactoring of the COOL mechanism includes a change to the front-end for translating source COOL coordinators into annotated Java classes. The annotations mark the lock_ and unlock_ methods of the coordinator class and identify the requires, on_entry, and on_exit instruction blocks within these methods.

The COOL mechanism introduces shadow types for lock, unlock, requires, on_entry, and on_exit computations. The lock and unlock shadows represent executions of the lock_ and unlock_ methods. The requires, on_entry, and on_exit shadows represent executions of the corresponding COOL expressions and statements. They map to blocks of instructions within the lock_ and unlock_ methods. The bodies of the requires, on_entry and on_exit constructs are Java expressions and statements. The mechanism represents them using the base shadow domain (field-get and field-set shadows).

The COOLMechanism aspect advises the weaveClass and the reify methods of the platform. The after advice to the weaveClass method introduces a shadow (target) class a coordinator field and getter methods, and transforms the constructor of the class:

after(ClassFile cf): args(cf) 
   execution(void Platform.weaveClass(ClassFile)) {
      ClassFile coordAspect = findAspect(cf);
      if (coordAspect!=null) {
         addCoordField(cf, coordAspect);
         transformConstructor(cf, coordAspect);
         addGetterMethods(cf, coordAspect);
      }
   }

The advice around the reify method is similar to the corresponding advice in the AjMechanism aspect: if the argument class is a COOL coordinator class, then the advice provides a shadow representation for it; otherwise, the advice proceeds.

The COOL mechanism also provides an implementation for match and order. match selects lock and unlock pieces of advice by matching the coordinator classes against the method-execution shadows. The order method schedules the lock advice to run before the unlock advice.

7. Case Study: An AWESOME Weaver for COOLAJ

Plugging the AjMechanism and the COOLMechanism aspects into the composition Platform produces a multi-weaver with a default behavior. It lets aspects advise join points within requires, on_entry, and on_exit expressions of coordinators. It lets coordinators synchronize methods that are defined within aspects; and it allows coordinators and aspects to co-advice the same method. Although this default behavior is reasonable, a specific multi-extension composition of AspectJ and COOL may require different semantics. In this section we specify such a multi-extension AOP language named COOLJ. We implement a weaver for COOLJ by customizing the default multi-weaver using the Config aspect.

7.1 Informal Specification for COOLJ

The specification for COOLJ is independent of the AWESOME architecture. It is based only on the syntax and semantics of the AspectJ and COOL languages; not on their implementation. COOLJ is specified as a conservative composition of AspectJ and COOL, i.e., it follows as much as possible the original semantics of Aspect and COOL. Specifically, in COOLJ an aspect is woven into classes and aspects according to the weaving semantics of AspectJ. Similarly, a coordinator is woven into classes according to the weaving semantics of COOL. The specification for COOLJ differs from the default behavior when it comes to dealing with foreign advising and co-advising:

7.1.1 Foreign Advising

In COOLJ, aspects advise executions of coordinators through field-get and field-set join points that are located within requires, on_entry, and on_exit expressions; and through advice-execution join points that represent lock and unlock computations (i.e., executions of lock_ and unlock_ methods of the COOL coordinator classes).

The foreign advising specification poses several restrictions on advising join points within a coordinator:

• In the COOLJ specification, an access (read or write) to a condition field can only be advised with before or after advice. This way aspects cannot override values of these fields, but are still able to observe their access patterns. This restriction is important for protecting the synchronization logic of a coordinator.

• An execution of a lock or an unlock computation is advisable by aspects as an advice-execution join point. However, aspects are limited to advising these join points with before and after advice only. This restriction ensures that the locking and unlocking operations imposed by coordinators are not overridden by aspects, and always apply in the correct order.

The specification permits coordinators to advise methods that are declared within aspects in the same way as methods within classes. The specification restricts coordinators (aspects) from advising any synthetic code introduced by the foreign mechanism, e.g., coordinators do not advise advice methods in aspect classes, and aspects do not advise getter
methods that are introduced into the coordinated classes by the COOL mechanism.

7.1.2 Co-advising

The COOLAJ co-advising specification coordinates the collaborative application of aspects and coordinators on the same program method:

- The lock (unlock) advice of COOL is executed before (after) the before, around, and after advice of AspectJ.
- From the perspective of AspectJ aspects, COOL advice executes in the control flow of the method execution join point it advises.

**Example** For illustration, consider the LogAdviceOnStack aspect (Listing 9). Under the semantics of AspectJ, the aspect logs all advice (except for its own) woven at Stack method-execution shadows. The

```java
Listing 9. LogAdviceOnStack

```{r}
public aspect LogAdviceOnStack {
  pointcut scope():
    !cflow(within(LogAdviceOnStack));
  pointcut tgt(): execution(* Stack.*(..));
  before(): scope() &&
    cflow(tgt()) && !cflowbelow(tgt()) {
      System.out.println(thisJoinPoint);
    }
}
```

pointcut selects not only tgt() join points, but also join points within aspects that advise the tgt() join points. In particular, LogAdviceOnStack would advise join points within the Logger aspect (Listing 8), if the two are used together with the Stack class (Listing 3). If this aspect is run together with the Stack class (Listing 3) and the Stack coordinator (Listing 4) under the semantics of COOLAJ, then: (1) it logs executions of the coordinator; and (2) an execution of the LogAdviceOnStack advice that prints the method-execution join point is synchronized by the coordinator (along with the original method body).

7.2 Customization

We realized a multi-weaver for COOLAJ by providing a Config aspect with three pieces of advice: one implementing the co-advising rules, and the other two realizing the foreign advising rules.

7.2.1 Customizing Co-advising

The AspectJ weaver realizes the semantics of the cflow pointcut designator by introducing special advice at program shadows. Specifically, every shadow that matches an argument pointcut of a cflow pointcut is wrapped with a CFlowPush advice and a CFlowPop advice. The CFlowPush advice runs before any other advice at a join point, and pushes the join point on the AspectJ’s join point stack. CFlowPop runs after all the other advice and pops the join point off the stack. These advice thus mark the start and the end of a join point’s control flow. For example, the pointcut of the LogAdviceOnStack’s advice causes the AspectJ weaver to weave the CFlowPush and CFlowPop pieces of advice at the Stack.push method-execution shadow.

Config implements the co-advising rules of COOLAJ by advising the Platform.order method. The advice orders COOL advice to be woven between the CFlowPush and CFlowPop advice, but around any other AspectJ advice:

```java
private Advice[] mvAdv(Advice[] advs, Class fromAdv, Class toAdv) {
  int fromPos = elTypePos(advs, fromAdv);
  if (fromPos < 0) return advs;
  int toPos = elTypePos(advs, toAdv);
  if (toPos < 0) toPos = advs.length;
  if (fromPos < toPos) toPos--; //
  return move(advs, fromPos, toPos);
}
```

where LockAdv and UnlockAdv classes respectively implement lock and unlock advice of COOL; elTypePos returns a first position of an object of a given class in the array; and the move method moves an element of an array from one position to another. Specifically, move(advs, fromPos, toPos) moves an element at the position fromPos of the advs array to the position toPos, and shifts elements between fromPos (exclusively) and toPos (inclusive) to the left, if fromPos < toPos, or to the right, if fromPos > toPos. mvAdv is an auxiliary method that co-orders COOL and AspectJ advice.

If the multi-extension advice array contains no COOL advice then Config does not affect it. Otherwise, if the array contains the cflow advice then Config orders LockAdv (UnlockAdv) to be woven immediately before CFlowPush (CFlowPop), so that at run time the LockAdv (UnlockAdv) advice runs immediately after (before) the CFlowPush (CFlowPop) advice. If the array contains no cflow advice, then COOL advice is scheduled to be woven the last, thus dominating the AspectJ advice at run-time.

7.2.2 Normalizing Shadow Types

To allow aspects to advise lock and unlock computations as advice-executions, Config normalizes lock and unlock shadows of COOL with advice-execution shadows of AspectJ.
by advising calls to the \texttt{match} and \texttt{order} methods of the \texttt{AJMechanism} aspect:


define advice [] around (Shadow shadow): 
    args (shadow) &&
    (call (Advice[] \texttt{AJMechanism.match}(..)) ||
     call (Advice[] \texttt{AJMechanism.order}(..))) 
    
    return proceed (isLockOrUnlock (shadow) ?
        maskAsAExec (shadow) : shadow);

where \texttt{isLockOrUnlock} tests if shadow is a \texttt{lock} or \texttt{unlock} shadow, and \texttt{maskAsAExec} masks the \texttt{C\textsc{ool}} shadow as an \texttt{advice-execution} shadow. As a result, \texttt{AspectJ} advises the \texttt{lock} and \texttt{unlock} shadows as if they were \texttt{advice-execution} shadows.

### 7.2.3 Restricting Advisability

\texttt{Config} restricts the advising of join points within \texttt{C\textsc{ool}} coordinators by advising the executions of the \texttt{AJMechanism.match} method:


define advice [] around (Shadow shadow): 
    args (shadow) &&
    execution (Advice[] \texttt{AJMechanism.match}(..)) 
    
    Advice[] advs = proceed (shadow); 
    if (isCondFieldAccess (shadow) ||
        isLockOrUnlock (shadow))
        advs = removeElType (advs, AroundAdvice.class); 
    return advs;

where \texttt{isCondFieldAccess} checks if the shadow represent access to a condition field of a \texttt{C\textsc{ool}} coordinator, and \texttt{removeElType} removes all elements of a given type from the array.

### 8. Evaluation

To evaluate our approach, we integrated our multi-weaver framework with the \texttt{ajc} AspectJ compiler. The main \texttt{ajc} class runs the front-end and eventually weaves bytecode classes by invoking the \texttt{weave} method on the \texttt{org.aspectj.weaver.bcel.BcelClassWeaver} class. We modified this method to call instead the \texttt{Platform.weaveClass} method for weaving. This permitted to “plug” a specific multi-weaver into \texttt{ajc} by putting a corresponding implementation of the \texttt{Platform} class on the class path, and running the AspectJ compiler as usual. We evaluated the pluggability, correctness, and performance.

#### 8.1 Third-party Composition

First, we evaluated the pluggability feature of \texttt{AWESOME} by constructing three different weavers from the same building blocks. The building blocks are jar files containing compiled aspects and classes. \texttt{platform.jar} is the stripped down platform containing the \texttt{Platform} class and the abstract \texttt{Mechanism} aspect. The jars \texttt{ajm.jar} and \texttt{coolm.jar} contain the concrete independently developed aspects \texttt{AJMechanism} and \texttt{COOLMechanism} for \texttt{AspectJ} and \texttt{COOL}, respectively. \texttt{config.jar} contains the \texttt{Config} aspect for customizing the composition of \texttt{COOL} and \texttt{AspectJ}.

We verified that it is possible, using the command line, to construct weavers for \texttt{AspectJ}, \texttt{COOL}, and \texttt{COOLAJ} from the four building blocks. We constructed a stand-alone \texttt{AspectJ} weaver, named \texttt{ajw}, by plugging just the \texttt{AspectJ} mechanism into the platform. The command line is (Figure 6):

\begin{verbatim}
ajc -inpath platform.jar;ajm.jar -outjar ajw.jar
\end{verbatim}

where \texttt{ajc} is the original (non-refactored) version of the \texttt{AspectJ} compiler. The \texttt{inpath} option directs \texttt{ajc} to weave classes within jar files. The \texttt{outjar} option directs the compiler to save the woven classes into a separate jar file. To construct a stand-alone \texttt{COOL} weaver, named \texttt{coolw}, we plugged only the \texttt{COOL} mechanism (Figure 7):

\begin{verbatim}
ajc -inpath platform.jar;coolm.jar -outjar coolw.jar
\end{verbatim}

and to construct a multi-weaver, named \texttt{awesomew}, that combines \texttt{AspectJ} and \texttt{COOL}, we ran (Figure 8):

\begin{verbatim}
ajc -inpath platform.jar;ajm.jar;coolm.jar;config.jar -outjar awesomew.jar
\end{verbatim}

To compile and run a multi-extension aspect program, a file with unwoven bytecode \texttt{unwoven.jar} (an unwoven program including aspect and base classes) was passed to the multi-weaver to produce a woven file:

\begin{verbatim}
java -cp <weaver>.jar;aspectjtools.jar
org.aspectj.tools.ajc.Main -inpath unwoven.jar -outjar woven.jar
\end{verbatim}
8.2 Testing

Second, we tested the three weavers to determine with high confidence that indeed \texttt{ajw} implements the semantics of AspectJ; \texttt{coolw} implements the semantics of COOL; and \texttt{awesomew} realizes the specification for COOLAJ. We did this by observing the runtime behavior of test programs; by inspecting their woven bytecode; by analyzing join point traces; and, when possible, by comparing the results to programs compiled with \texttt{ajc} or \texttt{abc} [5]. Because the framework is based on \texttt{ajc}, which is assumed correct, we focused our tests on a coverage of the newly introduced and refactored behavior.

8.2.1 Testing \texttt{ajw}

The \texttt{ajw} weaver can be evaluated by comparing \texttt{ajw}-woven to \texttt{ajc}-woven bytecode. In fact, the main difference between \texttt{ajw} and \texttt{ajc} is in the design and implementation of the \texttt{reify} process. In the implementation of \texttt{ajw} we disentangled the monolithic \texttt{reify} process of \texttt{ajc} into a common platform \texttt{reify} method and an AspectJ-specific advice of the AspectJ mechanism. The other processes were either left unchanged (e.g., \texttt{match} and \texttt{order}), or undergone a coarse-grained (and assumed behavior-preserving) transformations (e.g., \texttt{mix}). Thus, we hypothesize that \texttt{ajw} is a behavior-preserving refactoring of \texttt{ajc}, if the \texttt{reify} processes of the two exhibit the same behavior, i.e., given a Java class or an AspectJ aspect they build identical shadow representations.

To test the \texttt{reify} process and reason about its shadow representation, we generated an exhaustive join point trace by weaving together three classes and aspects: \texttt{Stack.java} (Listing 3); \texttt{LogAll.aj}, and \texttt{TouchAll.aj}. The \texttt{LogAll} aspect advises with \texttt{before}, \texttt{around}, and \texttt{after} advice all the join points in the program, except those within the aspect itself (to prevent an infinite loop), and \texttt{logs} the join points to a file. \texttt{TouchAll} also advises everything but itself, but just “touches” the join points with an empty advice.

The woven bytecode is run by a main program that creates a \texttt{Stack} object and invokes \texttt{push} and \texttt{pop} in a single thread. The execution produces an exhaustive trace of the join points within \texttt{Stack} and within \texttt{TouchAll}. This trace provides a good insight into behavior of the \texttt{reify} process, because it covers almost all types of join points and includes join points within both Java classes and AspectJ aspects.

We executed \texttt{ajw}-woven and \texttt{ajc}-woven bytecode using the same test program and obtained identical join point traces. We therefore conclude that, at least on this benchmark example, \texttt{ajw} behaves the same as \texttt{ajc}, and is likely to exhibit an \texttt{ajc}-equivalent behavior in general.

8.2.2 Testing \texttt{coolw}

We tested whether the runtime behavior of \texttt{coolw}-woven bytecode complies with the dynamic semantics of COOL [27] (i.e., if multiple threads are properly synchronized on the woven target methods). We compiled together \texttt{Stack.java} (Listing 3) and \texttt{Stack.cool} (Listing 4), which employs all features of COOL (i.e., \texttt{selfex}, \texttt{mutex}, \texttt{requires.on_entry}, \texttt{on_exit}, and access to \texttt{private} field of a coordinated class from a coordinator aspect). As part of our tests we inspected the \texttt{StackCoord.java} file that was constructed by the COOL front-end from \texttt{Stack.cool}, and we inspected the \texttt{coolw}-woven bytecode that was produced by weaving the \texttt{StackCoord} coordinator class into the \texttt{Stack Java} class.

We ran the woven program and observed its runtime behavior. The test program created a \texttt{Stack} instance with a very small capacity (size of 5), and invoked its methods concurrently by five reader and five writer threads. A reader thread attempted to remove 5000 objects from the stack, while a writer thread attempted to add 5000 objects onto the stack. The test program completed successfully (i.e., executed all the threads to completion without throwing an exception), indicating, with a high probability, that the behavior of the woven bytecode is correct. Additional inspection of traces verified that the stack was properly synchronized.

We tested the front-end translator by comparing the generated \texttt{StackCoord.java} against the manual translation presented in Listing 5. \texttt{StackCoord} is said to pass the test, if we can conclude that its \texttt{lock} and \texttt{unlock} methods encode the same behavior as the corresponding \texttt{lock} and \texttt{unlock} methods in Listing 5. We tested \texttt{coolw} by comparing the bytecode of the woven \texttt{Stack} class against the the manually-woven class presented in Listing 6.

All three tests succeeded. Our COOL implementation exhibited the correct dynamic and compilation semantics on the input program that comprised \texttt{Stack.java} and \texttt{Stack.cool}. We consider the input program to be a representative COOL application since it uses all the features in COOL. We thus conclude with a high degree of confidence that our COOL implementation would generally weave COOL programs correctly.

8.2.3 Testing \texttt{awesomew}

We hypothesize that \texttt{awesomew} implements correctly the semantics of COOLAJ. To test this hypothesis we verified that:

1. Given a program with only aspects and classes as input, \texttt{awesomew} weaves the program according to the semantics of AspectJ;
2. Given a program with only coordinators and classes as input, \texttt{awesomew} weaves the program according to the semantics of COOL;
3. Given a program with coordinators, aspects and classes as input, \texttt{awesomew} weaves the coordinators into their
matching classes according to the semantics of COOL; weaves the aspects into classes and other aspects according to the semantics of AspectJ; weaves aspects into coordinators according to the foreign advising specification; and coordinates the weaving of multi-extension advice according to the co-advising specifications of COOLAJ.  

The first two cases were validated using the same testing strategy as for ajw and coolw and by comparing the output of awesomew to that of ajw and coolw. The details are omitted. To validate the foreign advising and co-advising behavior in the third case, we compiled Stack.java, Stack.cool, LogAll.aj, and TouchAll.aj, and we tested the woven bytecode.

We verified the weaving of aspects into coordinators by inspecting the join point trace within the control flow of the StackCoord class. awesomew correctly weaves aspects into coordinators, if the trace complies with the COOLAJ foreign advising specification, (which defines the shadow representation of COOL coordinators; the normalization between COOL and AspectJ shadow types; and mapping between AspectJ advice types and coordinator-located shadows). In particular, we verified that the trace contains only expected join points, that it reflects executions of lock_ and unlock_ methods as advice-execution join points, and that around advice of TouchAll and LogAll are properly filtered (e.g., not applied at advice-execution join points).

We also tested the ordering of multi-extension advice on a program that contains Stack.java, Stack.cool, LogAdviceOnStack.aj (Listing 9), and TouchAll.aj. The execution trace of a Stack method reflected that: (1) execution of StackCoord is advised by LogAdviceOnStack; and (2) the first and the last advice-execution join points around a Stack method execution join point that are not in the control flow of LogAdviceOnStack are executions of StackCoord, lock_, and unlock_, respectively. The first result shows that StackCoord advice executes in the control flow of the join point it advises. The second result shows that StackCoord advice takes precedence over AspectJ advice at the same join point.

awesomew passed all these tests. We verified that awesomew weaved our input programs according to the COOLAJ semantics. The input programs provide a good coverage of the COOLAJ specification. Therefore, we conclude with high confidence that awesomew performs a correct weaving of COOLAJ programs.

8.3 Performance

Finally, we evaluated the runtime performance of the bytecode produced by the framework weavers. This is intended to verify that the quality of woven bytecode is unaffected by the improved design of the weaver. Specifically, we validated the following hypotheses:

1. The runtime performance of ajw-woven bytecode is the same as ajc-woven;
2. The runtime performance of coolw-woven bytecode is the same as coolx-woven;
3. The runtime performance of awesomew-woven bytecode is the same as:
   (a) ajc-woven, for AspectJ programs;
   (b) coolx-woven, for source COOL programs;
   (c) ajcoolx-woven, for COOLAJ programs.

where coolx and ajcoolx are weaving algorithms for COOL and COOLAJ that are applied manually.

We ran a multi-threading COOL program that creates a Stack object and invokes its methods using ten writer-reader threads. A writer-reader thread performs 5000 pairs of push-pop operations. The test program reported the average running time of a thread (in milliseconds) over series of 10 runs. We also ran a single-threaded AspectJ program that created a Stack object, and invoked its push and pop methods 5000 times each. We measured the average running time of executing the operations over a series of 10 runs.

Figure 9 summarizes the measured execution times. Programs compiled with ajw and coolw are as efficient as those compiled with ajc and coolx. Programs compiled with awesomew are within 4% efficiency compared to (optimal) code woven mechanically using the ajcoolx algorithm and ajc as a back-end compiler. This indicates that the framework design overhead on the performance of the woven bytecode is negligible, i.e., there is almost no overhead to supporting the plugin architecture.

9. Related Work

The vision of domain specific aspect-oriented extensions dates back to early days of AOP (e.g., [8, 9]), but very few of the related works deal with making such extensions available concurrently. Those that do, leave much to be desired in terms of composability, customizability, and efficiency.

---

17 Coordinators never weave other coordinators.
Reflex [37] and XAspects [36] do not support the level of composability or customizability that is necessary for resolving foreign advising. These frameworks implement the composition by translating source aspects in foreign extensions to aspects in a common target language. The translation introduces and exposes in the target aspects synthetic join points that do not exist in the source. However, in Reflex and in XAspects, foreign aspects cannot distinguish the synthetic from the genuine join points. Moreover, Reflex and XAspects provide no mechanism (or composition rules) for customizing the foreign advising behavior, thus preventing the integrator from being able to correct the faulty resolution of this feature interaction. As a result, aspect programs compiled in Reflex- and XAspects-based multi-extension weavers may exhibit incorrect behavior [22, 24].

In contrast, AWESOME provides automatically a default reasonable resolution of foreign advising, thus significantly simplifying the problem of resolving the interactions. The integrator may fine-tune the default behavior, but does not necessarily need to.

In Reflex there is ample support for configuring co-advising at the aspect level. A programmer can resolve interactions between aspects in a specific aspect program. AWESOME, on the other hand, supports customizability at the language level. A language designer can resolve the interactions between aspect extensions, thus affecting the behavior of all multi-extension programs. Extending AWESOME with aspect-level support for fine-tuning co-advising is a topic for future work.

Pluggable AOP [22] is a third-party composition framework that supports the composition of dynamic aspect mechanisms into an AOP interpreter. In Pluggable AOP, an aspect mechanism is a transformer of an AOP interpreter. Among the related frameworks, only Pluggable AOP addresses foreign advising by treating a foreign aspect mechanism as an open module [1] that can determine which join points within its aspects are advisable and which are hidden. However, there is no control in Pluggable AOP over how these join points are advised. Pluggable AOP is also restricted in its co-advising customizability. It allows the integrator to customize the co-advising behavior only indirectly by ordering the aspect mechanisms. Moreover, Pluggable AOP is not designed for efficiency. It is impractical for use in industrial settings.

In comparison, AWESOME supports flexible language-level customizability of both the co-advising and the foreign advising interactions; and employs an efficient compile-time weaving scheme.

AWESOME was demonstrated to successfully compose real-world extensions. Pluggable AOP, in contrast, uses “toy” languages as a proof of the concept. To the best of our knowledge, Reflex has not been shown to work with AspectJ. We only found a plugin that implements a limited subset of AspectJ. The plugin, however, does not advise AspectJ aspects correctly, emphasizing the general limitation of the Reflex framework to support foreign advising [24].

The composition of COOL [27] and AspectJ [20] presents an interesting case study with a representative complexity. The two are sufficiently different, thus demanding interesting design decisions to make them work together. Similar to AWESOME, XAspects too explored a composition of COOL and AspectJ. However, the XAspects weaver exhibits incorrect weaving behavior that may result in deadlock [22].

AWESOME is not limited to COOL and AspectJ. It can be generally applied to combine any reactive aspect mechanisms. To the best of our knowledge, all existing join point and advice aspect extensions fall into this category. Other more disparate aspect-oriented extensions are either non-reactive aspect mechanisms, which are not composable, e.g., Hyper/J [34, 38], or they are not “oblivious” [7, 13, 14] and can be composed trivially, e.g., Demeter [25, 26]. Aspect-oriented features other than advising, such as introductions, are easier to compose because it is easier to detect and resolve conflicts [16].

The problem of feature interactions in a multi-extension composition does not arise in the context of a single-extension AOP language. Therefore, related works on single-extension weavers [3, 5, 19, 29, 30, 39, 42] do not address or solve this problem.

10. Conclusion

This work studies the composition implementation problem in constructing multi-extension weavers. We present a practical third-party composition framework, named AWESOME, for composing multiple aspect extensions. The AWESOME framework was built systematically. It implements a specified set of composition requirements. It provides a default resolution of feature interactions in the composition. It also provides means for customizing the default resolution to comply with a given composition specification. AWESOME was tested and evaluated on real-world aspect languages. The runtime performance of compiled aspect programs is practically unaffected by the extensible design of the framework, making AWESOME also useful in practice.

AWESOME is unique in its approach to composing aspect extensions by assembling an aspect compiler. In AWESOME, an aspect mechanism is a plugin to the compile-time aspect weaver. The AWESOME framework simplifies the creation of new extensions, because writing a plugin is much simpler than writing a complete compiler. In order to evaluate our approach, we refactored the AspectJ ajc compiler and used it as a basis for our multi-weaver platform. But the refactoring of open source compilers is not a part of the integration methodology.

The ajw, coolw, and awesomew weavers themselves are also a modest contribution of this work. These AspectJ, COOL, and COOLAJ compilers do not just have a cool design, but also awesome performance.
References


