ABSTRACT

“Two-Way” Obliviousness in General Aspect-Oriented Modeling

Nathan V. Roberts, M.S.

Mentor: Eunjee Song, Ph.D.

A key problem in software development is producing systems that are maintainable even as the concerns at play evolve. Aspect-oriented programming (AOP) seeks to foster maintainability by isolating the specifications of cross-cutting concerns, allowing them to be modified in relative isolation from the rest of the system. Research in aspect-oriented modeling (AOM) aims to develop a model-layer analogue of AOP, allowing integration with accepted modeling practices. Aspects usually allow developers of the primary model to be oblivious to the aspects that modify the primary model; because of this, aspects can be closely coupled to potentially transient details of the primary model. When those details change, the aspects that depend on them may no longer have the desired effect. In this thesis, we examine three approaches to AOM, and introduce a novel solution to the problem of obliviousness by extending a graph-transformational approach to AOM.
“Two-Way” Obliviousness in General Aspect-Oriented Modeling

by

Nathan V. Roberts, B.A.

A Thesis
Approved by the Department of Computer Science

__________________________
Donald L. Gaitros, Ph.D., Chairperson

Submitted to the Graduate Faculty of
Baylor University in Partial Fulfillment of the
Requirements for the Degree
of
Master of Science

Approved by the Thesis Committee

__________________________
Eunjee Song, Ph.D., Chairperson

__________________________
Paul C. Grabow, Ph.D.

__________________________
James A. Marcum, Ph.D.

Accepted by the Graduate School
August 2008

__________________________
J. Larry Lyon, Ph.D., Dean

Page bearing signatures is kept on file in the Graduate School.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td></td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td></td>
<td>xi</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td></td>
<td>xii</td>
</tr>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1</td>
<td>AOP Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1.2</td>
<td>Obliviousness and Fragility in AOP Systems</td>
<td>3</td>
</tr>
<tr>
<td>1.1.3</td>
<td>Aspect-Oriented Modeling</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>UML Diagrams</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>Research Summary</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>Scope of the MATA Extension</td>
<td>9</td>
</tr>
<tr>
<td>1.5</td>
<td>Structure of This Thesis</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>Literature Review</td>
<td>11</td>
</tr>
<tr>
<td>2.1</td>
<td>AOP in the Literature</td>
<td>11</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Overcoming Developer Resistance to AOP</td>
<td>11</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Obliviousness: The Whole Point, or a Problem?</td>
<td>12</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Tight Coupling between Base and Aspects</td>
<td>12</td>
</tr>
<tr>
<td>2.1.4</td>
<td>Symmetric Composition and System Organization</td>
<td>13</td>
</tr>
<tr>
<td>2.1.5</td>
<td>Composition Order and Evolving Method Signatures</td>
<td>13</td>
</tr>
<tr>
<td>2.1.6</td>
<td>Summary AOP Observations</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Selecting Three AOM Approaches</td>
<td>15</td>
</tr>
<tr>
<td>2.2.1</td>
<td>The Eight AOM Approaches Discussed by Schauerhuber et al.</td>
<td>15</td>
</tr>
</tbody>
</table>
### 2.2.2 Summary Evaluation of SDWeaver, WEAVR, and MATA

3 Three Current AOM Approaches in Detail

3.1 Klein et al.’s Sequence Diagram Weaving

3.1.1 Introduction

3.1.2 Formal Definitions

3.1.3 Matching Algorithm and Performance Evaluation

3.2 Motorola’s WEAVR for State Model Weaving

3.3 MATA for General Model Composition

3.3.1 MATA Introduction

3.3.2 Performance Concerns

3.3.3 Relation to Klein et al.’s Part Matching

3.4 Summary Comparison of the Three Approaches

4 Extending MATA for Two-Way Obliviousness

4.1 Introduction: Obliviousness

4.2 Hiding Mutable Details with ≪hide≫ and ≪reveal≫ Stereotypes

4.2.1 Ambiguities in Sequence Diagram Weaving, Revisited

4.2.2 Limitations

4.2.3 Syntax and Semantics

4.3 Allowing Flexible Implementations: Model Interfaces, Construals, and Badges

4.3.1 Methods and Method Invocations: A Distinction

4.3.2 State Machine Model Interface Example

4.3.3 ≪modelInterface≫ Syntax
6.1.4 Tool Workflow ................................................. 90

6.2 Implementation and Enhancement of the Model Interface Proposal .... 91

6.2.1 Implementation .............................................. 91

6.2.2 Optional Elements and Elements with Arbitrary Multiplicity .... 91

6.2.3 Model Interface Inheritance ................................. 92

6.2.4 Model Interface Examples .................................. 92

6.2.5 User Interface Development ................................. 92

6.3 Conclusion ...................................................... 92

APPENDIX

A Klein Matching Algorithm and Performance Analysis .................. 95

A.1 Matching Algorithm ............................................ 95

A.2 Performance Analysis: Experiment and Optimization ............... 97

A.2.1 Performance of the Enclosed Part Algorithm: An Experiment .... 97

A.2.2 Optimization Using a Lazier Transitive Closure ................. 100

B MATA Performance Exploration .................................. 103

B.1 Repetition of Experiment from Section A.2 ........................ 103

C Tool UI Ideas .................................................. 106

C.1 Badging Icons .................................................. 106

C.2 Interface View ................................................ 106

C.3 Interface Painter ............................................... 107

C.4 Automation .................................................. 108
## LIST OF FIGURES

1.1 A sequence diagram demonstrating separation of concerns 2

1.2 UML Stereotype Example 6

3.1 An aspect for Klein et al.’s sequence diagram weaver 25

3.2 Base sequence diagram for each part type 26

3.3 A sequence diagram that imposes an ordering constraint 27

3.4 A sequence diagram for enclosed part emulation 29

3.5 An aspect definition for WEAVR 32

3.6 State machines at two levels of abstraction 32

3.7 Aspect composition using MATA 34

3.8 A MATA model for general part matching 36

3.9 MATA model that is not a safe match 37

4.1 Two-way obliviousness in aspects and the base system 41

4.2 Sequence Diagram with ≪hide≫ and ≪reveal≫ stereotypes applied 42

4.3 Matcher’s View of Base Model from Fig. 4.2 44

4.4 Aspect for base model from Fig. 4.2 45

4.5 Composition of Fig. 4.2 and Fig. 4.4 45

4.6 A sequence diagram with the ≪hide≫ stereotype applied 46

4.7 Aspect Applicable to Fig. 4.6 46

4.8 Composition with the ≪hide≫ and ≪reveal≫ Stereotypes 47

4.9 A model interface for transactions 54

4.10 A model, with two Transaction construals 55
4.11 An aspect for Transactions ........................................... 56
4.12 Transaction Model Interface, Possible Extension ................. 57
4.13 Model Interface for Authorization State Machine .................. 58
4.14 Aspect for a State Machine Using Model Interfaces .............. 59
4.15 Concrete State Machine, Construed as Model Interfaces ....... 59
4.16 Badge Associations and Attributes ................................ 63
4.17 Transaction Construal with \texttt{belongsTo} ≪ badge ≫ construal ... 66
4.18 Message Deletion Aspect ............................................. 66

5.1 Partition Unreachable by Klein et al.’s Approach .................. 71
5.2 MATA Solution for the Partition Unreachable by Klein et al. .... 72
5.3 Partition Unreachable by MATA ....................................... 73
5.4 General two-phase commit protocol ................................. 78
5.5 The \texttt{finalCommit} model interface. ............................... 80
5.6 The \texttt{commit} model interface. ................................... 81
5.7 Base model for the two-phase commit protocol .................... 82
5.8 Our version of the diagram in Fig. 5.7 .............................. 83
5.9 First graph transformation rule for the 2PC protocol .............. 84
5.10 Second graph transformation rule for the 2PC protocol ......... 85
5.11 Third graph transformation rule for the 2PC protocol ............ 86
5.12 Fourth graph transformation rule for the 2PC protocol .......... 86

A.1 Aspect for composition timing experiment ......................... 98
A.2 Base for composition timing experiment .......................... 98
A.3 Running time of SDWeaver composition ............................ 99
A.4 Performance of SDWeaver vs. lazy transitive closure modification . . . . 102

B.1 Running time for MATA composition . . . . . . . . . . . . . . . . . . . . 104
B.2 Base model for more natural test case . . . . . . . . . . . . . . . . . . . . 104

C.1 State Machine with Model Interface Icons . . . . . . . . . . . . . . . . 107
C.2 State Machine with Model Interface Icons . . . . . . . . . . . . . . . . 108
C.3 Interface Painter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
LIST OF TABLES

4.1 Implications of each of the possible values for the sticky attribute of the
    ≪hide≫ stereotype for element deletion. . . . . . . . . . . . . . . . . . . . . . 51

4.2 The weaving rules for each of the possible values for the sticky attribute
    of the ≪hide≫ stereotype, in the context of sequence diagrams. . . . . . 52

A.1 Running times for SDWeaver composition . . . . . . . . . . . . . . . . . . . . . 99

A.2 Timing results for the Klein et al.’s enclosed part algorithm, with a mod-
    ified transitive closure construction, using the aspect in Fig. A.1 and the
    base in Fig. A.2. Times are in seconds. . . . . . . . . . . . . . . . . . . . . . 101
ACKNOWLEDGMENTS

I would like to thank Eunjee Song, for her support and guidance of my work in this thesis. I thank the entire thesis committee for their consideration, suggestions, and willingness to participate in the important process of evaluating my work. I also thank the faculty and staff of the Baylor Computer Science department for all that they have taught me and done for me throughout my time at Baylor.
CHAPTER ONE

Introduction

1.1 Motivation

1.1.1 AOP Introduction

Among other things, object-oriented programming (OOP) allows programs to be divided into distinct units, each of which contains relatively isolated sets of features. This is known as the separation of concerns, and at its best OOP succeeds in achieving it for many, but not all, kinds of concerns. An example appears in Fig. 1.1. Here, the Web Server object contains code related to handling requests for web pages. It delegates responsibility for collecting data related to the ticker symbol passed in to the Stock Data Repository; similarly, the HTML Rendering Engine object has responsibility for producing a correctly formatted HTML web page. If NASDAQ changes its mechanism for stock price retrieval, then only the code in Stock Data Repository will need revision; if the latest version of Internet Explorer is tripping over the HTML produced, only the code in the HTML Rendering Engine needs to be modified.

However, some concerns defy this kind of decomposition of the system. One oft-cited example of this is logging: a logging policy will specify what sorts of system actions should be recorded in the log. One such logging policy might be to log all public method invocations. Public and private methods will, in a typical system, be present in most classes in the class hierarchy. Our example logging policy thus cuts across the class hierarchy. If we used traditional programming techniques to implement this policy, code

\footnote{The basic idea of modularization as a technique for clear and flexible programs is due to Parnas (Par79), although the phrase “separation of concerns” appears to have been coined by Dijkstra in 1974 (Dij74).}
belonging to this concern would be present in nearly every class implementation in the system. In AOP parlance, the logging code is scattered throughout the system. The concerns which OOP cannot separate are known as cross-cutting concerns, because they cut across the so-called dominant decomposition of the program into parts, which in OOP is the class hierarchy.

There are two major problems with the traditional approach to implementing this logging policy. The first is that the policy itself is laborious to implement, because very similar code will have to be added to each class implementation. Not only is this tedious, but it is also prone to programmer error. The second problem is that the implementation is not robust to changes in policy. If we decide that we now want the log to include not only the fact that the public method was invoked, but also a representation of the arguments with which the method was invoked, then we must edit every public method in the system.

How can we solve this? One idea that addresses the first problem is to write a script that does the scattering for us. The script could process all of our code, determine whether the methods are public or private, and insert code at the start of each public method. The problem of changing system concerns remains, but could easily be
addressed by including this script in the build process for the system: the script thus becomes a full-class citizen in the system specification; when the logging policy changes, it is this script which should be edited. The system in which the logging code is scattered throughout is an artifact of the build process, rather than something that developers have to work with directly.

This is quite close to what AOP allows programmers to do, except that rather than having to write a script to parse the codebase and insert code for each cross-cutting concern, the AOP environment provides a mechanism (known as a pointcut) for specifying the important points in the code (known as join points), and what code should be inserted at those points (known as advice). The compiler or the runtime environment then does the work of the hypothetical script, effectively providing a standard mechanism for solving this kind of problem, saving the developer the effort of implementing scripts each time, and encouraging the developer to think in these terms, which ideally will lead to a reduction in redundancy and a concomitant increase in maintainability of the systems thus developed. We call the dominant decomposition of the system into parts the base system; aspects are specifications of the features that cut across the base system, specified as pointcuts with advice.

1.1.2 Obliviousness and Fragility in AOP Systems

This ideal, however, meets with some interesting challenges in practice. A key question, which we examine in the context of a review of the AOP literature in Chapter 2, is to what extent the use of AOP should change the way the authors of code related to non-cross-cutting concerns do their work. One answer is that they should be allowed to proceed just as they always have done, entirely oblivious to the fact that aspects may
modify their code in various ways (by, for example, adding logging logic). This answer is problematic, however, because it can encourage aspect developers to rely too much on details of the base system implementation which are subject to change. When changes occur, an aspect may no longer match the code its author intended it to match. This type of problem motivates the main contribution of this thesis, an extension to Whittle & Jayaraman’s MATA (Modeling Aspects using a Transformation Approach) (WJ07), which we describe in Chapter 4.

1.1.3 Aspect-Oriented Modeling

A model is an abstraction of a system that allows a clearer view of the system’s structure and behavior by hiding certain details. In software, machine code can be thought of as a model of the various physical processes involved in computing, assembly language code a model of machine code, and higher-level programming languages models of lower-level ones. When we speak of a modeling language, we usually mean a language at a higher level of abstraction than typical high-level programming languages, such as Java or C++ (and usually modeling languages will be visually expressed in diagrams, rather than code). At present, aspect-oriented development sits almost exclusively at the same level of abstraction as C++ or Java code—or possibly, precisely one layer above it, since a language like AspectJ is arguably an abstraction layer atop Java. Because aspects are a feature of the decomposition of the system into logical units, we can expect higher-level models to include aspects (rather than hiding them); the fact that our modeling languages do not now include aspects can be seen as a deficiency of those languages, rather than an intrinsic feature of modeling languages.
We make models to understand better the systems we are developing, and this improved understanding brings many benefits. One immediate benefit of modeling is the ability to detect design errors early, before we invest time and money in writing implementation code for a flawed system. When a modeling domain is limited (to, say, engineering switches for telecommunications), the details obscured by models can have sufficient regularity that code can be generated from models: in such a case, the model can itself be thought of as the implementation. The vision of Model-Driven Architecture (MDA)\(^2\) is to increase the number and variety of systems in which this is possible.

There are, then, good reasons for wanting to model aspects. Approaches such as Motorola’s WEAVR, which includes a tool that allows code to be generated from aspect models, within the telecommunications domain, suggest that we may indeed be able to develop a full, general aspect modeling toolset (CvdBE07a; CvdBE07b).

1.2 UML Diagrams

Throughout this thesis, we make use of diagrams specified in the Unified Modeling Language (UML) (Gro07). Fig. 1.1 is an example of a sequence diagram, which specifies how messages get passed between objects in the system.\(^3\) These messages might be simple method invocations, but they could just as easily be messages sent over the internet. Two other kinds of diagrams will be of interest. Class diagrams specify the structure of a system, showing which classes are associated with each other, and what

\(^2\) http://www.omg.org/mda

\(^3\) It is worth emphasizing that sequence diagrams in this thesis make only minimal claims about event timing; in general, message events are only partially ordered. The two ordering rules that do apply are:

(1) for any message \(m\), \(m_{\text{send}} \leq m_{\text{receive}}\), and

(2) for any two events \(e_1, e_2 \in E\) along the lifeline for an object \(A \in I\) where \(e_1\) occurs before \(e_2\) on \(A\)’s lifeline, \(e_1 \leq e_2\).

More detail can be found in Section 3.1.2.
methods are available. State machine diagrams, like sequence diagrams, are behavioral diagrams, but these describe behavior in terms of states and transitions.

Two other UML concepts that will be useful to us are profiles and stereotypes. Profiles are extensions to the UML language, and are a standard way to add new features and concepts, such as aspect modeling, to UML. Stereotypes, which may be included within profiles, are a way to modify the intent of the elements that they are applied to. Stereotypes may include attributes called tagged values in order further to specify the modified intent. For example, the Stock Data Repository class in Fig. 1.2 has the ≪keepCache≫ stereotype applied, indicating that the objects of that class should keep a cache, with a cacheSize value of 1000, indicating that 1000 records should be kept in the cache.

1.3 Research Summary

Our research focused on three AOM approaches, extending one (MATA) to offer a combination of its features with that of another (WEAVR), in awareness of the weav-
ing concerns raised by the third (SDWeaver\textsuperscript{4}). Klein et al.’s approach, implemented in a prototype tool called SDWeaver, offers a mechanism for specifying aspects as sequence diagrams as well as a composition algorithm to weave those aspects with the base diagrams they match. Whittle & Jayaraman’s MATA, an approach that treats aspects as graph transformations, gives a succinct, intuitive visualization of aspect models. Cottenier et al.’s WEAVR approach offers a full toolset, from modeling to code generation, for state machine diagrams within the telecommunications domain. The principal contributions of our research are, then, as follows:

- a close look at Klein et al.’s approach to aspect-oriented sequence diagram weaving (KFJ07), including a detailed analysis of their pointcut specifications;

- an examination of Whittle & Jayaraman’s graph-transformational MATA approach to AOM (WJ07), including reduction of two of Klein et al.’s pointcut types to their MATA equivalents; and

- an extension of MATA, inspired by certain features of Motorola’s WEAVR (CvdBE07a; CvdBE07b), to allow aspect models to be more resilient to the evolution of the base systems they modify.

Additionally, we explore the computational efficiency of Klein et al.’s algorithm in Appendix A. We also explore MATA’s performance on comparable examples in Appendix B. Finally, we offer a few initial user interface ideas to support our extension of MATA in Appendix C.

\textsuperscript{4} SDWeaver is the name of the weaving class within Klein et al.’s prototype. We use “SDWeaver” throughout this thesis to refer to either Klein et al.’s prototype or their approach to sequence diagram weaving.
Klein et al.’s sequence diagram weaving approach, of which they have developed a prototype (SDWeaver), is a formal specification of sequence diagrams and pointcuts in set-theoretic terms. Our experiments revealed that SDWeaver is not as efficient as it might be—indeed, with some minor modifications we reduced the running time from $O(n^3)$ to $O(n^2)$, and we believe that, even so modified, the algorithm remains suboptimal.\(^5\) We also argue that two of the four pointcut types that they propose do not represent generally useful categories of pointcuts. However, the formalism provided by Klein et al. and the motivating problem—namely, that using unadorned sequence diagrams as pointcuts and/or as advice leaves the specification ambiguous—both were useful to us as we evaluated MATA and developed our extensions to it.

Whittle & Jayaraman’s MATA is a more general approach, in the sense that it allows sequence diagrams, class diagrams, and state machine diagrams in aspect specifications. We used a prototype of MATA, and used it to perform the same experiments that we had done with SDWeaver. MATA’s asymptotic performance appears to be worse than SDWeaver’s, but we do have some experimental-analytical reasons to believe that our experimental setup belongs to a pathological class of diagrams, so that MATA’s performance is not near as bad as the test made it look (details can be found in Appendix B). We also showed that the two types of Klein et al.’s pointcuts that we believe represent useful categories are also representable using MATA, thereby demonstrating that, with the exception of the two pointcut types that we argue are not useful, MATA is more general than SDWeaver—that is, with those exceptions, every aspect that can be expressed in SDWeaver can be expressed in MATA, and there are aspects that can be expressed in MATA that cannot be expressed in SDWeaver.

\(^5\) Our experimental inputs can be processed by a naive algorithm in linear time; we have not determined whether linear time is achievable in general. Details can be found in Appendix A.
The heart of our contribution, however, lies in two extensions that we propose for MATA. They both have to do with allowing model designers to shape more precisely how the models will be seen by the pointcut matcher. Because this allows potentially transient details of the design to be hidden from aspects, so that pointcuts will not depend on those transient details, the system will be more resilient to design changes. The first extension consists of two UML stereotypes, ≪hide≫ and ≪reveal≫, which control which model elements are visible to the pointcut matcher. The second extension is more general (indeed, the first can be seen as a special case of the second), providing model interfaces against which pointcuts may be written. Concrete models may implement a model interface by means of a construal.

1.4 Scope of the MATA Extension

The extensions we propose to MATA in Chapter 4 are described with detailed syntactic and semantic rules; an implementation of these extensions is left as an area for future work. Although we provide sufficient detail that specification of a complete UML profile should be straightforward, we also leave such a specification to future work. We evaluate our extension using criteria detailed in Section 2.2, and by using it to specify a two-phase commit protocol.

1.5 Structure of This Thesis

The remainder of this thesis is structured as follows. Chapter 2 investigates the current AOP literature, gleaning some information about what problems AOP has yet to solve, and looks to a survey of AOM approaches for some criteria which may be used to evaluate the three AOM approaches we examine, as well as the extensions we propose.
Chapter 3 examines, in some detail, three AOM approaches. Section 3.1 examines the work of Klein et al. in sequence diagram weaving. Section 3.2 examines Motorola’s WEAVR tool, which uses state models to specify aspects. Section 3.3 examines Whittle & Jayaraman’s MATA tool, which uses graph transformations to specify aspects. Chapter 4 details our extension to the MATA. Chapter 5 evaluates our extension according to the criteria from Chapter 2, and by using it to specify a two-phase commit protocol. Chapter 6 concludes with some remarks regarding future work.
CHAPTER TWO

Literature Review

2.1 AOP in the Literature

AOP has recently enjoyed a great deal of popularity; a 2006 edition of IEEE Software (MS06) was devoted to the subject. In the introduction to that issue, they note that frameworks available supporting AOP include AspectJ, Spring, JBossAOP, AspectC++, and Aspect#. A basic distinction is whether weaving is done at compile time (static weaving) or at run time (dynamic weaving). In either case, AOP brings with it a number of challenges. In this subsection, we examine several articles to determine key challenges in AOP, with the hope of gleaning some insight into AOM.

2.1.1 Overcoming Developer Resistance to AOP

Lesiecki describes an experience using AspectJ in a commercial software development project, extending and improving the Adbase system used by Video Monitoring Services (Les06). The experience was generally quite positive. Lesiecki emphasizes the importance of understanding (and being able to visualize) relationships between aspects and base code. He observes that the change from conventional OOP to AOP required the developers not only to overcome a certain resistance to change, but also to update their mental models to consider problems in terms of their natural concern decomposition and use aspects to implement the concerns that cut across the system, rather than

---

1 http://www.eclipse.org/aspectj
2 http://www.springframework.org
3 http://jboss.com/products/aop
4 http://www.aspectc.org
5 http://www.castleproject.org/index.php/AspectSharp
thinking of AOP’s scope as limited to a few special problems (such as logging). As their paradigms shifted, however, developers repeatedly found new applications for aspects.

2.1.2 Obliviousness: The Whole Point, or a Problem?

One of the consequences of AOP is that it allows a developer to be oblivious to the aspects which will later modify his code. Filman & Friedman consider obliviousness a virtue for an AOP system, because it means that developers of base code do not have to do any work to allow aspects to be added (FF00).

Sullivan et al. provide a counterpoint, arguing that this simply shifts the burden of labor from base developer to aspect developer (SGS+05). Obliviousness may also tend to undermine the confidence that a developer has that a program element will continue to behave in the expected fashion, because an aspect may modify that element in the future. The strong coupling between aspects and base classes to be discussed in Section 2.1.3 can be seen as a consequence of obliviousness: because aspects are allowed to match any portion of the base model, the system as a whole can become fragile, because any modification to the base may break a match. For these reasons, Sullivan et al. argue for an interface layer to decouple base designs from their features which aspects may match and modify. Griswold et al. extend this idea to produce a crosscut programming interface (XPI) (GSS+06).

2.1.3 Tight Coupling between Base and Aspects

A key problem in AOP is that there tends to be tight coupling between base classes and the aspects that modify them. Refactoring and modification are made more difficult, and code becomes harder to understand (GSS+06). Cottenier et al. have
developed an approach (CvdBE07a) whereby two sets of state machine specifications, at separate layers of abstraction, can be used to reduce coupling, so that the system is substantially easier to maintain and extend. By so doing, they are able to “infer” join points; join points are specified at the more abstract layer, but matched concretely, so that the matches may be deep within the system, and may depend on the implementation without requiring the aspect specification to refer to the implementation. This approach is implemented in WEAVR, which we describe in Section 3.2, and which inspires the extension to MATA that we propose in Chapter 4.

2.1.4 Symmetric Composition and System Organization

Chavez et al. compare implementing a system in AspectJ and implementing the same system using Hyper/J, a general tool, not specifically intended for AOP, that allows multi-dimensional separation of concerns in Java (CGL01). Hyper/J allows symmetric composition, in which there is no hard distinction between the base model and the aspect model. They conclude that Hyper/J is more flexible and powerful, but that AspectJ’s distinction between base and aspect is a useful way to organize a system, especially because it simplifies the composition rules considerably. AspectJ also suffers from the fact that composition rules are not cleanly separated from the concerns themselves.

2.1.5 Composition Order and Evolving Method Signatures

McDirmid et al. introduce Jiazzi, a component system for Java, which can be used to do AOP (MFH01). Jiazzi components can be built from standard Java binaries. Components support separate compilation, which means that each unit is compiled separately from every other unit. Units are of two types: atoms, which are compiled Java
classes and interfaces with a package specification, and *compounds*, comprised of atoms and other compounds. Jiazzi provides a linker which builds a standard Java binary from the separately-compiled Jiazzi units. Jiazzi supports mixins, which can be used to allow “open” classes, which let you add new methods and such to existing classes without breaking existing inheritance relationships. McDirmid et al. also describe using Jiazzi to do AOP in the context of a maze game (MH03). Jiazzi’s open signatures provide the ability to continue to match methods even when they are modified by other aspects. Xin et al. compare Jiazzi with AspectJ, concluding that Jiazzi is better for system-wide structuring and independent manipulation of features, but AspectJ does better at extending and modifying existing Java code (XMEH). For the latter, Jiazzi may require more “invasive” code modification.

2.1.6 Summary AOP Observations

We thus note several salient points regarding AOP. Lesiecki suggests (Les06) that developers’ resistance to change must be overcome, which in turn suggests that developer tools should be as good and efficient as possible. Chavez et al.’s discussion (CGL01) underscores a related point: sometimes having less flexibility is actually beneficial; here, AspectJ’s requirement of a base model promotes good aspect design. Most current AOM approaches also require having a base model; an exception is Baniassad & Clarke’s Theme/UML approach (CB05).

Finally, a key question in designing an AOP development environment is whose responsibility it is to specify what code modifications may be made by aspects. Put another way, what are the environment’s constraints (if any) with regard to obliviousness? Our view, discussed most fully in Section 4.1, is that the author of a piece of
code (or of a model) is the one best suited to determine what are the salient features of that code—what is prone to change, and what will remain fixed over time. Therefore, the base author(s) ought to be the one(s) to specify which parts of a model may be matched by pointcuts. In Chapter 4, we propose two ways to facilitate this division of responsibility.

2.2 Selecting Three AOM Approaches

Schauerhuber et al. examined eight AOM approaches, which they used to identify a set of basic ingredients of aspect-oriented languages (SSK+07). In light of these, they produced a set of evaluation criteria, which they then used to evaluate each of the eight approaches. In this section, we describe salient features of each of those approaches; we then turn to the AOM approaches that we examine in detail in Chapter 3, in light of some of the more revealing criteria from Schauerhuber et al.

2.2.1 The Eight AOM Approaches Discussed by Schauerhuber et al.

2.2.1.1 Ortiz et al.’s MDA Approach to Modeling Web Services. Ortiz et al. offer an MDA approach to modeling web services (OHCA05); the domain of their approach is thus limited. The web services as a black box; therefore, aspects cannot make structural modifications. They offer two types of join points, service execution join points and service call join points; these are the points at which new behaviors can be added using their approach.

2.2.1.2 Conejero et al.’s AspectNotification UML Profile. Conejero et al. have specified a UML Profile for one fairly specific aspect, which models the notification
functionality of the Observer pattern for distributed Corba applications (CHR05). It has been successfully used in a real-world project.

2.2.1.3 Groher et al.’s sUFA. Groher et al. propose sUFA (Standard UML for Aspects) (GBS05), which offers support for aspectual collaborations—that is, a collection of classes that collaboratively realize an aspect. sUFA builds upon UML for Aspects (UFA), a way of specifying aspects graphically that uses UML in a non-standard way. sUFA instead specifies a profile to extend UML in an appropriate manner; this is a standard, accepted mechanism for extending UML. sUFA employs a symmetric understanding of aspects (that is, there is no hard distinction between base models and aspect models), and allows aspects to specify both behavioral and structural modifications. sUFA does not provide a pointcut mechanism; instead, join points must be specified one at a time. sUFA is quite new; there is only a single modeling example available, and no known real-world use.

2.2.1.4 Pawlak et al.’s JAC Design Notation. Pawlak et al. (PSD+05; Paw02) also extend UML using a profile, primarily in order to support aspect modeling in the JAC framework, an open source framework that acts as a middleware layer for aspect components. Pointcuts are specified in a textual language that can include regular expressions and keywords. There is also a «group» stereotype that allows aggregation of an arbitrary set of base elements for the purpose of applying an aspect to them. The «group» stereotype is similar in intent to the model interface extension we propose to MATA in Chapter 4.

6 http://jac.objectweb.org
2.2.1.5 Stein et al.’s Aspect-Oriented Design Model. The Aspect-Oriented Design Model (AODM) offered by Stein et al. provides UML modeling support for AspectJ; it allows graphical specification of the way that aspects relate to base elements (SHU; SHU02). It does not, however, include a graphical representation of advice, and its pointcut model is very much like that of AspectJ, which is relatively limited. The authors, however, have introduced JPDD (join point designation diagrams), a pointcut specification mechanism designed to support any UML-based approach to AOM, which supports name and signature pattern matching, including wildcards (SHU06).

2.2.1.6 Aldawud et al.’s AOSD Profile. Aldawud et al. offer the AOSD Profile, intended to be independent of the choice of AOP language (AEB03; EAB05). They support class diagrams (for structural aspects) and state machine diagrams (for behavioral aspects) in aspect and base specifications. One interesting feature of this approach is that it includes explicit concern about the impact that aspects will have on the base systems they modify—some aspects, like logging, will not change the behavior of the base system at all; others, like the observer design pattern, are intended to change the functionality of the base system. The former are called synchronous aspects; the latter are called asynchronous aspects.

2.2.1.7 Clark et al.’s Theme/UML Approach. Clarke et al.’s Theme/UML approach calls the major concerns related to a system themes (CB05; CW02; CW05); the idea is that the authors of a system identify its themes, and produce models for each of them. Thus themes are like aspects that are entirely symmetric; that is, base themes are not treated any differently from crosscutting themes. Their approach supports all diagram types, and three composition relationships—merge (to combine elements without
replacing any), override (to replace elements in one model with elements from another),
and bind (a specialization of merge, to allow the pattern-based combinations that typify
aspects). Pointcuts can be specified using literal matches and wildcards; both behav-
ioral and structural modifications are possible. This is one of the most mature AOM
approaches; there are many modeling examples in the literature, and Clarke & Baniassad
have also written two case studies (CB05).

2.2.1.8 France et al.’s Aspect-Oriented Architecture Models. France et al. of-
fer the Aspect-Oriented Architecture Models (AAM) approach (FRGG04), a platform-
independent approach that supports communication diagrams for behavioral aspects
and class diagrams for structural ones, using a non-standard UML extension to improve
diagram readability (the details of which are not specified by the authors). These are
specified as templates that can be bound, using text to specify the binding, to concrete
applications; bound aspects are then instantiated in a contextual aspect model, which can
then be used for composition with base models. In a later article (RFG+05), the authors
introduce a signature-based weaving mechanism that merges elements whose signatures
(that is, whose salient structural features) match; this only applies to structural aspects.

2.2.2 Summary Evaluation of SDWeaver, WEAVR, and MATA

We now turn to a brief evaluation of the three AOM approaches that we examine
in more detail in Chapter 3, in light of some of the more revealing criteria from Schauer-
huber et al (SSK+07). These criteria fall into four categories: language, the kinds of
aspects supported, maturity, and tool support.
2.2.2.1 Language. The language criterion of interest to us is “Diagrams.” An aspect can be specified in behavioral and/or structural terms; which of UML’s behavioral and structural diagram types are supported?

- Klein et al.’s SDWeaver - sequence diagrams are supported.

- Cottenier et al.’s WEAVR - state machine diagrams are supported.

- Whittle & Jayaraman’s MATA - state machine, sequence, and class diagrams are supported.

2.2.2.2 Kinds of Aspects Supported. Closely related to the kinds of diagrams supported, this criterion determines whether behavioral, structural, or a combination of behavioral and structural aspects are supported.

- Klein et al.’s SDWeaver - behavioral aspects are supported.

- Cottenier et al.’s WEAVR - behavioral aspects are supported.

- Whittle & Jayaraman’s MATA - behavioral, structural, and combination aspects are supported.

2.2.2.3 Maturity. The criterion for maturity of primary interest to us is whether the approach has been used for real-world applications.

- Klein et al.’s SDWeaver - no known real-world applications.

- Cottenier et al.’s WEAVR - several years of real-world use by Motorola.

- Whittle & Jayaraman’s MATA - no known real-world applications.
2.2.2.4 Tool Support. Schauerhuber et al. evaluate tool support by considering whether there is *modeling support*, meaning that aspect specifications can be checked by a tool for syntactical and semantical correctness, whether there is *weaving support*, meaning that a tool will perform aspect weaving, and whether there is *code generation support*, meaning that the tool has facilities to translate woven models into code.

- Klein et al.’s SDWeaver - modeling and weaving support are provided; code generation is not.

- Cottenier et al.’s WEAVR - modeling, weaving, and code generation are all supported.

- Whittle & Jayaraman’s MATA - modeling and weaving support are provided; code generation is not.

2.2.2.5 Other Criteria. In addition to the criteria used by Schauerhuber et al., we propose two criteria of interest:

- *formalism* - does the approach provide formal specifications of its elements? Formal specifications can aid in evaluating the correctness as well as the efficiency of weaving algorithms.

- *decoupling support* - does the approach provide ways of encouraging decoupling aspects from the base models they modify?

Klein et al. provide a formal specification for their approach, which will be helpful to us as we reason about their match types, which define the pointcuts available in their approach. Cottenier et al. provide decoupling support, in the form of a kind of state
machine diagram interface that specifies relatively fixed elements of the model. This decoupling is what we describe as “two-way” obliviousness in Chapter 4, which will motivate our extension of MATA.

2.2.2.6 Summary Comparison of Approaches. In summary, then, MATA allows the most general models, while SDWeaver and WEAVR only provide support for sequence diagrams and state machine diagrams, respectively. SDWeaver has the most formal specification, with set-theoretic definitions of each of its constructs. WEAVR is the most mature, with real-world deployment and good mechanisms for decoupling of aspects from base models.
Three Current AOM Approaches in Detail

In this chapter, we examine three current AOM proposals in some detail. Section 3.1 examines Klein et al.’s approach to weaving sequence diagrams at the metamodel level.\footnote{The \textit{metamodel} of UML defines what kinds of diagrams can be specified, and the constraints on those diagrams (Gro07). The reason that Klein et al. must work at the metamodel level is because they define two new kinds of sequence diagram, the \textit{basic sequence diagram} and the \textit{combined sequence diagram}, which we describe in Section 3.1.1; to introduce a new diagram type or further constrain one of UML’s diagrams requires metamodel extensions.} Section 3.2 turns to Cottenier et al.’s WEAVR, an approach that uses abstract state machines to specify interfaces that aspect pointcuts may match against; composition is done using concrete implementations of those interfaces, which may be considerably more complex. Finally, in Section 3.3, we look at Whittle & Jayaraman’s MATA, a graph-transformational approach that provides mechanisms to specify aspects in terms of class diagrams, sequence diagrams, and/or state machine diagrams. In Chapter 4, we offer two extensions to MATA, in an attempt to provide a generalization of WEAVR’s state machine interfaces.

3.1 Klein et al.’s Sequence Diagram Weaving

3.1.1 Introduction

Klein et al. have developed a way to do aspect modeling using sequence diagrams (KFJ07). They note that the natural approach—to describe a pointcut in one sequence diagram, and the advice as a second sequence diagram—leaves the aspect specification ambiguous, both in what counts as a match, and how weaving will be performed. Con-
sider, for example, the pointcut in Fig. 3.1(a): which of the diagrams in Fig. 3.2 should match?

To support their work, Klein et al. introduce a somewhat simplified sequence diagram called a \textit{basic sequence diagram}, or \textit{bSD}. bSDs are limited to finite event sets (indefinite looping is excluded) and do not support branching. The idea is to decompose the behavior you wish to describe into some basic building blocks, the bSDs, to facilitate aspect specification and matching; these can then be composed to form arbitrary sequence diagrams (which Klein et al. refer to as \textit{combined sequence diagrams}, or \textit{cSDs}).

A general problem in doing aspect weaving and composition of sequence diagrams is to what extent and in what way the precise sequence of events should matter when doing matching. On the one hand, having a given set of messages in order and without any messages between them may matter a great deal, and you may not want to count as a match anything that interposes other events in the sequence; on the other hand, this may matter very little, and you may be happy to accept any sequence diagram that contains the messages in the specified order, whether there are other messages between them or not. Further complicating matters, if several pointcuts match a bSD, the application of advice to that bSD may prevent the match of another, which may or may not be what you would like. There is therefore some need for flexibility in the kind of matching that is specified.

To solve these problems, Klein et al. specify four types of matches, defined as types of sequence diagram \textit{parts}, in increasing order of strictness: \textit{general}, \textit{safe}, \textit{enclosed}, and \textit{strict}. If a sequence diagram X matches another sequence diagram Y according to, say, the strict part match, then we say that Y is a strict part of X.
This section is structured as follows. We define key terms (bSDs and the part
types) in Section 3.1.2. In Section 3.1.3, we briefly describe the results of some experi-
ments we ran using a prototype provided by Jacques Klein; the details can be found in
Appendix A.

3.1.2 Formal Definitions

3.1.2.1 bSDs Defined. Klein et al. define a bSD as a tuple \((I, E, \leq, A, \alpha, \phi, \prec)\), where:

- \(I\) is the set of objects participating in the interaction,
- \(E\) is a finite set of events (one for each message sent, and one for each message
  received),
- \(\leq\) is a partial ordering induced by (send, receive) message event pairs and object
  lifelines\(^2\),
- \(A\) is a set of actions (message names),
- \(\alpha\) is a mapping from \(E\) to \(A\) (naming the events),
- \(\phi\) is a mapping from \(E\) to \(I\) (associating each send event with its sender and
  each receive event with its recipient), and
- \(\prec\subseteq E \times E\) pairs the send event of each message with its receive event.

\(^2\) A lifeline in a sequence diagram represents an individual participant in the interaction. A lifeline is
depicted by a rectangle with a vertical line (which represents the “lifetime” of the participant) extending
from it; e.g., \(I_1\) and \(I_2\) in Fig. 3.1(a) are lifelines (Gro07, pp. 489-491).
Figure 3.1: An aspect for Klein et al.’s sequence diagram weaver, specified as a pointcut indicating what message sequence to match, and advice indicating how the matched message sequence should be transformed by the aspect.

The partial ordering of events $\leq$ can be determined by taking the transitive closure of pairwise event orderings determined by (send, receive) message event pairs and object lifelines, according to two rules:

1. for any message $m$, $m_{\text{send}} \leq m_{\text{receive}}$, and

2. for any two events $e_1, e_2 \in E$ along the lifeline for an object $A \in I$ where $e_1$ occurs before $e_2$ on $A$’s lifeline, $e_1 \leq e_2$.

3.1.2.2 Part Types: General, Safe, Enclosed, and Strict. The most flexible type of match is the general part, which allows any messages to be interspersed among the matched messages. Each of the bSDs in Fig. 3.2 match the pointcut in Fig. 3.1 according to this match type; the bSD in Fig. 3.1(a) is a general part of each of those bSDs.

A safe part is like a general part, except that the extra messages are not allowed to impose any additional constraints on the ordering of the matched events. The bSD in Fig. 3.1(a) is a safe part of Fig. 3.2(a), Fig. 3.2(b), and Fig. 3.2(c). To see why it is
Figure 3.2: Several possible base sequence diagrams. The captions indicate the most restrictive match type for which the pointcut in Fig. 3.1(a) will match each base sequence diagram.
Figure 3.3: A sequence diagram that imposes an ordering constraint on the messages $a_{receive}$ and $b_{send}$: $b_{send} \leq a_{receive}$. This is exactly the opposite of the constraint imposed by Fig. 3.2(d).

not a safe part of Fig. 3.2(d), notice that, in Fig. 3.1(a), the events $a_{receive}$ and $b_{send}$ are unordered. However, in Fig. 3.2(d), we have $a_{receive} \leq d_{send} \leq b_{send} \Rightarrow a_{receive} \leq b_{send}$.

Because message $d$ adds additional constraints to the event ordering, the pointcut is not a safe part of the bSD in Fig. 3.2(d). Although Klein et al. do not say so explicitly, we believe the reason this part type is called “safe” is that it is not possible for a set of matches to impose conflicting ordering constraints on the messages in the pointcut; consider the base sequence diagram in Fig. 3.3, where we have $b_{send} \leq d_{send} \leq d_{receive} \leq a_{receive} \Rightarrow b_{send} \leq a_{receive}$, exactly the opposite of the constraint imposed in Fig. 3.2(d).

An enclosed part does not allow any other events to be interposed between matched events, but does allow a message $m$ to be wrapped “around” matched events, with $m_{send}$ prior to the matched events\(^3\), and $m_{receive}$ after the matched events. In Fig. 3.2(b), $d$ is sent before the messages $a$ and $b$ are received, and received after they are sent.

\(^3\) More precisely, there exists an event $\alpha$ in the pointcut such that $m_{send} \leq \alpha$, but there does not exist any event $\beta$ in the pointcut such that $\beta \leq m_{send}$. Similarly, to say that $m_{receive}$ comes after the pointcut events is to say that there is some event $\gamma$ in the pointcut such that $\gamma \leq m_{receive}$, and that no event $\delta$ exists such that $m_{receive} \leq \delta$. 

27
A strict part does not allow any other events to be interposed between the matched events; neither does it allow a message to span the matched events. The pointcut in Fig. 3.1(a) is a strict part of the bSD shown in Fig. 3.2(a).

3.1.2.3 Critique of the Enclosed Part Distinction. We do not believe that the strict part type provides any meaningful distinction from the enclosed part type. To see why, consider Fig. 3.4. If we identify $d_{1\text{send}}$ in this figure with $d_{\text{send}}$ in Fig. 3.2(b), and define the Relay object as one that simply sends messages it receives from $I1$ to $I2$, allowing us also to identify $d_{2\text{receive}}$ in Fig. 3.4 with $d_{\text{receive}}$ in Fig. 3.2(b), then this diagram specifies exactly the same thing as Fig. 3.2(b), but this one will match the pointcut in Fig. 3.1(a) even with a strict part match, as we will now demonstrate.

Call the bSD in Fig. 3.1(a) $M'$, and the bSD in Fig. 3.4 $M$. We would like to show that $M'$ is a strict part of $M$. According to Klein et al.’s definitions, to do so we will need to find bSDs $X$ and $Y$ such that $M = X \cdot M' \cdot Y$, where $\cdot$ is the sequential composition operator. Here, we present only part of the definition of this operator, the part which is of interest for our proof. For further details, see (KFJ07).

Let $X$ be a bSD containing the message $d_1$, $Y$ be a bSD containing $d_2$. The only element of the tuple $(I, E, \leq, A, \alpha, \phi, \prec)$ for which $M = X \cdot M' \cdot Y$ is not obvious is $\leq$. Klein et al. define the partial order on events for a sequential composition $M_1$ and $M_2$ as $\leq_{1\circ2} = (\leq_1 \cup \leq_2 \cup \{(e_1, e_2) \in E_1 \times E_2 | \phi_1(e_1) = \phi_2(e_2)\})^*$, where $^*$ is the transitive closure. That is, the new partial event ordering is simply the one induced by the composed bSDs, together with a lifeline ordering rule, which states that if events from $M_1$ and $M_2$ are on a common lifeline, those in $M_1$ should precede those in $M_2$.

\footnote{To be precise, where our definition has $\cup$, Klein et al. have $\uplus$, the disjoint union of two multisets. The distinction only makes a difference when the two bSDs share elements, which does not occur in our discussion.}
We have:

\[ \leq_X = \{(d_{1\text{send}}, d_{1\text{receive}})\}, \]

\[ \leq_Y = \{(d_{2\text{send}}, d_{2\text{receive}})\}, \]

and

\[ \leq_{M'} = \{(a_{\text{send}}, a_{\text{receive}}), (b_{\text{send}}, b_{\text{receive}}), (a_{\text{send}}, b_{\text{send}}), (a_{\text{receive}}, b_{\text{receive}})\}. \]

Further, \( \leq_{X \bullet M'} = \leq_X \cup \leq_{M'} \cup \{(d_{1\text{send}}, a_{\text{receive}}), (d_{1\text{send}}, b_{\text{receive}})\} \). Thus,

\[ \leq_{X \bullet M' \bullet Y} = \leq_{X \bullet M'} \cup \leq_Y \cup \{(d_{1\text{receive}}, d_{2\text{send}}), (b_{\text{send}}, d_{2\text{receive}}), (a_{\text{send}}, d_{2\text{receive}})\} = \leq_M, \]

as required.

This will be relevant in Section 3.3, when we discuss using of Whittle & Jayaraman’s MATA approach to emulate Klein et al.’s various part types—we will be able to emulate enclosed and general parts, but strict parts as such will be excluded. The argument above suggests that the limitation of being unable to specify strict matches is not an important one. In Section 3.3.3, we will also argue that being unable to specify safe part matches as such is not a great limitation, either.
3.1.3 Matching Algorithm and Performance Evaluation

For Klein et al., a pointcut can be specified as a bSD with a selection of the type of part to require for a match. They detail a matching algorithm with variants for each part type. We include a detailed description of the algorithm, and an analysis of its performance, in Appendix A. The examples we use are ones for which a naive algorithm would run quickly (completing in time linear in the size of the input diagrams), but Klein et al.’s prototype SDWeaver implementation is considerably slower (completing in cubic time). If we imagine deploying tools similar to SDWeaver in large-scale software systems, poor asymptotic performance will be a concern. The fact that a naive approach to our example would do so much better suggests that there is room for improvement in the algorithm.

3.2 Motorola’s WEAVR for State Model Weaving

The second approach to AOM that we examine in detail has been developed by Motorola, in the form of their WEAVR tool. As we describe in Chapter 2, Cottenier et al. note that base model and aspects are often tightly coupled, making later system modifications difficult and prone to error (CvdBE07a). They dismiss the kind of solution proposed by Griswold et al. (GSS+06), in which join points are moved to interfaces, because traditional interfaces provide too little information about the runtime behavior of their components. Instead, they propose using two layers of state machines, one that acts as a kind of interface, and one that provides sufficient information that code may be generated from it. One of their key successes is the ability to “infer” join points—although the pointcuts are specified in terms of the states in the more abstract state
machines, they actually match against and are woven into the corresponding, more concrete state machines, and they may be “deep” within these.

In Fig. 3.5, an aspect is defined, matching any transition named access() that ends in the unauthorized state, adding code that throws a SecurityException. In Fig. 3.5(a), the throwSecurityException advice of Fig. 3.5(c) is bound to the security-Violation pointcut of Fig. 3.5(b). The access() transition and the unauthorized state belong to the more abstract state machine defined in Fig. 3.6(b); for any concrete implementation of that abstract state machine, such as that in Fig. 3.6(a), a realization mapping like the one in Fig. 3.6(c) must be defined. This realization mapping specifies that the actionPerformed() transitions from Init to the Next and Done states in Fig. 3.6(a) both implement (realize) the access() transition from Init to authorized in Fig. 3.6(b). The realization mapping thus adds a layer of meaning to what is given in Fig. 3.6(a): it states that both these transitions are authorized transitions. The mapping also specifies that the actionPerformed() transition from Init to Previous in Fig. 3.6(a) implements the transition from Init to unauthorized. This means that the transition from Init to Previous is unauthorized. WEAVR can then take advantage of this additional information to weave in the throwing of the SecurityException shown in Fig. 3.5(c) at that transition.

In addition to decoupling aspects from implementation details of the base model, this approach has one substantial advantage—namely, that it has withstood the test of real-world usage, in Motorola’s network infrastructure business unit. The context of its deployment in the telecom industry may be the reason their tool is limited to state models: because the systems they develop are essentially state machines, this is the only sort of aspect specification that is supported.
Figure 3.5: An aspect definition in WEAVR. Example taken from Cottenier et al. (CvdBE07b).

Figure 3.6: State machines at two levels of abstraction, and a mapping between them, in WEAVR. Example taken from Cottenier et al. (CvdBE07b).
Because WEAVR has an interface layer between aspects and the base system, aspects are insulated from changes to the base system. When a base developer changes the base system, it is her responsibility to create a new realization mapping to replace any that are no longer valid. The small additional work required of the base developer compares with a potentially large amount of work on the part of the aspect developer: the aspect developer not only would need to rewrite the pointcuts affected by the base system change, but before he could do so, he would have to learn all the relevant details of that change (which the base developer, as author of the change, already knows). Because of these development process advantages, WEAVR’s approach serves as the inspiration for the extension to MATA that we detail in Chapter 4.

3.3 MATA for General Model Composition

3.3.1 MATA Introduction

A third approach, and one on which much of our work in Chapter 4 is based, is Whittle & Jayaraman’s MATA (Modeling Aspects using a Transformation Approach) (WJ07). This treats aspects as graph transformations, and modifies Whittle et al.’s earlier work (WAM06) by, among other things, consolidating the matched graph (i.e., the left-hand side of the transformation rule) and the output graph (i.e., the right-hand side) into a single diagram. In the earlier work, Whittle et al. articulate a view of models as an abstraction hierarchy, with machine code at the lowest layer, and each layer above that a model for the layer below. For them, an aspect-oriented model is then a model that crosscuts other models at the same level of abstraction. They use the example of a use case: many use cases will cut across implementation-layer models, but only a use case that cuts across other use cases counts as an aspect-oriented use case.
Figure 3.7: Aspect composition using MATA. The aspect refactors the display mechanism for the FractalGenerator, replacing calls to the Display’s `refresh()` method with calls to DisplayBuffer’s `drawIntoBuffer()` method.

MATA allows designers to specify aspects that match and transform arbitrary UML models. MATA includes a succinct language for the expression of aspects as graph transformations; a single, simple diagram can capture both pointcut and advice. In Fig. 3.7, an aspect refactors the display mechanism for a class in the base model. The `<<create>>` stereotype indicates that new elements (the `DisplayBuffer` lifeline and the `drawIntoBuffer()` message) should be added to the model; the `<<delete>>` stereotype indicates that existing elements (the `Display` lifeline and the `refresh()` message) should be removed from it.

### 3.3.2 Performance Concerns

Because it relies on graph transformations which in turn rely on solving a subgraph homomorphism problem, which in general is known to be NP-complete, we may wonder whether this approach is computationally feasible—that is, if the models grow large, will
model composition be possible within a reasonable amount of time? The Attributed
Graph Grammar system (AGG), the tool on which MATA relies to do transformations,
uses some of the special constraints of the transformation problem to translate it into a
problem that can be solved more efficiently,\(^5\) so there is some hope that this question
can be answered in the affirmative, although Whittle & Jayaraman do not address it
directly. Although a full analysis of this question is outside the scope of the present
work, we have done some initial exploration, which we present in Appendix B.

3.3.3 Relation to Klein et al.’s Part Matching

Here, we compare MATA’s sequence diagram matching to the four types defined by
Klein et al. MATA provides flexible matching through two types of wildcard operators,
the \textit{any} interaction fragment, which will match any sequence of messages involving any
objects, and messages with names of the form \texttt{|messageName}, which will match any
single message between the indicated objects. In literal matches (that is, in the absence
of wildcard operators), MATA works precisely the way that Klein et al.’s enclosed part
does. One consequence of this is that MATA at its most strict is not as strict as Klein
et al.’s strict part, but, as we saw in Section 3.1.2.3, the additional strictness does not
seem to add any meaningful distinction.

The \textit{any} interaction fragment is a new interaction operator defined by Whittle &
Jayaraman, which matches any messages sent along any lifeline(s). The general part
match can be emulated by MATA if \textit{any} fragments are inserted between each pair of
messages in the match. For example, to do a general part match using the pointcut in
Fig. 3.1, the MATA model could be as shown in Fig. 3.8.

\(^5\) The graph transformation problem is translated into a constraint satisfaction problem (CSP); CSPs
are a class of problems for which there are many results that can aid in finding solutions efficiently
(Rud00).
What of Klein et al.’s safe part match? Recall that this is a match that requires that the messages outside the matched part do not impose additional ordering constraints on the messages in the matched part. For example, Fig. 3.2(d) is not a safe part match for Fig. 3.1 because message $d$ imposes the additional constraint that $a_{receive} \leq b_{send}$. Could we then, in this example, simply exclude any sequence diagrams that have messages from $I2$ to $I1$ that fall between $a_{receive}$ and $b_{send}$? This is not enough; consider Fig. 3.9, in which a pair of messages similarly imposes $a_{receive} \leq b_{send}$ upon the matched messages. A safe match for Fig. 3.1, then, requires that there be no totally ordered sequence of messages $x_1, x_2, \ldots, x_n$, where $x_1$ is sent from $I2$ after $a_{receive}$ and $x_n$ is received at $I1$ before $b_{send}$.

We can see, then, that specifying a safe part match using MATA’s wildcards is not generally possible, and the cases in which it is possible are precisely those cases for which the safe part match is the same as the general part match: namely, those whose messages are already totally ordered, so that there is no possibility of imposing additional ordering constraints. We do not, however, believe this to be a very severe limitation. Although the safe part match provides an interesting guarantee—as discussed in Section 3.1.2.2, it
Figure 3.9: A MATA model that is not a safe match for the pointcut specified in Fig. 3.1.

requires that none of the matched sequence diagrams impose opposite constraints upon
the incomparable events within the pointcut—it does not seem that this guarantee will
generally be useful. If there had to be a single ordering to the messages in the pointcut
(if, for example, all the join points somehow represented the same message sequence),
then of course this would be a useful, perhaps even a necessary, guarantee. However, the
usual case in aspect oriented modeling and programming is that a pointcut expresses a
pattern which is understood to have potentially many instantiations, each of which may
use the flexibility provided within the pattern in different ways, entirely independent of
the other instantiations.

3.4 Summary Comparison of the Three Approaches

We have seen then, three approaches to AOM in some detail. Klein et al.’s sequence
diagram weaving provides a rigorous, formally specified mechanism for weaving aspect
sequence diagrams and base sequence diagrams together. Their technique does provide
some flexibility in matching by using different part definitions, but we do not believe
that either strict part or safe part matching will be especially useful in practice; the
general/enclosed part distinction seems to provide a lot more discriminatory value.
Cottenier et al.’s WEAVR approach allows the specification of two levels of state machine diagram, the more abstract of which can be thought of as the interface, and the more concrete of which can be thought of as the implementation. A realization mapping relates elements in the interface to those in the implementation. This approach has the advantage that it does not require any consistent naming from the base developer, and that it can hide potentially transient details of implementation from the aspect developer, so that aspects can be more robust to changes in the base implementation. WEAVR’s greatest strength, then, lies in the kind of development process that it encourages.

Finally, Whittle & Jayaraman’s MATA is a very general approach, with support for sequence, state machine, class, and package diagrams. The approach is visually appealing, and capable of succinctly expressing both pointcut and advice for an aspect on a single diagram. Limited wildcard matching is supported; with this exception, names in matching diagrams must be identical. MATA’s greatest strengths lie in its generality and its ability to succinctly and visually express aspects.

In Chapter 4, we extend MATA, retaining its generality and much of its succinctness, in an effort to incorporate mechanisms to support the kind of separation of developer concerns that is WEAVR’s great strength.
CHAPTER FOUR
Extending MATA for Two-Way Obliviousness

4.1 Introduction: Obliviousness

Filman & Friedman claim that AOP is quantification and obliviousness—quantification being the ability to refer to possibly disparate elements of the system and apply modifications to them; obliviousness being the ability to specify the system without any special effort to allow the quantification. They argue that the more oblivious an AOP environment is, the better; ideally, they would like to tell the base developer, “Just program like you always do, and we’ll be able to add the aspects later” (FF00, p. 6). For them, obliviousness is exactly the base developer’s ability to work without any awareness of the aspects that may later be woven into his or her code. However, as Sullivan et al. argue, this kind of obliviousness may simply shift the burden of effort to the aspect developer, who must ensure that the pointcuts match appropriate places in the base developer’s code—and that they continue to match as the system evolves (SGS+05). We may also wonder whether, in the presence of a new paradigm such as AOP, we should hope for any developers to continue to work just as they always have done; it seems likely that most developers will need to make at least minor adjustments to the way that they work. In fact, it seems unlikely that even Filman & Friedman’s base developer working in an ideal environment will go on working just as before: because the previously entangled concerns are disentangled by AOP, the base developer now needs to do less.
Let us step back a moment and consider why obliviousness of any kind should be a virtue. The basic principle extends back to the idea of *separation of concerns*, which we discussed in Chapter 1. Among the reasons that this is a desirable feature for a development environment are that it is easier to reason about the way a system will behave relative to a particular feature or concern, that it is easier to change the details of one system feature in relative isolation from others, and that generally code for well-isolated concerns will be easier to reuse when those concerns arise in other systems. Consider, however, the kind of separation of concerns Filman & Friedman are promoting: the base is produced in isolation from the aspects, but the aspects have inevitable dependence on the base. If the base is modified, the aspects may break. We might call this *one-way obliviousness*: the base developer gets to be oblivious, but the aspect developer must be the furthest thing from oblivious—we might even say she must be *omniscient*, relative to the portion of the base system the aspect is intended to match, and perhaps she should also be *prescient*, able to predict the ways in which the base system may change in the future.

A moment ago, we called the aspect’s dependence on the base inevitable. It is so because the aspect modifies the base; and we believe that the reason that Filman & Friedman’s obliviousness is one-sided is precisely this. However, the fact of dependence on the base does not require that said dependence be direct, so one strategy for making things more balanced is to introduce a layer of indirection, an interface on which both aspects and base can rely.

Both Sullivan et al. (SGS+05) and Cottenier et al. (CvdBE07a) promote what we might call a *two-way obliviousness*, in which no one gets to be oblivious with respect to the *fact* that aspects are being woven, but in which both parties can be oblivious about
Figure 4.1: The high-level relationship between aspects and the base system, in what we call two-way obliviousness. The authors of the base system are responsible for implementing the agreed-upon interface; authors of aspects may write pointcuts that match elements within this interface and advice that transforms them.

the mutable details of the other’s design, by agreeing upon an immutable interface, which the base developer agrees to implement, and over which the aspect developer may quantify; that is, the aspect developer may write pointcuts that match elements in the agreed-upon interface, and transformations to be applied to those elements. The high-level structure this imposes is depicted in Figure 4.1.

What we propose in this chapter are two extensions to MATA, whose succinct, clear diagrams we prefer, to allow two-way obliviousness. The first is simpler and may work well for relatively small projects; the second adds complexity in the form a full interface layer and should work well for more general projects.

4.2 Hiding Mutable Details with ≪hide≫ and ≪reveal≫ Stereotypes

Given that the basic problem we are trying to solve is that aspect developers see too much of the base system, a natural idea is to add a ≪hide≫ stereotype that allows a model author to hide certain model elements from pointcut matching. Effectively, what the author is then saying is that this element is a detail that may change with later
revision. Given that such details may be the rule rather than the exception, we propose that when this stereotype is applied to container elements, this should mean that it applies to all their contents as well.\(^1\) Within sequence diagrams, however, we do make one exception: lifelines are never hidden.\(^2\) Of course, it may be that certain elements within a container should be visible, while the rest remain hidden; for this reason, we also introduce a \(<\text{reveal}>\) stereotype which can reveal an element that was hidden at a higher level of the containment hierarchy.

Notice that this mechanism provides a lightweight imitation of WEAVR’s abstract state machine interfaces—the principal differences being:

---

\(^1\) Here, we are following the way that MATA handles its \(<\text{create}>\) and \(<\text{delete}>\) stereotypes. MATA introduces a \(<\text{context}>\) stereotype which allows elements within containers marked with \(<\text{create}>\) to be part of the pointcut (i.e., \(<\text{context}>\) elements exist prior to the weaving of the aspect). \(<\text{context}>\) is analogous to our \(<\text{reveal}>\) stereotype.

\(^2\) We disallow this for reason of simplicity: if a lifeline were hidden, that participant could not receive exposed messages; it seems to us that a simpler way to achieve a similar result is simply to hide the messages that are sent to that lifeline.
• the “interfaces” are defined within the base model itself, in the form of exposed elements;

• in addition to state machine diagrams, the mechanism supports class diagrams and sequence diagrams; and

• matches are name-literal, in the sense that there is no equivalent to WEAVR’s realization mappings (and therefore any “interface” implementations must use the same names for the elements that take part in the interface).

Fig. 4.2 shows an example using the «hide» and «reveal» stereotypes. The example is an interactive fractal generator. In this diagram, the Controller object passes the FractalGenerator a `generate()` message, which will produce a fractal image. The Controller then specifies how that fractal should be colored with the `applyColorScheme()` message. By applying the «hide» stereotype to the diagram as a whole and «reveal» to two messages, the author of this diagram has specified that the `generate()` and `applyColorScheme()` message events are relatively fixed—most precisely, that they are suitable for aspect matching—while the other messages are not. For example, the `createFractal()` message passed to the iterated function system IFS might be an artifact of design decisions that could later be reversed;\(^3\) by using «hide» and «reveal» thus, the author reserves the right to revisit these decisions. A representation of this model as it would be seen by the composer for aspect matching purposes can be found in Fig. 4.3.

Figure Fig. 4.4 shows an aspect that adds a backup of the color scheme previously used by adding two messages. Figure Fig. 4.5 shows the composed model that would

---

\(^3\) Iterated function systems are just one way in which fractal images can be generated, so it is quite appropriate that this be considered a contingent detail of the implementation.
result from the application of this aspect to the base model. Notice that, although the aspect did not specify the ≪reveal≫ stereotype, it has been added, because elements are visible by default and the aspect did not specify that they were hidden; thus, to distinguish their status from that of the container (which is hidden), they must be revealed.

4.2.1 Ambiguities in Sequence Diagram Weaving, Revisited

A difficulty, however, arises with respect to the way that sequence diagrams are woven. Consider the example in Fig. 4.5. We glossed over an ambiguity in the earlier discussion: should the new sendColorScheme() message from the FractalGenerator be sent before or after it sends its existing createFractal() message? It may be that in this case, and perhaps in many cases, either order agrees with what is intended, but of course it will be best if our modeling language provides us a mechanism to specify which order is correct.
Figure 4.4: Backup Aspect. An aspect that would match the model from Fig. 4.2, and add method invocations to backup the color scheme.

Figure 4.5: Composed Model. The model that would result from the application of the aspect in Figure Fig. 4.4 to the base model in Fig. 4.2.
Figure 4.6: Base Model. A sequence diagram with the ≪hide≫ stereotype applied to an element that should not be matched.

Figure 4.7: Display Refactoring Aspect. An aspect that could be applied to the model in Figure 4.6. The aspect refactors the display mechanism for the FractalGenerator, replacing calls to the Display’s refresh() method with calls to DisplayBuffer’s drawIntoBuffer() method.
Another, more serious, ambiguity is apparent if we consider what happens if an aspect deletes an exposed message. Consider the model in Figure 4.6. If the exposed refresh() method were deleted by an aspect like the one in Figure 4.7, what should happen to the didRefresh() method? Because didRefresh() is an implementation detail that is hidden from the pointcut matcher, the aspect author has no opportunity to specify what should be done with it. Since the aspect does not specify what should be done with the hidden message, it is quite natural to keep it, as shown in Figure 4.8. However, in a case like this one, the correct action, according to the intent of the various authors, would be to delete the didRefresh() message, since it is conceptually part of the refresh() message. However, this will not always be the appropriate response; it could be that the subsequent, hidden message is conceptually independent of the earlier message, and thus should continue to be sent even when the earlier message is deleted.
The case of the `didRefresh()` message suggests an interesting distinction: we can define two kinds of relationship that may obtain between the visible message events (which here act as the interface layer) and the invisible messages near them:

- an invisible element may be *conceptually independent of* the visible ones, or
- an invisible element may be *conceptually dependent on* one or more visible elements.

Observe that if an invisible element depends on a visible element, then it should share the visible element’s fate: if the `refresh()` message is deleted, the `didRefresh()` message should likewise be deleted. Conceptually, the dependence relation suggests that the invisible element is a hidden implementation detail of the revealed concept. An independent hidden element is then one that is intrinsic to its container; even if all revealed elements within the container are deleted, the hidden element should remain.

We solve this by adding a *sticky* tag definition to the `≪hide≫` stereotype. Using the sticky tag, elements can be “glued to” other elements. When an exposed element has hidden elements stuck to it, those elements will share its fate if it is deleted. The details are discussed in Section 4.2.3.

4.2.2 Limitations

This first part of our proposed extension to MATA has a significant limitation relative to WEAVR; the names of elements in the concrete model must match the names of elements in the aspects. Because of the naming limitation, pointcuts will not often be reusable across aspects, and aspects will not generally be reusable across base models. To some extent, this can be addressed with some strict naming conventions, but it will remain the case that the naming constraint reduces the dimensions of freedom,
and will make many tasks more cumbersome. For one example, consider the state machine example from Section 3.2: to cast the appropriate transition as \texttt{unauthorized}, the \texttt{actionPerformed(evt)} transition would have to be renamed \texttt{access()} and the \texttt{Previous} state would have to be renamed \texttt{unauthorized}, which would sacrifice the intelligibility of the diagram. Applying a similar transformation for the \texttt{authorized} state would actually result in an invalid diagram, since two distinct states would then share the same label.

This limitation will serve as motivation for the second part of our proposal, in Section 4.3; we will in fact use our extension to specify precisely this example.

\subsection*{4.2.3 Syntax and Semantics}

We allow \texttt{≪hide≫} and \texttt{≪reveal≫} to be applied to elements within class diagrams, sequence diagrams, and state machine diagrams. For sequence diagrams, asynchronous and synchronous messages are supported; lifelines are excluded, in order to simplify the specification and because something very similar to hiding a lifeline can be accomplished by hiding all messages along that lifeline.

The semantics for the \texttt{≪hide≫} and \texttt{≪reveal≫} stereotypes can be divided into three parts: how hiddenness is determined, how hidden elements are treated when related exposed elements are deleted, and how new elements are woven with existing, hidden elements. First, however, we establish the syntax for the sticky attribute of the \texttt{≪hide≫} stereotype, which is the only attribute in either of the new stereotypes.

\subsubsection*{4.2.3.1 Syntax for the sticky Attribute}

For all UML model elements (except for state models, for which stickiness is unnecessary), the \texttt{sticky} attribute may take
values of: **stickyOutside** or **unsticky**. UML containers can additionally take on the **stickyWithin** value. Messages within sequence diagrams may be assigned any one of the following **sticky** values: **sendStickyAbove**, **sendStickyBelow**, **receiveStickyAbove**, **receiveStickyBelow**, **stickyOutside**, or **unsticky**. Interaction fragments within sequence diagrams may be assigned **stickyAbove**, **stickyBelow**, **stickyOutside**, or **unsticky**.

For the sticky attribute assignment to be valid, there must be an exposed element defined to which the sticky value refers. For example, for a hidden message to have a sticky value of **sendStickyAbove**, there must be an exposed message in the sequence diagram which is prior to the sending of the hidden message. The meaning of each value ascription is defined in Section 4.2.3.2.

### 4.2.3.2 Semantics for the sticky Attribute

Table 4.1 defines the meaning of each possible sticky value assignment, and the implications for element deletion. Table 4.2 describes the effect of the sticky value on weaving semantics for sequence diagrams.

### 4.2.3.3 Determining Hiddenness

An element is hidden if and only if either:

- the `≪hide≫` stereotype has been applied to the element, or

- the element’s immediate container is hidden and the `≪reveal≫` stereotype has not been applied to the element.

New elements introduced by aspects, like all other elements, default to being exposed (that is, not hidden). An aspect may apply the `≪hide≫` stereotype to both new elements and to previously existing elements. When weaving exposed elements into a
Table 4.1: Implications of each of the possible values for the sticky attribute of the `<hide>` stereotype for element deletion.

<table>
<thead>
<tr>
<th>Attribute Value</th>
<th>Referent</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsticky</td>
<td>None.</td>
<td>The element will remain defined, independent of any other element’s fate.</td>
</tr>
<tr>
<td>stickyOutside</td>
<td>The element’s immediate container.</td>
<td>The element will be deleted if its immediate container is deleted.</td>
</tr>
<tr>
<td>stickyWithin</td>
<td>The element’s exposed contents.</td>
<td>The element will be deleted if all of its exposed contents are deleted.</td>
</tr>
<tr>
<td>receiveStickyAbove</td>
<td>The immediate predecessor to the element’s receive event.</td>
<td>The element will be deleted if its immediate predecessor is deleted.</td>
</tr>
<tr>
<td>receiveStickyBelow</td>
<td>The immediate successor to the element’s receive event.</td>
<td>The element will be deleted if its immediate successor is deleted.</td>
</tr>
<tr>
<td>sendStickyAbove</td>
<td>The immediate predecessor to the element’s send event.</td>
<td>The element will be deleted if its immediate predecessor is deleted.</td>
</tr>
<tr>
<td>sendStickyBelow</td>
<td>The immediate successor to the element’s send event.</td>
<td>The element will be deleted if its immediate successor is deleted.</td>
</tr>
</tbody>
</table>
Table 4.2: The weaving rules for each of the possible values for the sticky attribute of the \textless hide\textgreater stereotype, in the context of sequence diagrams.

<table>
<thead>
<tr>
<th>Attribute Value</th>
<th>Referent</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsticky</td>
<td>None.</td>
<td>Other elements may be inserted immediately before or after this element.</td>
</tr>
<tr>
<td>stickyOutside</td>
<td>The element’s immediate container.</td>
<td>Other elements may be inserted immediately before or after this element.</td>
</tr>
<tr>
<td>receiveStickyAbove</td>
<td>The immediate predecessor to the element’s receive event.</td>
<td>No messages may be inserted between this message’s receive event and its immediate predecessor. Other messages may be inserted immediately after this one.</td>
</tr>
<tr>
<td>receiveStickyBelow</td>
<td>The immediate successor to the element’s receive event.</td>
<td>No messages may be inserted between this message’s receive event and its immediate successor. Other messages may be inserted immediately prior to this one.</td>
</tr>
<tr>
<td>sendStickyAbove</td>
<td>The immediate predecessor to the element’s send event.</td>
<td>No messages may be inserted between this message’s send event and its immediate predecessor. Other messages may be inserted immediately after this one.</td>
</tr>
<tr>
<td>sendStickyBelow</td>
<td>The immediate successor to the element’s send event.</td>
<td>No messages may be inserted between this message’s send event and its immediate successor. Other messages may be inserted immediately prior to this one.</td>
</tr>
</tbody>
</table>
hidden container, the weaver applies the ≪reveal≫ stereotype: that is, no elements will be hidden simply by virtue of being added to a hidden container.

4.2.3.4 Handling Deletions. When an exposed element is deleted, all hidden elements that are stuck to it are likewise deleted.

When a state in a state diagram is deleted, all transitions into and out of that state diagram must be deleted. If all transitions into a hidden state have been deleted, the hidden state is also deleted. If a hidden transition’s origin state or its target state is deleted, then the transition is also deleted.

4.2.3.5 Weaving New Elements. There is the possibility of naming conflicts when new elements are created. This is a syntax error; it is a syntax error independent of the exposure status of the pre-existing element. (Recall that the exposure of the element affects only its visibility to the aspect matcher, not its accessibility within the system.)

4.3 Allowing Flexible Implementations: Model Interfaces, Construals, and Badges

WEAVR accomplishes its two-way obliviousness by means of two layers of state machines, the more abstract of which is used in pointcuts. We would like to extend MATA to do something similar, thereby allowing us to take advantage of MATA’s succinct, expressive diagrams and its support for class and sequence diagrams while maintaining WEAVR’s capacity for two-way obliviousness and aspect reuse. To accomplish this, we introduce the notions of model interfaces, construals, and badging.

Model interfaces allow model patterns to be defined in a way that goes well beyond what traditional class interfaces allow, because they may include sequence diagrams,
state machine diagrams, as well as class diagrams. Just as traditional class interfaces specify structural features which classes may implement, model interfaces specify the structural features that a concrete model may implement. Model interfaces can be thought of as a generalization of WEAVR’s more abstract state machines.

An example of a model interface can be found in Fig. 4.9. The model interface is for transactions, specifying that there must be a `perform()` method defined (which will perform the transaction), that there must be a boolean `success()` method defined (which will allow us to determine whether the transaction was successful), and that there must be a `rollback()` method defined (which will reverse the transaction).

Construals are a collection of “is a” claims, relating a concrete model to a model interface. Badges are individual “is a” claims, relating model interface elements to concrete model elements. Taken together, these are a generalization of WEAVR’s realization mappings. In Fig. 4.10, the Account class is construed twice: once with `withdraw()` as the transaction, and once with `deposit()` as the transaction. A word about the notation: we are using a shorthand; here `≪transaction.Subject[0,1]≫` should be understood as two stereotypes, both of which badge the Account class as the Subject from the Transaction model interface. [0,1] indicates that those stereotypes have construal iden-
 identifiers 0 and 1. Since there are two construals on a single model, we must use construal
tifiers to indicate which badges belong to which construal.

A model interface can be used in an aspect to apply a transformation to all model
elements in the system that are construed as examples of that interface. Fig. 4.11
specifies a transaction aspect, which would be woven with the two sequence diagrams
from Fig. 4.10.

4.3.1 Methods and Method Invocations: A Distinction

Messages within model interface sequence diagrams are an interesting case; here
the critical distinction is between a method within the class and a particular invocation
of that method. Sequence diagrams deal with invocations, rather than methods in
general. Therefore, a sequence diagram within a model interface means that there is a
particular invocation of interest, and that it will need to be badged, within a concrete

Figure 4.10: A model, with two Transaction construals.
Figure 4.11: An aspect, specifying the way that transactions should be performed and rolled back.

sequence diagram, as such. To support this, we introduce a distinction between static and dynamic badges; dynamic badges are badges that apply to a single invocation only.

For example, suppose we added a sequence diagram to Fig. 4.9, as shown in Fig. 4.12. Then, to form a syntactically correct construal of this, the concrete model would need not only to badge appropriate methods within its classes (these would be static badges), but also to apply a dynamic badge to a single invocation of the method statically badged as `perform()`. A consequence of this setup is that only one match of that invocation will be made per construal; in a case like the Transaction model interface, this is probably not what we want: probably when we match `perform()`, we would like to match any invocation of the class’s method that has been thus badged.

Additionally, method invocations within model interfaces may or may not have unique class methods associated with them. If the message name is one that is defined for the receiving class by the model interface, then the badged invocation must be of a method thus badged; in Fig. 4.12, the `perform()` invocation must call the `perform()`
Figure 4.12: The model interface from Fig. 4.9, with a sequence diagram added. To implement this model interface, not only would a concrete model need to have appropriately badged class methods, but it would also need to apply, within a sequence diagram, a dynamic perform() badge to a single invocation of the class method that is badged as perform().

method within the Subject class. If the message name is one that is undefined, then implementers are free to apply the interface’s invocation badge to any invocation of a message that has appropriate sender, receiver, and arguments.

4.3.2 State Machine Model Interface Example

In Fig. 4.13(a), a model interface for the unauthorized states and the transitions to them from the state machine example from Cottenier et al. (CvdBE07b), which we discussed in Section 3.2, is shown. Fig. 4.13(b) shows a model interface for the authorized state transitions. Note that, because a transition has unique start and end states, we only need the transition to be mapped in the model interface, and can allow the start and end states to remain anonymous; leaving elements unnamed is one
way to specify wildcard matches in MATA. A transformation rule corresponding to the pointcut and advice given by Cottenier et al. (CvdBE07b) is shown in Fig. 4.14. Finally, a concrete state machine with elements badged according to their participation in the <<unauth>> and <<auth>> model interfaces, is shown in Fig. 4.15.

Our rendering of the example is slightly more fully specified than what we saw in Section 3.2, because we explicitly identify the failure state, where the earlier rendering assumed the weaver could appropriately interpret a state labeled ‘thisStateMachine<<FAILURE>>.’ WEAVR may in fact be able to do so, but a benefit of our approach is that the author of a concrete state machine may mark which of its states should be the failure state, and the failure state could be different for various unauthorized states.

We allow construal matches to be either strict or loose. If loose matches are selected, then MATA will continue to match as it always has done, except that it will also match elements that are construed as context elements within an aspect. If strict
Figure 4.14: An aspect, pointcut and advice, using the model interface from Fig. 4.13(a), for the state machine from Section 3.2.

```
throw new SecurityException("This protocol on" + " component " + thisJoinPoint::getThisClassName() + " is not allowed!");
```

Figure 4.15: The concrete state machine from Section 3.2, badged with ≪unauth≫ and ≪auth≫ model interfaces, according to the rule mapping specified.
matches are selected, then a construal is required; that is, just having elements with names and associations that are the same as those in the aspect’s model interface is neither necessary nor sufficient; they have to claim to “implement” the interface. Whether matches are strict or loose is a feature of the aspect; for compatibility with MATA as it exists today, matches are loose by default.

This proposal has several advantages over MATA by itself:

- It allows two-way obliviousness, because the authors of the base system and the authors of aspects need only know about the model interface; neither needs to know other details of the other’s work.

- It allows flexible naming; where MATA requires either that names within the aspect match the base exactly or that a wildcard match be used, model interfaces allow concrete elements to implement the interface, regardless of their names.

- It allows join point reuse; by referring to an externally defined model interface, aspects can share and reuse join point definitions.\footnote{It is worth emphasizing that, by including one element of a model interface as a matched element in an aspect, the entire model interface is matched (that is, guaranteed to be there). Thus, our extension allows for a much more succinct pointcut specification in cases where only a small subset of the required elements are actually modified by the aspect. By consolidating pointcut and advice into a single diagram, Whittle & Jayaraman made pointcut reuse more difficult. Our approach makes this possible once again, while retaining and perhaps even enhancing the intuitiveness of MATA, because our diagrams only need contain the elements that are most relevant to the rule being specified.}

- It allows performance optimization, in the case of strict matches. Because we only have to look for elements that are construed with a particular interface when doing strict matching, an index of construals could easily be built, and weaving may be considerably faster than less-structured graph transformations would allow.
Also, allowing loose matching means that, just as in MATA, quick and dirty specifications are entirely admissible, as are matches unanticipated by the base designer (although we do consider taking advantage of the latter to be dangerous; certainly it violates the principle of obliviousness).

**4.3.3 ≪modelInterface≫ Syntax**

The ≪modelInterface≫ stereotype can be applied to a UML package, and has the following attribute:

- **name** - a character string identifying the model interface

- **matchType** - **strict** or **loose**

Elements within the package are elements of the model interface. If **matchType** is **strict**, then only elements that are explicitly badged as belonging to this model interface will be matched by aspects employing this model interface. If **matchType** is **loose**, then elements that match the names of elements in this model interface but are not explicitly badged as belonging to this model interface will be matched, if the model does not contain a suitable badged element (that is, explicitly badged elements will always be matched first).

**4.3.4 ≪badge≫ Syntax**

The ≪badge≫ stereotype can be applied to any UML element, and has the following attributes:

- **miName** - the name of the model interface
- miElementId - the name of the model interface element with which this element is associated

- badgeType - either isA or belongsTo

- subElementMap - if the badgeType is isA, then this is a mapping from the sub-elements of ≪MName.elementId≫ to the sub-elements of the badged element

- construalId - an integer used to associate this badge with other badges belonging to the same construal

- subElementName - when a badge is used in a subElementMap, this name is used to identify the concrete sub-element that is being badged

- construalAction one of create, delete, context, or moved

- invocationType either static or dynamic, defined for messages in sequence diagrams.

The miElementId must be sufficiently qualified to distinguish it from other elements. When the badgeType is isA, then a mapping may be defined between sub-elements and the elements; the most common example is methods within a class, but it may also be used for arbitrary containers. A conceptual view of the associations and attributes of badges can be seen in the class diagram in Fig. 4.16.

4.3.5 Syntax and Semantics

Model interface references are made using standard UML package qualifiers; if, for example, the FractalGenerator model interface you wish to refer to is in a package called FractalsOhMy, then you can refer to it as FractalsOhMy:FractalGenerator.
Figure 4.16: A conceptual view of the associations and attributes of the <<badge>> stereotype. We describe one possible mechanism for implementation in the text, using miName and miElementId to determine the associated interface element, and using miName and construalId to determine the construal of which this badge is a member.

4.3.5.1 **belongsTo Syntax.** A concrete element can only belong to a model interface element of the same type, or to the model interface element of one of its containers.

4.3.5.2 **Construal Syntax.** The badging of an element as a member of a model interface determines that a construal is being made (that is, there is no separate construal operator). If there is a construal to one element of a model interface, then there must be isa badges for each of the other elements in that model interface—that is, there are no optional elements in model interfaces. (The possibility of optional model interface elements is something that we consider in Chapter 6.)
4.3.5.3 Badging Syntax. construalId need not be specified if there is only one construal to the model interface. However, only one concrete element may be given an isa badge for a single element in any given construal.

When there are sub-elements in the model interface, then subElementMap must contain mappings to each of the sub-elements in the model interface. It is permissible for multiple sub-elements of ≪MIName.elementId≫ to be mapped to the same sub-element of the badged item; that is, the mapping need not be one-to-one. However, every sub-element of ≪MIName.elementId≫ must be mapped to some sub-element of the badged item.

The case of method mappings is particularly interesting, because two methods may have different numbers and types of arguments. Therefore, each badging of a concrete method as a model interface method that takes arguments requires a subElementMap to be specified for the badge.

The way we specify the subElementMap is as a set of badges, each of which has the same construalId as the concrete element’s badge, and whose miElementId has the form miContainerElementId:subElementId. In this way, we can nest badges to arbitrary depth.

Specifying badges for methods is an area in which we believe a tool with a good user interface will be critical. A tool can provide hints based on argument names and types, so that in the majority of cases, the user may be able simply to accept the tool’s suggested mapping.

It is permissible for items to be badged with both belongsTo and isa badges. This will have the effect that multiple model interface elements belong to each other, as will any concrete elements badged as any of the model interface elements thus grouped.
**ConstrualAction** has defaults according to MATA’s rules—if neither the element nor its container has been stereotyped with ≪create≫ or ≪delete≫, then **construalAction** is context; if the element has been stereotyped, then **construalAction** defaults to the value of the element’s stereotype.

4.3.5.4 Deletion Semantics. When an aspect specifies that a model interface element should be deleted, all concrete elements—both those of type isA and those of type belongsTo—badged as the model interface element are deleted. belongsTo can be seen as a generalization of the concept of stickiness from Section 4.2; elements belonging to a particular model interface item can be understood as being “stuck to” it.

For example, consider the **Account** class in Fig. 4.17. Here, the logWithdraw() invocation belongs to the withdraw() invocation. The aspect in Fig. 4.18 deletes the perform() invocation, and therefore not only the withdraw() invocation but also the logWithdraw() invocation which belongs to it. The refreshDisplay() invocation would remain unaffected.⁵

4.3.5.5 Creation and Movement Semantics. Element creation in class diagrams proceeds just as it does in MATA. This is also true in state machine diagrams, with one exception: sometimes we want to “move” a matched transition from one state to another state. This is what we did in the example in Fig. 4.14. What is meant by this is that the concrete transition should no longer end in its present end state, but should instead end in the one specified by the transformation rule. This can be specified using a

---

⁵ It is worth noting that, at present, our approach does not provide a mechanism by which the logWithdraw() invocation could belong to the withdraw() invocation, unless, as here, the model interface includes the withdraw() invocation. This is a limitation, which might be addressed in future work. One idea would be to use something like the sticky attribute we defined for the ≪hide≫ stereotype.
Figure 4.17: A construal of an Account class as the Transaction model interface from Fig. 4.12. The invocation of logWithdraw() belongs to the invocation of withdraw(); the invocation of refreshDisplay() does not belong to any other elements.

Figure 4.18: An aspect that deletes the perform() invocation from the model interface in Fig. 4.12.
construalAction value of moved—the moved element’s miElementId and construalId will be used to determine which element is meant. It is an error to move an element whose original is not deleted.

For sequence diagrams, the rules for message insertion are just as they were with the ≪hide≫ stereotype’s sticky attribute, the only difference being that instead of using the sticky semantics to determine how messages should be grouped, now we use the belongsTo to determine grouping. Message invocations may be moved within a sequence diagram by specifying a construalAction of move; just as with state transitions, this requires that the original be deleted.

For example, if the aspect in Fig. 4.11 were applied to the model in Fig. 4.17, the invocation of success() (as well as the opt fragment) would succeed the invocation of logWithdraw(), but precede that of refreshDisplay().

Only state transitions and messages within sequence diagrams allow construalAction to have a value of move.

4.3.6 Construal Creation and Deletion

Note that aspects can add construals as well as remove them, by using appropriate construalAction values. The fact that elements belonging to a model interface may be deleted makes this all the more important: for the construal to remain valid, the deleted member must be replaced, by badging another element with the appropriate badge. We leave to the tool implementer the decision of when and whether to allow partially implemented model interfaces—it may be desirable, for example, to allow several aspects each to produce a piece of a complete model interface. A related question is what to do when deletion does leave a model interface invalid. One option is to immediately and
automatically remove the model interface; another would be to warn the user; another still would be to require the explicit deletion of the other badges belonging to the model interface.

4.3.7 Relationship between ≪hide≫ and ≪reveal≫ and Model Interfaces

Finally, we note that the ≪hide≫ and ≪reveal≫ stereotypes in Section 4.2 can be thought of as a lightweight mechanism to specify model interfaces in-line. The model interface elements are then precisely the exposed elements, and elements that are “stuck” to a revealed element belong to that model interface element. Model interfaces thus specified do have several limitations compared to the full model interfaces: names in aspects must match precisely, aspects are harder to reuse, and the sticky attribute does not generally give as much flexibility in grouping as belongsTo badges do.\textsuperscript{6}

4.4 Extension Summary

We thus have specified an extension to MATA that facilitates pointcut and aspect reuse, allows flexible element naming, and, most importantly, allows two-way obliviousness through the use of model interfaces. For simpler specifications, it may be that ≪hide≫ and ≪reveal≫ provide enough power and flexibility; for more sophisticated ones, the more general model interfaces should be used. Using our extension, aspect and base authors can both do their work without knowing the possibly transient details of others’ work; all they need to do first is agree upon the model interfaces. Some initial ideas about how a user interface might be designed to support model interfaces can be found in Appendix C.

\textsuperscript{6} As mentioned above, one exception to this is in the context of sequence diagrams, where for model interfaces that do not specify dynamic invocations, we are unable to group invocations together; for these, the sticky attribute is actually more expressive.
CHAPTER FIVE
Evaluation

In this chapter, we evaluate our approach in several ways. In Section 5.1, we use the criteria from Chapter 2 to compare our approach to the others that we have examined. Next, in Section 5.2 we consider the expressiveness of aspect specifications. In Section 5.3, we evaluate each of the four AOM approaches in terms of their provisions for one- and two-way obliviousness. Then, in Section 5.4, we apply our approach to an example from a case study by Whittle et al. (WAM06). Finally, we conclude with an overall evaluation of our approach.

5.1 Evaluation According to Evaluation Criteria

In Section 2.2, we compared the AOM approaches of Klein et al., Cottenier et al., and Whittle & Jayaraman according to several criteria:

- diagram types supported (sequence diagrams, state machine diagrams, and/or class diagrams)
- aspect types supported (behavioral, structural, or both)
- maturity (real-world use or not)
- tool support (modeling, weaving, and/or code generation support)
- degree of formalism
- support for decoupling aspects from base systems
Like MATA, our approach supports all three diagram types and both behavioral and structural aspects. Because it is a new proposal and not yet implemented, our approach has not yet seen real-world use. Similarly, it does not yet have tool support, although, because it extends MATA, adding tool support for modeling and weaving should be simpler than it otherwise might be. Our approach, while less formal than that of Klein et al., is quite specific in terms of both syntax and semantics. Like WEAVR, our approach supports decoupling aspects from base systems, which we call two-way obliviousness. We discuss this further in Section 5.3.

5.2 Aspect Matching and Expressiveness

Each of the four aspect approaches we have examined—Klein et al.’s sequence diagram weaving, Cottenier et al.’s state machine models using WEAVR, Whittle & Jayaraman’s graph transformations using MATA, and our proposed extension of MATA—provides a way for an aspect author to quantify over other models in the system, as well as a way to transform those models. In this section, we examine each of the four approaches with respect to the flexibility and expressiveness of their quantifications.

Observe that when we specify a pointcut to match, say, sequence diagrams, we have in mind both a set of sequence diagrams that should match and a set that should not match. This is to say that we have in mind a partition of the space of sequence diagrams, and we write the pointcut to express that partition. As we will see, the approaches vary in their ability to express an arbitrary partition in a pointcut, together with—in the case of WEAVR and our extension to MATA—an interface specification and mappings to that interface.
Figure 5.1: A partition of the space of sequence diagrams which cannot be reached by any of the part types specified by Klein et al. The `noMessage()` message with an ‘X’ through it indicates that no messages should be sent from `FractalGenerator` to `Controller` between `FractalGenerator`’s receipt of the `generate()` message and its receipt of `applyColorScheme`.

In this section, we will consider the spaces of sequence diagram partitions available to each of Klein et al.’s approach, MATA, and our extension to MATA. We do not consider WEAVR, because it uses state machine diagrams, and we would expect its space of possible state machine diagram partitions to be identical with those in our extension to MATA. In this section, our goal is simply to get a rough sense for the relative flexibility of the various approaches in matching.

5.2.1 Expressiveness of Klein et al.’s Pointcuts

Klein et al. allow four types of matches, varied in terms of the strictness of the match. It is not difficult to come up with desirable partitions that are not expressible with these four types of matches. Consider the partition specified in Fig. 5.1—which can easily be seen as an effort to match diagrams like Fig. 4.2, while excluding diagrams like Fig. 4.5. Using Klein et al.’s approach, the pointcut will need to match the `generate()` and `applyColorScheme()` messages, but also to allow an un-
Figure 5.2: MATA’s pointcut for the partition specified by Fig. 5.1.

named message `someMessage()`. Because the `someMessage()` message is sent on the FractalGenerator lifeline between receipt of the `generate()` message and receipt of the `applyColorScheme()` message, then to make the match we want to make, we must use either a safe part, or a general one. Either of these, however, will also match sequence diagrams with a message where the partition specifies there should be no message.

5.2.2 Expressiveness of MATA’s Pointcuts

MATA provides a somewhat more traditional dimension of flexibility in matching; it allows two kinds of wildcards in sequence diagrams—any single message can be matched by naming its placeholder in the rule with something of the form `\textbackslash x` and a sequence of messages of arbitrary length, origin, and destination may be matched using its `any` fragment, shown in Fig. 3.8. This does allow it to produce a rule that would establish the partition that failed for Klein et al.; a solution is shown in Fig. 5.2.

This is quickly seen to be a special case; it is not generally possible to specify partitions that exclude messages in one direction or another. For example, consider Fig. 5.3. Here, we want to match diagrams that have zero or one message where the
Figure 5.3: A partition of the space of sequence diagrams which cannot be reached by a MATA pointcut.

message possibleMessage is shown. The solution in Fig. 5.2 will fail when no message is there; in order to match both when there is a message and when there is no message, we must use the any fragment—this, however, will also allow messages at the excluded point, where the message noMessage() is shown on the diagram.

There are also some partitions that Klein et al.’s pointcuts can reach that MATA cannot—namely, those corresponding to safe matches and strict matches. However, we argued in Chapter 3 that these were not interesting partitions.

5.2.3 Expressiveness of Pointcuts in Our MATA Extension

Because our extension provides a superset of MATA’s functionality, it is evident that its expressiveness (that is, its space of partitions specifiable by pointcuts) will at least match that of MATA, for sequence diagrams that do not use the ≪hide≫ stereotype. It is also evident, however, that the rules have changed somewhat by the addition of the ≪hide≫ stereotype; this allows a sequence diagram author to specify what will be seen. If said author hid a message at the noMessage() point in any of the above partitions, then the matcher would be (by design) unable to discriminate between
that sequence diagrams and the ones that ought to match. We have argued that this is a desirable feature, because the author of the diagram is the one with the best insight as to what should be matched.

The model interface mechanism provides a very powerful mechanism for quantification; we can determine the features (messages) of interest, and specify these in a model interface, and then badge those that ought to match appropriately. Our pointcuts can quantify over messages with different names without matching arbitrary messages, which neither MATA nor Klein et al.’s approach can do.

5.3 Managing Obliviousness

Our central goal in extending MATA is to work towards a general mechanism for aspect-oriented modeling, which MATA provides, that tells a compelling story regarding obliviousness, which MATA by itself does not do. As we argued in Section 4.1, the “one-way” obliviousness that naive aspect-oriented environments provide promotes fragile coupling between aspects and the base features they modify. Environments such as WEAVR offer what we have called two-way obliviousness, allowing an interface of sorts to be established that allows authors of both aspects and the base system to work together without fragile dependence on potentially transient details of other authors’ designs. In this section, we consider obliviousness in the context of each of the four AOM approaches.

5.3.1 Obliviousness in Klein et al.

Klein et al.’s four match types allow some control over which aspects match which base models. Although they do not explicitly consider obliviousness, in a sense this
is what is at play in the selection of the match type: a more flexible match might be thought of as more oblivious, because it has fewer dependencies on the details of the base implementation, and a stricter match will be less oblivious because it depends heavily on those details. Because the level of obliviousness is selected when the pointcut is specified, it is the responsibility of the aspect designer rather than the base designer—which fits well with Filman & Friedman’s definition of obliviousness (FF00).

The person who knows best whether an element ought to be visible, and in which contexts (that is, to which aspects), is the author of that element. A difficulty with Klein et al.’s approach is that the decision of which elements are “hidden” lies entirely in the hands of the pointcut author and the one who specifies the type of part to use in matches. Thus, Klein et al.’s approach offers exactly the one-way obliviousness that we have argued against.

5.3.2 Obliviousness in WEAVR

By virtue of addressing the problem of coupling between aspects and the base systems they modify, WEAVR has also addressed obliviousness: the solution is to explicitly specify states that can be matched within an “interface-layer” state machine. They thus effectively deny that one-way obliviousness is a virtue; their approach requires the involvement of the base-level designer, with the possible exception of cases in which the interface-layer state machine can be specified without modifying the implementation-layer state machine. We agree; as we have argued in Section 4.1, the base designer is generally the one who ought to specify the aspectual interface. Thus, WEAVR offers two-way obliviousness for state machine diagrams.
5.3.3 Obliviousness in MATA

MATA by itself provides little by way of support for two-way obliviousness. Some flexibility in matching is provided by wildcards, but generally matched elements will have names explicitly specified in aspects, requiring aspects to see into the naming and structural details of the base systems they modify. This is exactly the situation we saw with Klein et al.: MATA offers only one-way obliviousness.

5.3.4 Obliviousness in Our MATA Extension

Our $\ll$hide$\gg$ and $\ll$reveal$\gg$ stereotypes provide a lightweight mechanism by which model authors (whether aspect authors or base authors) can control the visibility of structural elements to the aspect matcher. This, however, does not provide any flexibility of naming; it still requires names in aspects to match names in base elements. Flexible naming of aspect elements is important precisely because aspects usually are applied because of some non-dominant (that is, cross-cutting) characteristic of the base elements; therefore, the names of the base elements will usually mean little in the context of an aspect. Our model interface extension allows base elements to be named entirely independently of the aspects that modify them; it only requires that the appropriate non-dominant characteristics of elements be specified using badges. Our MATA extension thus offers multiple mechanisms for two-way obliviousness for all the diagrams that MATA supports.

5.4 Extended Example: Two-Phase Commit

Whittle et al. report a case study on a weather station that had two models that needed to implement a two-phase commit protocol (WAM06). They specify this
using their RBML-based graph transformational approach, which we mentioned briefly
in Section 3.3. An interesting feature of this problem is that it employs primitives that
MATA does not provide, including the ability to specify that all of a group of objects has
received a particular message (e.g., \texttt{prepareToCommit}). These primitives are detailed in
the next paragraph. It seems likely that MATA could be extended to provide support for
such primitives; this is an area for future work. In this section, we will use our extension
to MATA to specify a two-phase commit protocol with a set number of participants
(namely, two). Our solution will work for an arbitrary number of clients; only the base
model makes any assumption about the number of participants, and it is easy to see
how this also could be extended to an arbitrary (but fixed) number of participants.

The additional notations not available in UML 2.0 that Whittle et al. use in
their solution are as follows: they use a \texttt{multiobject} tag to indicate that there are
multiple objects represented by a single lifeline, and in messages to (from) the multi-
object lifeline; they use \{\texttt{all}\} to specify that each object receives (sends) the message;
and they use \{\texttt{exist}\} to indicate that some object receives (sends) the message. Their
diagram, showing a general two-phase commit protocol and employing these notations,
is shown in Fig. 5.4. These additional notations are not available to MATA, although
their does not seem to be any reason why they could not be added. Nevertheless, for the
purposes of this example, we use only MATA and our Model Interface extension. This
creates some additional complexity, but the exercise of using our extended MATA to
solve the problem will give us some sense of how it might be used in real-world systems,
and some sense of both its power and its limitations. One limitation we have already
encountered: it would be more powerful still if it were extended to support the additional
notations Whittle et al. employ.
Recall that a two-phase commit (2PC) protocol requires the following two phases:

1. **prepare to commit** If any participant refuses to commit, all participants abort.

2. **commit** If any participant cannot complete the commit, all participants abort.

Once all participants have signaled successful commitment, the commit is considered to be complete.

The messages in Fig. 5.4 such as \{all\} \textit{prepareToCommit} are sent in parallel; therefore, the standard UML equivalent is to specify each participant’s lifeline separately, and to use the \texttt{par} interaction fragment type to indicate that the messages are sent in parallel.

Our model will need to have several parts:

- a concrete model, taken from Whittle et al.’s case study (modified to express it in standard UML notation),
• one or more model interfaces which the concrete model can be construed to implement, and

• one or more aspects, expressed as graph transformations of the model interface elements.

Our concrete model can be found in Fig. 5.8. We have elected to include two clients, where Whittle et al., thanks to the ≪multiobject≫ notation, included an arbitrary number of them. (Our rules and model interface definitions do not rely on having exactly two clients; they can be applied to an arbitrary number of clients.) The figure includes two construals: a commit and a finalCommit construal. The reason for having two model interfaces which apparently express the same thing will be clear in a moment; it has to do with the way our transformation rules are applied.

We define three model interfaces commit, finalCommit, and 2PC. commit, shown in Fig. 5.6, ensures that clients have the required methods defined, and that they receive a prepareToCommit message from the Server. finalCommit, shown in Fig. 5.5, defines a unique “final” prepareToCommit message (they are all sent in parallel, so what is meant here is just that the message is the last one listed on the diagram) and the sending of the enable message, which occurs upon success of the 2PC protocol.

We define four graph transformations, R1, R2, R3a, and R3b. R1 must be applied before R2 and R2 before R3a or R3b, but R3a and R3b may be applied in either order. The effect of each rule is as follows:

• R1 produces the overall 2PC structure with exactly one of the clients.

• R2 weaves the other clients into the structure produced by R1, but does not yet send any abort messages to clients other than the one that caused the abort.
Figure 5.5: The `finalCommit` model interface.
• R3a adds the missing `abortRefuseToCommit` messages.

• R3b adds the missing `abortIncompleteCommit` messages.

Why could we not simply use something like R1 for each of the clients, rather than introducing a second rule R2? Repeated application of R1 to multiple clients in the base diagram would result in a series of single-client 2PC protocols. We require clients to run in parallel in each step of 2PC; repeating R1 would serialize them. Once R1 has been applied, however, we have established good join points for each parallel grouping, and R2 takes advantage of this.

A similar question may be asked of R3a and R3b—why can we not include the `abort` messages in R1 and R2? A sensible implementation of 2PC will usually propagate `abort` messages immediately, so that clients that have not finished processing can save
themselves the effort. We also require that the abort messages be sent to all clients. The main difficulty here, then, lies in the fact that MATA does not include support for multiple simultaneous matches. We cannot say something like “send this message to all clients.” Instead, we must find a way to specify this as a pattern, repeated application of which will amount to sending the message to all clients, without introducing spurious matches. This is precisely what R3a and R3b do.

It is worth noting that the specification is incomplete, because the alt interaction fragments for each client’s choice between readyToCommit and refuseToCommit, as well as that between commitComplete and commitIncomplete, do not include the required boolean guard clauses. These were omitted from Whittle et al.’s specification, so we have followed them here, but it would be a simple enough matter to add boolean methods isReadyToCommit() and commitIsComplete() to the Client model interface, and use these to determine each client’s branching behavior. Similarly, we do not include a
Figure 5.8: Our version of the diagram in Fig. 5.7. The «modelInterface.member» notation should be understood as shorthand for a «ModelInterface» stereotype badging the element as the specified member of the specified model interface. Similarly, «lastClient» should be understood as shorthand for «finalCommit.lastClient». With the exception of that shorthand, the model is in standard UML notation.
Figure 5.9: R1, the first graph transformation rule for the 2PC protocol. This must be applied before R2 (Fig. 5.10) can be applied.
Figure 5.10: R2, the second graph transformation rule for the 2PC protocol. This can only be applied after R1 (Fig. 5.9) has been applied, and must be applied before R3a (Fig. 5.11) or R3b (Fig. 5.12) can be applied.
Figure 5.11: R3a, the third graph transformation rule for the 2PC protocol. This can only be applied after R1 (Fig. 5.9) and R2 (Fig. 5.10) have been applied.

Figure 5.12: R3b, the fourth graph transformation rule for the 2PC protocol. This can only be applied after R1 (Fig. 5.9) and R2 (Fig. 5.10) have been applied.
mapping from the WA Clients’ methods to corresponding methods in the FinalCommit and Commit model interfaces, as that level of detail was not provided by Whittle et al.

The fact that we required four graph transformation rules for this specification may be an indicator that our modeling language could be made more succinct; the \{all\} and \{exist\} constructs used by Whittle et al. in their case study (WAM06) would be a good start. At the same time, however, the fact that we are able to complete the specification without such powerful constructs is an indicator of the expressiveness of our modeling language.

5.5 Evaluation Conclusions

We have seen that our proposed extension to MATA fares well according to the evaluation criteria we discussed in Chapter 2, with the exception of maturity and tool support; this is just to say that it is a new proposal not yet implemented. It also is at least as expressive as the other approaches we examined, and provides the support for two-way obliviousness that was its motivating feature. We have successfully specified a relatively complex two-phase commit aspect using our approach. Much remains to be done, however, and to this we turn in Chapter 6.
CHAPTER SIX

Conclusion and Future Work

AOP has already begun to prove its usefulness by allowing more thoroughgoing separation of concerns than OOP by itself allows. However, one problem with current AOP is its encouragement of a fragile dependency of aspects on base classes. As we develop AOM techniques, we have the opportunity to correct this, by introducing language features that encourage better design. Our model interfaces are an example of this.

It is also worth noting that, as we abstract further from the hardware—as we move to higher level models—there is both greater opportunity and greater necessity for including further information regarding our intent for model elements. When machine code is written to perform a complex calculation, there is little chance for ambiguity, and also relatively little chance for an intelligent compiler to do meaningful performance enhancement. That same calculation, however, could be accelerated for specific hardware (say, a multi-core system, or a GPU), if the intent of the calculation is codified in a way that is mathematically precise and does not make assumptions about the underlying hardware.¹ One advantage of an approach like our model interfaces is that it provides the opportunity to add layers of meaning by stereotyping model elements as members of

¹ Software Transactional Memory (STM) is one example of a recent attempt to provide language constructs that make the specification of a program nearer to the intent of its author (HMPJH05); it allows the author to specify groups of atomic instructions, with the aim of flexible, painless concurrency. The compiler and runtime environment guarantee that code within atomic blocks will run atomically; that is, none of the memory touched by any of the instructions may be altered from outside the current thread while the atomic block is executing; if it is, then the block will be rolled back and re-executed. The compiler and runtime are then free to execute blocks in parallel. The important point for our purpose here is that, historically, programs have either been wholly sequential or have required explicit threading instructions, even though in the usual case a programmer would be happy with a large number of possible approaches to threading. STM provides a way to relax unnecessary requirements (sequential execution of logically independent code) while retaining necessary ones (sequential execution of code that depends on earlier code).
particular model interfaces; we hope that future compilers and other MDA tools can take advantage of this information to ensure correctness of woven models and executables, as well as to produce efficient executables across a variety of hardware.

Future work can be divided into two broad areas:

- further exploration of and enhancements to MATA, and
- implementation of, and further enhancement of, our model interface proposal

### 6.1 Further MATA Enhancements and Exploration

#### 6.1.1 Efficiency Analysis and Enhancement

The existence of the simple models that cause pathological asymptotic performance suggests that MATA has a good deal of room for performance enhancement. A detailed analysis of MATA’s translation of UML models into graphs, and of AGG’s transformation of those graphs into CSPs, should be performed. Very likely this analysis would explain the pathological performance that we saw. Given the rich features of the graphs that MATA is transforming—nodes are elements with types and names, both of which are often used to do matching and each of which could be used to produce an index for faster matching—we are optimistic that an efficient implementation will be feasible.

#### 6.1.2 Linguistic Richness

As we saw in Chapter 5, there are several linguistic features that would enable MATA’s diagrams more succinct and able to express a wider variety of pointcuts. The addition of the ≪multiobject≫ stereotype, and the {all} and {exist} message mod-
ifiers, as well as some more sophisticated variants of these—e.g., one that would allow one to specify that, say, at least two of the multiple objects responded—would make it easier to specify the behavior of sets of objects succinctly. Pointcuts could also be made more expressive by adding exclusionary matching—the ability to say that there is no element with a given name in specified association to other matched model elements.

6.1.3 Code Generation

Currently, MATA does not offer any code generation features. It would be nice to provide this, at least for state machine diagrams where the specificity of the diagrams is such that code generation is feasible. Providing that much, together with an implementation of our model interface proposal, would make MATA’s features a superset of those that WEAVR offers.

6.1.4 Tool Workflow

MATA is an unpublished prototype, and currently uses IBM’s Rational Software Modeler™ (RSM), but RSM is only used to produce standard UML2 model files. RSM is susceptible to the critique presented by Stephens (Ste08); there is quite a lot of repetitious pointing and clicking that would be unnecessary if a dedicated MATA tool were developed. We believe that it would also be possible to develop a simple textual format for specifying models; part of the toolset could then be a mechanism to translate such a format into the UML2 format that MATA uses. The textual format could be used until a full graphical version were developed, and even then, the textual format could prove useful to script authors attempting to automate model specification.
6.2 Implementation and Enhancement of the Model Interface Proposal

6.2.1 Implementation

Clearly, a critical element of our future work will be to implement our proposed model interface extension of MATA—and, using this, to add support for the `<<hide>>` and `<<reveal>>` stereotypes as well. We believe that it will be possible to take advantage of much of the work that MATA already does, by transforming concrete graphs according to the model interface construals contained within them. We hope to do that transformation without solving subgraph homomorphism problems; instead, we intend to use the badges within each model to produce an index of badged elements.

6.2.2 Optional Elements and Elements with Arbitrary Multiplicity

We would like to add support to model interfaces for optional elements. Clearly, this does limit the assumptions that developers can make about the elements, and it is this kind of concern that kept us from including optional elements in this proposal. Very likely, there should be some mechanism for specifying which elements of a model interface might not be provided by all concrete instances. Similarly, we would like to allow model interface elements with multiplicities: we would like to allow all the multiplicities that UML allows to be applied to any element. For example, you could apply multiplicity * to the sequence diagram in Fig. 4.12 to allow no invocations of the method, as well as an arbitrary number of them. An optional element can be seen as the special case in which the element’s multiplicity is 0..1.
6.2.3 Model Interface Inheritance

Our MATA extension could be enhanced further by allowing model interfaces to inherit from other model interfaces. For example, our specification of the \texttt{finalCommit} model interface in Chapter 5 could have been more succinct had it been able to inherit the elements in the \texttt{commit} model interface.

6.2.4 Model Interface Examples

The 2PC example we saw in Chapter 5 suggests that certain aspects may be broadly applicable. Development of a library of such examples would save individuals from repeating the effort of specifying the model interfaces, but, and perhaps more importantly, would demonstrate and promote effective use of the mechanism.

6.2.5 User Interface Development

The presence of potentially many model interface construals within a concrete model poses the danger that the model diagram would become cluttered and unwieldy. Care must be taken when developing tools to support model interfaces, so that a user can easily apply a model interface to a concrete model and can quickly see which concrete elements implement a given model interface. In Appendix C, we discuss a few initial ideas for an efficient, effective user interface.

6.3 Conclusion

If AOP is in its infancy, AOM has yet to be born. As we work toward a robust approach to AOM, we will need good tools that allow our models to communicate clearly and succinctly, that also promote good collaboration between developers. We have argued that two-way obliviousness is one way in which good collaboration can be
achieved, and proposed an extension to MATA that allows two-way obliviousness, while retaining MATA’s clarity, succinctness, and generality.
APPENDIX A

Klein Matching Algorithm and Performance Analysis

A.1 Matching Algorithm

Klein et al. propose an algorithm to match pointcuts, defined as a bSD with a part type selection. Jacques Klein provided us with his implementation of the enclosed part algorithm, which has allowed us to determine their algorithm somewhat more fully, as well as to perform the experimentation and analysis found in Section A.2. Given a bSD $M$ and a pointcut $P$, this proceeds as follows:

1. For every object $i \in I_P$, construct a set $V(i)$ of sets of all events $v$ on the lifeline of $i$, such that $v$’s set of action names is equal to the set of names of events in the projection of $M$ onto $i$.\(^1\)

To construct $V(i)$, consider the projection of $M$ onto $i$. The events in this projection are totally ordered, because the projection is onto a lifeline, and events on a lifeline are totally ordered. $v$’s events are likewise totally ordered. For the enclosed part, we allow events to come before and after the matched events, and even “around” them, in the sense described above, but if we consider the sending and receiving of a message as separate events, then what we require is exactly that the matched events be in the appropriate order and contiguous.

Thus if there are $k$ events in the projection of $P$ onto $i$, to find a match in this

\(^1\) Although Klein et al. do define the projection from a bSD onto one of its objects, here the projection is from $M$ onto an object in $P$, which they do not define in the article. By inspecting the code provided by Klein, we determined that the projection of $M$ onto $i$ uses the name of $i$ (an object in the pointcut $P$) to select an object of the same name in $M$. (It appears that Klein et al.’s definition of a bSD should be augmented with a set of object names and a mapping from the set $I$ of objects, to make formal definition of this projection possible.)
step we seek a set \( v \) of \( k \) contiguous events, with action names that match those in the projection of \( P \) onto \( i \). Note that the action name comparison here is done setwise; the order of the names is not yet considered.

Klein’s implementation thus constructs a set of contiguous event sets in \( i \) of length \( k \) using a sliding window and iterating over the events of \( i \).

(2) Iterate through the set of matches \( v \in V(i) \), eliminating any whose event ordering on \( i \) does not match that of the projection of \( P \) onto \( i \).

(3) Determine the “earliest” match, using a \( \text{min}() \) function (which Klein et al. define, but is not necessary to our discussion); it is this function which selects matches along individual lifelines in such a way that there is a match across all lifelines. This is a join point, which will be used (in step 6) for weaving.

(4) Remove the events in the join point, and any events which precede them, from the copy of the bSD used for matches. Because of the use of the \( \text{min}() \) function, we know that the preceding events removed have already been passed over: they do not match \( P \).

(5) Repeat steps 1-4 until there are no more matches.

(6) For each join point found, weave in the advice at that join point. This is a non-trivial procedure, given that there may have been messages around and, for safe and general parts, between the matched messages. Klein et al. propose an operator called the left amalgamated sum to do this in a sensible way.

A few comments are in order here. In several places, Klein’s approach discards information that would be useful later in the algorithm. For example, step 2 could be
eliminated altogether if, in step 1, we bail out of a putative event-set match as soon as an event fails to match.²

\[2\]

A.2 Performance Analysis: Experiment and Optimization

A.2.1 Performance of the Enclosed Part Algorithm: An Experiment

We used Klein et al.’s Kermeta-based SDWeaver to verify experimentally our performance predictions. Consider the pointcut and advice in Fig. A.1, and the base in Fig. A.2. Because there are \(n\) copies of the message sequence that the advice matches, and the algorithm deletes each match as it proceeds (so that a single instance of a message cannot be matched more than once in the course of the algorithm), there will be \(n\) matches.

In our experiment, we ran Klein’s prototype implementation on these sequence diagrams, allowing \(n\) to vary from 1 to 10, recording the time taken to complete the weaving for each. The results are in Table A.1.

To get a sense for the asymptotic behavior as \(n\) grows, we graphed the results (shown in Fig. A.3).

Observe that the first cycle of Klein et al.’s algorithm will reduce our experimental input for \(n\) to that for \(n - 1\), because step 4 deletes the matched messages. Define \(f(n)\) the time that the algorithm takes to make and weave its first match. This is arguably the more interesting value, since typical real-world examples will usually have \(O(1)\) matches,

² For general and safe part matches, this strategy would require modification, since the failure of the next event to match does not necessarily indicate that the putative match fails. For these, Klein et al. define the sets formally, but do not specify the manner of their construction. Although we have Klein’s implementation for the enclosed part, we do not have it for any of the other part types. Their description of these algorithms, however, retains a separation between the set construction and the determination of the minimal match, which we believe will impose a considerable performance cost: we suggest combining steps 1 and 2 for this reason.
Figure A.1: The aspect we used in our experiment to determine the asymptotic performance of Klein et al.’s enclosed part weaving.

Figure A.2: The sequence diagram we used as base in our experiment. $n$ was allowed to vary from 1 to 10.
Table A.1: Timing results for the Klein et al.’s enclosed part algorithm using the aspect in Fig. A.1 and the base in Fig. A.2.

<table>
<thead>
<tr>
<th>Repetitions</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.965</td>
</tr>
<tr>
<td>2</td>
<td>5.125</td>
</tr>
<tr>
<td>3</td>
<td>12.395</td>
</tr>
<tr>
<td>4</td>
<td>26.641</td>
</tr>
<tr>
<td>5</td>
<td>52.213</td>
</tr>
<tr>
<td>6</td>
<td>96.038</td>
</tr>
<tr>
<td>7</td>
<td>162.971</td>
</tr>
<tr>
<td>8</td>
<td>266.106</td>
</tr>
<tr>
<td>9</td>
<td>416.594</td>
</tr>
<tr>
<td>10</td>
<td>620.384</td>
</tr>
</tbody>
</table>

Figure A.3: The time SDWeaver took to compose the base in Fig. A.2 with the aspect in Fig. A.1.
rather than the $O(n)$ matches our experiment had. The values in Table A.1 are then of the form $F(n) = \sum_{k=1}^{n} f(n)$. Thus $f(n) = F(n) - F(n - 1)$. Assuming that $F(n)$ is a polynomial in $n$, this means that $O(f(n)) = \frac{O(F(n))}{n}$.

We then used a curve-fitting tool\(^3\) to approximate the asymptotic behavior of $F(n)$; we found that this is $O(n^4)$. Given the argument above, $f(n) = O(n^3)$. This fits well with our analytic expectations; the part of Klein et al.’s implementation that was dominant has running time $O(n^3)$.

### A.2.2 Optimization Using a Lazier Transitive Closure

This result is clearly far from optimal; after all, a naive algorithm could do both matching and weaving in $O(n)$ for the experimental input. Klein et al.’s algorithm is, of course, more general, so we may not expect to do as well as this, but $O(n^3)$ seems a little extravagant. Therefore, we looked for ways we could speed up the implementation Klein provided us.

As described above, one way to simplify the approach is to eliminate step 2 by adding a verification of event ordering to the first step. We implemented this, but were surprised to find that it made no measurable difference to the running time of the algorithm. Analyzing the timing more carefully, we found that the dominant term was the time spent in a transitive closure construction, used to accomplish step 4’s deletion of all events prior to the ones matched.

The transitive closure implemented by Klein et al. produced a hash table which could be used for quick look-up of all events that were, according to the partial ordering

---

\(^3\) The tool can be found at zunzun.com; this examines over 1800 types of function, and returned the closest approximations. The nearest two functions were exponential, and could be eliminated based on the fact that the running time of the program should continue to grow (specifically, should never diminish) as $n$ increases; each of these functions had maxima at a positive, finite value of $n$. The third-nearest function was a fourth-degree polynomial, which fits well with our analytic expectations.
Table A.2: Timing results for the Klein et al.’s enclosed part algorithm, with a modified transitive closure construction, using the aspect in Fig. A.1 and the base in Fig. A.2. Times are in seconds.

<table>
<thead>
<tr>
<th>Repetitions</th>
<th>SDWeaver</th>
<th>With Lazy Transitive Closure</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.965</td>
<td>1.473</td>
</tr>
<tr>
<td>2</td>
<td>5.125</td>
<td>3.698</td>
</tr>
<tr>
<td>3</td>
<td>12.395</td>
<td>7.347</td>
</tr>
<tr>
<td>4</td>
<td>26.641</td>
<td>13.295</td>
</tr>
<tr>
<td>5</td>
<td>52.213</td>
<td>20.781</td>
</tr>
<tr>
<td>6</td>
<td>96.038</td>
<td>31.782</td>
</tr>
<tr>
<td>7</td>
<td>162.971</td>
<td>44.34</td>
</tr>
<tr>
<td>8</td>
<td>266.106</td>
<td>61.698</td>
</tr>
<tr>
<td>9</td>
<td>416.594</td>
<td>82.032</td>
</tr>
<tr>
<td>10</td>
<td>620.384</td>
<td>109.245</td>
</tr>
</tbody>
</table>

≤, prior to a given event (as well as those that come after it according to ≤)—because the order is only partial, this construction involves determination of which events are comparable to a given event. Unfortunately, the construction of the transitive closure is an expensive operation; its asymptotic running time is $O(n^3)$.

Observe, however, that the transitive closure of the entire event set is considerably more than is required in step 4: we need only determine the events prior to (not the events after) the events matched in the pointcut (and only the events matched, not the entire event set). We thus implemented something equivalent to the transitive closure, constructed in an appropriately lazy fashion.

Our modification produced considerable speedup in our experiment; a comparison of the running times is shown in Table A.2. Performing the same analysis as we used to evaluate the running time of Klein et al.’s original implementation, we found that the
asymptotic running time of this modified algorithm was $O(n^2)$. The results are graphed in Fig. A.4.

Examining the timing detail, it is now evident that the majority of the time is spent composing advice at the joinpoints; for example, 87 of the 109 seconds for $n = 10$ are spent in composition. It therefore seems likely that the algorithm could complete in linear time on our experimental input, if the left amalgamated sum were replaced by a linear composition method. Although it lies outside the scope of the present work, we do believe that a linear composition method is eminently achievable.
APPENDIX B

MATA Performance Exploration

As we mentioned in Section 3.3.2, there are some reasons to wonder about MATA’s computational performance when weaving aspects, as models grow. In this appendix, we discuss some initial experiments.

B.1 Repetition of Experiment from Section A.2

We used the same models as we used to test the performance of Klein et al.’s sequence diagram weaving in Section A.2. We let \( n \), the number of copies of the matched messages, vary from 1 to 16. The results are in Fig. B.1. Note that this experiment was run on different hardware from the one we ran on Klein et al.’s prototype, so the results are not directly comparable; what was of primary interest to us was the shape of the graph.

We again used a curve fitting program to analyze the data, but here our results were less conclusive; the best fit functions was of the form \( \frac{p(x)}{q(x)} \), where both \( p(x) \) and \( q(x) \) are third-degree polynomials, but since that had a constant asymptote for large \( x \), we could eliminate it (we expect to be at least linear asymptotically, and probably significantly superlinear). The second-best fit was an exponential function: \( 870(e^{0.55x-6.3}+1)^{0.29} - 870 \).

We could not eliminate this function; whereas for Klein et al.’s algorithm, we had analytic reasons to expect that the running time would be a polynomial in \( n \), here we had no such reasons.\(^1\) In fact, although AGG aims to reduce the average case complexity of graph

\(^1\) Given the complexity of the CSP translation, it seems entirely possible that the function describing the scaling behavior for our tests would not even be among those considered by zunzun.com’s curve-fitting software.
transformations, it cannot reduce the worst case to a polynomial function, because that would amount to proving that $NP = P$ (Rud00); thus, we can expect there will be some kinds of graphs for which AGG scales badly.

As we saw in Section A.2, we might expect that our experimental setup has introduced an additional $O(n)$ factor to our time complexity, because there are $n$ matches in our base model, and after the first match has been made, the problem reduces to the one solved in the previous test. Thus we might expect that for a base sequence diagram such
as the one in Fig. B.2, in which there are 22 messages (two more than in our previous $n = 10$ test) and exactly one match, the time taken should be the difference between our prior $n = 10$ and $n = 9$ tests: approximately 43 seconds. We tried this, and found that MATA performed far better than we would have expected: it completed in just under 3 seconds. (We also performed the same experiment for a similar graph with 41 messages in all, ending in a single match; MATA produced the composition in 12.5 seconds.) For this reason, we believe that something about the way that MATA translates the UML model into a graph, combined with AGG’s translation of the subgraph homomorphism problem into a CSP, creates pathological performance when the number of matches is very large. On the other hand, as we mentioned in Section A.2, our particular example is one for which a naive approach would be quite effective: by simply reading the messages in order and comparing them with the graph transformation rule, we could produce the transformed graph in linear time.

We believe that MATA’s relatively poor scaling performance in our trials is not an inherent limitation of a graph-transformational approach, though it remains to be seen whether this can be corrected by modifying the way that MATA uses AGG, or whether more fundamental changes need to be made to the way that MATA implements its graph transformations. Our main reason for believing that better performance can be achieved is the amount of structure that is present in the graphs that MATA transforms: elements have both types and names, and the graphs are usually sparse. The types and names can both be used to build element indexes; the sparseness means that the number of edges that must be examined to determine whether there is a matching subgraph will be limited. The model interfaces approach that we propose in Section 4.3 could provide even greater gains in efficiency, if an index of model interface construals were built.
APPENDIX C

Tool UI Ideas

In this section, we consider a few ideas for a user interface that would allow users to specify aspect models more effectively and efficiently.

C.1 Badging Icons

One way to make it easy to quickly identify which elements belong to a given model interface is to include small icons (non coincidentally, known as badges) for each element of a model interface. For example, in Fig. C.1, a red “Do not enter” symbol might adorn model states that are badged as “unauthorized,” and a green dot could adorn those that are “authorized.” Conveniently, UML stereotypes definitions can include icons (Gro07). When a model interface is specified, the user would be given a chance to specify icons for each element. If the user did not select one, a textual representation of that model interface element will be displayed instead. The user interface should allow visual badging to be turned on and off at a global level (i.e., for all model interfaces) or at a per-interface level, allowing the user to see the interfaces of current interest to her.

C.2 Interface View

We propose that there be a way to see a particular model in terms of its participation in a given model interface. That is, if you take a state diagram, and view it in terms of the Authorization model interface, all the authorized and unauthorized state transitions will be highlighted, as in Fig. C.2. Each interface element could automatically be assigned a highlight color, and a key could appear. Concrete elements that do
not participate in the model could be grayed out. A similar user interface is detailed by Dong et al. for the very similar purpose of casting elements to roles within design patterns (DYZ07).

### C.3 Interface Painter

To allow quick specification of interface construals, the tool could have an “interface painter,” which would allow the user to badge concrete elements with a selected element from the model interface in a single click. For example, the user could specify that she wished to add a `<<Transaction>>` construal; the UI would then prompt her to specify each of the required elements, highlighting candidate elements and/or graying out non-candidates. Fig. C.3 shows a mock-up in which the user is prompted for each of the three elements of `<<Transaction>>`. Good keyboard shortcuts would make the interface painter even more efficient; for example, the user might be able to use arrow keys to move within candidate elements, or to start typing the name of a candidate to
select that element, and to badge the element by pressing Enter. Similarly, the user should be able to move within the construal’s badge set by using keyboard shortcuts.

\[\text{Figure C.2: The model from Fig. 4.15, with colors assigned to each model interface element, and unbadged elements grayed out.}\]

\[\text{C.4 Automation}\]

As we have previously discussed, many developers chafe at being required to point and click. No matter how efficient we make the mouse interface, some developers will still want a keyboard alternative. Keyboard shortcuts can go a long way in this, but we suggest two additional provisions for such programmers:

- Provide a simple textual language that may be used to generate UML models for MATA (the language employed by USE could be a good starting point). Other tools, including custom tools by individual developers, could provide output in this language. Simplicity is key here; of course programmers can work directly with the UML2\(^1\) file format that MATA accepts as input, but because these

---

\(^1\) See http://www.eclipse.org/modeling/mdt/, especially the uml2 package provided by Eclipse.
Figure C.3: An interface painter for the `transaction` model interface. For each model interface element, candidate concrete elements are highlighted; the user can select among them.

- are general, there is considerable additional complexity, and the files are not as readable, nor as concise, as a custom file format could be.

- Provide some simple libraries to allow programmatic generation of model files. This way, the tool can efficiently serve needs the UI designer cannot anticipate. Alternative UIs could be developed atop these libraries. (As with the previous provision, the same work could be accomplished by working directly with Eclipse’s um12 component, but by providing facilities dedicated to MATA models, that work could be made much simpler.)
BIBLIOGRAPHY


Omar Aldawud, Tzilla Elrad, Jeff Gray, Mohamed Kandé Jörg Kienzle, and Dominik Stein, editors. 7th International Workshop on Aspect-Oriented Modeling, October 2005.


Siobhán Clarke and Robert J. Walker. Generic aspect-oriented design with Theme/UML. In Filman et al. (FECA05), pages 425–458.


Iris Groher, Stephan Bleicher, and Christa Schwanninger. Model-driven development for pluggable collaborations. In Aldawud et al. (AEG+05).


D. L. Parnas. On the criteria to be used in decomposing systems into modules. pages 139–150, 1979.


Dominik Stein, Stefan Hanenberg, and Rainer Unland. Designing aspect-oriented cross-cutting in UML.


