

MODEL-DRIVEN ASPECT ADAPTATION  
TO SUPPORT MODULAR SOFTWARE EVOLUTION

by

JING ZHANG

JEFF GRAY, COMMITTEE CHAIR  
BARRETT BRYANT  
ANIRUDDHA GOKHALE  
MARJAN MERNIK  
CHENGCUI ZHANG

A DISSERTATION

Submitted to the graduate faculty of The University of Alabama at Birmingham,  
in partial fulfillment of the requirements for the degree of  
Doctor of Philosophy

BIRMINGHAM, ALABAMA

2009

Copyright by  
Jing Zhang  
2009

# MODEL-DRIVEN ASPECT ADAPTATION TO SUPPORT MODULAR SOFTWARE EVOLUTION

JING ZHANG

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCES

## ABSTRACT

Software maintenance and evolution are the most costly and time consuming activities during the software development life cycle. One of the biggest challenges of software evolution is to adapt a software system to the ever-changing requirements from users or operating environments. An ideal goal is to encapsulate these requirements into a high-level abstraction, which can be used to drive large-scale adaptation of the underlying software implementation. Model-Driven Engineering (MDE) is one of the enabling techniques that support this objective, in that it allows the domain experts or application designers to synthesize various software artifacts from high-level models that represent domain concepts or system design logic. The state-of-the-art MDE techniques, however, lack support for advanced processes and constructive methods involved within the context of the evolution of software systems. With respect to large legacy systems written in disparate programming languages, the primary problems of evolution are the difficulty of adapting the legacy source to match the evolving requirements specified in the corresponding models and the incapability of developing and applying evolutionary tasks in a modular way.

In order to overcome such difficulties, this dissertation introduces a Model-Driven Aspect Adaptation (MDAA) framework that unites the MDE and Aspect-Oriented Software Development (AOSD) approaches to support modular software evolution driven by changing requirements. AOSD offers an advanced technique that supports invasive adaptation by weaving aspect modules that encapsulate the evolutionary crosscutting changes into the software system. By combining MDE and AOSD, the evolutionary change requirements are specified in the high-level aspect models, which

in turn drive the generation of the low-level aspect code used to perform legacy system evolution. This way, the whole software evolution process is performed in a modular manner, which enables the changeability, comprehensibility and independent development of the evolved implementation.

The dissertation provides two case studies to demonstrate the applicability and benefit of the approach, according to two distinct instantiations of the MDAA framework. One is based on the paradigm of Domain-Specific Modeling (DSM), which leverages a DSM modeling environment, a model transformation engine and a program transformation system to provide evolution support for legacy systems from the high-level modeling abstraction that depicts the application domain concepts. Another case study is based on a more general modeling mechanism – UML activity modeling. An Aspect-Oriented Activity Modeling (AOAM) approach is implemented to facilitate modular evolution for UML activity models. In addition, the aspect models drive the generation of the corresponding AspectJ code, which in turn performs the evolutionary adaptation of the underlying legacy source.



## DEDICATION

*This work is dedicated to my parents, my husband Danyu,  
and whoever encouraged and supported me to reach so far.*

## ACKNOWLEDGMENTS

First and foremost, I would like to express my sincerest gratitude to my research advisor, mentor and role model, Dr. Jeff Gray, for providing me with the great opportunity to work on this research. Jeff is always there to listen and to give advice. He teaches me how to question thoughts and present ideas. Moreover, he encourages me to engage in a number of professional activities such as conferences and program committees. I especially appreciate his tremendous support and continuous assistance during my remote study for the past two years. Jeff, without your persistent encouragement and enthusiastic supervision, I would never be able to reach where I am today. UAB is lucky to have you, as are the many students who are under your guidance. You are surely deserving of “Alabama Professor of the Year!”

I would also like to extend my gratitude to the other members of my committee. Dr. Barrett Bryant led me to the world of programming languages and compilers. The knowledge I learned from his course has not only been very useful to my PhD research, but also very practical to my work in industry. Dr. Marjan Mernik helped me enrich and deepen my understanding of the concepts of “Domain-Specific Languages” through his enlightening course. Dr. Aniruddha Gokhale and Dr. Chengcui Zhang, thank you for your time and efforts on serving on my committee and providing many valuable comments that improved the presentation and contents of this dissertation.

To the staff members in the Department of Computer and Information Sciences (CIS): Kathy Baier, Fran Fabrizio and Janet Tatum, I appreciate your constant support and kind help throughout my entire graduate study.

I am extremely grateful to all of my dear student colleagues in the CIS Department.

To Yuehua Lin, it was my great pleasure to work with you under the same programs (and in the same office) for four years. You're not only a great labmate, but also a good friend and a big sister who offers tremendous care and help in my life. To all the other Softcom lab members: Fei Cao, Faizan Javed, Alex Liu, Suman Roychoudhury, Robert Tairas, Hui Wu, Xiaoqing Wu and Wei Zhao, I really enjoyed the moments we shared together on classes, seminars and discussions. To other CIS friends: Xin Chen, Yonghui Chen, Zhijie Guan, Yin Liu, Ying Liu and Ying Sun, without your friendship and support, Alabama could never be called a "sweet home" to me.

My special thanks are due to my past and present colleagues at Motorola Labs. To Aswin van den Berg, my first manager and mentor, I sincerely appreciate your internship and job offer, which turned me from a student to an independent researcher in the real-world industry. Without the unique opportunity of working on the Motorola WEAVR project, this dissertation could never be completed. To Thomas Cottenier, I truly cherish the time we worked together on the WEAVR project. To John Strassner, my second manager, you opened the world of research on autonomic computing to me, and guided me to commercialize new technologies to build high-quality software-based products. To Michael Jiang, my last supervisor, thank you for being there and encouraging me all the time. To Rick Kane, my current manager, I'm grateful for your full support on my PhD research. To all of my colleagues in the software and middleware research lab, Scott Chen, Judy Fu, Yan Liu, Srinivasa Samudrala, and Kabe Vanderbaan, thank you all for being a great team to work with all these years.

My deep appreciation and heartfelt gratitude goes to my family. To my dear parents, Huancheng Zhang and Aifang Huang, thank you for bringing me into the world in the first place, for giving me the freedom to make my own decisions, and for your everlasting dedication in my whole life. To my beloved husband, Danyu Liu, having you in my life is the best gift that I can ever and will ever receive. None of this would have been possible without your unconditional love and support.

Last but not least, I appreciate the financial support from the DARPA Program Composition for Embedded Systems (PCES) and the National Science Foundation (NSF) that funded parts of the research presented in this dissertation.

## TABLE OF CONTENTS

	<i>Page</i>
ABSTRACT . . . . .	ii
DEDICATION . . . . .	iv
ACKNOWLEDGMENTS . . . . .	v
LIST OF FIGURES . . . . .	xv
LIST OF TABLES . . . . .	xvi
LIST OF ABBREVIATIONS . . . . .	xvii
CHAPTER	
1 INTRODUCTION . . . . .	1
1.1 Challenges in Software Evolution . . . . .	2
1.2 Criteria for Software Evolution . . . . .	8
1.3 Research Objectives . . . . .	10
1.4 Outline . . . . .	13
2 BACKGROUND . . . . .	15
2.1 Software Restructuring . . . . .	15
2.2 Reverse Engineering . . . . .	16

2.3	Model-Driven Engineering . . . . .	17
2.3.1	Unified Modeling Language . . . . .	20
2.3.2	Model-Driven Architecture . . . . .	22
2.3.3	Domain-Specific Modeling . . . . .	23
2.3.4	DSM Evolution . . . . .	29
2.4	Model Transformation . . . . .	32
2.4.1	Model Migration . . . . .	32
2.4.2	GReAT . . . . .	33
2.4.3	ATL . . . . .	34
2.5	Program Transformation . . . . .	35
2.5.1	Design Maintenance System . . . . .	36
2.6	Aspect-Oriented Software Development . . . . .	38
2.6.1	Aspect-Oriented Programming . . . . .	40
2.6.2	Aspect-Oriented Modeling . . . . .	46
3	ASPECT-ORIENTED ACTIVITY MODELING . . . . .	52
3.1	Activity Modeling . . . . .	53
3.2	Aspect-Activity Model Specification . . . . .	55
3.2.1	Aspect-Activity Metamodel . . . . .	58
3.3	Compositions in Aspect-Oriented Activity Modeling . . . . .	61
3.3.1	Pointcut Composition . . . . .	64
3.3.2	Advice Composition . . . . .	65
3.3.3	Aspect Composition . . . . .	68
3.4	Case Study . . . . .	73
3.4.1	Background . . . . .	73
3.4.2	Modeling Timeout Handler Activity Aspect . . . . .	76
3.4.3	Deploying the Timeout Handler Aspect on a Base Model . . . . .	77
3.5	Summary . . . . .	79
4	LEGACY EVOLUTION THROUGH MODEL-DRIVEN ASPECT ADAP- TATION . . . . .	80
4.1	The Model-Driven Aspect Adaptation Framework . . . . .	81

4.1.1	Model Extractor . . . . .	85
4.1.2	Aspect Metamodel . . . . .	86
4.1.3	Model Composer . . . . .	87
4.1.4	Aspect Code Generator . . . . .	88
4.1.5	Program Composer . . . . .	89
4.2	DSM-Driven Aspect Adaptation . . . . .	90
4.2.1	Background: Bold Stroke . . . . .	93
4.2.2	Embedded Systems Modeling Language . . . . .	96
4.2.3	Applying MDAA to Bold Stroke . . . . .	101
4.2.4	Experimental Results . . . . .	118
4.2.5	Discussion . . . . .	120
4.3	Activity-Based System Evolution through MDAA . . . . .	122
4.3.1	Modeling Failure Handler Aspects . . . . .	124
4.3.2	Aspect Code Generation . . . . .	127
4.3.3	Discussion . . . . .	129
4.4	Aspect Mining from a Modeling Perspective . . . . .	129
4.4.1	Clone Detection for Aspect Modeling Mining . . . . .	131
4.4.2	Case Study: Aspect Mining in ESML . . . . .	135
4.5	Summary . . . . .	137
5	RELATED WORK . . . . .	140
5.1	Research on Aspect-Oriented Activity Modeling . . . . .	140
5.2	State-of-the-Art on Model-Driven Legacy Evolution . . . . .	142
5.2.1	Architecture-Driven Modernization . . . . .	142
5.2.2	Model-Driven Legacy Migration . . . . .	144
5.2.3	Model-Driven Modernization of Complex Systems . . . . .	144
5.2.4	Model-Driven Software Evolution . . . . .	146
5.2.5	Model-Driven Engineering for Software Migration . . . . .	147
5.3	Approaches on Aspect Mining . . . . .	149
6	FUTURE WORK . . . . .	151
6.1	Aspect-Oriented Modeling . . . . .	151

6.2	Model-Driven Aspect-Adaptation for Software Evolution . . . . .	154
7	CONCLUSION . . . . .	157
7.1	Aspect-Oriented Activity Modeling . . . . .	157
7.2	Model-Driven Aspect-Adaptation . . . . .	158
7.3	Lessons Learned . . . . .	159
	LIST OF REFERENCES . . . . .	161



## LIST OF FIGURES

<i>Figure</i>	<i>Page</i>
1.1 Model Evolution . . . . .	3
1.2 Legacy Evolution . . . . .	4
1.3 Models and Source Code Co-Evolution . . . . .	5
1.4 Research Overview . . . . .	11
2.1 Two Types of Software Restructuring . . . . .	16
2.2 Four-Layer Modeling Architecture . . . . .	19
2.3 A UML Example of the Four-Layer Modeling (Adapted from [95]) . .	21
2.4 MDA Overview . . . . .	23
2.5 Domain-Specific Modeling Development (Adapted From [140]) . . . .	25
2.6 A Petri Net Domain in GME . . . . .	28
2.7 Evolution of Models and Source Code in Terms of Metamodel Changes	31
2.8 Form of the Model Migration solution (Adapted from [166]) . . . . .	33
2.9 Model Transformation through ATL (Reprinted from [104], with per- mission from Frédéric Jouault) . . . . .	35
2.10 DMS Overview (Reprinted from [14], with permission from Ira Baxter)	37
2.11 Crosscutting Concerns . . . . .	39
2.12 Aspect Weaving . . . . .	41
2.13 Crosscutting Concerns in Models (From [79]) . . . . .	47
2.14 C-SAW Overview (From [79]) . . . . .	49

2.15	The ECL Aspect Code for Inserting Precondition Element to the Component Model in Figure 2.13 (From [79, 195]) . . . . .	50
3.1	Simplified Activity Metamodel (Adapted from [95]) . . . . .	54
3.2	An Order Processing Activity Model (Adapted from [95]) . . . . .	57
3.3	Aspect-Activity Modeling Profile . . . . .	58
3.4	Tracing Aspect, Pointcuts and Advice . . . . .	60
3.5	Aspect Weaving on Activity Models . . . . .	61
3.6	Augmented Order Processing Activity Model with the Tracing Aspect	62
3.7	Pointcut Composition in AOAM . . . . .	65
3.8	Advice Composition in AOAM . . . . .	66
3.9	Aspect Composition in AOAM . . . . .	69
3.10	Results of Join Point Selection for Aspect Deployment Strategy in Figure 3.9 . . . . .	72
3.11	Extracted INFM Alarm Correlation Activity Base Model . . . . .	75
3.12	A Timeout Handler Aspect . . . . .	76
3.13	Integrating the Timeout Handler Aspect with the INFM Alarm Cor- relation Base Model . . . . .	78
4.1	Model-Driven Aspect Adaptation Framework . . . . .	82
4.2	Roles in the MDAA Process . . . . .	84
4.3	Metamodel-Driven Model Extraction (Adapted from [33]) . . . . .	86
4.4	Integration of Models by a Model Composer . . . . .	88
4.5	One-to-one Mapping Between Aspect Model and Aspect Source Code	89
4.6	DSM-Driven Source Adaptation . . . . .	92
4.7	The Update Method in Bold Stroke <code>BM_ClosedEDComponentImpl.cpp</code>	95
4.8	ESML Component Metamodel (From [138]) . . . . .	97
4.9	Bold Stroke Component Interaction in ESML . . . . .	98

4.10	Internal Representation of <code>BM_ClosedEDComponent</code> in ESML . . . . .	100
4.11	ESML Concurrency Aspect Metamodel . . . . .	105
4.12	ESML Concurrency Metamodel Aspect View . . . . .	105
4.13	ESML Component Metamodel (Augmented with the Concurrency Metamodel) . . . . .	106
4.14	Metamodel Composition Algorithm . . . . .	107
4.15	Internal Representation of the <code>BM_ClosedEDComponent</code> in ESML (Augmented with the Concurrency Aspect) . . . . .	109
4.16	ECL Specification for Adding External Locking to ESML Component Models (Adapted from [122]) . . . . .	110
4.17	A Set of Generated Locking Transformation Patterns and Rules in the DMS Rule Specification Language . . . . .	112
4.18	ESML Logging Metamodel . . . . .	115
4.19	ECL Specification for Adding <code>LogOnMethodExit</code> in ESML Models (From [83]) . . . . .	116
4.20	Internal Representation of the <code>BM_ClosedEDComponent</code> in ESML (Augmented with the Logging Aspect) . . . . .	117
4.21	A Set of Generated Logging Transformation Patterns and Rules in the DMS Rule Specification Language . . . . .	118
4.22	The <code>Update</code> Method in Bold Stroke <code>BM_ClosedEDComponentImpl.cpp</code> (Enhanced with the Logging Aspect) . . . . .	119
4.23	MDAA Interpreters for Different Aspect Models . . . . .	121
4.24	Activity-Based System Evolution through MDAA . . . . .	123
4.25	Pattern Failure Management Aspect Model . . . . .	126
4.26	Pattern Failure Analyzer Model . . . . .	127
4.27	The Generated AspectJ Program for the Pattern Failure Handler Aspect	128
4.28	A Metamodel Example . . . . .	133

5.1	The XIRUP Process Model (Adapted from [20]) . . . . .	145
5.2	Model-driven Migration Process in Sodifrance (Adapted from [67]) . .	148

## LIST OF TABLES

<i>Table</i>	<i>Page</i>
2.1 Comparison of DSM to Programming Language, and Database Definition	26
3.1 Graphical Symbols in Activity Diagram . . . . .	56
3.2 Symbols for Denoting Aspect Relationships . . . . .	70
4.1 Some Mappings between Bold Stroke Source and ESML Metamodel .	99
4.2 Comparison of Manual Modification to the MDAA Approach . . . . .	119
4.3 INFM Software Failures and Resolution Table for Alarm Correlation .	125
4.4 Three Levels of Similarity . . . . .	132
4.5 Comparison of DSM-based and UML-based MDAA . . . . .	138

## LIST OF ABBREVIATIONS

<b>AAM</b>	Aspect-Oriented Architecture Model
<b>ADM</b>	Architecture-Driven Modernization
<b>AMMA</b>	ATLAS Model Management Architecture
<b>AMT</b>	Aspect Mining Tool
<b>AOAM</b>	Aspect-Oriented Activity Modeling
<b>AODM</b>	Aspect-Oriented Design Model
<b>AOM</b>	Aspect-Oriented Modeling
<b>AOP</b>	Aspect-Oriented Programming
<b>AOSD</b>	Aspect-Oriented Software Development
<b>API</b>	Application Programming Interface
<b>AQML</b>	Adaptive Quality Modeling Language
<b>AST</b>	Abstract Syntax Tree
<b>ASTM</b>	Abstract Syntax Tree Metamodel
<b>ATL</b>	Atlas Transformation Language
<b>BNF</b>	Backus-Naur Form
<b>BPEL4WS</b>	Business Process Execution Language for Web Services
<b>C-SAW</b>	Constraint-Specification Aspect Weaver

## LIST OF ABBREVIATIONS (continued)

<b>CBS</b>	Computer-Based System
<b>CBSE</b>	Component-Based Software Engineering
<b>CCM</b>	CORBA Component Model
<b>CDMA</b>	Code Division Multiple Access
<b>CMM</b>	Capability Maturity Model
<b>COM</b>	Component Object Model
<b>CORBA</b>	Common Object Request Broker Architecture
<b>DARPA</b>	Defense Advanced Research Projects Agency
<b>DDD</b>	Domain-Driven Development
<b>DDL</b>	Data Definition Language
<b>DMS</b>	Design Maintenance System
<b>DRE</b>	Distributed Real-time and Embedded
<b>DSL</b>	Domain-Specific Language
<b>DSM</b>	Domain-Specific Modeling
<b>DSML</b>	Domain-Specific Modeling Language
<b>DSTL</b>	Domain-Specific Transformation Language
<b>ECL</b>	Embedded Constraint Language
<b>ESML</b>	Embedded Systems Modeling Language
<b>FCO</b>	First-Class Object
<b>FSM</b>	Finite State Machine

## LIST OF ABBREVIATIONS (continued)

<b>GME</b>	Generic Modeling Environment
<b>GPL</b>	General-Purpose Language
<b>GReAT</b>	Graph-REwriting And Transformation Language
<b>GUI</b>	Graphical User Interface
<b>IDE</b>	Integrated Development Environment
<b>INFM</b>	Intelligent Network Fault Management
<b>KDM</b>	Knowledge Discovery Metamodel
<b>M2M</b>	Model-to-Model Transformation
<b>MDA</b>	Model-Driven Architecture
<b>MDAA</b>	Model-Driven Aspect Adaptation
<b>MDE</b>	Model-Driven Engineering
<b>MIA</b>	Model-In-Action
<b>MIC</b>	Model-Integrated Computing
<b>MM</b>	Model Migration
<b>MoBIES</b>	Model-Based Integration of Embedded Systems
<b>MoDSE</b>	Model-Driven Software Evolution
<b>MOF</b>	Meta-Object Facility
<b>MOMOCS</b>	MOdel driven MOdernization of Complex Systems
<b>OCL</b>	Object Constraint Language
<b>OMG</b>	Object Management Group



## LIST OF ABBREVIATIONS (continued)

<b>PCES</b>	Program Composition for Embedded Systems
<b>PDG</b>	Program Dependence Graph
<b>PDM</b>	Platform Definition Model
<b>PIM</b>	Platform-Independent Model
<b>PN</b>	Petri Net
<b>PSM</b>	Platform-Specific Model
<b>QoS</b>	Quality of Service
<b>QVT</b>	Query/View/Transformation
<b>RSL</b>	Rule Specification Language
<b>SOA</b>	Service-Oriented Architecture
<b>UML</b>	Unified Modeling Language
<b>XIRUP</b>	eXtreme end-User dRiven Process
<b>XML</b>	Extensible Markup Language

## CHAPTER 1

### INTRODUCTION

Software permeates all aspects of our lives. It spans from small personal embedded devices (e.g., mobile phones and digital watches) to large enterprise solutions (e.g., avionics and automobiles systems), from human-centered applications (e.g., computer-aided design systems and e-commerce service platforms) to autonomous systems (e.g., robot software and self-managed networks). The proliferation of software in daily life has increased the level of responsibility placed on software applications [60, 176]. Nevertheless, software often inevitably falls short of expectations as demands for new requirements increase. New user requirements or improved execution environments demand additional functionality that makes existing software applications obsolete. This is called *software aging* [144] – a phenomenon that occurs when software applications fail to meet the changing needs and environments. In order to overcome the negative effects of software aging, future requirements will necessitate new strategies to support the requisite adaptations across various software artifacts (e.g., documentation, design models, source code and test cases) [25, 132].

According to Bennett and Rajlich [29], software evolution is a type of software maintenance task that takes place at the later stage of the software development life cycle after the initial development was accomplished successfully. The goal of evolution consists of adapting the software application to the ever-changing, often unanticipated requirements from users or operating environments. It should be noted that software evolution is different than revolution. It is assumed that software and

its evolved new version are more similar than different. Software revolution, on the contrary, often refers to the activity of wiping out the old software and rewriting or even redesigning a new solution.

Software evolution is characterized by its high cost and slow speed of implementation. Numerous investigations have shown that the relative cost for maintaining software and managing its evolution represents more than 90% of its total cost [59, 135]. One study showed that about 65% of software evolution activities were found to be perfective maintenance tasks (i.e., new user/functionality requirements) [121] and another study found that about 75% of the evolution cost was spent for providing adaptive enhancements (i.e., changing environment requirements) [182]. About two decades ago, there were an estimated 120 billion lines of source code being maintained, primarily COBOL and FORTRAN programs [180]. Gartner estimates that there are over 230 billion lines of COBOL and RPG code in existence with 5 billion added annually. This dissertation is intended to address the issue of software evolution by providing an initial experimentation on legacy system evolution with advanced software engineering techniques.

## 1.1 Challenges in Software Evolution

As observed by Lehman [119], continuous change is the first law of software evolution, i.e., “software that is being used must be continually adapted or it becomes progressively less satisfactory.” Because of this intrinsic nature of software, the need for software to evolve constantly poses stiff challenges for software engineers. The investigation of software evolution has received considerable attention in the research literature. The workshop on “Challenges in Software Evolution (ChaSE)” [56] was particularly dedicated to identifying substantial obstacles to software evolution research and practice, and to propose and discuss challenges in software evolution. As summarized in [130] and [132], eighteen different categories of challenges were

identified during the workshop, among which a representative list of the most important challenges contains:

### 1. Supporting Model Evolution

Most state-of-the-art evolution techniques and supporting tools are primarily focused on source code. For instance, program transformation [27, 40] and program refactoring [71, 142] are the two well-known techniques that are both applied to the program representation of the software for facilitating evolution. Much less evolution support can be found at the design and modeling stages. With more adoption of Model-Driven Engineering (MDE) techniques (e.g., Domain-Specific Modeling (DSM) [80] and UML modeling [38]), models are becoming first-class citizens during the development process. As shown in Figure 1.1, during the evolution process, there exists some difference,  $\Delta_M$  (i.e., maintenance delta as defined in [26]), between the old models and the new ones, which capture additional system requirements. Models can be evolved under the same modeling paradigm (i.e., the meta-level specification that defines the valid syntax and semantics of models) or driven by the modeling paradigm evolution. As the size of system models grow, techniques and tools that automate complex change evolution are urgently needed.

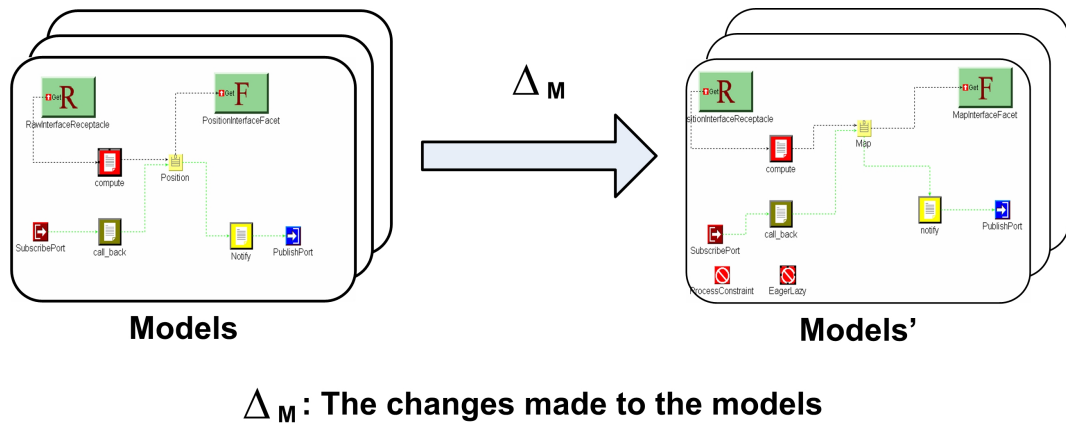


Figure 1.1: Model Evolution

## 2. Supporting Legacy Evolution

In contrast to modern software systems that are usually derived from a higher level of abstraction via advanced techniques and have built-in support for easy extension and modification (e.g., MDE offers system extensibility through high-level models), legacy systems are often manually written with limited or no support for evolution. Often, business requirements are expressed by source code that is hardcoded deep within an application. It is a great challenge to maintain and evolve such systems in terms of their low-level implementation, because even a small modification in the requirements may trigger drastic manual changes in large portions of the source code [83]. Legacy evolution is usually accomplished through source code evolution, as shown in Figure 1.2.

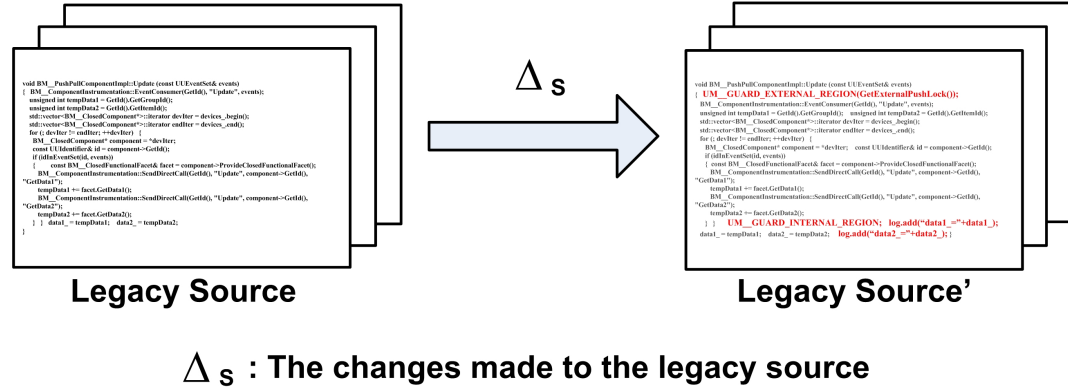


Figure 1.2: Legacy Evolution

## 3. Supporting Co-evolution

Co-evolution, also called *coupled transformation*, is defined by Lämmel [115] as: “two or more artifacts of potentially different types are involved, while transformation at one end necessitates reconciling transformations at other ends such that global consistency is reestablished.” In a software system that contains multiple software artifacts (e.g., documentation, design models, source code

and test cases), co-evolution techniques are indispensable for maintaining the fidelity of the whole system. A typical co-evolution scenario, as shown in Figure 1.3, is concerned about the consistent evolution between the high-level abstract models and the low-level source code. An advanced mechanism is needed to map the model changes ( $\Delta_M$ ) to the source code changes ( $\Delta_S$ ) and enforce the conformity between the models and underlying code.

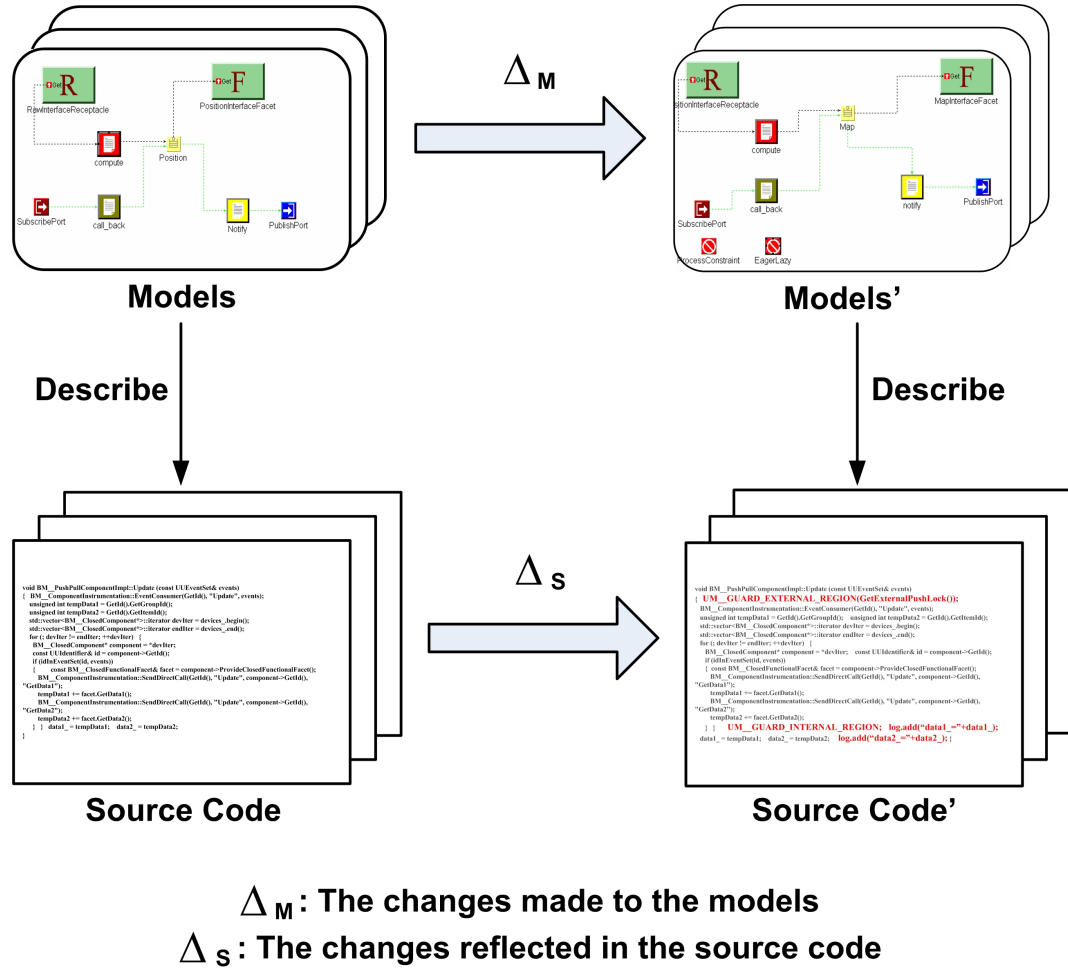


Figure 1.3: Models and Source Code Co-Evolution

With the assistance of state-of-the-art MDE techniques, source code evolution is usually performed through modification of the existing models, from which a new application will be regenerated completely. Consequently, the application

source is actually replaced instead of updated. A challenge arises when the MDE approach is to be applied to literally several hundred billion lines of legacy source [181] in production use today <sup>1</sup>. Reproducing a large code base each time is extraordinarily expensive and inefficient, especially when the evolutionary changes are applied incrementally with most of the source remaining unaffected. Thus, complete regeneration is not always a plausible solution for source code evolution when applied to existing legacy applications, nor is it feasible or even possible for very large systems.

To apply model-based techniques to large legacy systems, it is beneficial to have an approach that is *transformational* (i.e., one that actually modifies the source code representation) rather than *translational* (i.e., one that generates the source code) <sup>2</sup>. However, support for parsing and invasively transforming legacy source code from high-level models is not well-represented in the research literature. This is because of the following challenges:

- (a) It is often the case that a slight change in the system requirements would necessitate extensive modifications that are widely spread over a vast section of the source code. With the DSM approach, this situation can be mitigated by providing a Domain-Specific Modeling Environment (DSME) that is customized for the domain experts so that they are able to manipulate the system at a higher level of abstraction, instead of the low-level source code. The desired result is to achieve modularization such that a change in a design decision is isolated to one location within the model [79]. However, this approach alone cannot solve the problem because small changes in the models might still necessitate drastic changes throughout

---

<sup>1</sup>The key difference here is between modeling and generation of brand new (also known as “green-field”) applications, versus the application of modeling to support evolution of legacy systems with existing code.

<sup>2</sup>The difference between transformational and translational approaches will be further illustrated in Chapter 2.

the source code. The question remains as to how the underlying existing legacy source is to be modified from the models. It is one of the key challenges to maintain the fidelity between the mapping of model properties and the corresponding source code.

- (b) Many legacy systems are usually large (e.g., hundreds of thousands, or even millions of lines of source code) and represented by a variety of programming languages. In order to transform such diverse systems, different parsers are needed for each language. In addition, if support for a new language is required, an individual new parser for this particular language must be necessarily included in the transformation toolsuite. Developing industrial-scale parsers to support all languages, and integrating them within the modeling tool, is prohibitively time-consuming (if not unfeasible) [116].
- (c) Even if a mature parser is available and applicable for handling all of the languages in the underlying source, a full-featured program transformation engine is also required in order to perform the invasive [19] adaptations to the large legacy source base. Using a program transformation system is also an arduous task and requires skills that many programmers do not possess.

This dissertation is devoted to addressing these three challenge problems in the software evolution area by using advanced software engineering techniques. Before the elaboration of the approach, the criteria first must be identified for evaluating the quality of the software evolution techniques.



## 1.2 Criteria for Software Evolution

A long-standing goal of software engineering is to construct software that is easily modified and extended. A desired result is to achieve modularization such that a change in a design decision is isolated to one location. During the software development process, *modularization* represents a design rationale that concerns decomposing software systems into a set of manageable building blocks, or modules. Representative examples of modular software development techniques are object-oriented design and programming [155] (i.e., modularization by classes or objects), Component-Based Software Engineering (CBSE) [100] (i.e., modularization by components), Service-Oriented Architecture (SOA) [58] (i.e., modularization by services), and Aspect-Oriented Software Development (AOSD) [2] (i.e., modularization by crosscutting concerns or aspects).

The goal of modular software development is to improve the flexibility and comprehensibility of a software system while reducing its development time. In order to achieve this goal, the system must be modularized using certain criteria, such as changeability, comprehensibility and independent development, as promoted by David Parnas in the early 1970s in his influential paper “On the Criteria to Be Used in Decomposing Systems into Modules” [143]. These three criteria were originally created for the software development process. However, it is interesting to note that they are also suitable to be used for evaluating the quality of software evolution techniques.

- **Changeability**

In a highly modularized system, changeability typically means that individual modules should be able to change without radical impact to the rest of the system. A key enabler is *information hiding*, i.e., hiding the design decisions that are likely to change and thus protecting other modules from modification if the design decisions are changed. However, no matter how well a system is

decomposed into loosely-coupled modules, there always exist some evolutionary requirement concerns that cut across the chosen modularization mechanism. This is generally referred to as the problem of “the tyranny of the dominant decomposition” [177], from which most of the modularization mechanisms suffer, because they only allow a single means of decomposition. In the software evolution process, it is indispensable to provide support for encapsulating these crosscutting concerns into isolated modules such that changes to these concerns can be conducted in a localized manner.

- **Comprehensibility**

Comprehensibility is another major objective of modular development. It offers the possibility of studying the system one module at a time. The whole system can therefore be better designed, developed and analyzed, because the system is better understood at a higher level of abstraction. The concept of *abstraction* is complementary to that of information hiding, in that it presents a simplified view of something – depicting only the relevant aspects and ignoring irrelevant details. Comprehensibility is also a crucial factor in the software evolution process in that high-level evolutionary requirements need to be well-understood and thus systematically translated into the low-level change adaptation. As stated by Favre [62], “large scale software evolution should be driven by much higher levels of abstraction.” This necessitates intrinsic encapsulation and abstraction of the evolutionary changes such that different changes can be studied and specified individually.

- **Independent Development**

In a large software system, the total development time can be shortened and the cost can be reduced if separate groups are able to work on each module concurrently with little communication. Independent development is extremely

useful during the software evolution process. As the complexity of a system increases, different types of evolutionary changes may require different evolution techniques. Also, different parts of the system may exhibit different rates of change [131]. By dividing evolution tasks into individual modules that can be developed and maintained by different groups, the whole evolution process can be sped up considerably and the complexity of the evolution can be reduced drastically.

The above three criteria, when applied to the software evolution process, all point to one principle – *modular evolution*. In this dissertation research, evolution is treated as just another type of software development task. Similar to modular development, *modular evolution* enables the system maintainer to develop and apply evolutionary changes in a modular way. Each module encapsulates one kind of change requirement and may be designed and programmed independently. Advanced composition techniques are thus needed to integrate these evolution modules into the base system to produce the new version of the evolving system. Aspect-oriented software development (AOSD) [2] is such a key enabling technology that supports modular evolution to the greater extent. In AOSD, the unit of modularity is called an *aspect*, which enables the modularization of concerns such as evolutionary change requirements that usually cut across multiple types and objects. The composition mechanism provided by AOSD is referred to as *aspect weaving*, whereby aspects are woven into the base modules by a specific AOSD compiler called an *aspect weaver*. The general concepts of AOSD will be further introduced in Chapter 2.

### 1.3 Research Objectives

The research described in this dissertation investigates advanced techniques that provide modular evolution solutions for addressing the three challenge problems described in Section 1.1. An overview of the research is shown in Figure 1.4. First,

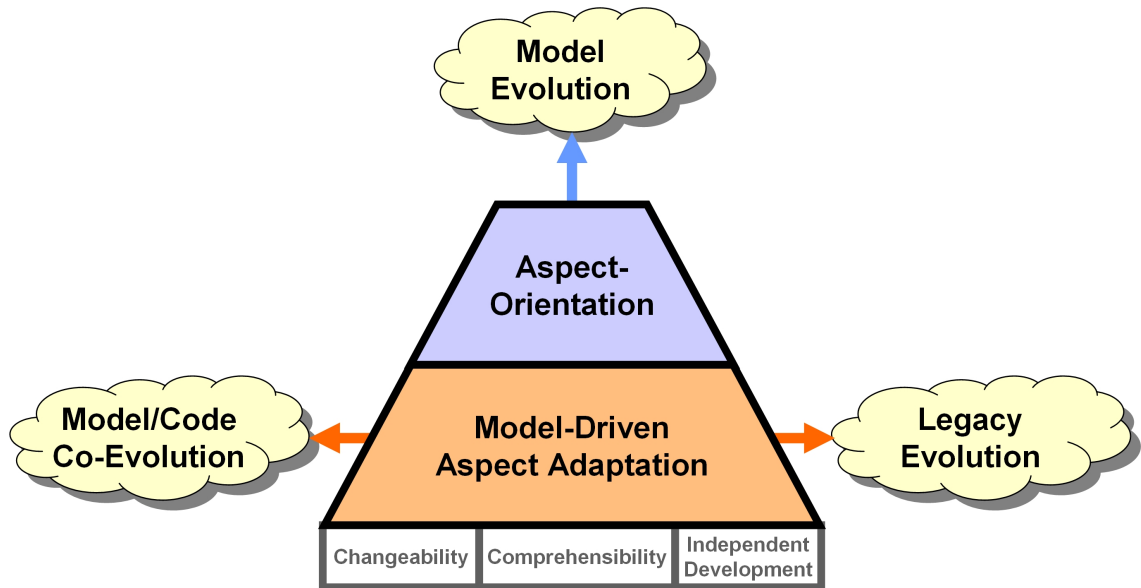


Figure 1.4: Research Overview

an aspect-oriented approach is applied to support the modular evolution of models at a higher level of abstraction. Then, a model-driven aspect adaptation approach is initiated by uniting the model-driven approach with aspect-oriented techniques to provide the systematic co-evolution between the models and the corresponding source code. Co-evolution is also the key enabler to support the evolution of the legacy code from the high-level change requirements that are represented by models. The overall research is conducted with an aim to satisfy the three evolution criteria as described in Section 1.2, i.e., changeability, comprehensibility and independent development.

Specifically, the contributions described in this dissertation can be summarized by the following two objectives:

- **Aspect-Oriented Activity Modeling**

The first part of the research is focused on supporting evolution for a particular kind of model, i.e., UML activity model. Activity modeling is a well-adopted design and specification paradigm that is usually employed to describe the control flow and data flow of a system using a set of graphical notations. When a

new evolution task requires modification that spreads across multiple different actions in an activity model, it may be difficult to comprehend and change. Aspect-Oriented Activity Modeling (AOAM) offers a solution to handle such kind of evolutionary change (i.e., crosscutting concerns) by introducing some of the concepts and methods of Aspect-Oriented Software Development (AOSD) into the activity modeling space. During the evolution process, each evolutionary requirement can be encapsulated within a specific abstraction module called an *aspect*, which intends to break “the tyranny of the dominant decomposition” [177] of the activity modeling constructs. The key benefit of AOAM is to enable the model engineer to study, specify, analyze and change one requirement individually with great flexibility (i.e., apart from the original model and the other requirement modules). The aspects are automatically integrated with the original model using an underlying model composition engine called an *aspect model weaver*. The research contributions lie in the introduction of a specific modeling language that allows the model engineer to specify activity aspects, as well as the implementation of an aspect weaver that supports different types of composition mechanisms.

- **Model-Driven Aspect Adaptation**

The second part of the research advocates a synergy of Model-Driven Engineering (MDE) and AOSD for facilitating model/code co-evolution as well as legacy evolution. A generic framework called Model-Driven Aspect Adaptation (MDAA) has been developed to support this objective. Using the MDAA framework, a modeling paradigm is pre-defined to express high-level domain requirements or application design concepts.

During the system evolution phase, the business users or domain experts specify the new requirement changes in a high-level aspect modeling language (e.g., AOAM). Those models are then interpreted to generate low-level aspect code

that will invasively modify a large cross-section of the corresponding code base by an underlying code weaver. The model engineers do not need to understand the accidental complexities of the aspect code or the source code written in programming languages. That process is transparent and is driven by an aspect code generator that maintains the consistent mappings between the models and the source code.

When MDAA is leveraged to assist in legacy evolution, the legacy source first needs to be extracted into the modeling space by a model extractor, based on a modeling paradigm that is appropriate to describe the domain knowledge or design architecture of the legacy system. The problem of legacy evolution can be addressed by the solution of model/code co-evolution provided by the MDAA framework. Such an approach offers a capability for performing wide-scale source adaptation of legacy systems from requirements properties described in high-level models. The key contribution of this research lies in the construction of the framework that is composed of a collection of methods, components, modeling artifacts and tools. Two instantiations of the framework are implemented for supporting different styles of modeling paradigms for evolving legacy systems. In addition, an aspect mining technique is presented to support aspect identification on high-level models, which intends to improve software modularization and allows a legacy system to benefit from AOSD.

## 1.4 Outline

The rest of the dissertation is structured as follows. A background introduction of the related literature is provided in Chapter 2. The chapter starts by giving an overview of the existing software restructuring techniques, which are foundations that support software evolution. Then, the concept of MDE is introduced, including two different implementation approaches. One is based on the generic UML standard,

and another is based on DSM. Lastly, the technical approaches supporting AOSD are given, both at the modeling level and at the programming level.

Chapter 3 introduces the details of implementing the AOAM approach that supports modular evolution for UML activity models. A brief overview of the activity metamodel definition is given, upon which the aspect-activity specification is defined via the UML profiling mechanism. The approach is illustrated through a case study based on an industry inspired problem.

Chapter 4 presents the implementation of the MDAA framework. The framework is instantiated to accommodate two different modeling approaches, i.e., UML and DSM. The experimental results and comparison between the two approaches are provided to evaluate the feasibility of the approach. The aspect mining approach on models is also presented in this chapter.

Related work is discussed in Chapter 5, including research on applying aspect-oriented techniques to models and approaches that support legacy evolution.

Lastly, Chapters 6 and 7 offer future extensions of this research, as well as summary remarks.

## CHAPTER 2

### BACKGROUND

This chapter introduces some relevant background material as preliminaries for the following chapters. An overview of software restructuring approaches is first presented, which prepares the introduction of the key enabling techniques that are involved in the Model-Driven Aspect Adaptation (MDAA) research, as will be demonstrated in Chapters 3 and 4. These enabling techniques are reverse engineering, Model-Driven Engineering (MDE), model transformation, program transformation and Aspect-Oriented Software Development (AOSD).

#### 2.1 Software Restructuring

Research into software restructuring techniques, and the resulting tools supporting the underlying science, has enhanced the ability to modify the structure and function of a software representation in order to address changing stakeholder requirements [86]. As shown in Figure 2.1, software restructuring techniques can be categorized as either horizontal or vertical. The research into horizontal transformation concerns modification of a software artifact at the same abstraction level. This is the typical connotation when one thinks of the term *transformation* [185], with examples being model transformation [32] and particularly aspect weaving at a higher modeling level [1], as well as program transformation [183] and Aspect-Oriented Programming (AOP) at the implementation level [109]. Horizontal transformation systems often



lead to invasive composition of the software artifact [19]. In contrast, vertical transformation is typically more appropriately called *translation* (or synthesis) [185] because a new artifact is being synthesized from a description at a different abstraction level (e.g., reverse engineering [47], generative programming [52], and specifically model-driven software synthesis [139]).

Software restructuring techniques are essential enablers to support software evolution. The following sections will introduce a number of techniques that fall into these two distinct types of software restructuring approaches.

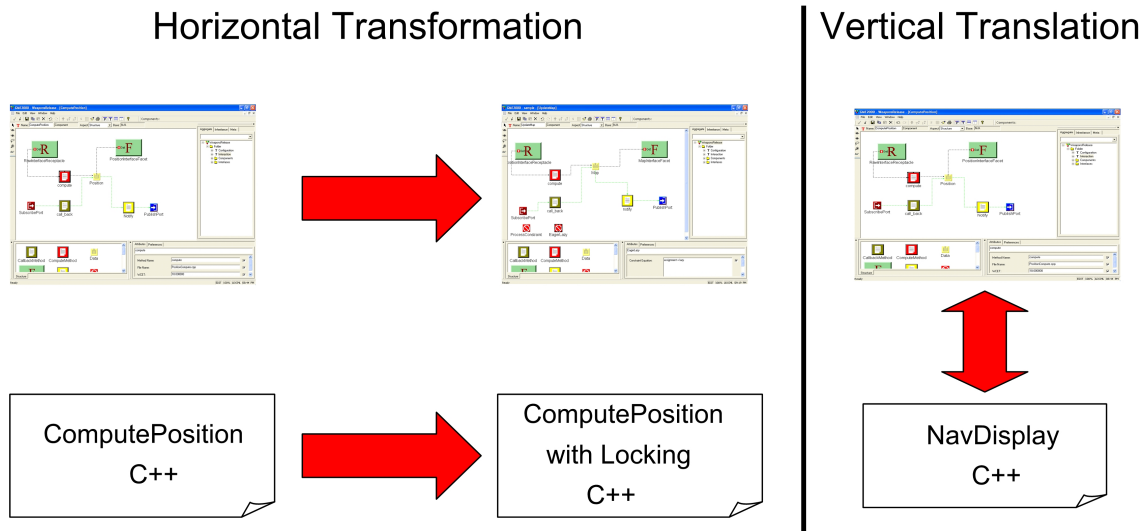


Figure 2.1: Two Types of Software Restructuring

## 2.2 Reverse Engineering

Whether it is for fixing a glitch or rebuilding a machinery engine, it is possible to retrieve much of the information (mostly undocumented) by deconstructing and observing individual pieces of the whole. This is the basic concept behind reverse engineering – taking apart an object to see how it works in order to understand it, improve it or build a copy out of it. Reverse engineering is used for many reasons:

for document discovery, for product analysis, for learning purposes, for improving the product or making it interoperable with other systems/platforms.

The practice, taken from older industries, is now frequently adopted in the area of computer hardware and software. Software reverse engineering involves “the process of analyzing a subject system to create representations of the system at a higher level of abstraction,” as defined in [47]. More specifically, it is referred to as “going backwards through the software development cycle” [97]. In this sense, the output of the implementation phase (i.e., mostly referred to as the source code form of the system) is reverse engineered back to the requirements analysis and design phase, in an inversion of the traditional waterfall model [152] that represents *forward engineering*.

Reverse engineering is a broad term and can take many forms. In one form it involves disassembly (i.e., translation of machine code into assembly code) or decompilation (i.e., translation of the output from a compiler into source code written in a relatively high-level programming language such as C++ or Java). In another form it deals with design recovery, either from the source code or any other available software artifacts. In this research, reverse engineering is applied in the latter case, in particular, to the extraction of high-level models from the examination of source programs, as well as to the identification of crosscutting concerns (i.e., aspects) on the models. As will be described in Chapter 4, reverse engineering is the first step toward legacy source adaptation from evolving requirements that are specified by high-level domain or design models.

## 2.3 Model-Driven Engineering

The word *modeling* comes from a Latin word *modellus* [157]. It describes a human activity to deal with the real-world via the media of models that are essentially abstract representations of real-world objects. The approach of using models during the software development process has a long-standing history (e.g., the Moore and

Mealy state machine models [128, 136] have been widely adopted since the early 1950s).

With the initiative of the *de facto* standard Unified Modeling Language (UML) [38], model-based development has become more popular. Yet, many developers consider a model as pure documentation and the relationship between models and the actual implementation is only intentional but not formal [168]. That is, designers put the domain requirements or design logic in models, according to which, developers then implement the system in some programming language. Therefore, the models and the code are under development and maintenance independently. This poses two significant disadvantages. First, it purely depends on the developers' interpretation to lead to the executable implementation from the models. Additionally, every software system is subject to change during all the phases of its life cycle. The documentation represented in the models and the implementation codified in the source program are inevitably out of sync and thus need to be adapted to each other scrupulously. This is an arduous task if it cannot be automated.

Model-Driven Engineering (MDE) [31] offers a distinct approach to supporting the development of Computer-Based Systems (CBSs). The key differentiator is that models are not treated as pure documentation, but are first-class citizens during the development process. MDE allows for modeling software at a much higher abstraction level, using business concepts rather than developing on a low technical implementation level (e.g., with programming language terms such as *class* or *function*). That is, modeling allows one to specify “what to do” instead of “how to do it.” The whole implementation is thus completely driven or automated from the models by the supporting model manipulation tools. With a model-driven approach, software development not only becomes more efficient but also enables stakeholders without programming experience to participate and contribute to the development cycle [173].

MDE is based on a classical four-layer modeling architecture [89], which is com-

posed by four different layers or levels of abstraction. As shown in Figure 2.2, the topmost layer (i.e., the meta-metamodel, or M3 layer) defines an abstract language and framework for specifying, constructing and managing metamodels. It forms the foundation for the whole architecture and conforms to itself. The metamodel, or M2 layer, is an instance of the meta-metamodel and defines a language to describe a model. The models of the underlying system, represented by concepts defined in the corresponding metamodel at the M2 layer, are at the model or M1 layer. Finally, at the M0 layer are objects from the real-world system. The architecture also provides a framework that enables exchanging metamodels and models among different metamodeling environments, which is critical for tool interoperability. The four layer modeling architecture can also be put into a *modeling pyramid* [64], as the lower layer is dependent on and more concrete than the higher level of abstraction.

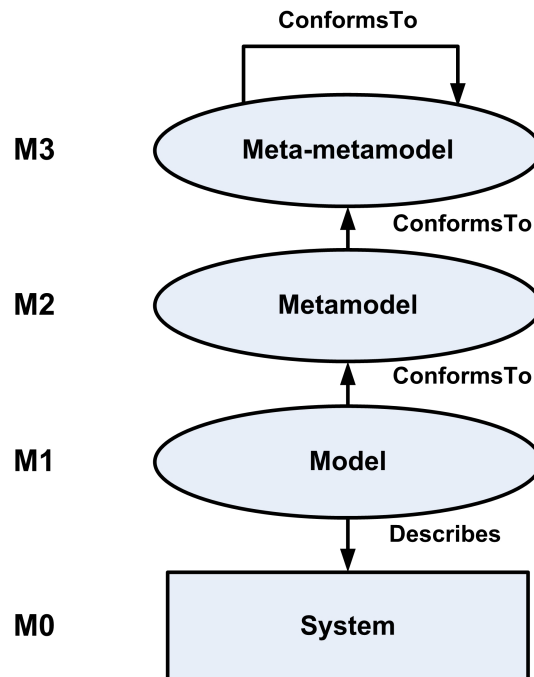


Figure 2.2: Four-Layer Modeling Architecture

### 2.3.1 *Unified Modeling Language*

The growing number of heterogeneous models that are used to facilitate software development has divided the modeling community into two different camps. One camp advocates special-purpose models that tend to focus on individual application domains. An approach called Domain-Specific Modeling (DSM) [82] is a development paradigm that supports this objective. Another camp is interested in general-purpose models that tend to address a wider range of problems that span all different aspects in various systems. The Unified Modeling Language (UML) [38] is the *de facto* standard that falls into this camp. This section covers some basic ideas in UML. The concepts of DSM will be introduced later in Section 2.3.3.

UML was first initiated by the Object Management Group (OMG) in 1997 as a general-purpose modeling language for describing the architecture of software systems. Later in 2003, the growing requirement of MDE necessitated the advent of a major revision of UML. UML is the *de facto* standard formalism that can be used in a wide range of application domains. As such, it contains many diagrams and constructs that represent various perspectives of a system from different viewpoints. The static structural view captures the structure of the system using objects, attributes, operations and relationships. Examples are class diagrams and component diagrams. The dynamic behavior of the system is illustrated by collaborations among objects and changes to the internal states of objects. Examples are sequence diagrams, activity diagrams and state machine diagrams.

UML's definition is based on the Meta-Object Facility (MOF) [93], which is a M3 layer meta-metamodel that is used to build various types of metamodels, including the UML metamodel. An example of the MOF/UML implementation of the four-layer modeling architecture is illustrated in Figure 2.3.

It is generally recognized that there is no single universal modeling language that is suitable to cover every perspective for every domain. Therefore, UML is equipped

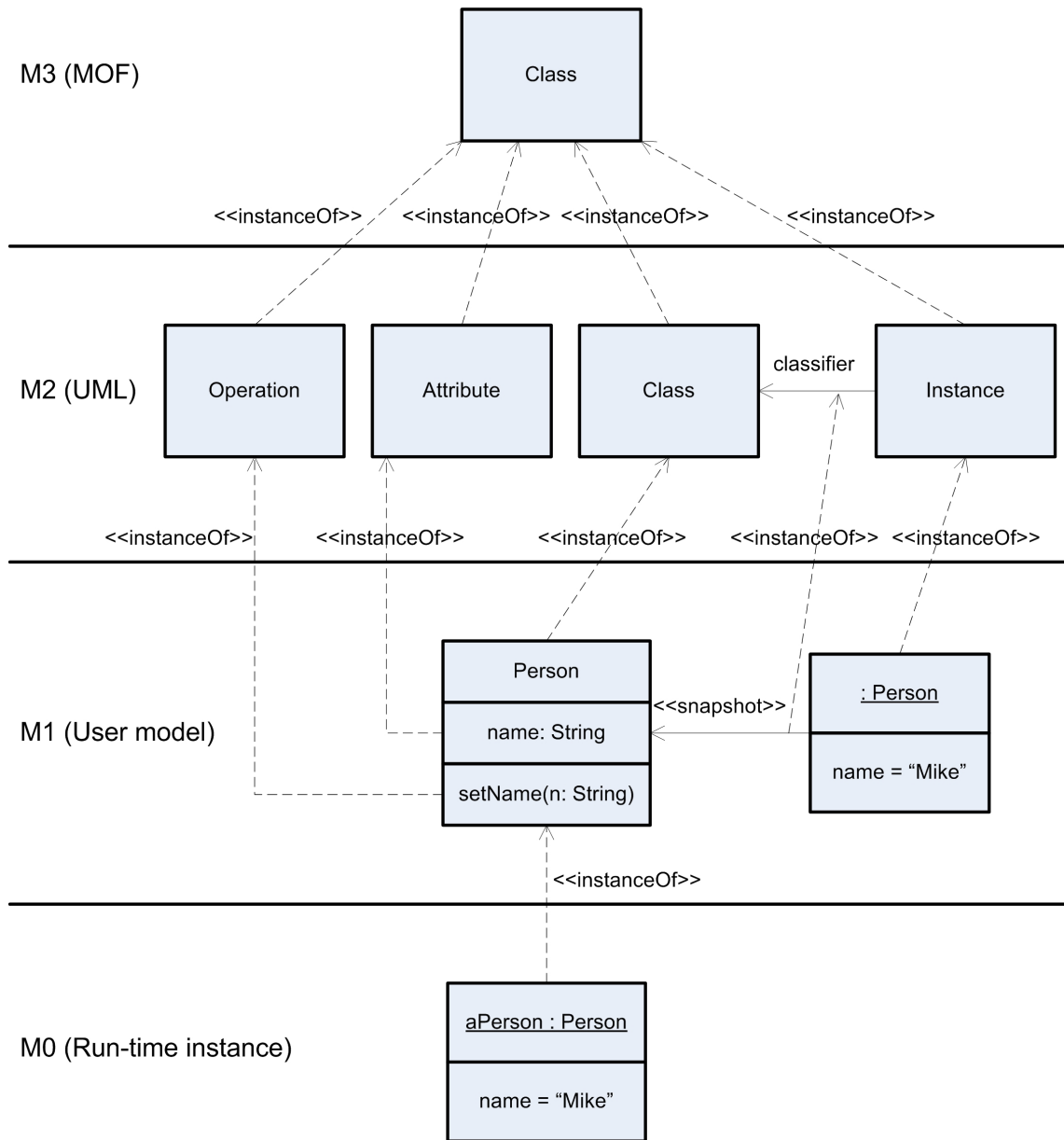


Figure 2.3: A UML Example of the Four-Layer Modeling (Adapted from [95])

with a built-in extension mechanism to accommodate additional concepts beyond those that are defined in the UML specification. There are two options to carry out the extension – the constrained approach via profiles or the augmented approach via MOF.

The constrained approach is realized by the UML profile mechanism. A UML profile provides a generic extension mechanism for customizing UML models for particular domains (e.g., financial or healthcare) and platforms (e.g., J2EE or .NET). It is defined by a collection of stereotypes, tagged values, and constraints that extend elements of the UML metamodel. A model engineer can adopt and implement the profile-based approach with all the generic UML tools that offer the profile feature. This is in contrast to the heavyweight approach that relies on the augmentation of the UML metamodel via the MOF specification (i.e., the metamodel and model spaces are extended), which often depends on vendors to revise their tools (unless a MOF-based metamodeling facility is in place). However, the augmented approach does not have the problem of restricted semantic power as exposed in the profile approach, because model engineers are free to use MOF’s rich set of modeling mechanism to create new modeling languages. In Chapter 3, a specific UML profile is constructed to define an aspect metamodel for activity diagrams.

There exist a number of tools that support development, analysis, testing and code generation of UML models. In this dissertation research, Telelogic TAU [13] was chosen as the experimental platform to perform model specification and evolution. Details will be presented in Chapters 3 and 4.

### *2.3.2 Model-Driven Architecture*

Model-Driven Architecture (MDA) is the OMG’s initiative for advocating the use of models as a complete specification of software artifacts. One of the major objectives of the MDA is to decouple design from architecture. The design concerns

the functional requirements while architecture provides the infrastructure through which non-functional requirements like scalability, reliability and performance are realized. Such separation allows the design specification and the technologies used to realize the architecture to be constructed and changed at their own pace. In MDA, system functionality and domain-related specifications are defined in Platform-Independent Models (PIM) (shown in Figure 2.4) using UML (mostly UML profile). Then, given a Platform Definition Model (PDM) (e.g., J2EE, .NET or any proprietary frameworks and platforms), the PIM is translated to a Platform-Specific Model (PSM) that contains the target platform’s specific concepts. The PSM may be subsequently transformed into more PSMs, which eventually lead to the actual implementation via code generation. Although this dissertation research is not focused on the MDA per se, one of our future works described in Chapter 6 will investigate the leverage of the MDA technology in terms of legacy system evolution.

### 2.3.3 Domain-Specific Modeling

From a modeling perspective, expressive power in software specification is often gained by using notations and abstractions that are aligned to a specific problem domain. According to Evans [60], “Every software program relates to some activity or interest of its user. That subject area to which the user applies the program is the

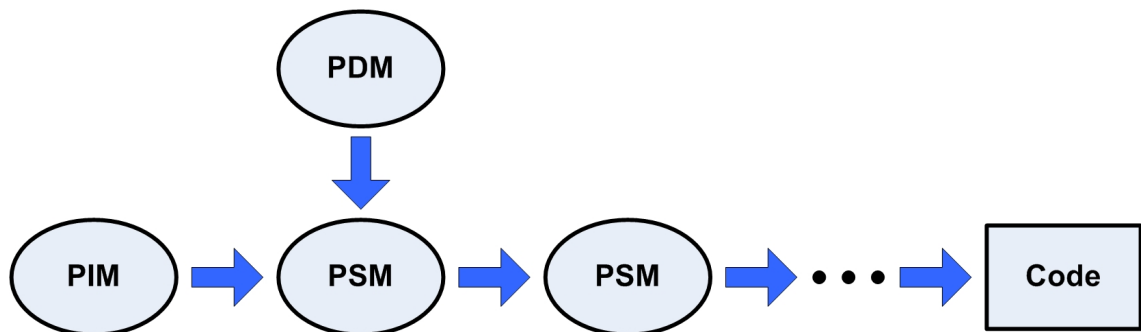


Figure 2.4: MDA Overview



domain of the software.” This can be further enhanced when graphical representations are provided to model the domain abstractions. Domain-Specific Modeling (DSM) [82, 80] techniques offer such support for customization of modeling tools that enable domain experts to construct visual models by making use of a graphical domain-specific language (DSL) [133] directly related to the domain concepts.

DSM is an instantiation of a broader design rationale called “Domain-Driven Design (DDD),” which aims to develop software for complex needs by deeply connecting the implementation to an evolving model of the core business concepts. It is based on two premises [4, 60]:

1. For most software projects, the primary focus should be on the domain and domain logic (as opposed to the particular technology used to implement the system)
2. Complex domain designs should be based on a model.

As shown in Figure 2.5, DSM development starts with the specification of a meta-model that represents the ontology [96] of a particular domain. That is, the meta-model identifies the pertinent entities of the domain, as well as their various associations and constraint rules. Once the domain has been defined, the metamodel is then used to construct a Domain-Specific Modeling Environment (DSME) through the phase of meta-level translation [117]. Subsequently, domain models can be created within this DSME, using the concepts and notations that are associated with the environment. Finally, these models need to be converted into the artifacts that are valuable to the domain experts. This is achieved by model interpreters (also called model compilers, or translators). A DSME may have multiple model interpreters associated with it that possess various semantic intuitions and permit synthesis or generation of different types of application artifacts. For example, one interpretation may synthesize a model to C++ program source code, whereas a different interpretation

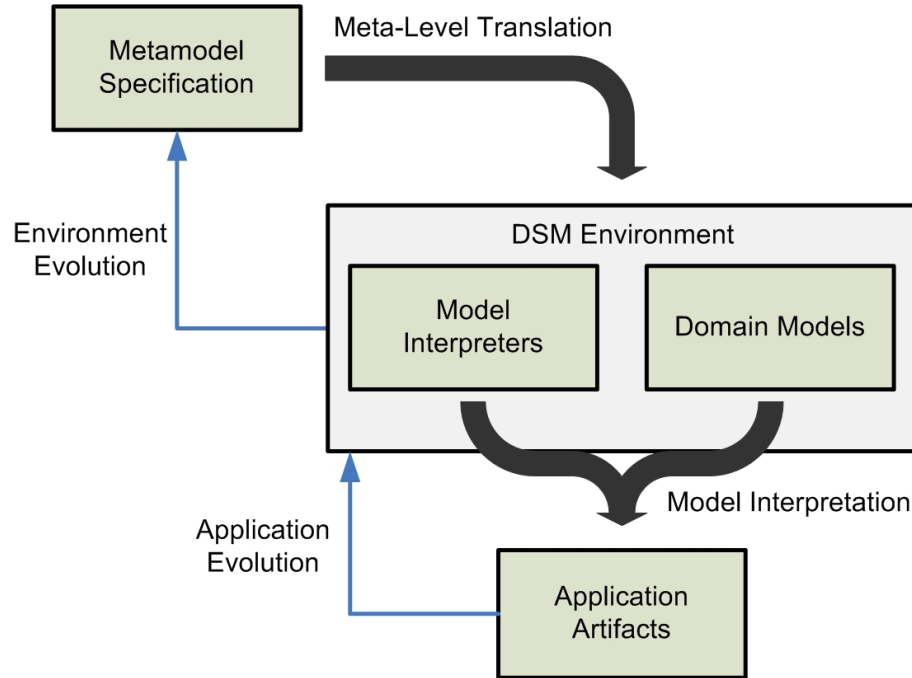


Figure 2.5: Domain-Specific Modeling Development (Adapted From [140])

may synthesize to a simulation engine or analysis tool [139]. A range of tools has been implemented in order to facilitate DSM development, such as the Generic Modeling Environment (GME) [68], DOME [101], MetaEdit+ [179] and Microsoft's Software Factory tools [85].

A comparison can be made between the metamodel and its instance model to that of a programming language definition and a particular program written in that language [24]. The metamodel defines the schema specification for expressing the correct syntax and static semantics of instantiations. Furthermore, a model interpreter, which captures the dynamic semantics of the domain models and generates application artifacts, is akin to a compiler [17] that generates machine code from a programming language. Table 2.1 compares DSM with the similar concepts in the areas of programming language and database definition.

Table 2.1: Comparison of DSM to Programming Language, and Database Definition

	<b>Domain-Specific Modeling</b>	<b>Programming Language Definition</b>	<b>Database Schema Definition</b>
<b>Information Schema Definition</b>	Meta-metamodel (e.g., MOF)	Backus-Naur Form (BNF)	Data Definition Language (DDL)
<b>Schema Definition</b>	Metamodel for a specific domain (e.g., Petri Net)	Grammar for a specific language (e.g., Java)	Table, constraint, and stored procedure definitions for a specific domain (e.g., payroll database)
<b>Schema Instance</b>	Domain model (e.g., Petri Net model of a teller machine)	A program written in a specific language	Intension of a database at a specific instance in time (e.g., the June 2008 payroll instance)
<b>Schema Execution</b>	Model interpreter	Language compiler or interpreter	Transactions and behavior of stored procedures in an executing application

### The Generic Modeling Environment

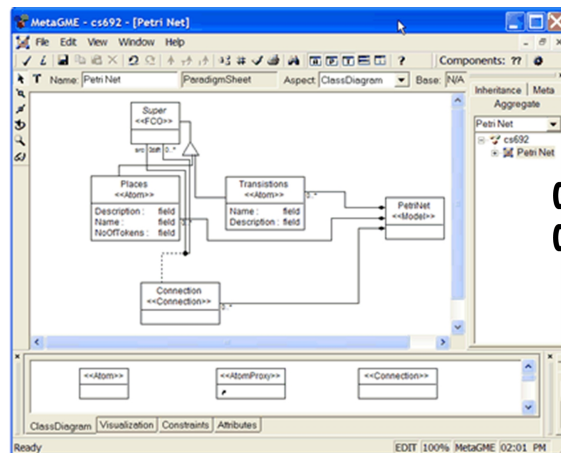
Model-Integrated Computing (MIC) [175] has been refined at Vanderbilt University over the past decade to assist with creation and synthesis of computer-based systems. In MIC, multiple-view models are used to capture the information relevant to the system, represent the dependencies and constraints among different modeling views, and automatically synthesize different kinds of software artifacts. As a variant of the Model-Driven Architecture (MDA) [73], a key application area for MIC is those domains that tightly integrate the computational structure of a system and its physical configuration (i.e., embedded system domains such as avionics and automotive

software). In such systems, MIC has been shown to be a powerful tool for providing adaptability in frequently changing environments.

A specific instance of the type of DSM supported by MIC is implemented using the Generic Modeling Environment (GME) [68]. The GME is a UML-based metamodeling environment that can be configured and adapted from meta-level specifications (i.e., the modeling paradigm) that describe the domain. When using the GME, a modeling paradigm is loaded into the tool to define an environment containing all of the modeling elements and valid relationships that can be constructed in a specific domain [117]. A set of generic modeling concepts are supported to represent entities, relationships and attributes. An *atom* is the most basic type of entity that cannot have any internal structures. A *model* is another type of entity that can contain other modeling types. A *connection* represents the relationship between two entities. *Attributes* are used to record state information and are bound to atoms, models, and connections. Model interpreters supply an ability to generate other software artifacts (e.g., code or simulation scripts) from the models.

As an example, Figure 2.6 illustrates a simplified Petri Net [147] domain as implemented in the GME. The left part of the figure indicates a basic metamodel to represent Petri Nets. The metamodel is described in a UML class diagram [38] with OCL constraints [188] (not shown). It defines places and transitions of a Petri Net (PN) [147], as well as various visualization attributes. From this metamodel, a new PN modeling environment is generated (i.e., bootstrapped from within the GME). The middle of the figure defines an instance of the metamodel that represents a solution to the dining philosopher’s problem [54] as specified in the Petri Net modeling language. The GME permits model interpreters to be associated with specific domains as tool plug-ins. A model interpreter traverses the internal structure of a model and generates different artifacts during the interpretation. The GME provides an API for accessing the internal model structure to permit interpreters to be written in C++ or

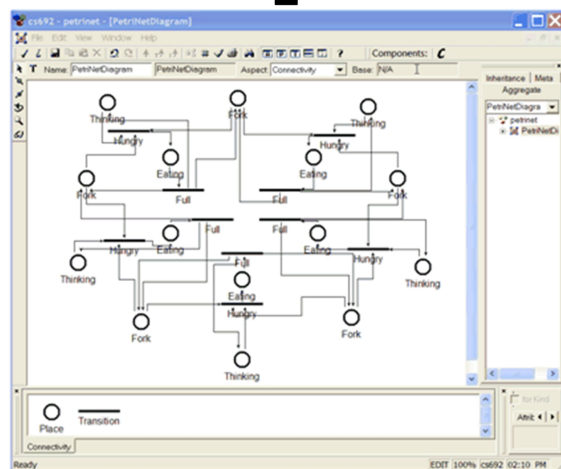
Metamodel



↑ Conforms To

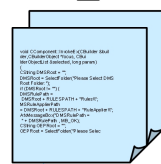
Conforms To

Model



↑ Interpret

Interpreters in C++/Java



Synthesize

Software Artifacts



Figure 2.6: A Petri Net Domain in GME

Java. The bottom of Figure 2.6 symbolically represents the generated artifacts from the interpretation of the dining philosopher’s model; these artifacts could be source code, an XML representation of the model, or some other translation.

Although the example in Figure 2.6 was chosen for simplicity, the GME has been used to create very rich modeling environments containing thousands of modeling components [118]. The GME is a core tool for several dozen research projects with hundreds of users. In this research, the GME is selected as one of the experimental platforms to perform legacy evolution from high-level domain models. However, the presented approach is domain-independent and tool-independent, in the sense that it can be customized to many different domains and tools within any DSM development context.

#### *2.3.4 DSM Evolution*

Based on different layers of specification, as shown in Figure 2.5, current DSM techniques are intended to provide support for two types of software evolution: application evolution and environment evolution. This section outlines these two categories and reveals the fundamental issues involved within each.

### **Application Evolution**

Application evolution (as shown on the bottom of Figure 2.5) is concerned with the evolution of the artifacts in an application domain. The issue of application evolution within the context of DSM corresponds to the model and code co-evolution challenge as described in Section 1.1 of Chapter 1. That is, a robust transformation mechanism is needed as the bridge to enforce the conformity between models and the underlying source code. Although program transformation systems have been under development for several decades, there is little investigation into the merging of mature program transformation systems within model-driven tool-suites. In Chapter

4, a Model-Driven Aspect Adaptation (MDAA) approach is presented to facilitate application evolution in terms of legacy source via the union of model-driven and aspect-oriented techniques.

## Environment Evolution

A special case of application evolution occurs when the changes are made to the DSME itself, as denoted on the mid-left of Figure 2.5. Changing stakeholder requirements often necessitate the need for evolution of the modeling language associated with a domain. The evolution of a domain requires that changes be made to the underlying metamodel. As shown in Figure 2.7, with the evolution of the metamodel, the models and underlying source code that were defined under the previous metamodel are often made invalid under the new metamodel. There exists some difference,  $\Delta_{MM}$ , between the old metamodel and the new one, which captures the evolving features of the domain. For the sake of reusability,  $\Delta_{MM}$  must reflect the difference between the old and new instance models (annotated as  $\Delta_M$ ) such that the new models can preserve the original semantics under the new metamodel definition.

The problem of schema evolution is common across many software development activities (e.g., database schema evolution [120]). To understand this phenomenon better, consider the evolution of a programming language and a compiler defined for a specific definition of the language. If the language were to evolve (e.g., Ada 83 to Ada 95) to a new syntax and semantics, the previous programs may no longer be valid and the previous compilers will not work under the new definition. After a programming language definition changes, it may be necessary to evolve the previous programs defined by the language (e.g., when older programs use deprecated features in previous Java libraries).

Regarding the notion of model evolution in the presence of metamodel schema changes (i.e., the automatic mapping from  $\Delta_{MM}$  to  $\Delta_M$ ), work has already been done

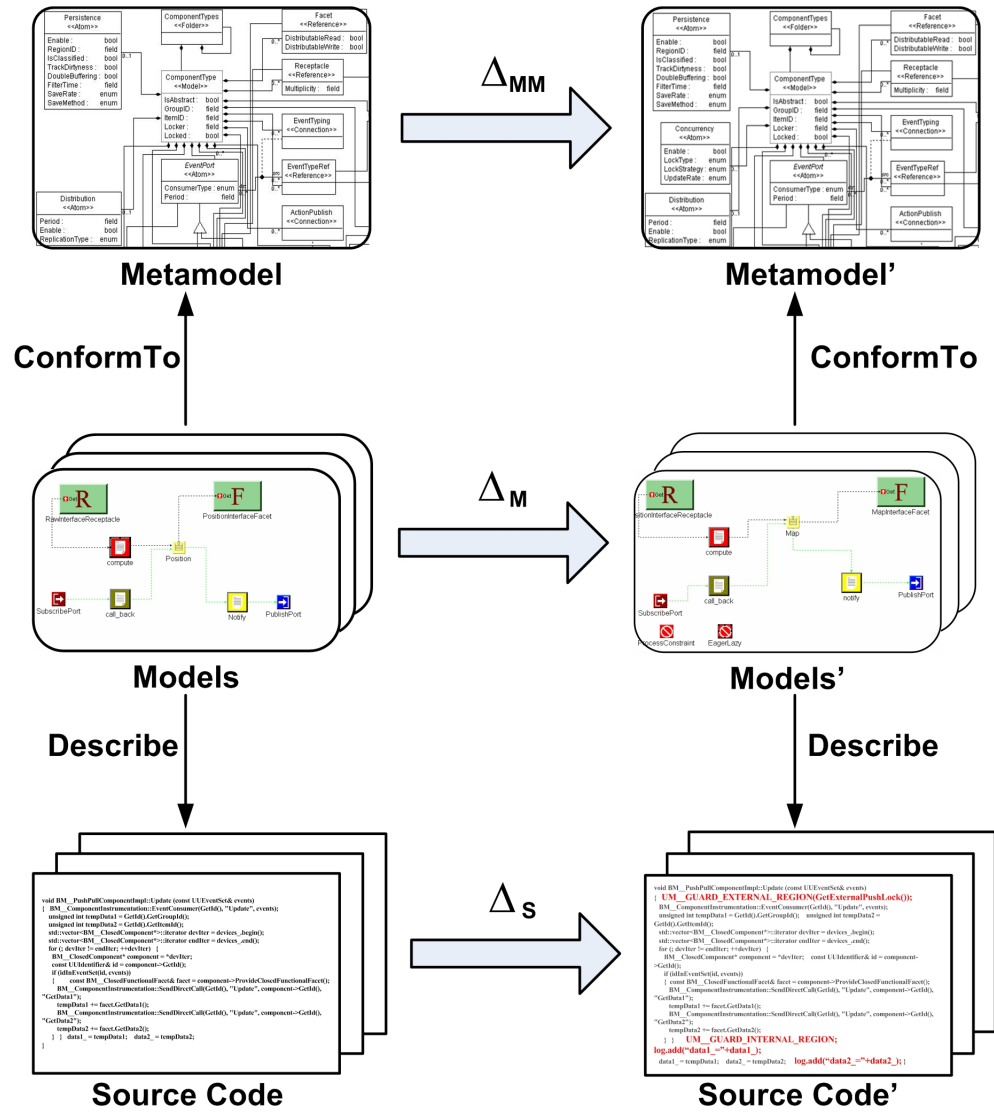


Figure 2.7: Evolution of Models and Source Code in Terms of Metamodel Changes



by others to address this problem [103, 167]. The typical approach is to perform model migration from one representation schema to another using a model transformation engine, as will be described in the next section.

## 2.4 Model Transformation

Model transformation is at the core of MDE [159] and represents the process of taking one or more source models as input to produce one or more target models as output by following a set of transformation rules. A significant amount of research has been performed on supporting model evolution by means of various model transformation technologies, as noted in several surveys on model transformation [53, 129].

### 2.4.1 Model Migration

Sprinkle [166, 167] first generalized and formalized the concepts of model migration (MM) to solve the domain environment evolution problem. Model migration is defined as:

“a total function that operates on a model database,  $M$ , and produces a model database,  $M'$ , using a set of  $m$ -functions (i.e., partial functions that operate on a model,  $m$ , which is contained in  $M$ ). In the absence of a defined  $m$ -function for  $m$ , the model is isomorphically copied into the model database,  $M'$ .” [166]

This definition indicates:

- Each to-be-modified model is associated with a unique mapping function.
- Mapping functions are provided ONLY for the to-be-modified models.
- A model is kept unchanged in the absence of the mapping function.

In MM, the mapping pattern is specified by a visual model composed of entities from the old and new metamodels along with an algorithmic specification for the

model transformation. Figure 2.8 illustrates the form of the MM solution, which operates on a set of domain models to produce another domain model set that will comply with the syntax and semantics of a distinct new metamodel.

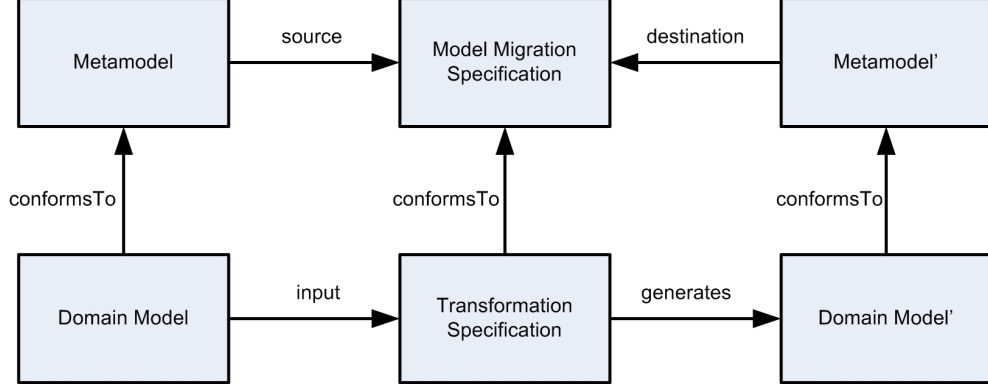


Figure 2.8: Form of the Model Migration solution (Adapted from [166])

#### 2.4.2 GReAT

The Graph-REwriting And Transformation Language (GReAT) [16, 106] is a general model transformation language, following a similar mapping specification as in the MM solution. The difference between model migration (evolution) and transformation is that migration assumes a metamodel and its evolved form are more similar than different; thus, mapping functions are only provided for the differences of two models. However, a transformation approach often converts one model in entirety to another, which requires mapping functions for all of the model elements in the ontology [166].

Graph grammars and graph rewriting [36] have been developed for more than two decades as techniques for formal modeling and tools for high-level programming. The goal of GReAT is to allow the operational specification of rather complex model transformations by utilizing well-known graph grammar and graph transformation [153] methodologies. GReAT adopts a very expressive language to specify patterns, graph rewriting rules and transformation control flow. Besides multiplicities for graph nodes,

the language introduces multiplicities for edges in graph rewriting rules for simultaneous manipulation of sets of rule matches. Additionally, it offers sophisticated control structures, such as sequences, non-determinism, hierarchical expressions, recursion, and branching. The metamodel for all GReAT input graphs (and thus individual nodes and edges) is UML.

The GReAT language is supported through a graph rewriting engine implemented as a plug-in within the GME. The engine works as an interpreter – it takes the model transformation in the form of a data structure, and interprets and executes the model transformation on an input graph to produce an output graph. Because GReAT makes extensive use of the Universal Data Model (UDM) [21] that contains a generic set of APIs, it is suitable to handle all types of model transformations.

### 2.4.3 *ATL*

The Atlas Transformation Language (ATL) [103], which is developed under the Eclipse Model-to-Model transformation (M2M) project [7], aims to define and perform model transformations based on several proposed standards of the Object Management Group (OMG) - the Meta-Object Facility (MOF) and the Query/View/Transformation (QVT) [94] standards [104]. The ATL is a model transformation language that serves as both a metamodel and a textual concrete syntax. It is also a hybrid of declarative and imperative languages. The declarative constructs specify the mappings between source and target patterns according to the source and target metamodels, respectively. When the declarative approach falls short in specifying complex rules, ATL offers an action block with imperative constructs that are used to implement sequences of instructions, such as assignments, loops and even external code invocations. An ATL transformation specification is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models.

The underlying operational mechanism of ATL is shown in Figure 2.9. *Tab* is a transformation program whose execution results in automatic creation of the model *Mb* from *Ma*. These three entities are all model specifications that conform to *MMt*, *MMb*, and *MMa* MOF metamodels, respectively. *MMt* corresponds to the abstract syntax of the transformation language. ATL is a key building block that defines model transformation facilities in the ATLAS Model Management Architecture (AMMA) platform [35], a framework that offers basic facilities to manipulate models, and onto which a variety of different modeling tools may be plugged.

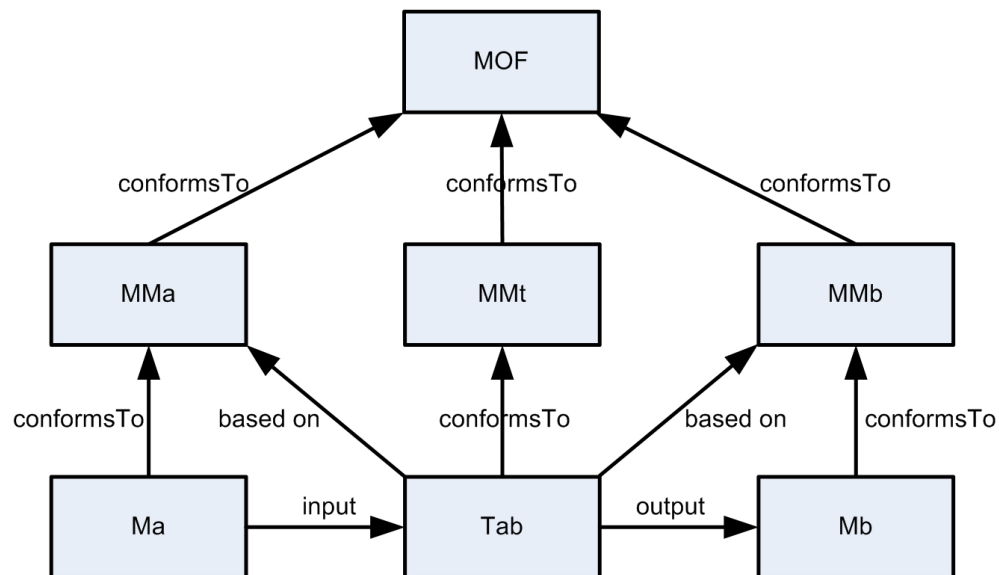


Figure 2.9: Model Transformation through ATL (Reprinted from [104], with permission from Frédéric Jouault)

## 2.5 Program Transformation

Program transformation is about changing one program into another. It is used in a wide range of software engineering areas such as compilation, refactoring, evolution, and optimization. The objective of a program transformation system is to increase the productivity of programmers and reusability of programs by conducting transformation tasks in an automated manner.

A typical program transformation system is built on top of the chosen program representation (e.g., parse/syntax trees or graph), the kind of transformation it specializes for (e.g., compilation/decompilation or refinement, compile-time or run-time transformation), as well as the mechanism adopted for specifying and executing transformations (e.g., term/functional rewriting or strategy-based) [185]. Although a number of techniques have been investigated and developed to support certain kinds of transformations, very few have made it into practice in terms of efficiency and scalability. The next section will introduce a practical, commercial program transformation system that has proven to be applicable to large-scale applications for software analysis, modification, and enhancement.

### *2.5.1 Design Maintenance System*

As a type of program transformation technique, source-to-source transformation systems provide the ability to parse many different programming languages and to perform transformations on the abstract syntax trees (ASTs). The Design Maintenance System (DMS)<sup>3</sup> is a source-to-source transformation engine and re-engineering toolkit developed by Semantic Designs ([www.semdesigns.com](http://www.semdesigns.com)). The core component of DMS is a term rewriting engine that provides powerful pattern matching and source translation capabilities [26, 27]. In DMS terminology, a language domain represents all of the tools (e.g., lexer, parser, pretty printer) for performing translation within a specific programming language. In addition, DMS defines a specific language called PARLANSE as well as a set of APIs (e.g., Abstract Syntax Tree API and Symbol Table API) for writing DMS applications to perform sophisticated program analysis and transformation tasks (see Figure 2.10).

DMS provides pre-constructed domains for several dozen languages (e.g., C++, Java, COBOL, and Pascal). Moreover, these domains are very mature and have

---

<sup>3</sup>In this thesis, there is a distinction between the acronyms DSM and DMS.

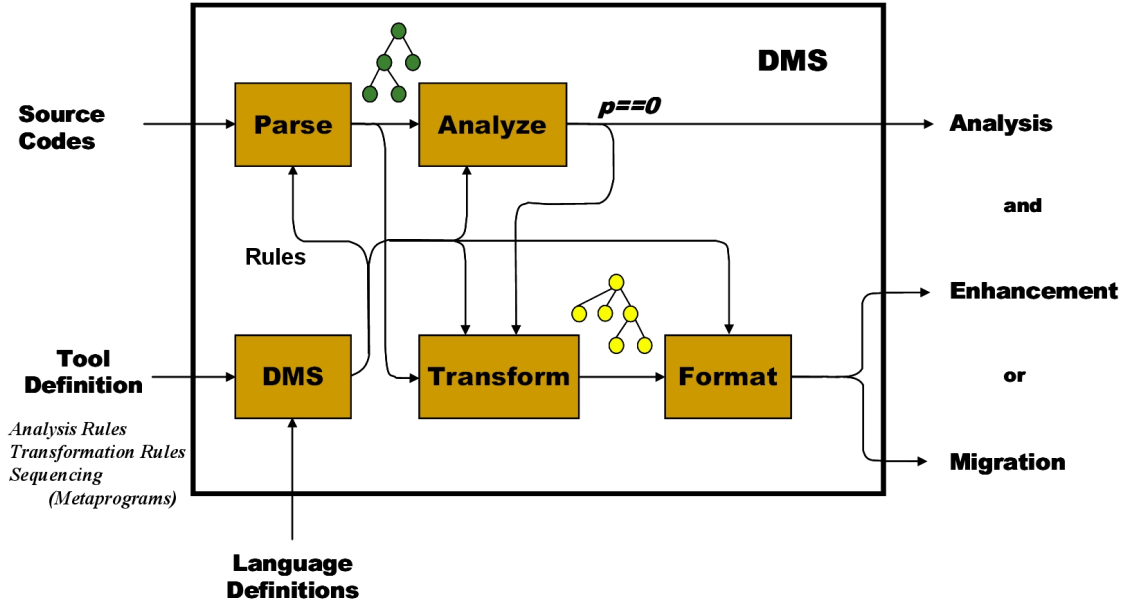


Figure 2.10: DMS Overview (Reprinted from [14], with permission from Ira Baxter)

been used to parse several million lines of code, including the millions of lines of the targeted system explored in the experimental validation plan of the research presented in this thesis [83, 193, 195]. Utilization of mature parsers that have been tested on industrial projects offers a solution to the evolution challenges mentioned in Section 2.3.4. Furthermore, the underlying rewriting engine of DMS provides the machinery needed to perform invasive [19] software transformations on source code. Examples of DMS transformation rules will be given in Section 4.2.

In addition to DMS, there are other popular program transformation systems, such as ASF+SDF [40], TXL [49] and Stratego [186]. DMS, as well as other transformation systems, aim to make it easier to define languages and transformations over those languages. For the technique described in this research to have a real impact, the ability to parse large code bases in multiple languages is essential toward providing a framework for supporting evolution of legacy applications. For this particular reason, DMS was chosen as the underlying program transformation system based on our previous research collaboration with the vendor of DMS (Semantic Designs). From

this collaboration, we were assured that DMS was capable of parsing and transforming millions of lines of code base, which served as one of the experimental platforms for this dissertation research.

## 2.6 Aspect-Oriented Software Development

Traditional software development and modularization mechanisms suffer from an inherent problem – “the tyranny of the dominant decomposition” [177]. That is, the system can be modularized using only one kind of decomposition mechanism at a time, which results in a single concern (e.g., a specific functionality, requirement, feature or task) scattered across many different modules. In other cases, numerous concerns may be tangled within the boundary of one single module. Such concerns are referred to as *crosscutting concerns* that are hard to modularize into distinct modules using the chosen decomposition techniques [109]. As illustrated in Figure 2.11, logging, error handling, synchronization and authorization are representative examples of crosscutting concerns that spread across the base functionality of many system modules.

The occurrence of tangling and scattering often leads to several negative impacts on the software quality in terms of comprehensibility, adaptability and evolvability:

1. Discovering or understanding the representation of a specific crosscutting concern that is spread over the system hierarchy is difficult, because the concern is not localized in one single module. This limits the ability to reason analytically about such a concern.
2. Changing concerns to reflect evolving requirements is also difficult and time consuming, because the engineers must go into each relevant module and modify the specific elements one by one. The change process is error-prone and affects productivity and correctness [79]. As a real-world case study reported in [151],

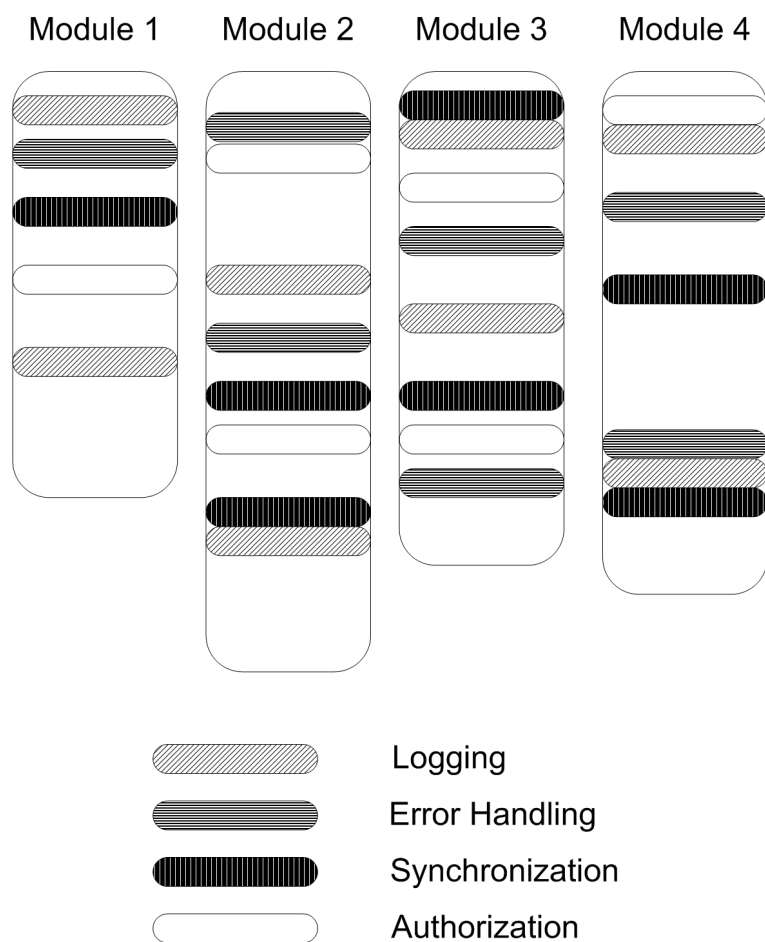


Figure 2.11: Crosscutting Concerns



a change to the logging strategies on an Apache Tomcat server would “require the developer to consider 47 of the 148 (32%) Java source files comprising the core of Tomcat.” This is because the logging concern is spread over 47 different locations without a cohesive container to modularize this concept.

Aspect-Oriented Software Development (AOSD) [2] offers a powerful technology for handling such concerns, whereby the crosscutting is explicitly specified in a standalone module called an *aspect*. Thus, a logging module only deals with logging and is not tangled with other business logic. An aspect contains information about what is the behavior of the crosscutting concern and where it is to emerge. Taking account of such information, a specialized composition engine called an *aspect weaver* is responsible for integrating the crosscutting concerns represented by aspects into the base system modules. The whole integration process is thus called *aspect weaving*. As depicted in Figure 2.12, the four different concerns (i.e., logging, error handling, synchronization and authorization) are extracted and modularized into separate aspects, respectively. This way, the system modules can be considered highly cohesive in terms of the clean separation of functionality. Crosscutting concerns are no longer distributed over different modules. Therefore, the system is easier to understand, maintain and evolve.

AOSD can be beneficial at various levels of abstraction and at different stages of the software life cycle. The next two sections will introduce aspect-oriented approaches at the programming language level as well as higher levels of abstraction (e.g., the modeling level).

### 2.6.1 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) deals with crosscutting concerns at the implementation level. A distinguishing characteristic of AOP is that “it allows programming by making quantified programmatic assertions over programs written by

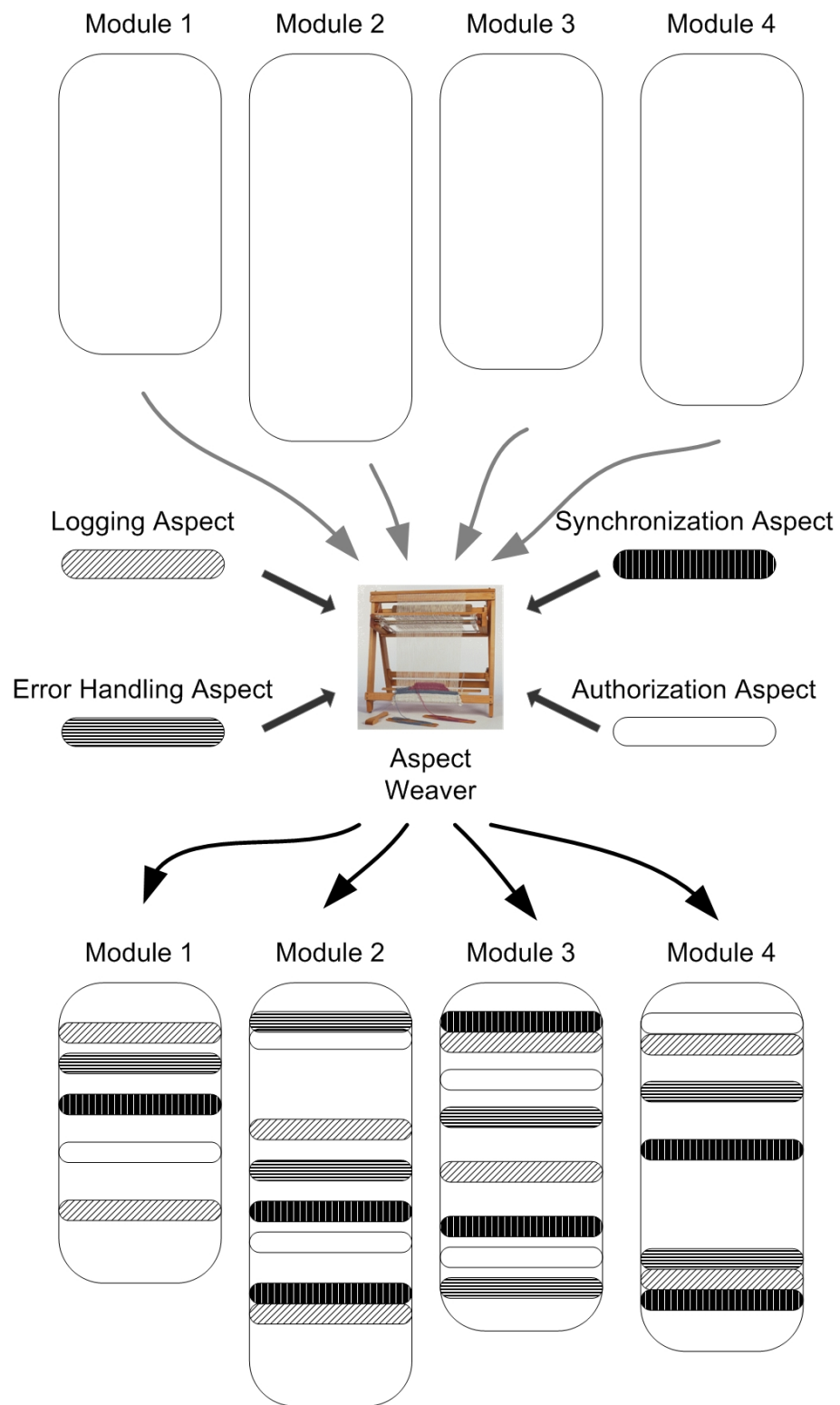


Figure 2.12: Aspect Weaving

programmers oblivious to such assertions” [66]. The *quantification* lies in the ability to identify specific locations in the program that may have arisen under certain conditions. The term *obliviousness* means that programmers are unaware of the fact that their programs are interwoven by external behavior introduced by aspects. This indicates three essential constituents of an AOP implementation:

1. The constructs to specify different kinds of *conditions* (or *locations*) during the program execution.
2. The constructs to specify different kinds of *behavior* (or *actions*) that crosscut the base program.
3. The mechanism to compose crosscutting behavior with the base program.

A wealth of technologies and tools are currently available that support AOP, such as AspectJ [108], AspectC++ [165], JAC [146] and JAsCo [174]. Based on different target programming paradigms (e.g., functional languages, object-oriented programming, or component-based programming) and weaving requirements (e.g., compile-time or run-time weaving), different technologies may adopt various approaches to implementing the above three constituents of AOP. The next section will focus on the introduction of AspectJ – the most popular and mature language to offer AOP capabilities.

## AspectJ

AspectJ had its genesis from the early work in 1996 by Kiczales and his team at Xerox PARC [15], and ever since then, it has become the most widely-used AOP implementation. It is often regarded as a general-purpose aspect language in that it provides a rich set of constructs to capture a wide variety of different kinds of crosscutting concerns.

As an extension to Java, AspectJ is specifically tailored for the object-oriented programming paradigm. It stays as close as possible to the original Java syntax and concepts, and uses Java to implement whatever behavior an aspect should provide. In AspectJ terminology, an aspect is a new class-like language construct that contains several special entities:

### 1. **Aspect**

An aspect is a modular unit that implements a crosscutting concern, comprising pointcuts, advice, and inter-type declarations. An abstract aspect contains one or more abstract pointcuts and can be specialized by the extending aspects.

### 2. **Pointcut**

A *pointcut* denotes certain conditions or locations in the program flow. These locations are referred to as join points that are well-defined points in the execution of a program. Due to the essential nature of object orientation, the join points in AspectJ include method or constructor call or execution, the initialization of a class or object, field read and write access, and exception handlers. A pointcut is specified by a declarative expression called a pointcut designator that is used to determine whether a given join point matches during run-time execution. A pointcut supports quantification over the source code. A pointcut can be declared as abstract without a body definition. Abstract pointcuts can be specialized by providing a concrete pointcut definition in the extending aspects. This is similar to the way methods can be declared abstract and later defined in sub-classes. Some of the primitive pointcuts and combinators provided by AspectJ are as follows:

#### **call(MethodPattern)**

every method call join point whose signature matches MethodPattern

**execution(MethodPattern)**

every method execution join point whose signature matches MethodPattern

**! Pointcut**

every join point not picked out by Pointcut

**Pointcut1 && Pointcut2**

each join point picked out by both Pointcut1 and Pointcut2

**Pointcut1 || Pointcut2**

each join point picked out by either Pointcut1 or Pointcut2

**(Pointcut)**

each join point picked out by Pointcut

**cflow(Pointcut)**

every join point in the control flow of each join point P picked out by Pointcut, including P itself

**cflowbelow(Pointcut)**

every join point below the control flow of each join point P picked out by Pointcut, excluding P itself

**within(TypePattern)**

every join point where the executing code is defined in a type matched by TypePattern

**3. Advice**

Pointcuts represent where the crosscutting concerns emerge. *Advice* specifies the actual implementation of a crosscutting concern. Advice binds an action body to a pointcut, meaning that the body executes at join points that the pointcut matches. There are three different kinds of advice. *Before advice* executes just before the program execution flow reaches the join points that are picked up by the specified pointcut. For instance, before advice on a method call would run

before the method is invoked. Similarly, *after advice* runs after the join point execution and *around advice* runs in place of the join point. AspectJ provides a special reference variable, *thisJoinPoint*, which contains reflective information about the current context of the join point for the advice to use.

#### 4. Inter-type declaration

Pointcuts and advice are often used to intercept a program's execution flow dynamically during run-time. There is another type of weaving called an *inter-type declaration* that enables a static addition to the structural hierarchy of the program. It allows a programmer to add fields, methods, or interfaces to existing classes from within the aspect. A new type of class can also be declared to extend (or implement) the existing class (or interface).

There are different ways to weave aspects into the base program. *Source weaving* requires access to program source files and produces source files that are augmented with aspects. *Bytecode weaving* performs the weaving at the bytecode level and is able to work with any Java program in the form of a class file. With *load-time weaving*, the weaving process is deferred until the point that a class loader loads a class file and registers the class to the underlying JVM. AspectJ started with the implementation of source-level weaving at the beginning, delivered a bytecode weaver later, and now offers a number of mechanisms to support load-time weaving. Some research has focused on a more advanced mechanism called dynamic AOP that enables *run-time weaving* [6]. AspectJ does not currently support the full feature of run-time weaving, though it is claimed that advice can be enabled and disabled in aspects dynamically.

In this dissertation research, AspectJ has been used as one of the underlying aspect languages supported in the Model-Driven Aspect Adaptation (MDAA) framework that will be introduced in Chapter 4. Examples of aspect specification in AspectJ will be given in Section 4.3.

### 2.6.2 *Aspect-Oriented Modeling*

As AOP handles crosscutting concerns at the programming language level, Aspect-Oriented Modeling (AOM) denotes a technology that focuses on specifying crosscutting concerns at a higher level of software abstraction. As a motivating example for AOM, Figure 2.13 illustrates a model of a system that contains concerns that cut across the modeling hierarchy. The top of the figure shows the interaction among components in a mission-computing avionics application (detailed introduction will be given in Chapter 4). The middle of the figure shows the internal representation of two components, which reveals the data elements and other constituents intended to describe the infrastructure of component deployment and the distribution middleware.

Among the components in Figure 2.13 are a lock element, a logging element, a condition element and two data elements (circled). Each of these modeling elements represents a system concern that is spread across the model hierarchy. The lock element (solid gray circle) identifies a system property that corresponds to the synchronization strategy distributed across the components. The collection of elements (dotted gray circle) defines the recording policy of a black-box flight data recorder. Some data elements also have an attached precondition (dotted black circle) to assert a set of valid values when a client invokes the component at run-time.

To analyze the effect of an alternative design decision manually, model engineers must change the synchronization or flight data recorder policies, which requires making the change manually at each component's location. The partial system model in Figure 2.13 is a subset of an application with more than 6,000 components. Manually changing a policy will strain the limits of human ability in a system that large. As an alternative solution, AOM offers a powerful mechanism to automate such change evolution in a modular way. The next section will introduce an AOM weaver that is tailored for addressing the crosscutting issue in models.

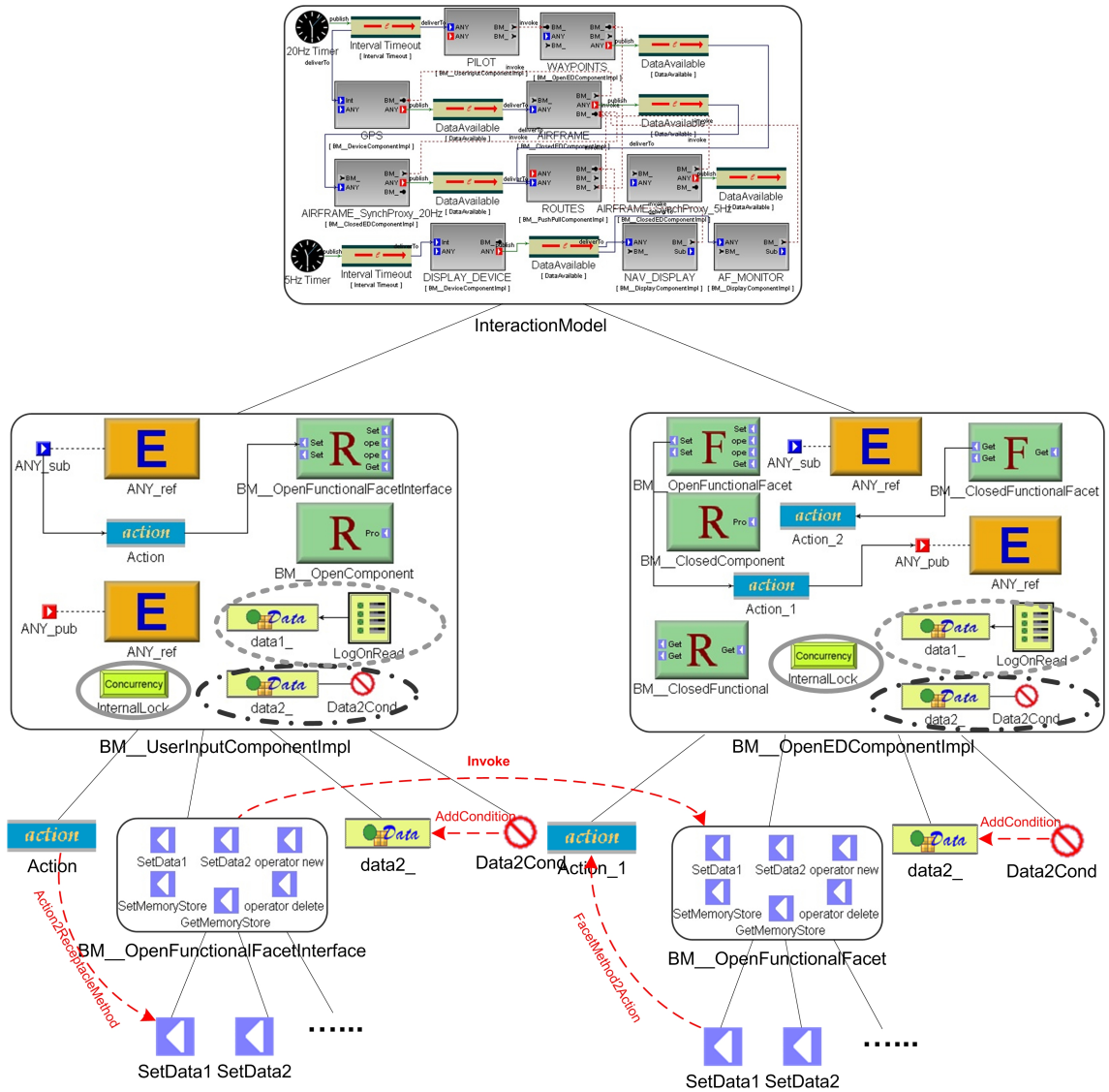


Figure 2.13: Crosscutting Concerns in Models (From [79])



## Constraint-Specification Aspect Weaver

The Constraint-Specification Aspect Weaver (C-SAW) is an AOM engine implemented as a plugin component for GME. C-SAW unites the ideas of AOSD [2] with MIC [175] to provide better modularization of model properties that are crosscutting throughout multiple layers of a model [77]. C-SAW offers the ability to explore numerous modeling scenarios by considering crosscutting modeling concerns as aspects that can be rapidly inserted and removed from a model. This permits a model engineer to make changes more easily to the base model without manually visiting multiple locations in the model. Until C-SAW, these transformations and translations have largely been performed manually in practice. Additional information about C-SAW, including software downloads and video demos, is available at: <http://www.cis.uab.edu/gray/Research/C-SAW>.

The C-SAW model transformation engine is depicted in Figure 2.14. In this figure, a source model serves as input to the model weaver, and the output is a target model that has a crosscutting concern dispersed across the original base. To perform this process, the transformation specifications describe the binding and parametrization of strategies to specific entities in a model. A transformation specification is composed of an aspect and several strategies. An aspect is the starting point of a transformation process. A strategy is used to specify elements of computation and the application of specific properties to the model entities.

The specification aspects and strategies are based on a special underlying language, called the Embedded Constraint Language (ECL) [84]. The ECL is an extension of the Object Constraint Language (OCL) [188], and provides many of the common features of OCL, such as arithmetic operators, logical operators, and numerous operators on collections (e.g., size, forAll, exists, select). ECL also provides special operators to support model aggregates (e.g., models, atoms, attributes), connections (e.g., connpoint, target, refs) and transformations that provide access to modeling

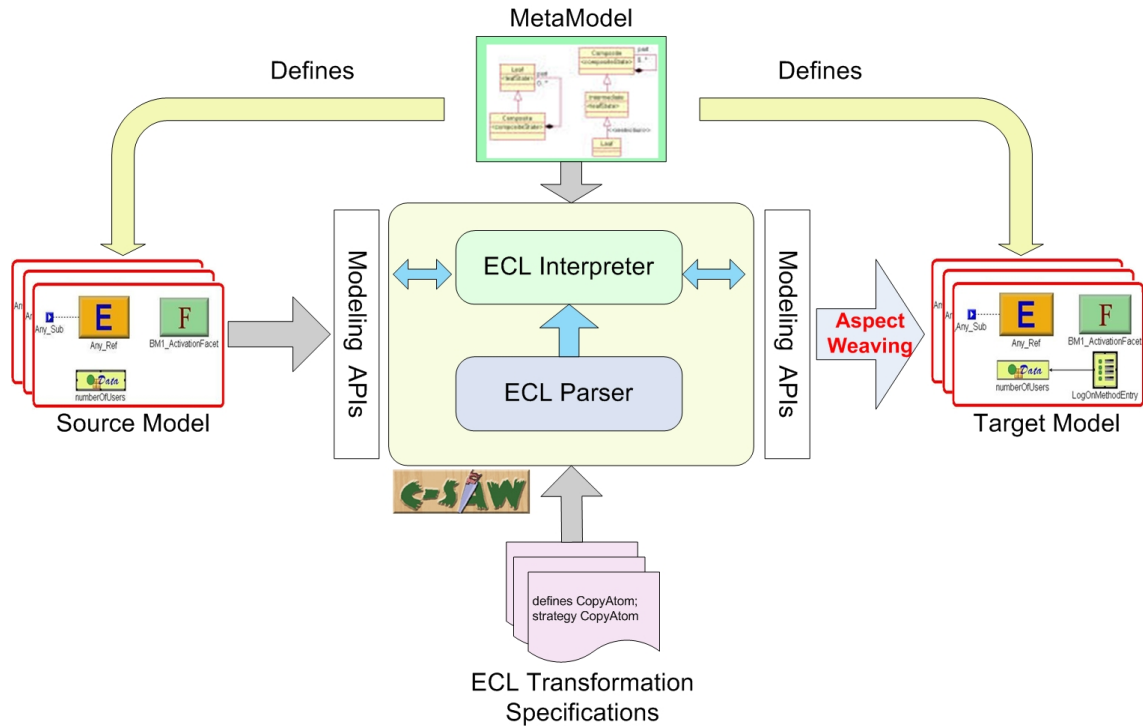


Figure 2.14: C-SAW Overview (From [79])

concepts that are within the GME (e.g., addModel, setAttribute, removeNode).

ECL is distinct from OCL with respect to side-effects and model manipulation features. OCL is a declarative language and cannot support operations to create, update or remove the entities within a model, whereas the use of ECL requires the capability to introduce side-effects into the underlying model. This is needed because the strategies often specify transformations that must be performed on the model. This requires the ability to make modifications to the model as the strategy is applied. ECL supports an imperative transformation style with a number of operations that can modify the structure of the model.

Figure 2.15 lists an example of the aspect specification in ECL for enforcing certain constraints to the data element in the component model as described previously in Figure 2.13. The **Precondition** aspect (line 22 to 26) specifies that all the data elements named **data2\_** must be associated with a precondition defining the valid range

```

1  defines Precondition, InsertPrecondition2Data, InsertPrecondition;
2
3  strategy InsertPrecondition2Data(data, condition, expr: string)
4  {
5      atoms()->select(a | a.kindOf() == data)->
6          InsertPrecondition(condition, expr);
7  }
8
9  strategy InsertPrecondition(condition, expr string)
10 {
11     declare parentModel : model;
12     declare dataAtom, conditionAtom : atom;
13
14     dataAtom := self;
15     parentModel := parent();
16     conditionAtom := parentModel.addAtom("Condition", condition);
17     conditionAtom.setAttribute("Kind", "Precondition");
18     conditionAtom.setAttribute("Expression", expr);
19     parentModel.addConnection("AddCondition", conditionAtom, dataAtom);
20 }
21
22 aspect Precondition()
23 {
24     rootFolder().findFolder("Component").models().
25         InsertPrecondition2Data("data2_", "Data2Cond", "value>100");
26 }

```

Figure 2.15: The ECL Aspect Code for Inserting Precondition Element to the Component Model in Figure 2.13 (From [79, 195])

of values (`value>100`). The strategies `InsertPrecondition2Data` and `InsertPrecondition` implements such a functionality and is invoked within the aspect under the context of component models. More examples and detailed explanation will be given in Chapter 4, where C-SAW is used as an underlying model transformation engine to perform aspect weaving within the Model-Driven Aspect Adaptation (MDAA) framework.

## Motorola WEAVR

Motorola has previously developed an industry-strength weaver [50] for enabling aspect-oriented weaving for transition-oriented state machine models [51, 191, 192].

The Motorola WEAVR follows the AspectJ approach in terms of language construct definition (e.g., aspect, pointcut, join point and advice). Based on the UML concepts that actions are executed during a transition from one state to another state, two distinct types of join points are supported in the WEAVR: action and transition join points, referring to the actions and transitions declared in the state machines, respectively. All these constructs are denoted by corresponding UML stereotypes that are defined by a UML profile. By weaving aspects into executable UML models, the platform-specific models and the source code can be generated in an automated manner. In addition, the WEAVR provides a join point visualization engine that allows the effects of an aspect on a state machine model to be visualized and validated. A simulation engine is also enabled that allows aspect models to be simulated, without breaking the modular structure of aspects. The aspect-oriented activity modeling approach, as will be introduced in Chapter 3, extends the Motorola WEAVR to support aspect weaving for activity-based behavioral models.

In addition to C-SAW and the Motorola WEAVR, a number of AOM approaches have been proposed based on different modeling paradigms and different purposes towards aspect-orientation. In contrast to C-SAW, most of these approaches are based on UML, such as the Theme/UML Approach [48], the Aspect-Oriented Design Model (AODM) [169] and Aspect-Oriented Architecture Models (AAM) [72]. However, very few proposals have tool support for aspect weaving. As stated in a recent survey paper [156], only two out of eight UML-based AOM approaches have real tool implementation, which indicates that research on AOM is still in its early phases and has a long way toward maturity. As a step toward this direction, an AOM approach for UML activity modeling will be presented in the next chapter. The implementation details and representative examples will also be given for illustrating the approach.

## CHAPTER 3

### ASPECT-ORIENTED ACTIVITY MODELING

Activity modeling is a core part of the UML that is frequently used to specify the dynamic behavior of a system. Activity models are often applied to document workflows in a system (e.g., the logic of a single operation, the scenario of a use case, or the flow logic of a business process). In UML 1.x [90], activity models were defined as a special case of state machines, mainly for describing a computational process in terms of control flow and data flow in state-transition-oriented systems. Since the adoption of the new UML 2.x specification [95], activity modeling has been redesigned and based on Petri Net (PN) [147] semantics instead of state machines, which “widens the number of flows that can be modeled, especially those that have parallel flows” [95]. It is believed that with such enriched expressive power and well-defined semantics, activity modeling will gain more popularity in the design and development of complex software systems.

Due to the increasing complexity of software systems, it is often the case that a single activity may be scattered over several different activity modules. Such activities are *crosscutting concerns* that are hard to modularize into separate units using existing activity modeling constructs. With the intent to support separation of crosscutting concerns involved in activity specification, this chapter applies an AOSD approach to activity modeling. An aspect-oriented extension to activity modeling is introduced for encapsulating crosscutting concerns in the constructs of aspects, which are systematically integrated with the base activities by an underlying aspect

model weaver. This work extends the Motorola state machine weaver (as introduced in Chapter 2) with support for activity models. The goal is to provide designers with more coherent and manageable activity modules through the clean separation of concerns.

The remainder of this chapter is structured as follows. Section 3.1 gives a brief overview of activity modeling, including the activity metamodel definition. Based on the activity metamodel, Section 3.2 presents an aspect-oriented extension to activity modeling. The underlying aspect composition mechanism is described in Section 3.3. Finally, a case study that illustrates the Aspect-Oriented Activity Modeling (AOAM) approach is given in Section 3.4, which specifies a timeout handler aspect for a network fault management system.

### 3.1 Activity Modeling

Activity modeling is intended to specify the behavioral aspects of a system. It is typically used to define a computational process in terms of the control flow and data flow among its constituent actions. This section provides a basic background introduction to activity modeling in order to set the context for our aspect-oriented enhancement to activity modeling.

Figure 3.1 shows the simplified activity metamodel in the Meta-Object Facility (MOF) [93] specification. An activity contains various kinds of nodes connected by edges to form a complete flow model. The sequencing of actions is controlled by control flow and object flow edges. An activity node can be an action, an object node or a control node. Some of the common kinds of actions are listed in Figure 3.1. An operation action may reference an activity specification, which means that the invocation of the operation involves the execution of the referenced activity. The send signal action and accept event action deal with signal and event transmission, respectively. An object node holds data that flow through the activity model. A pin

Figure 3.1: Simplified Activity Metamodel (Adapted from [95])

is an object node that can be attached to actions for expressing inputs and outputs. Control nodes are responsible for routing control and data flows in an activity. For instance, decision node and merge node are used to designate conditional behavior, but fork node and join node are used to delineate parallel behavior. Activities can be divided into different partitions that represent various kinds of activity groups for identifying actions that have some characteristics in common. Activity actions can also be grouped into an interruptible region, within which all execution can be terminated if an interrupting activity edge is leaving the region. The graphical symbols that are used in the activity modeling diagram are shown in Table 3.1.



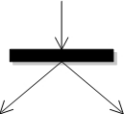
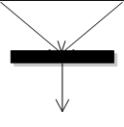
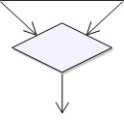
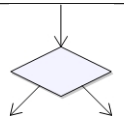




As an illustrative example, an order processing activity model is specified in Figure 3.2. Two concurrent flows are involved. One focuses on the normal procedure for order processing, including order receiving (indicated by the **Receive\_Order** action), order processing (i.e., the **Op\_Process\_Order** action), payment handling (denoted by the **Request\_Payment** send signal action and **Payment\_Confirmed** accept event action) and order shipment (i.e., the **Ship\_Order** action). Another flow indicates that during the same period of time that the first flow proceeds, the order will be cancelled whenever a **Cancel\_Order\_Request** event is received.

### 3.2 Aspect-Activity Model Specification

As the complexity of the described system grows, activity specifications also increase in complexity. This growth requires lifecycle maintenance for the concerns that crosscut different activity modules. For instance, a new requirement asking for a tracing capability that logs all the information of all operation actions results in appending a trace activity to every operation action. The trace activity is a cross-cutting concern that is hard to modularize into a single activity or action unit using existing activity modeling techniques. Such a concern can be extremely difficult to comprehend and change due to its scattering nature.



Table 3.1: Graphical Symbols in Activity Diagram

Node Type	Symbol
Initial Node	
Final Node	
Fork Node	
Join Node	
Merge Node	
Decision Node	
Operation Action Node	
Send Signal Action Node	
Accept Event Action Node	
Accept Time Event Action Node	

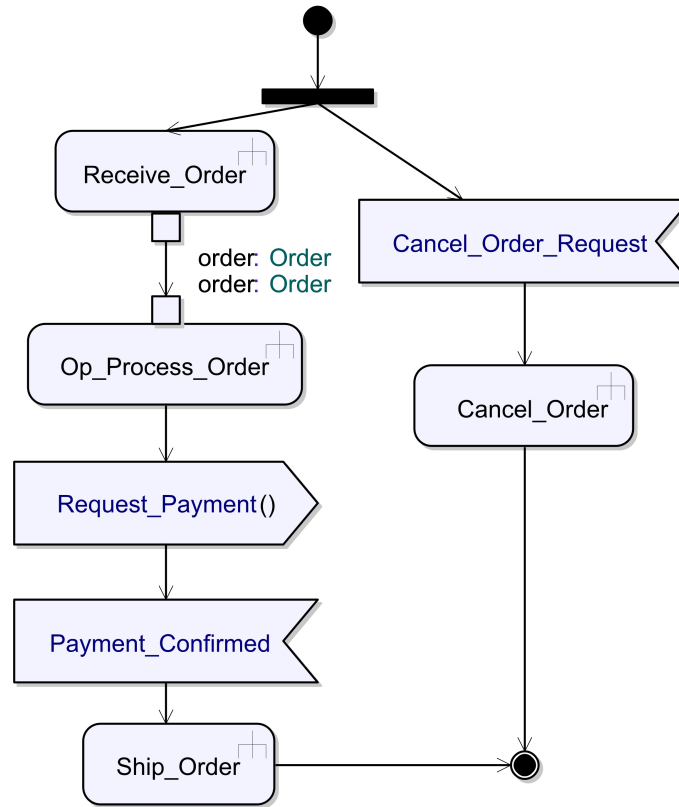


Figure 3.2: An Order Processing Activity Model (Adapted from [95])

The application of aspect-oriented approaches [1, 2] to activity modeling provides a solution to support this kind of modularization by encapsulating crosscutting concerns in a specialized unit called an *aspect*. Following the Aspect-Oriented Programming (AOP) [109] terminology, two fundamental constructs are involved in an aspect model. First, we need to specify “where” (i.e., the locations, or *join points*) in the models the crosscutting behavior emerges. Based on the activity metamodel definition, which defines an activity as being composed of a sequence of actions, join points refer to various kinds of actions that are allowed in activity modeling. A group of particular join points are represented in a *pointcut*, which defines a pattern to identify matching join points.

Second, we need to specify “what” (i.e., the behavior) makes up the crosscutting concern. In activity modeling, the concern behavior is implemented using an activity

model referenced by a special action called *advice*. An advice may contain a *proceed* operation action that refers to the current join point (i.e., a *proceed* is used to invoke the original matched join point action). An advice can also obtain the join point information through a set of predefined reflective APIs (e.g., `getSignature()` and `getKind()`) operations as shown in the `thisJoinPoint` class in Figure 3.3).

### 3.2.1 Aspect-Activity Metamodel

The aspect-activity modeling concepts are defined upon a light-weight extension of UML through profiles and stereotypes [95]. As shown in Figure 3.3, an aspect is a special activity that encapsulates a crosscutting concern. Pointcuts and advice are denoted as special actions that refer to different types of actions as defined in the activity metamodel. An aspect-activity model contains a binding diagram that defines which advice is bound to which pointcuts. Those bindings are realized by

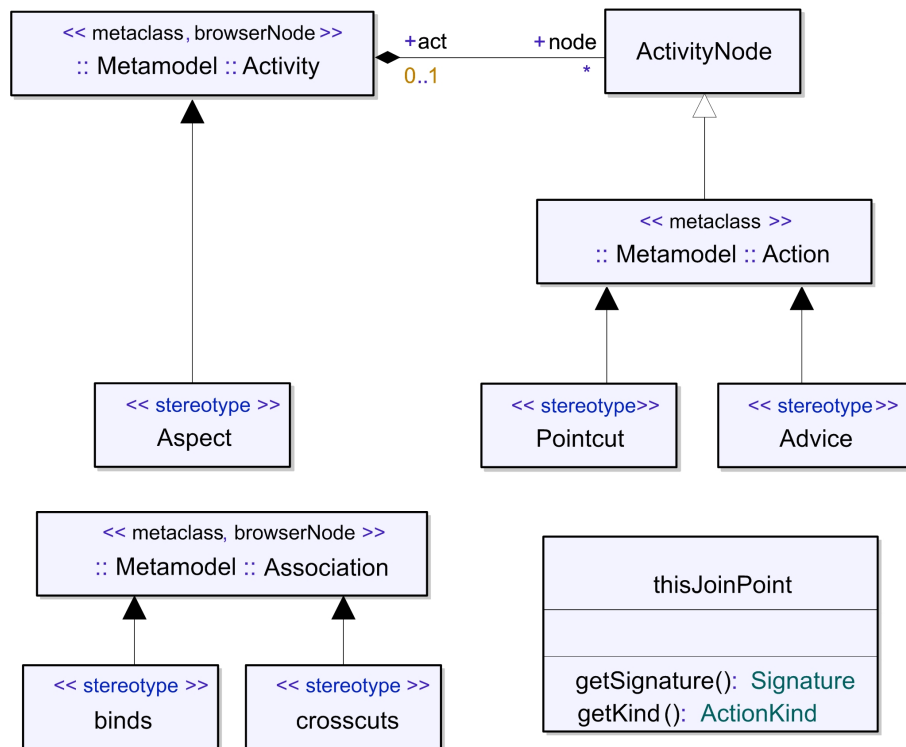


Figure 3.3: Aspect-Activity Modeling Profile

a stereotype named *binds*, which is a special association <sup>4</sup> that denotes the binding relationship between two modeling elements (i.e., the advice action and pointcut action in the aspect-activity metamodel definition). Aspects are deployed to the base activity models through a special association stereotyped by the name *crosscuts*. The *thisJoinPoint* class defines a set of APIs that are used to retrieve the reflective information of the matched join points, including the signature of an operation action and the kind of the join point action (e.g., operation action, send signal action or accept event action).

As an illustrative example, Figure 3.4 specifies a trace aspect model, with an aspect called **TracingAspect** applied to a base activity model. The purpose is to keep track of certain actions involved throughout the execution of the base activity flow. This aspect contains one advice that is bound to four different pointcuts. The pointcut **Cancel\_\*** denotes all of the operation actions whose names start with **Cancel\_** (e.g., the **Cancel.Order** operation action in Figure 3.2). The underlying pattern matching is based on the node type mapping as well as the regular expression mapping against the pointcut name. The pointcut **Op\_Process\_Order** refers to an operation action that has one parameter of the type **Order**. **Request\_Payment** and **Cancel\_Order\_Request** match to a send signal action and an accept event action, individually. The advice action **Trace** is implemented by an activity model, which extends the original join point action (denoted by **proceed**) with an operation action that logs the join point information (e.g., action signature or operation parameter value). An aspect model can also introduce inter-type members [108] that are to be inserted into the join point action implementations (e.g., the integer **flag** declared in **TracingAspect**).

The aspect and the base models are automatically composed together through a specialized aspect weaver for activity models, as indicated in Figure 3.5. The weaving procedure starts with instantiating advice based on the pointcuts they are bound to.

---

<sup>4</sup>In UML, an association specifies a semantic relationship that can occur between typed elements.

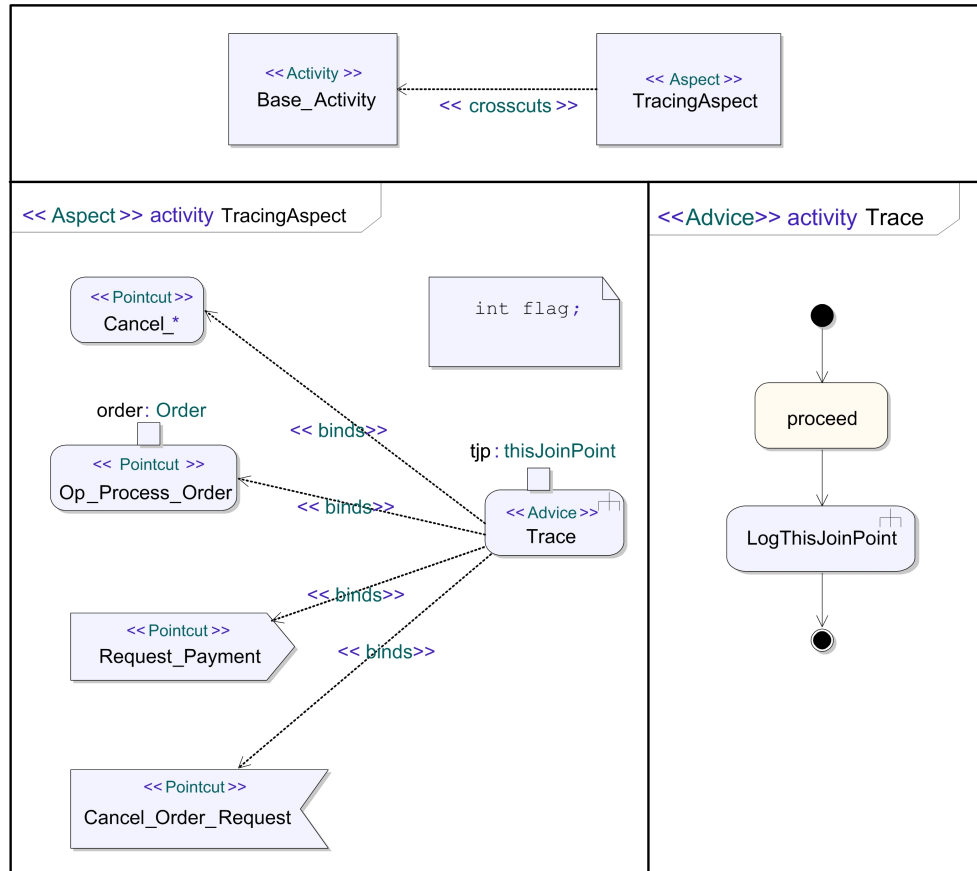


Figure 3.4: Tracing Aspect, Pointcuts and Advice

All of the calls to the reflective API are resolved based on the current join point. The `proceed` actions are replaced by the original join point action. These advice instances are in turn woven into the base models in one of the following two ways: wrapping or inlining. In the wrapping mode, the original join point action is replaced by an operation invocation to the corresponding advice instance. For the inlining version, the contents of all the advice instances are directly embedded into the base activity models. Figure 3.6 illustrates the inlining version of the composed activity model for the order processing example as shown in Figure 3.2. Four join points are matched according to the pointcut specification in Figure 3.4 and augmented with the `LogThisJoinPoint` action. The aspect-activity weaving process conforms to the one developed for aspect-oriented state machine in the current Motorola aspect weaver.

For more details about the Motorola aspect weaver, please refer to [51, 192].

### 3.3 Compositions in Aspect-Oriented Activity Modeling

One of the fundamental issues in AOSD is the potential conflicts that may occur in the presence of interactions among aspects (i.e., when multiple aspectual behaviors are superimposed at the same join point, different composition orders may reveal various inconsistency problems). In such circumstances, the aspects interfere with each other in a potentially undesired manner, either due to the side-effects caused by the aspects (e.g., several aspects change the state of the base system simultaneously) or due to the requirements enforced by the system (e.g., the logging aspect may be applied only in the presence of the encryption aspect because some particular systems require

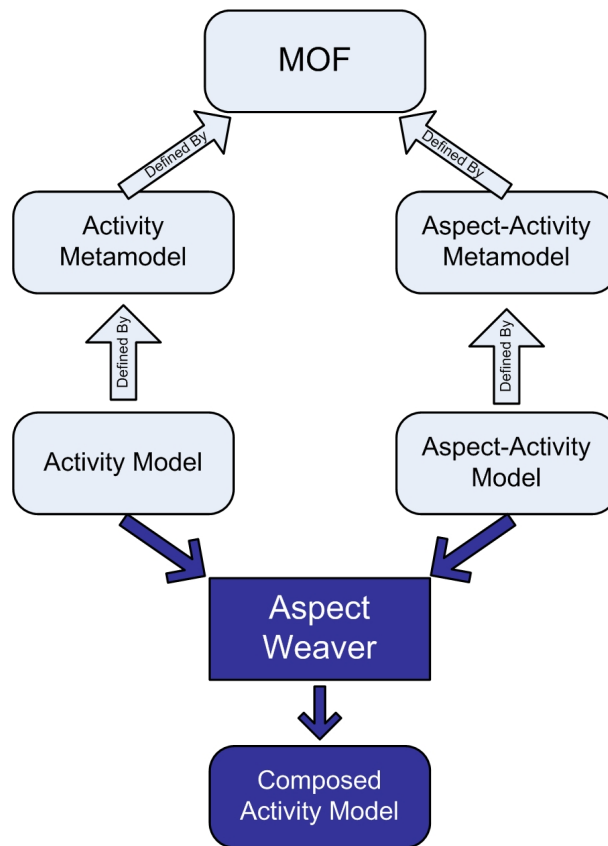


Figure 3.5: Aspect Weaving on Activity Models

all logged data to be encrypted). Durr et al. [57] use the term “semantic conflict” to designate the aspect interference problem. In their example, a user credit check aspect and a playlist creation aspect both select a particular method call as the join point. Without specifying any precedence relationships between aspects, unexpected behavior would occur. In [137], four requirements (i.e., monitoring salaries, checking salary raises, database persistence and XML representation persistence) were imposed on a personnel management system. Each of the requirements was realized by an aspect, which would be superimposed on the same join point and may affect each

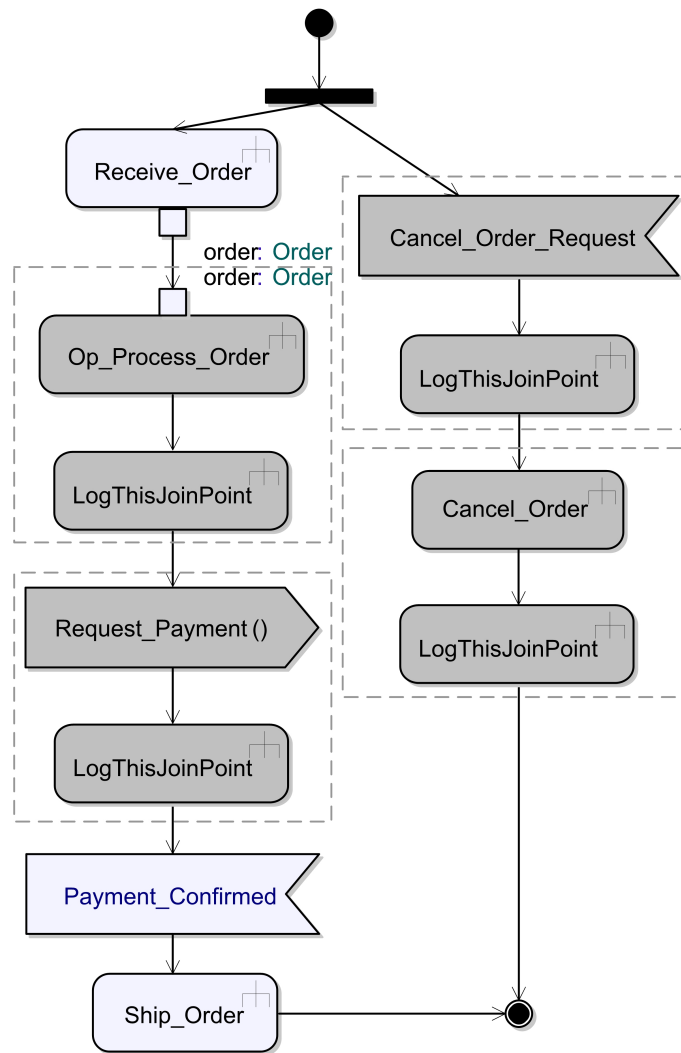


Figure 3.6: Augmented Order Processing Activity Model with the Tracing Aspect

others' functionality. In addition to these two cases, a number of aspect interference examples have been described in [110, 114, 145, 162].

Several techniques have been proposed and developed to resolve or reduce aspect interference. For the most light-weight approach, the execution orders between aspects are governed by declaring precedence relationships, such as in AspectJ [108] and some aspect modeling approaches [149]. Some other approaches extend the simple precedence declaration and introduce more complex dependencies and ordering relationships between aspects, such as [111, 137]. Aspect interactions can be identified through static analysis on the crosscutting concerns and the base module [55, 171]. Advanced approaches require extra behavior specifications from the user for each advice [57, 145] or each aspect [114, 162]. The conflict between aspect semantics can then be detected automatically based on the specified contracts.

The problem of aspect interference is intrinsic to every AOSD technique (i.e., interference is at the essence of aspects due to the focus of multiple concerns that may crosscut at common locations). As an initial step towards resolving the interference issue in AOAM, we adopt a light-weight approach following and extending the AspectJ [108] notation. Aspect precedence can be specified explicitly at the modeling level in order to reduce the occurrence of aspect interference in AOAM.

According to the distinct definition of the aspect constructs introduced in the last subsection, two kinds of interference problems may occur during the weaving process: advice-to-advice and aspect-to-aspect. (Note: pointcut-to-pointcut interference is not considered in this thesis because pointcuts do not own any behavior on their own. Research on the pointcut-to-pointcut interference is considered as part of future work.) This section introduces precedence declarations on advice and aspects. The explicitly specified precedence constraints reduce undesired interference at the shared join point and will be passed to the underlying model composition mechanism to compute a proper weaving order.



Three distinct categories of composition mechanisms (i.e., pointcut composition, advice composition and aspect composition) have been implemented in AOAM for reducing aspect interference as well as facilitating aspect reuse to a large extent. In the following sub-sections, each mechanism will be illustrated in detail. Comparisons to the corresponding AspectJ notation are also provided, with the intent to demonstrate the advantages of the AOAM composition approach.

### 3.3.1 *Pointcut Composition*

In AOAM, the pointcut composition semantics strictly follow the AspectJ semantics. Pointcuts can be composed with Boolean operators to build other pointcuts. The Boolean expression is specified in a separate text box within the composite pointcut diagram. (Note: Currently, a primitive and intuitive way is adopted to represent the pointcut compositions. However, the composition syntax can also be specified in a graphical notation through the extensions of the pointcut metamodel definition.) The supported Boolean operators are: **AND** (&&), **OR** (||) and **NOT** (!), indicating the intersection, union and negation of the set of the join point selections, respectively.

Furthermore, AOAM also supports **cflow**, **cflowbelow** and **within** pointcut designators (see the AspectJ interpretation of these designators in Section 2.6.1). **cflow(Pointcut)** picks out each join point in the control flow of the join points **P** picked out by **Pointcut**, including **P** itself. **cflowbelow(Pointcut)** is similar to **cflow(Pointcut)** except that the matched join points do not include **P** itself. **within(Scope)** picks out every join point within a specific scope (e.g., class or package). In this sense, **within** is usually combined with other pointcut designators, for the sake of further filtering join points.

As illustrated in Figure 3.7, **CompositePointcut** is constructed by two sub-pointcuts, **\*\_Order** and **Cancel\_\***, which means that **CompositePointcut** will pick out join points matched by **\*\_Order** (i.e., all the operation actions ending with **\_Order**) that are not in the control flow of any join point picked out by **Cancel\_\*** (i.e., any accept event

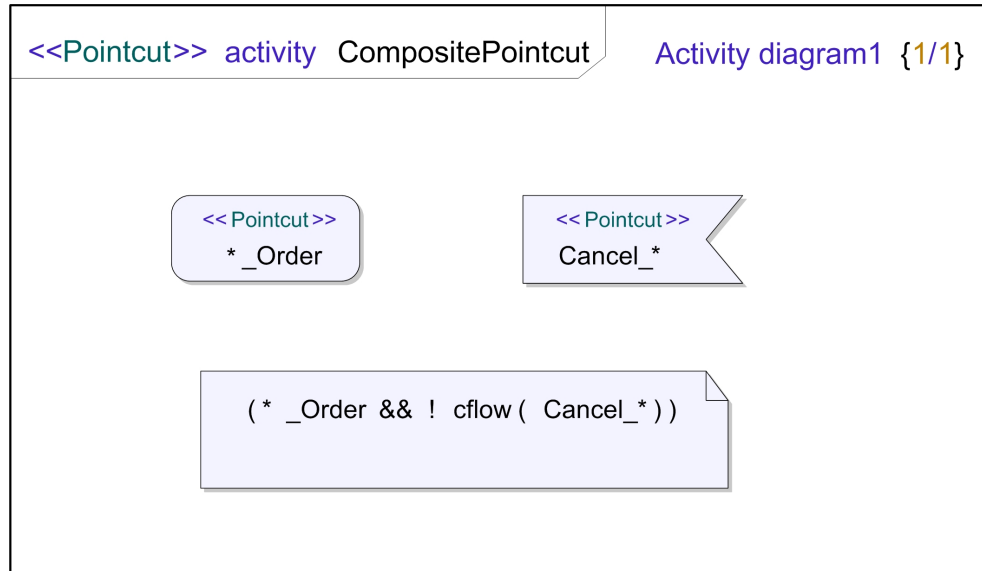


Figure 3.7: Pointcut Composition in AOAM

action starting with `Cancel_`). After applying pointcut matching to the order processing model in Figure 3.2, the resulting join points will be three operation actions: `Receive_Order`, `Op_Process_Order` and `Ship_Order`.

One advantage of our approach over AspectJ is that a pointcut can be directly referenced (e.g., through dragging and dropping in the model view) and reused in any other aspect. AspectJ, however, only allows the abstract aspect (see Section 2.6.1 for the explanation of abstract aspect) to be reused by inheritance. Concrete aspects extending an abstract aspect must provide concrete definitions of abstract pointcuts. Reusing pointcuts among multiple aspects is not possible in AspectJ.

### 3.3.2 Advice Composition

Advice composition binds and executes the advice instances that perform at the same join point in a certain appropriate order. In AOAM, advice are ordered based on the precedence relationships that are specified by the aspect developers. The `<<follows>>` relationship has been implemented between advice. As shown in Figure 3.8, `Advice2` follows `Advice1`, which means that at a particular join point, `Advice1` has

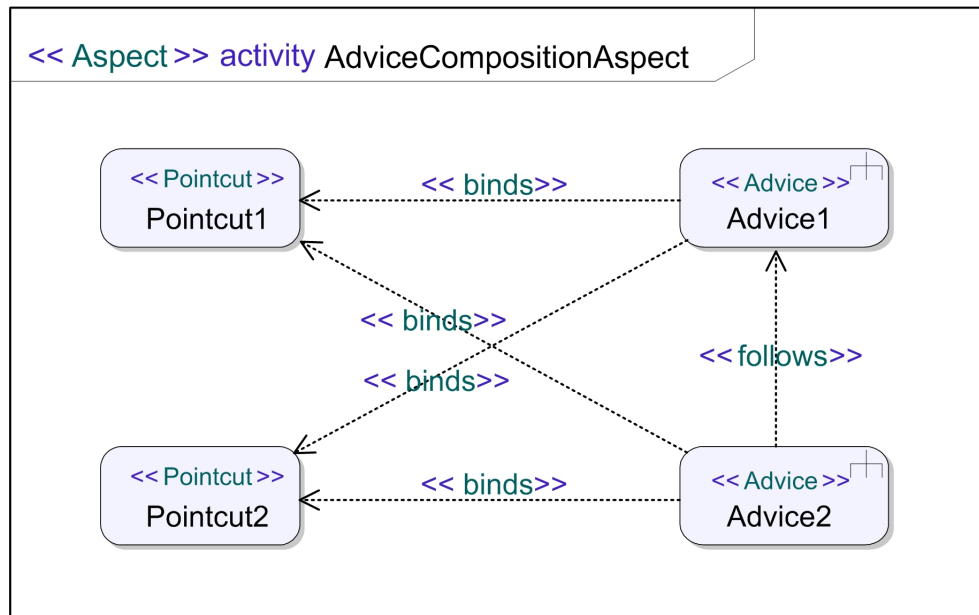


Figure 3.8: Advice Composition in AOAM

precedence over **Advice2**, and the instances of **Advice2** will be executed closer to the join point than the instances of **Advice1** (i.e., the *before* actions in **Advice1** instances will always be executed prior to the *before* actions in **Advice2** instances, and the *after* actions will be carried out in the opposite order). In the absence of an ordering constraint, the execution order of the corresponding advice instances is undefined and controlled by the underlying AOAM weaver.

Ordering relationships specify a partial order upon the execution of a set of advice instances. In order to obtain a composition and execution order, a topological sort is performed on the advice. Circular dependencies among the advice are detected when their corresponding pointcuts match to the same join point. Under such a circumstance, the AOAM weaver will abort with an error message, indicating the problematic advice involved in the circularity.

When executing an advice instance, the call to proceed will be redirected to the invocation of the advice instance with the next precedence, or the computation under

the join point if there is no further advice instance. In the case of Figure 3.8, suppose `Pointcut1` and `Pointcut2` both match a single join point (in the following, `PCT` replaces `Pointcut` and `ADV` replaces `Advice`). The advice instantiation order at this join point could be:

PCT1-ADV1, PCT2-ADV1, PCT1-ADV2, PCT2-ADV2

If both `Advice1` and `Advice2` contain a `proceed` action, the execution order of the woven model at this join point would be as follows:

1. Before actions in the advice instance: PCT1-ADV1
2. Before actions in the advice instance: PCT2-ADV1
3. Before actions in the advice instance: PCT1-ADV2
4. Before actions in the advice instance: PCT2-ADV2
5. Original join point action
6. After actions in the advice instance: PCT2-ADV2
7. After actions in the advice instance: PCT1-ADV2
8. After actions in the advice instance: PCT2-ADV1
9. After actions in the advice instance: PCT1-ADV1

By comparing our advice composition mechanism with AspectJ, we believe that our approach offers two advantages:

1. In AOAM, the concepts of pointcuts and advice are loosely decoupled. An advice is named, which allows it to be associated with not just one, but multiple pointcuts as long as they share compatible interfaces. Therefore, an advice can be directly referenced (e.g., through dragging and dropping in the model view) and reused in different aspects in a compositional way. In AspectJ, advice is

unnamed and can only be bound to one particular pointcut. The tight coupling between pointcuts and advice makes aspects difficult to reuse. The only way to reuse advice in AspectJ is by means of inheritance, which is known to be more brittle and less flexible than the composition-based solution [74].

2. In AspectJ, the precedence of advice relies completely on their textual locations in an aspect file. The underlying interpretation rules, as stated in the AspectJ Programmers Guide [3], declare that, “for two advice within a single aspect, if either is after advice, then the one that appears later in the aspect has precedence over the one that appears earlier; otherwise, the one that appears earlier in the aspect has precedence over the one that appears later.” These rules have limitations and cannot express all composition orders, as pointed out in [126]. Our approach resolves the above problems because there is only one advice type (i.e., around advice) in AOAM, which decreases the complexity of handling three different types (i.e., before, after and around) of advice as in AspectJ. Furthermore, by declaring the advice precedence explicitly, the interference between the advice is reduced.

### 3.3.3 Aspect Composition

Aspect composition is achieved through a deployment diagram (Figure 3.9), which is used to bind aspects to the base models, with the precedence relationships declared. Aspects can be bound to multiple base models through the stereotype `<<crosscuts>>` (e.g., `ExceptionAspect` and `TracingAspect` are both applied to the base activity). Aspects can also be deployed to other aspects or advice. In the absence of the `<<crosscuts>>` relationship, aspects will be applied to all the base activity models in the current active project (e.g., `LoggingAspect` and `EncryptionAspect`). The precedence relationships between aspects can be `<<follows>>`, `<<hidden_by>>` and `<<dependent_on>>`. The remainder of this section explains these three concepts in detail based on

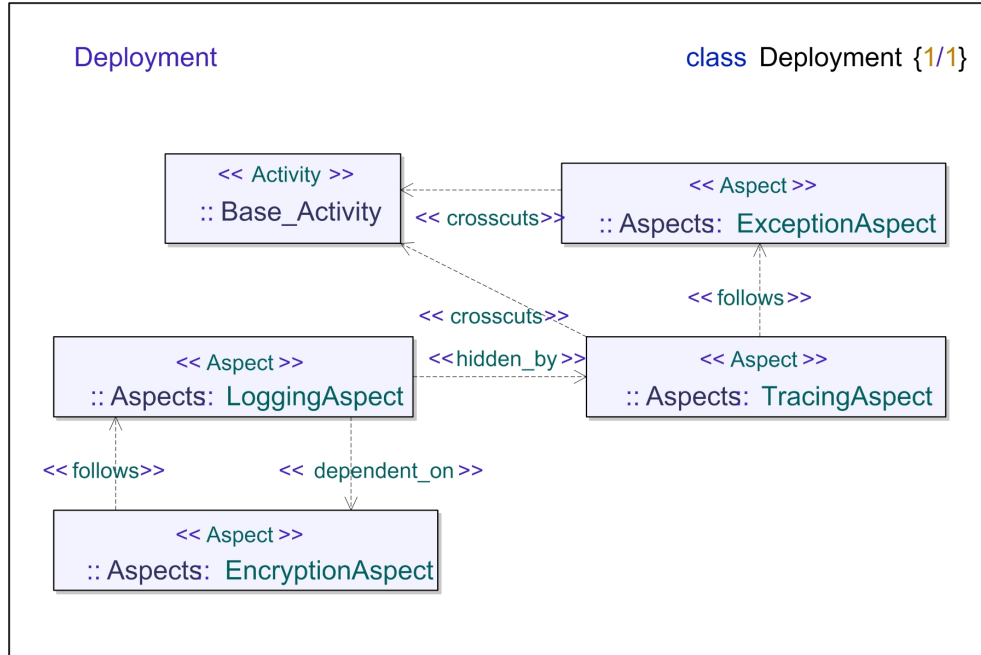


Figure 3.9: Aspect Composition in AOAM

the example provided in Figure 3.9. The symbols that are used in the relationship expressions between aspects are denoted in Table 3.2.

1. **TracingAspect follows ExceptionAspect:** TracingAspect (as defined previously in Figure 3.4 of Section 3.2.1) is used to print out signatures and parameter values of some particular action join points. ExceptionAspect encapsulates exception detection and handling mechanisms to certain actions. The `<<follows>>` relationship between these two aspects means that at a single join point, ExceptionAspect has higher precedence than TracingAspect. Therefore, all of the advice in the ExceptionAspect have higher precedence than the ones in the TracingAspect. In other words, the advice instantiations from TracingAspect will be executed closer to the join point than the ones instantiated from ExceptionAspect. The execution order of the woven model at this shared join point would be as follows:

Table 3.2: Symbols for Denoting Aspect Relationships

Symbol	Meaning
$<-$	(aspect) applied to
$=>$	implies
$\neg$	in absence
$\&\&$	and
$  $	or
$-$	subtract

- Before advice instances in **ExceptionAspect**
- Before advice instances in **TracingAspect**
- Join point action
- After advice instances in **TracingAspect**
- After advice instances in **ExceptionAspect**

2. **LoggingAspect** is **hidden** by **TracingAspect**: **LoggingAspect** stores the attributes and data of a particular interest into a database whenever they are used or modified. However, the system may not always want to log everything, such as those data that are being traced by **TracingAspect**. The `<<hidden_by>>` relationship inactivates **LoggingAspect** whenever it matches the same join point as **TracingAspect**. The correlation between **TracingAspect** and **LoggingAspect** can be described using the following expression:

$$\text{TracingAspect} => \neg \text{LoggingAspect}$$

This notation means that the presence of **TracingAspect** implies the absence of **LoggingAspect**. For each pointcut denoted as `PointcutLoggingAspect` in **LoggingAspect**, the actual corresponding pointcut exposed by this particular deployment strategy is:

$$\text{Pointcut}_{\text{LoggingAspect}}' = \text{Pointcut}_{\text{LoggingAspect}} \ \&\& \ \neg \text{Pointcut}_{\text{TracingAspect}}$$

3. **LoggingAspect is dependent on EncryptionAspect:** The `<<dependent_on>>` relationship enforces the **LoggingAspect** to be applied only in the presence of the **EncryptionAspect**. This is enforced because some systems may require all logged data to be encrypted (i.e., **LoggingAspect** will only be applied at the join points when **EncryptionAspect** and **LoggingAspect** both match. Therefore, **LoggingAspect** will be disabled at the other join points where it matches apart from **EncryptionAspect**). The relationship between **LoggingAspect** and **EncryptionAspect** is denoted as follows:

$$\text{LoggingAspect} \Rightarrow \text{EncryptionAspect}$$

This means that the presence of **LoggingAspect** implies that **EncryptionAspect** has to be present at the same join point as well. Therefore, under this particular condition, the actual pointcut exposed by **LoggingAspect** is:

$$\text{Pointcut}_{\text{LoggingAspect}}' = \text{Pointcut}_{\text{LoggingAspect}} \ \&\& \ \text{Pointcut}_{\text{EncryptionAspect}}$$

The resulting join point selection set for **LoggingAspect** is indicated by the striped area in Figure 3.10. In addition, as illustrated in Figure 3.9, **EncryptionAspect** also `<<follows>>` **LoggingAspect**, which forces encryption actions to be executed closer to the join point than logging procedures.

In order to detect and collect all of the join points by traversing the whole model in linear time, the base model is divided into several exclusive sets from the deployment diagram. The derived aspect composition order for Figure 3.9 is:



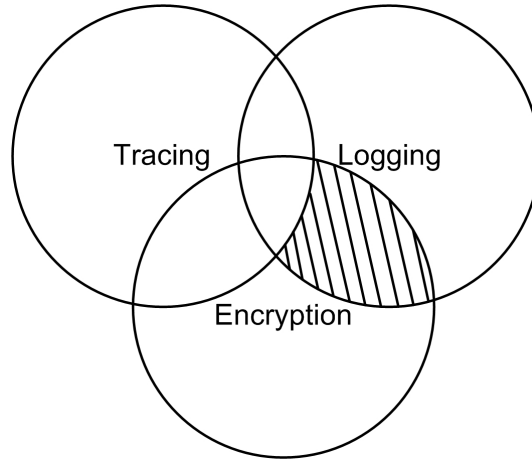


Figure 3.10: Results of Join Point Selection for Aspect Deployment Strategy in Figure 3.9

```

Base_Activity    <-    ExceptionAspect,
                    TracingAspect,
                    LoggingAspect',
                    EncryptionAspect
ALL - Base_Activity <- LoggingAspect',
                    EncryptionAspect

```

Within the scope of `Base_Activity`, `ExceptionAspect` will be applied first, followed by `TracingAspect`, `LoggingAspect` (with the new composite pointcut) and `EncryptionAspect`. For all of the other models that are not within the scope of `Base_Activity` (denoted by subtracting `Base_Activity` from `ALL` with a minus sign “-”), only `LoggingAspect` and `EncryptionAspect` will be applied. Circular and conflict relationships among the aspects will be detected and reported when they are superimposed at the same join point.

The advantages of our approach over AspectJ are:

1. In AOAM, aspects are explicitly deployed by means of a deployment diagram. Aspects can be bound to different fragments of the base models; in AspectJ, aspects are applied everywhere and the only way to apply an aspect to a certain

scope is to restrict every pointcut specification by using the “within” keyword, which are bound lexically to method or class names. This makes pointcuts and aspects less reusable.

2. The semantics of the `<<follows>>` relationship in AOAM correspond to the “declare precedence” form in AspectJ. In addition, AOAM is able to handle two more dependency relationships between aspects (i.e., `<<hidden.by>>` and `<<dependent_on>>`), which further restrict application of an aspect at the same join point. In AspectJ, however, when a pointcut matches a certain join point, the corresponding aspect is always applied.

This section introduced composition mechanisms implemented in AOAM in three distinct categories: pointcut, advice and aspect. From our experience, we have found that by integrating compositions at different granularity levels, the aspect expressiveness and reusability can be extended to a larger extent. By declaring precedence relationships between crosscutting concerns, aspect interference can be reduced and controlled by the aspect developers.

### 3.4 Case Study

To illustrate the presented approach, a case study is provided in this section that applies a timeout handler aspect to a real-world network fault management system using aspect-oriented activity models.

#### 3.4.1 Background

The Intelligent Network Fault Management (INFM) system [123] is being developed within Motorola for providing solutions to manage faults in a CDMA (Code Division Multiple Access) cellular network. One of the most important features of a fault management system is alarm correlation, which provides functionalities to filter

out informational alarms, report meaningful alarms that are regarded as actionable or as requiring operator attention and provide assistance in troubleshooting. Network operators rely on the alarm correlation feature to reduce the number of alarms to a limited number that could be handled within the required time constraints. The significant reduction, usually greater than 80% on average [124], is achieved by correlating the alarms using patterns and hidden correlations discovered by machine learning algorithms.

INFM carries out the alarm correlation by applying frequent pattern discovery algorithms, which use certain parameters to control how candidate patterns are constructed from the learning data (i.e., alarm instances). When the number of alarm instances is fairly large (e.g., thousands of alarms), the time complexity of the algorithm increases dramatically (e.g., the computation might take from tens of minutes to a couple of hours depending on the specific algorithms that are chosen), which causes a violation of real-time constraints and fails to provide a prompt correlation operation. This motivates the addition of a timeout handling capability to resolve such failures and maintain the healthy state of the fault management system.

Figure 3.11 illustrates a fragment of the simplified activity model for specifying the flow of the alarm correlation process logic in the INFM system. The control flow starts with configuring the data source for streaming event data from either a database server or an FTP site. After the connection is established, the process then initiates the parameters for the pattern discovery algorithm. The algorithm execution returns a list of discovered patterns among the alarm instances, such as “Alarm 1 is related to Alarm 2 and Alarm 3 with a certain correlation value within a certain time interval.” After the algorithm execution is completed, a special operation will be invoked to process the patterns that are generated (e.g., filter out those patterns whose correlation values are under a certain threshold).

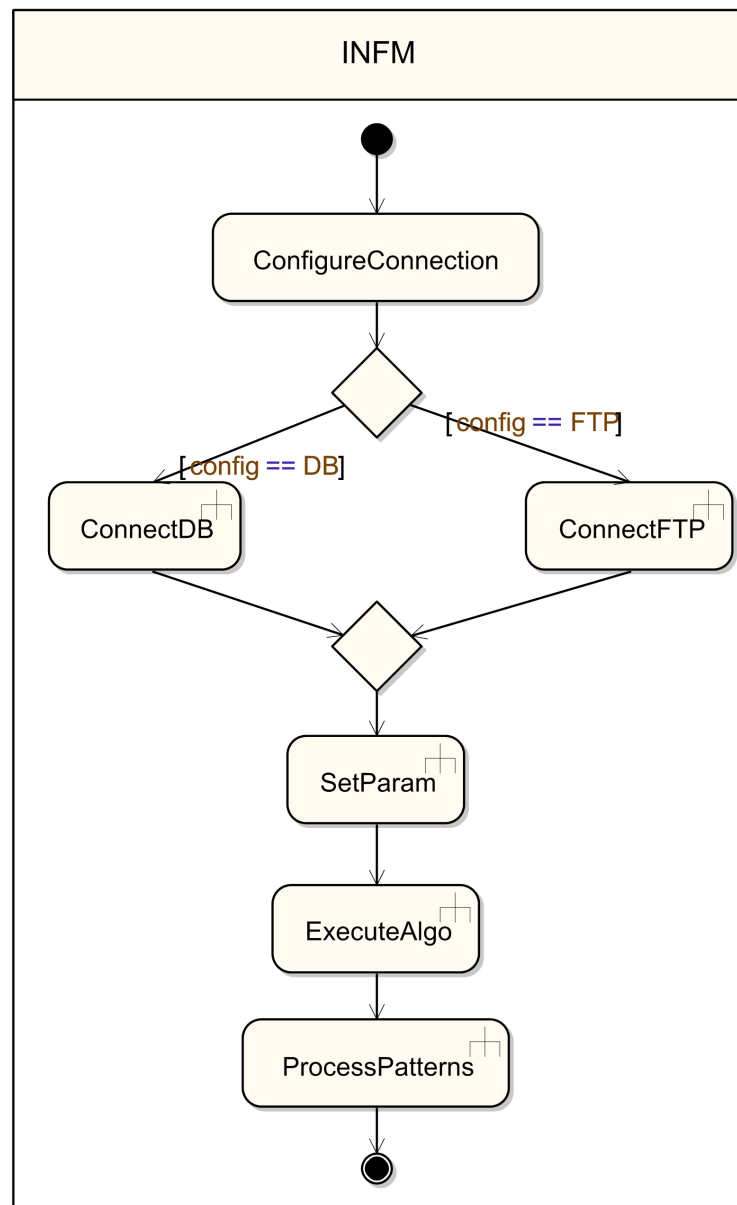


Figure 3.11: Extracted INFM Alarm Correlation Activity Base Model

### 3.4.2 Modeling Timeout Handler Activity Aspect

Timeout is among the most common software failures that can occur in almost every operation or service invocation. The timing failure is usually associated with certain time constraints, which can be a real-time constraint or a relative deadline with respect to certain events. For example, a time limit can be imposed on the alarm correlation process in a fault management system, as opposed to the relative deadline which states that “the alarm correlation must be completed before the next batch of alarms is received.”

Figure 3.12 shows an aspect-activity model for managing the timeout failure for

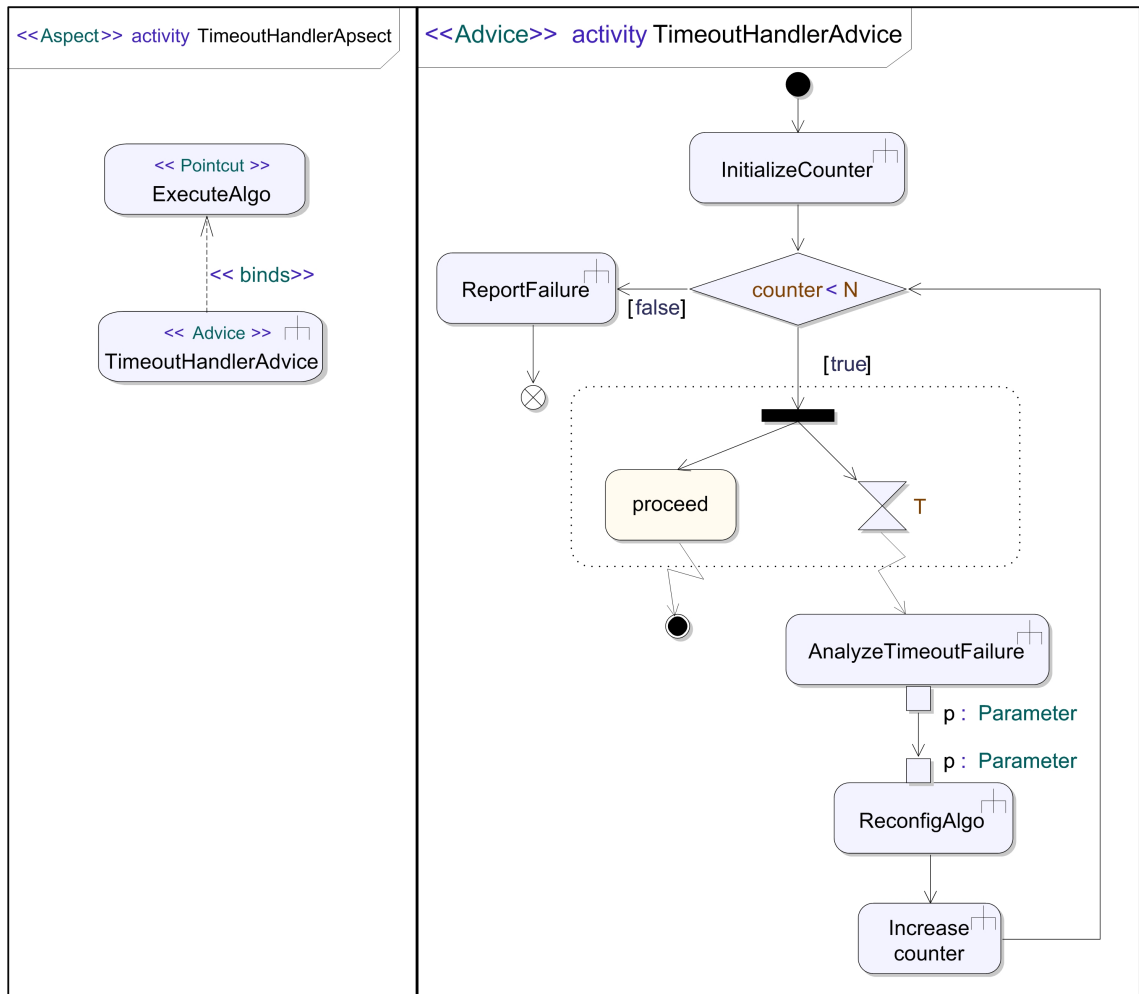


Figure 3.12: A Timeout Handler Aspect

the INFM alarm correlation system. The **proceed** action refers to an operation that has sensitive timing concerns and needs to be analyzed upon the timeout failure. The aspect model intercepts and wraps this action with a sequence of failure management activities. The aspect process first initiates the counter for the allowed number of iterations. If the counter already exceeds the allowed number of iterations, it means that the failure cannot be resolved and has to be reported. Thus, the whole application must be aborted. Otherwise, the process will start a timer with a value  $T$ , synchronously with the execution of the **proceed** action. The **proceed** action and the timer are surrounded by an interruptible activity region (denoted as a dashed rectangle with rounded corners), representing that whenever the flow leaves the region via interrupting edges, all of the activities in the region will be terminated. Specifically, if the **proceed** action completes execution successfully before it runs out of time, the control flow will return to the base process (via a bull's eye symbol) and continue with the next activity that follows the **proceed** action. Otherwise, the **proceed** process will be shut down properly and a timeout failure will be captured and passed to the failure analyzer and mitigator, which are responsible for determining the failure risk and calculating the corresponding mitigation strategies for reconfiguring algorithm parameters. The control loop for handling the timeout failure is thus realized by re-running the algorithm with the new parameter values.

### 3.4.3 *Deploying the Timeout Handler Aspect on a Base Model*

The timeout handler aspect is deployed to the base models of the INFM alarm correlation activity, particularly the **ExecuteAlgo** action. The base models and the aspect models are then integrated through the underlying AOAM weaver. The woven model resulting from the inlining mode of weaving is shown in Figure 3.13. The **proceed** action in the aspect specification is replaced by the **ExecuteAlgo** action. The initial and return symbol of the aspect models are connected to the pre- and post-

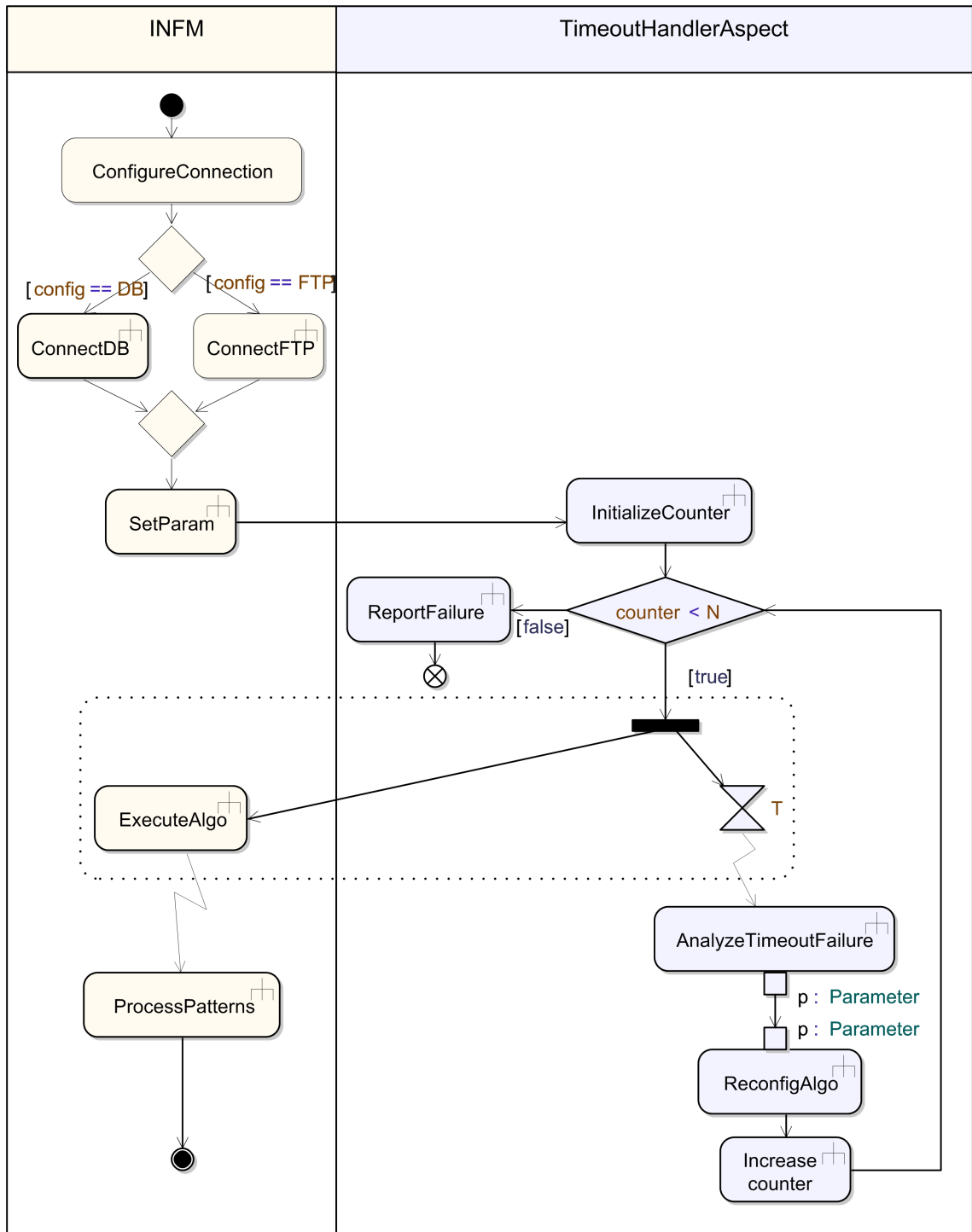


Figure 3.13: Integrating the Timeout Handler Aspect with the INFM Alarm Correlation Base Model

flow of the `ExecuteAlgo` action, respectively. By adopting the aspect-oriented approach to specifying activities, the aspect models (usually non-functional activities) are defined independently from the base functionality. From our experience in applying the AOAM weaver to this case study, we have found that such kind of separation of concerns can help improve the reusability, changeability, and maintainability of the system. Reusability is enhanced in AOAM by enabling composition at three different levels (i.e., pointcut composition, advice composition and aspect composition). The separation of concerns allows better modularity such that each evolutionary change can be localized in an individual aspect model, which results in a more maintainable system.

### 3.5 Summary

In this chapter, an aspect-oriented approach has been presented to support separation of crosscutting concerns in activity modeling. Aspect-specific constructs have been introduced as an extension to the activity models using UML profiles and stereotypes. The AOAM weaver has been implemented in Telelogic Tau [13] (a UML-based MDE environment), as an extension of the Motorola state machine weaver [50]. Other work related to the AOAM approach will be presented later in Chapter 5. In Chapter 4, we will show how this technique can be leveraged to facilitate activity-based legacy system evolution.

The contents presented in this chapter have been published in [191, 192, 198], among which [191] and [192] covered the underlying aspect composition mechanism offered by the Motorola AOM Weaver. The AOAM approach, along with the INFM case study, was presented in [198].



## CHAPTER 4

### LEGACY EVOLUTION THROUGH MODEL-DRIVEN ASPECT ADAPTATION

Legacy systems affect many aspects of daily life (e.g., software to control commercial transaction systems [18], avionics systems [160], network management systems [172], or even healthcare systems [141]). As noted in [181], “any application system that is functioning in a production environment within an enterprise can be considered as a legacy system.” In the context of this dissertation, legacy software systems particularly refer to the source code implementation that is hand-crafted rather than derived or generated from a high-level software abstraction. Some legacy software systems have poor or very limited requirements or design documentation, if there is any at all. Some have comprehensive documentation, but are hard to map requirement/design concepts into the code, because causal relationships are missing between the high-level requirement/design and the low-level source code. These software systems are notoriously difficult to maintain and evolve in terms of low-level source code, because it is often the case that even a small change in the requirements of the application domain might necessitate drastic changes in large portions of the source code. This is fundamentally different than the systems that are completely created by the model-driven software development process, in which the source code is systematically generated from the high-level requirements and design models.

The objective of Model-Driven Aspect Adaptation (MDAA) is the evolution of the source code of a legacy software system from the domain properties described in high-level requirement and design models. The domain evolution changes, which

usually represent crosscutting concerns that would affect multiple system modules, are specified in aspect models that are translated into the the aspects that are codified at the programming language level. These code aspects are in turn composed with the original legacy source using an underlying aspect weaver or program transformation engine. A key feature of the approach is the ability to accommodate evolutionary changes in a manner that does not require manual instrumentation of the actual source. An essential characteristic of the model-driven process is the existence of a *causal connection* between the models and the underlying source representation. That is, as model changes are made to certain properties of a system, those changes must have a corresponding effect at the implementation level. This process is called two-level weaving [78, 81], which allows aspects at the modeling level to drive widespread adaptations of the representative source code.

The rest of this chapter is organized as follows. Section 4.1 introduces a framework that supports legacy source evolution through MDAA. Sections 4.2 and 4.3 present two instantiations of the framework for realizing MDAA in terms of domain-specific models and UML activity models. Section 4.4 introduces an approach that supports automated identification of aspects in models, called aspect model mining.

## 4.1 The Model-Driven Aspect Adaptation Framework

Figure 4.1 depicts the general framework for realizing MDAA, which enables a round-trip engineering of legacy software systems from the source code to the higher level models, on which evolving domain requirements can be applied and transformed back to the source code. The MDAA framework is a generic representation of a collection of methods, components, modeling artifacts and tools (as highlighted in the gray boxes in Figure 4.1) that can be used to perform consistent evolution of both models and source code. The MDAA reengineering process starts with a model extractor analyzing legacy code and extracting high-level models that are defined based on a

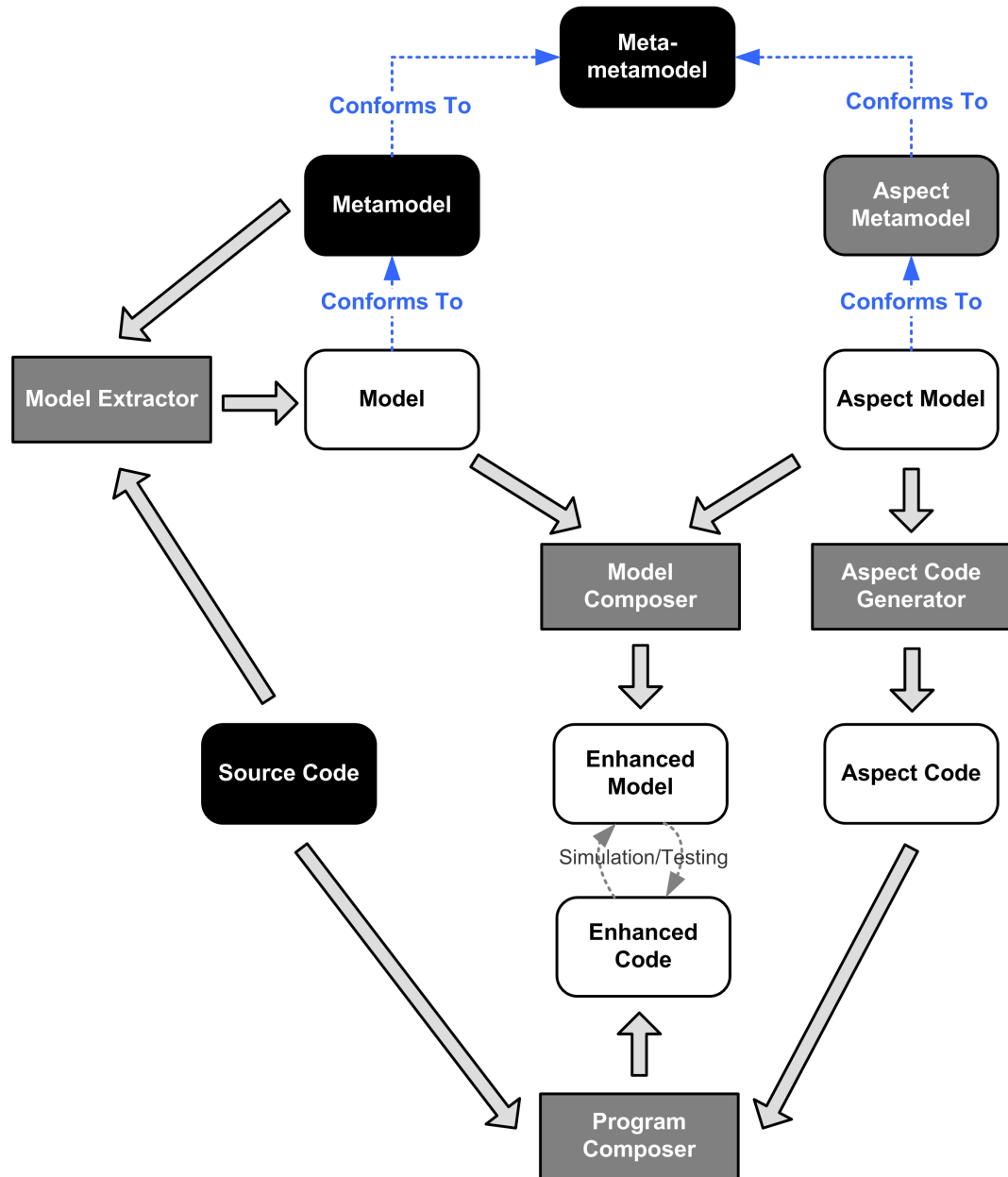


Figure 4.1: Model-Driven Aspect Adaptation Framework

specific metamodel [63]. This can be accomplished manually or through the use of reverse engineering techniques (e.g., MoDisco [33] proposed a generic and extensible metamodel-driven approach to model discovery). After the models that represent a particular aspect of the system are constructed, the new domain requirements that are specified in separate aspect models can be integrated with the extracted base model through model composition techniques (e.g., an AOM weaver). The aspect models are then translated into the aspect code through customized transformation generators that conform to the corresponding aspect metamodel. Finally, the generated aspect code serves as input to the underlying program composer (e.g., an AOP weaver), together with the legacy source, to produce the enhanced system that incarnates the new domain requirement features.

The legacy system evolution often requires the leveraging of knowledge and expertise in different engineering disciplines. The successful adoption of the MDAA approach necessitates the collaboration of model extraction engineers, aspect developers, as well as domain experts. Figure 4.2 visualizes these different roles that are involved in the MDAA approach. The solid black lines indicate the order of the tasks to be conducted. The dotted lines denote the dependencies between different tasks. In a typical MDAA process, the following three steps are involved:

1. During the preprocessing step, the model extraction engineers identify the base metamodel for the legacy domain and extract the base models that conform to this metamodel definition. Model extraction engineers must possess both programming skills (in the sense that they need to be able to understand and analyze the source code) and domain-specific knowledge (in the sense that they need to be able to identify the suitable metamodel).
2. Aspect developers are experienced in programming, especially in the area of developing both AOM and AOP technologies. For each new requirement that is introduced to the legacy system, the aspect developer creates an aspect meta-

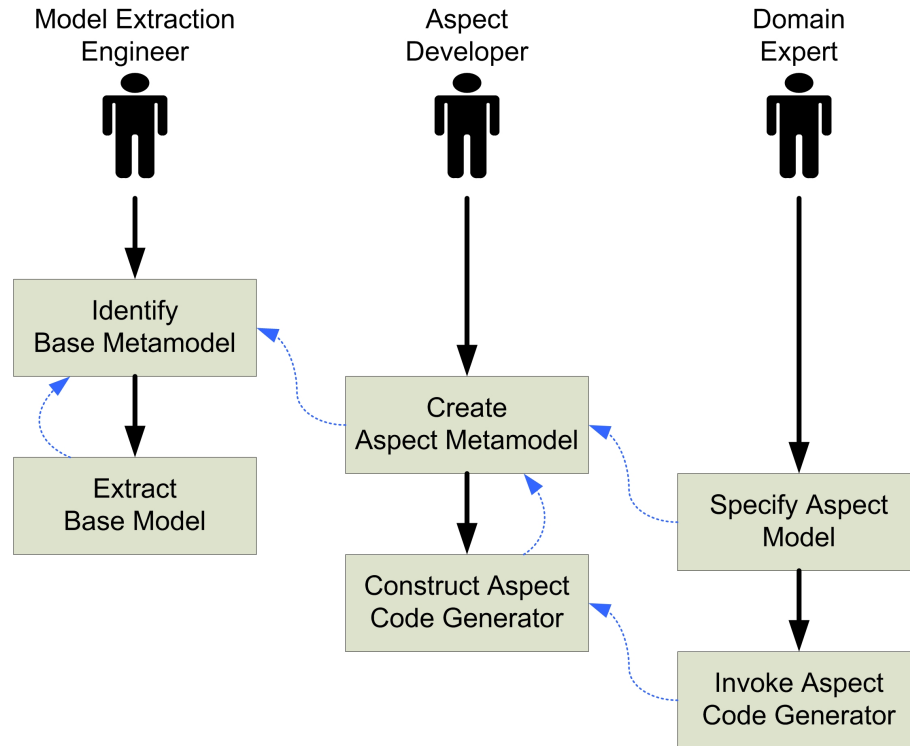


Figure 4.2: Roles in the MDAA Process

model that captures the requirement constructs and relationships. The aspect metamodel refers to the base metamodel that is produced during the preprocessing step. Each aspect metamodel is associated with one aspect code generator (usually written in a general-purpose language (GPL) like C++ or Java).

3. Domain experts are individuals who are both knowledgeable and experienced with application domains. However, they usually do not possess any specific programming skills. During the system evolution, the domain experts make changes to the models using the customized evolutionary requirement constructs defined in the aspect metamodel. After the changes are completely specified, they then invoke the aspect code generator, which results in the generation of corresponding aspect source that is used by the underlying program composer to perform legacy evolution.

Among various modeling techniques, domain-specific modeling (DSM) [82] and UML are the most popular ones to be adopted widely for specifying domain requirements and design. Sections 4.2 and 4.3 will present two instantiations of the MDAA framework in terms of DSM and UML (particularly, UML activity diagrams). The case studies illustrate the approaches and support a context for comparison.

The rest of this section will cover each of the components in the MDAA framework in detail, i.e., model extractor, aspect metamodel, model composer, aspect code generator, and program composer.

#### 4.1.1 *Model Extractor*

In order for a legacy system to benefit from the MDAA process, the initial step is to bring the legacy system into the modeling space, i.e., to construct the models that represent the base legacy system domain at a higher level of abstraction. A technique called metamodel-driven reverse engineering [65] can be adopted to realize model extraction from legacy source according to the corresponding metamodel definition.

Figure 4.3 depicts the general principle for conducting metamodel-driven reverse engineering. *Metamodel-driven* means that the whole model extraction process is guided by a metamodel. Thus, the very first step is to define the metamodel, which captures the essential concepts (i.e., structural and/or behavioral information) to represent the legacy system. Then, extractors need to be developed to extract necessary information from the system in order to build the model that conforms to the predefined metamodel.

Due to the highly heterogeneous nature (e.g., various techniques, platforms, and languages that are used) of a legacy system, different legacy systems might require distinct technologies to extract models. Even a single legacy system can be mapped to different models in various ways (e.g., one model can represent the static structural information of the system and another model can represent the dynamic behavioral

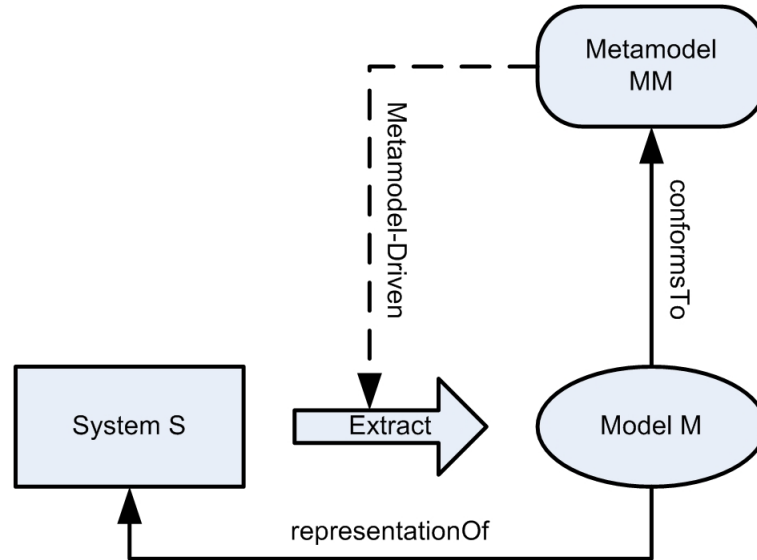


Figure 4.3: Metamodel-Driven Model Extraction (Adapted from [33])

aspects). Because of such diversity of the legacy system and the models, most of the model extractors have to be created manually and the extraction is an iterative process, though some can be semi-automated through the use of the existing information retrieval tools (e.g., a sequence of shell scripts including commands like “find” and “grep” can be treated as a specific extractor for generating a model of the file system that conforms to the metamodel for the UNIX system [34]). In Sections 4.2 and 4.3, two distinct approaches are introduced to extract structural and behavioral models from different legacy source.

#### 4.1.2 Aspect Metamodel

In this section, ideas and approaches from AOSD [2] are applied to accommodate legacy source evolution from high-level models. Evolutionary changes are represented by aspect models that conform to a specific aspect metamodel. In essence, an aspect metamodel is a modeling language with specific syntax and semantics to support aspect specification at the modeling level.

There are two different approaches for constructing an aspect metamodel. The

light-weight extension approach takes advantage of the UML extension mechanisms (e.g., stereotypes, tagged values and constraints) to refine the UML metamodel to support aspect modeling concepts. AOAM, as introduced in Chapter 3, falls into this category, in which the AspectJ concepts such as aspects, pointcuts, and advice are modeled as specialized UML activity constructs.

The heavyweight approach advocates that aspects are first-class entities in the models, which necessitates a self-contained aspect metamodel that is not tied to the base metamodel definition. This approach has more expressive power than the lightweight approach, but is more complex to implement.

Regardless of the disparate mechanisms of these two approaches, their aspect metamodels always share the same meta-metamodel with the base metamodel, which enables further development of a model weaver. Sections 4.2 and 4.3 will present the heavyweight and lightweight approaches to define an aspect metamodel, respectively.

#### *4.1.3 Model Composer*

Model composition (as shown in Figure 4.4) involves merging two or more models to obtain a single integrated model according to certain composition relations. A composition relation denotes how models are to be composed by identifying overlapping constructs in different models and specifying how models should be integrated. Such relations are either embedded in the models themselves (e.g., the pointcut designators in aspect models) or captured in a separate relation model (e.g., a weaving model [61] that denotes the mapping links between different models).

In the context of MDAA, an AOM weaver serves as a model composer, which binds the aspect model to the base model and produces a composed model that corresponds to an enhanced version of the represented system. In Section 4.2, C-SAW [77] along with a specialized metamodel composer is leveraged to realize aspect weaving for Domain-Specific Models (DSMs). Aspect-Oriented Activity Model (AOAM), yet



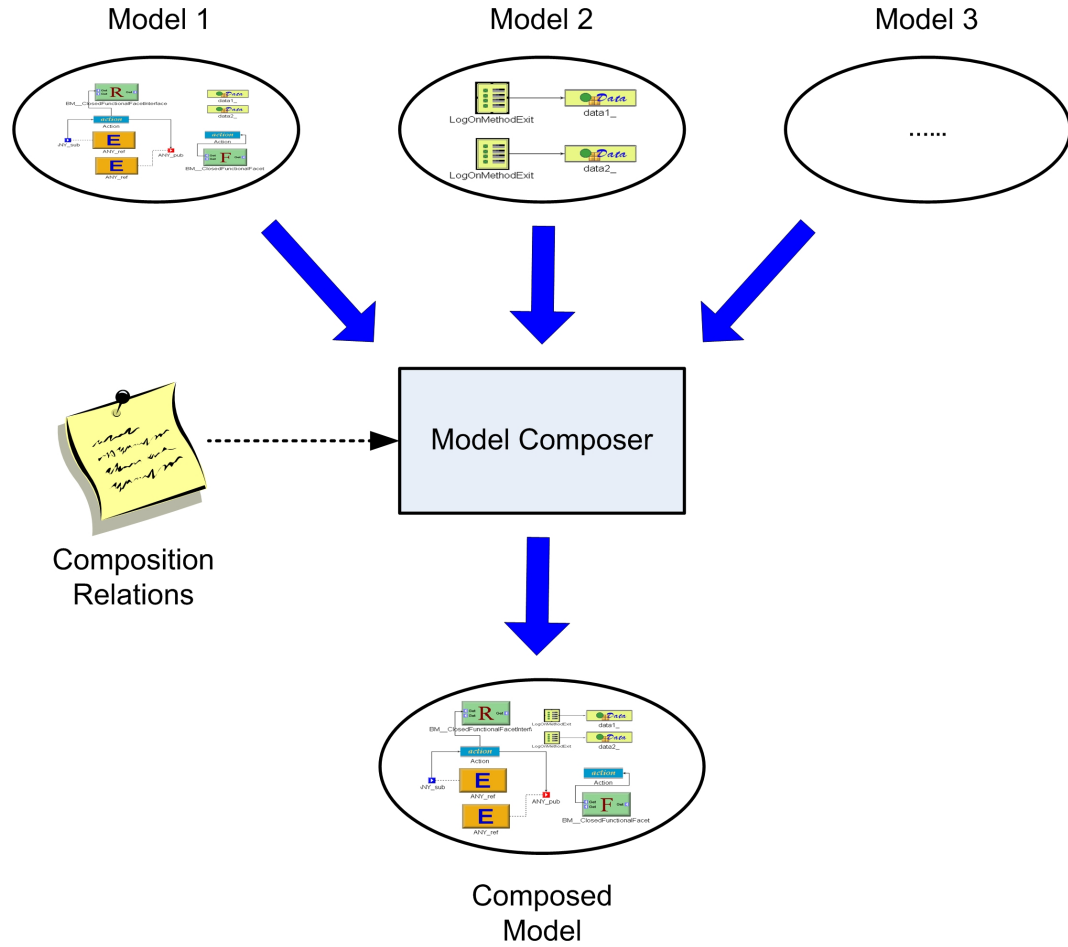


Figure 4.4: Integration of Models by a Model Composer

another model weaver with distinct aspect representation and weaving mechanisms, will be conducted for activity model composition in Section 4.3.

#### 4.1.4 Aspect Code Generator

In order to support execution on a target implementation platform, the model often has to be translated into a more concrete form, e.g., source code artifacts written in a GPL. In the MDE space, source code generation usually involves a certain kind of de-abstraction or concretization of the model, which results in the production of executable code. In the MDAA framework, the aspect models have to be mapped to the aspect representation at the source code level, in order to perform the source code

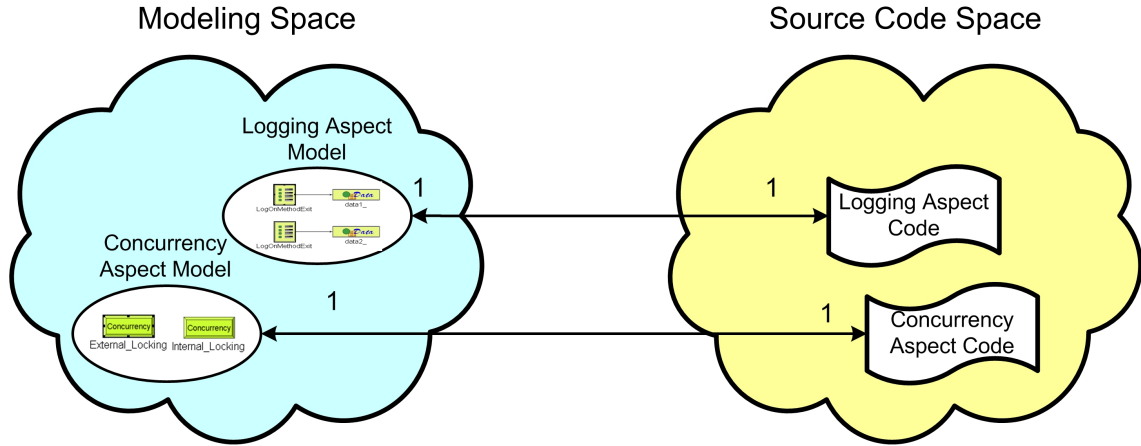


Figure 4.5: One-to-one Mapping Between Aspect Model and Aspect Source Code

adaptation for legacy evolution. This is accomplished through the use of an aspect code generator.

The aspect code generator is a key enabler for enforcing a causal connection between the model changes and the underlying change effect at the implementation level. As shown in Figure 4.5, there exists a one-to-one mapping between the model changes that are encapsulated in an aspect model and the source changes that are codified in an aspect source specification. Such a direct link is defined and maintained in the aspect code generator, which takes an aspect model, traverses and parses it according to its metamodel, and produces the corresponding aspect code that can serve as input to an underlying aspect weaver to perform the code level composition.

#### 4.1.5 Program Composer

The program composer takes the generated aspect code, together with the base legacy source, to produce a new system that is enhanced with the evolutionary changes that are codified in the aspects. Similar to model composition, program composition denotes a specific type of program transformation, i.e., the creation of integrated source code from other code. In Section 4.2, a generic program transformation engine, the Design Maintenance System (DMS), is leveraged as a program composer,

wherein aspects are specified as transformation rules in the Rule Specification Language (RSL), which allows source-level transformation-based on the internal Abstract Syntax Tree (AST) representation of the source code of a legacy system. Section 4.3 adopts a different composition approach by utilizing an existing aspect weaver – AspectJ, which enables transformation at the bytecode level.

## 4.2 DSM-Driven Aspect Adaptation

The MDAA framework must be instantiated or specialized to be applicable. Specifically, a particular model extraction procedure must be developed to retrieve the high-level design models. A modeling environment must be provided for enabling metamodel/model specification, as well as for implementing a model composer and aspect code generator. In addition, a program composer must be available to support the source code transformation. This section presents a DSM and program transformation based instantiation of the MDAA framework through the illustration of a case study on a legacy system called Bold Stroke [160]. DSM allows the domain experts to manipulate the system by directly using the concepts of the system domain. In our experimental approach, the domain-specific models are constructed in the Generic Modeling Environment (GME). The underlying program transformation system adopted is DMS [27], which has already been introduced in Section 2.5.1.

Figure 4.6 illustrates an overview of the MDAA realization within the context of DSM. The base metamodel that reflects the existing domain concepts and relationships is predefined within the GME metamodeling environment. The model extraction engineer then binds the modeling constructs to the underlying legacy source, in order to construct the base models that are valid under the definition of the base metamodel. After the legacy system is mapped into the modeling space, evolutionary changes can be performed by specifying the aspect models that conform to the aspect metamodels. In this research, C-SAW is adopted as the underlying engine for

model weaving, due to its built-in support for DSM as well as its scalable weaving power. Utilizing C-SAW, a model engineer can specify a property (e.g., “Record All updates to All variables in All components matching condition X”) from a single specification and have it weaved into hundreds of locations in a model. This permits plugging/unplugging of specific properties into the model <sup>5</sup>.

C-SAW, however, can only deal with models that are based on the same meta-model (i.e., the source model and the target model are under the definition of the same metamodel, as shown in Figure 2.14). This is categorized as *endogenous* transformation by [129] (in contrast to *exogenous* transformations that are between models under different metamodel definitions). In MDAA, a new aspect metamodel needs to be introduced to the existing base metamodel, in order to incorporate a new evolutionary change requirement. Therefore, a preprocessing step is required to compose the base metamodel and aspect metamodel before invoking C-SAW. The intention is to produce an enhanced metamodel upon which the aspect models encapsulated in an ECL specification can be inserted.

Each aspect metamodel is integrated with the base metamodel via the help of a generic metamodel composer. The composition mechanism implemented in the metamodel composer is based on matching and binding, i.e., searching for the overlapping model elements and inserting the unmatched ones in the aspect metamodel to the base model. The detailed algorithm for metamodel composition will be given in Section 4.2.3.

Due to the intrinsic characteristics of DSM, a different metamodel requires a distinct interpretation mechanism, in order to generate different software architectures for various purposes. In this sense, each aspect metamodel may correspond to a different underlying source weaving mechanism, which results in the association of a different MDAA interpreter to one aspect metamodel for generating different aspect

---

<sup>5</sup>This dissertation does not claim to make a contribution to C-SAW, but rather makes use of C-SAW to illustrate the MDAA approach.

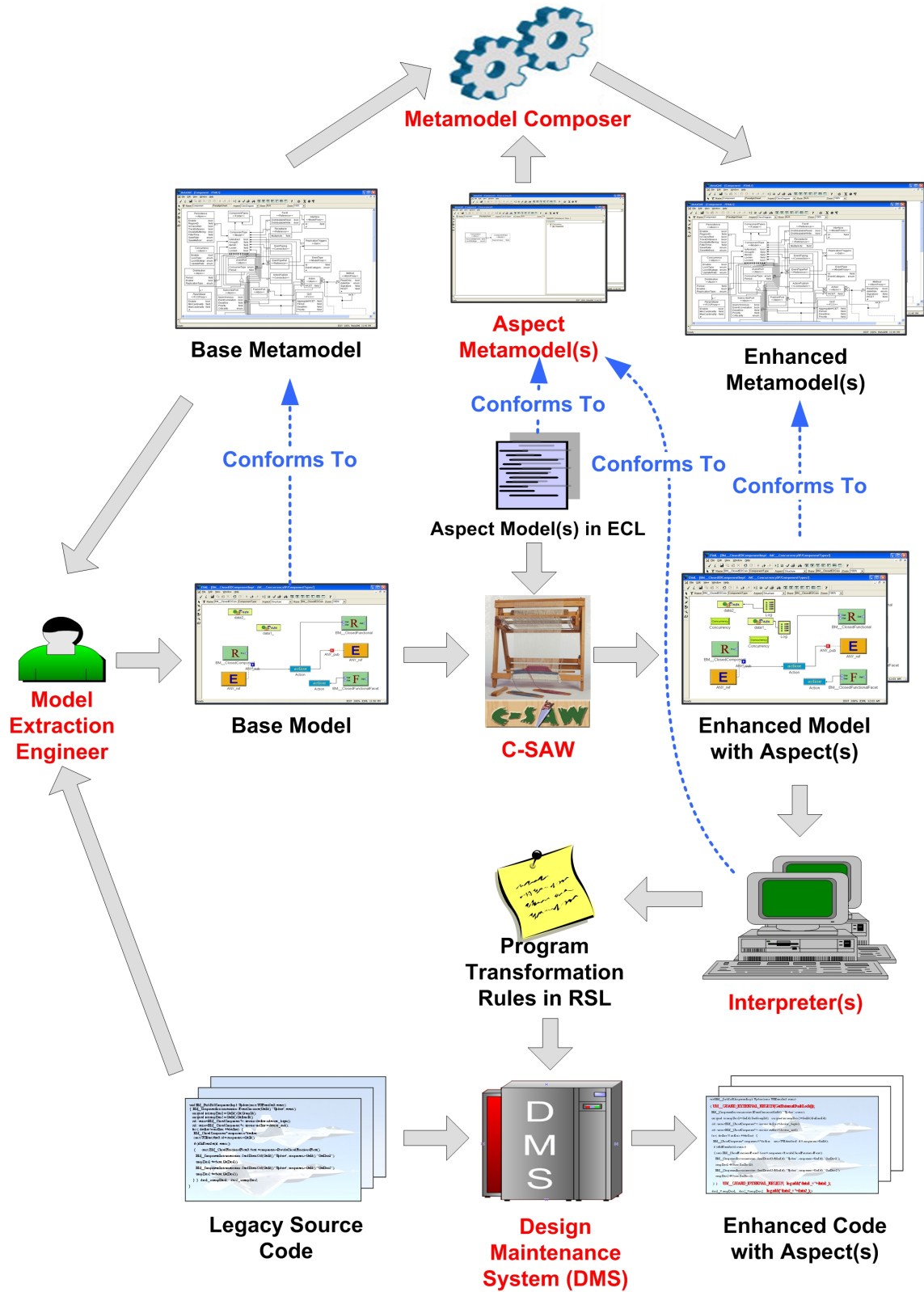


Figure 4.6: DSM-Driven Source Adaptation

code represented in DMS transformation rules. The overall benefit of the approach is large-scale adaptation across multiple source files that is driven by model properties. Such adaptation can be accomplished through minimal changes to the models. Such super-linearity is at the heart of the abstraction power provided by model-driven techniques [81, 176].

In summary of Figure 4.6, the model engineer simply makes changes to models using a higher level modeling language (either manually or by using C-SAW). Those models are then interpreted to generate transformation rules that will invasively modify a large cross-section of an application. It should be noted that the model engineer does not need to understand the accidental complexities of the transformation rule language. That process is transparent and is automated by the MDAA interpreter.

This section is structured as follows. Section 4.2.1 provides an overview of the Bold Stroke case study and sets the background and motivation for applying MDAA for legacy evolution. A domain-specific visual modeling environment for embedded systems is introduced in Section 4.2.2. The heart of the approach is contained in Section 4.2.3, which also presents two illustrative examples of the approach applied to concurrency control and a black-box flight data recorder. Finally, Sections 4.2.4 and 4.2.5 offer experimental results, discussion, as well as summary remarks.

#### *4.2.1 Background: Bold Stroke*

Bold Stroke is a product-line architecture written in several million lines of C++ that was developed by Boeing to support mission computing avionics applications for a variety of military aircraft [160]. As past participant researchers in DARPA's Program Composition for Embedded Systems (PCES), we had access to the Bold Stroke source code as an experimental platform on which to conduct our research on MDAA. The following section describes the Bold Stroke concurrency mechanism that will be used later as an example to demonstrate the applicability of MDAA.

## Bold Stroke Concurrency Mechanisms

To set the context for the following sections, the Bold Stroke concurrency mechanism is presented to provide an example for the type of transformations that can be performed in order to improve better separation of concerns within components that have been specified in a Domain-Specific Modeling Language (DSML).

There are three kinds of locking strategies available in Bold Stroke: Internal Locking, External Locking and Synchronous Proxy. The Internal Locking strategy requires the component to lock itself when its data are modified. External Locking requires the user to acquire the component's lock prior to any access of the component. The Synchronous Proxy locking strategy [158] involves the use of cached states to maintain state coherency through a chain of processing threads.

Figure 4.7 shows the code fragment in the **Update** method of the **BM\_ClosedEDComponent** in Bold Stroke. This method participates in the implementation of a real-time event channel [99]. In this component, a macro statement (Line 3) is used to implement the External Locking strategy. When system control enters the **Update** method, a preprocessed guard class is instantiated and all external components that are trying to access the **BM\_ClosedEDComponent** will be locked.

After performing its internal processing, the component eventually comes to update its own data. At this point, another macro (Line 11) is used to implement the Internal Locking strategy, which forces the component to lock itself. Internal Locking is implemented by the Scoped Locking C++ idiom [158], which ensures that a lock is acquired when control enters a scope and released automatically when control leaves the scope. Specifically, a guard class is defined to acquire and release a particular type of lock in its constructor and destructor. There are three types of locks: Null Lock, Thread Mutex, and Recursive Thread Mutex. The constructor of the guard class stores a reference to the lock and then acquires the lock. The corresponding destructor uses the pointer stored by the constructor to release the lock.

```

1 void BM__ClosedEDComponentImpl::Update (const UUEventSet& events)
2 {
3     UM__GUARD_EXTERNAL_REGION(GetExtPushLock()); // <-Locking Macro
4
5     BM_CompInstrumentation::EventConsumer(GetId(), "Update", events);
6     unsigned int tempData1 = GetId().GetGroupId();
7     unsigned int tempData2 = GetId().GetItemId();
8
9     /* REMOVED: code for implementing Real-time Event Channel
10
11     UM__GUARD_INTERNAL_REGION; // <-Locking Macro
12     data1_ = tempData1; /* REMOVED: actual var names (proprietary)
13     data2_ = tempData2;
14 }

```

Figure 4.7: The Update Method in Bold Stroke BM\_\_ClosedEDComponentImpl.cpp

The existence of locking macros, as shown in Figure 4.7, is representative of the original code base for Bold Stroke. During the development of that implementation, the concurrency control mechanisms implemented as locking macros occur in many different places in a majority of the components comprising Bold Stroke. In numerous configuration scenarios, the locking macros may evaluate to null locks, essentially making their existence in the code of no consequence. The presence of these locks (in lines 3 and 11 of Figure 4.7), and the initial effort needed to place them in the proper location, represents a point of concern regarding the manual effort needed for their initial insertion, and the future maintenance and evolution regarding this concern as new requirements for concurrency are added. The macro mechanism also represents a potential source of error for the implementation of new components - it is an additional design concern that must be remembered and added manually in the proper place for each component requiring concurrency control.

In Section 4.2.3, we remove the locking macros from the Bold Stroke source and show how the MDAA approach offers automated assistance in adding them back into the code only in those places that are implied by properties described in a model.



Before describing the details of the approach, however, it is essential to introduce the modeling language that is used to specify embedded systems like Bold Stroke.

#### 4.2.2 *Embedded Systems Modeling Language*

In this section, the Embedded Systems Modeling Language (ESML) [107] is described as a domain-specific graphical modeling language for modeling real-time mission computing embedded avionics applications. Its main goal is to address the issues arising in system integration, validation, verification, and testing of embedded systems. ESML has been defined within the GME and is being used on several US-government funded research projects sponsored from DARPA. The ESML was primarily designed by the Vanderbilt DARPA MoBIES team, and can be downloaded from the project website at <http://www.isis.vanderbilt.edu/Projects/mobies/>. There are representative ESML models for all of the Bold Stroke usage scenarios that have been defined by Boeing <sup>6</sup>.

Figure 4.8 shows the metamodel for the ESML component specification. The central element in the ESML component metamodel is **ComponentType**, which captures the modal behavior of a component. From the ESML metamodel, the GME provides an instantiation of a new graphical modeling environment supporting the visual specification and editing of ESML models (see Figures 4.9 and 4.10). The model of computation used for ESML leverages elements from the CORBA Component Model (CCM) [187] and the Bold Stroke architecture, which also uses a real-time event channel [99].

The ESML provides the following modeling categories to allow representation of an embedded system: a) Components, b) Component Interactions, and c) Component Configurations. Figure 4.9 illustrates the components and interactions for a specific scenario within Bold Stroke (i.e., the **MC\_BasicSP** scenario, which has components

---

<sup>6</sup>This dissertation does not claim to make a contribution to ESML, but rather makes use of ESML to illustrate the MDAA approach.

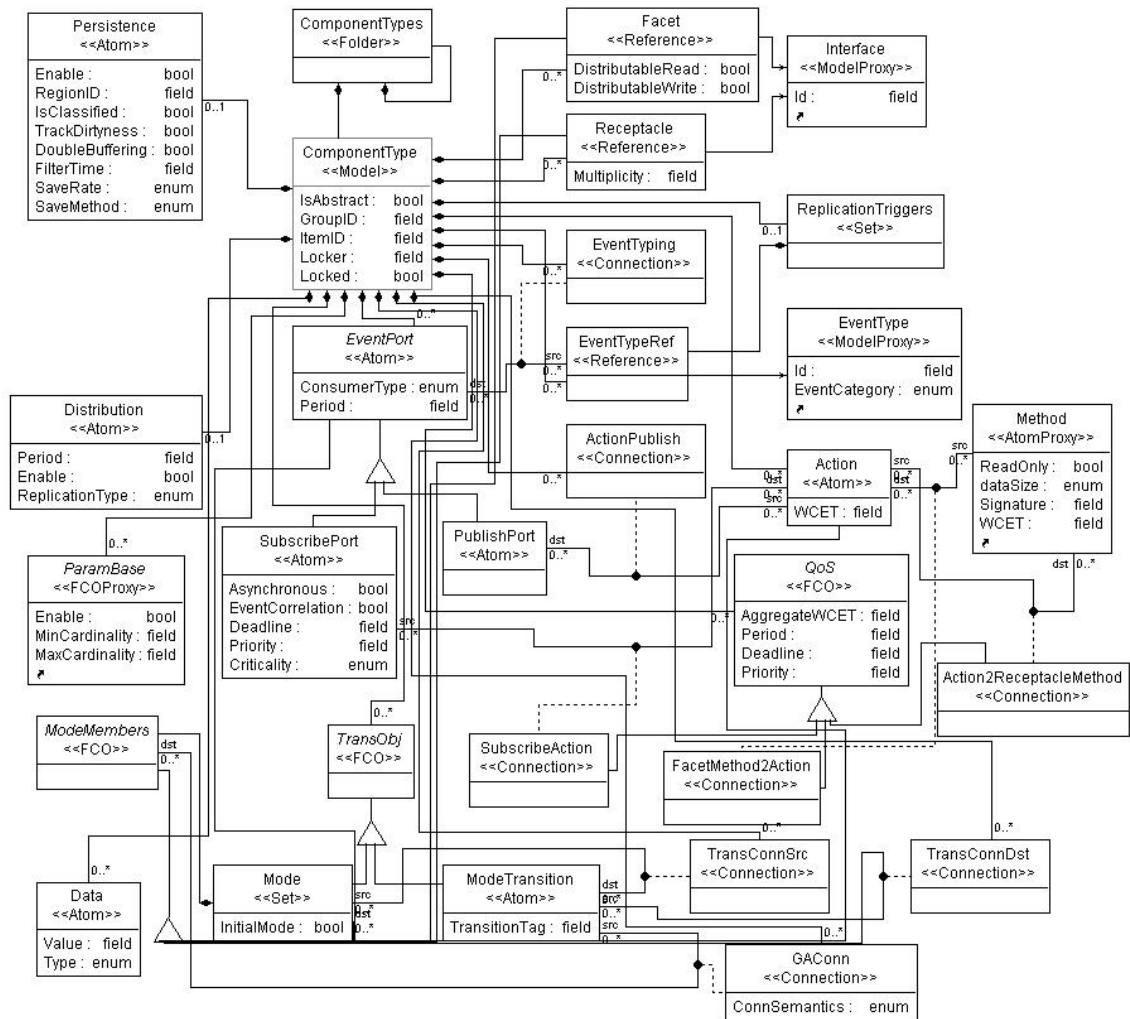


Figure 4.8: ESMML Component Metamodel (From [138])

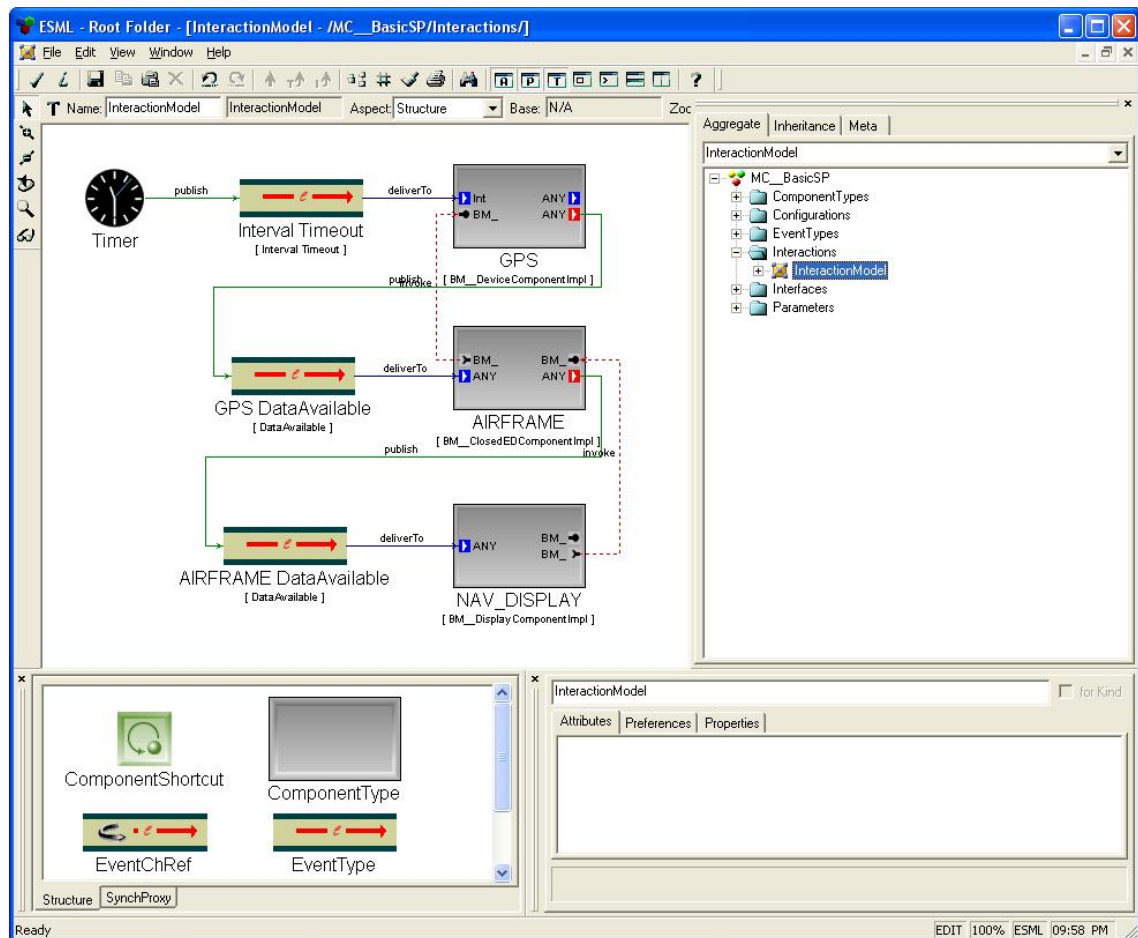


Figure 4.9: Bold Stroke Component Interaction in ESML

operating in a single-processor avionics backplane). This high-level diagram captures the interactions among components via an event channel. A system timer is also specified in this diagram.

Figure 4.10 shows a `ComponentType` model, labeled `BM_ClosedEDComponent`, which illustrates the ESML modeling capabilities for specifying the internal configuration of a component. For this component, the facet (i.e., functional features offered by the component) and receptacle (i.e., functional features required by the component) interface descriptors are specified, as well as internal data elements and events. The `BM_ClosedEDComponent` model is manually constructed by a model extraction engineer, based on the mappings between the Bold Stroke source code and ESML metamodel. Some of the mappings are listed in Table 4.1. For instance, a component class in Bold Stroke corresponds to the `ComponentType` model in an ESML specification. A component data member (e.g., `data1_` and `data2_` as shown in Figure 4.7) is mapped to a `Data` element as shown in Figure 4.10. Furthermore, the macro statements in the original source represent different locking strategies as specified in the ESML model.

Table 4.1: Some Mappings between Bold Stroke Source and ESML Metamodel

<b>Bold Stroke</b>	<b>ESML</b>
Component Class	<code>ComponentType</code>
Component Data Member	<code>Data</code>
<code>UM_GUARD_INTERNAL_REGION;</code>	<code>Concurrency.LockStrategy:</code> Internal Locking
<code>UM_GUARD_EXTERNAL_REGION</code> <code>(GetExtPushLock());</code>	<code>Concurrency.LockStrategy:</code> External Locking

The result of modeling in ESML is a set of diagrams that visually depict components, interactions, and configurations, as shown in Figures 4.9 and 4.10. The objective of the design is to create, analyze, and integrate real systems; thus, a number of

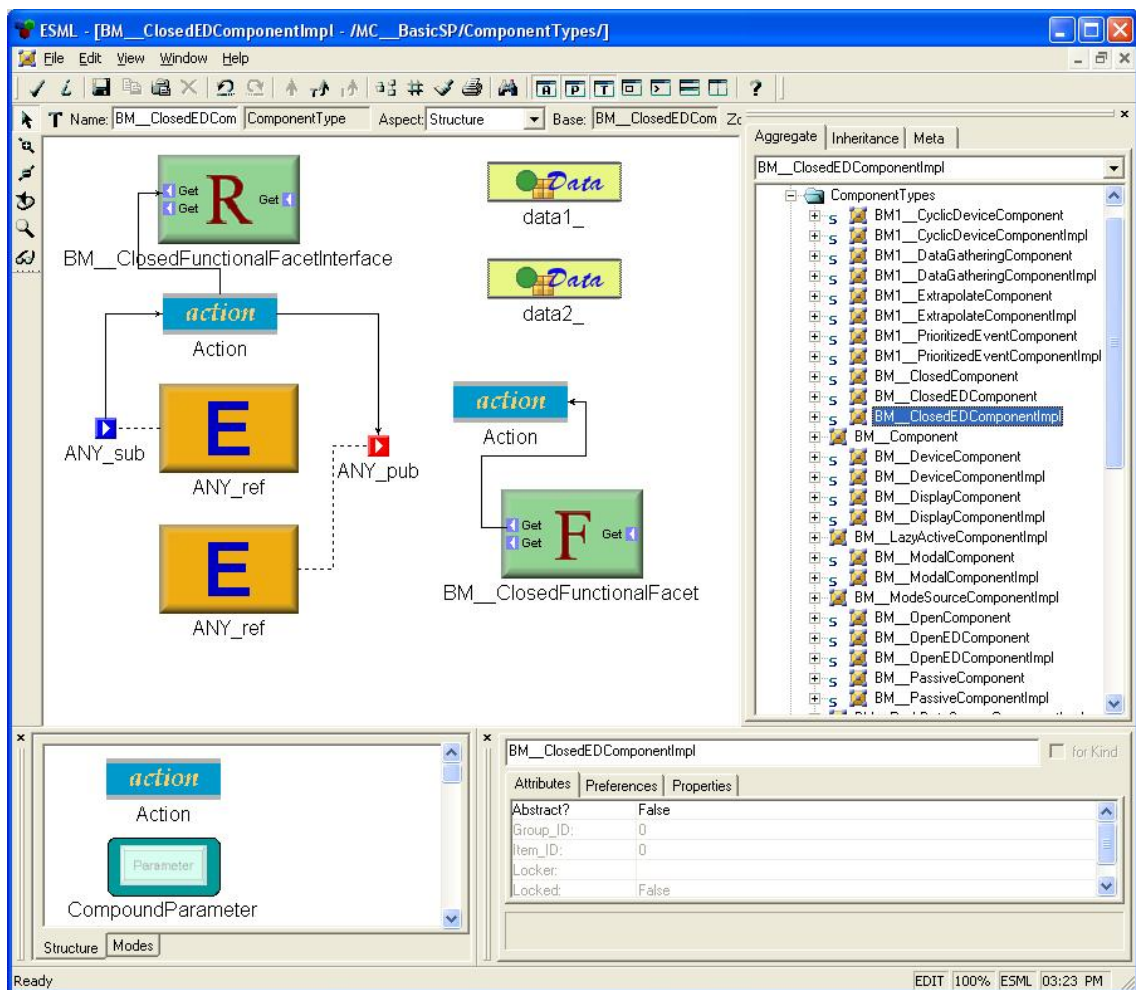


Figure 4.10: Internal Representation of `BM_ClosedEDComponent` in ESML

interfaces are defined to support these activities.

A very important feature of domain modeling within the GME is the capability of creating model interpreters. The modeling environment stores the model as objects in a database repository, and it provides an API for model traversal using a standard integration mechanism (i.e., COM [39]) provided by the GME. Using the API, it is possible to create model interpreters that traverse the internal representation of the model and generate new artifacts (e.g., XML configuration files, source code, or even hardware logic) based on the model properties. It is possible to associate multiple interpreters to the same modeling language.

A variety of model interpreters have been created for the ESML by the Vanderbilt DAPA MoBIES team. The **Configuration Interface** interpreter is responsible for generating an XML file that is used during load-time configuration of Bold Stroke. The locking macros of Figure 4.7 are configured from this generated file. The **Configuration Interface** provides an example of vertical translation that is more aligned with the synthesis idea for generating new artifacts, rather than a pure transformation approach that invasively modifies one artifact from descriptions in a model (as in Section 4.2.3). Another interpreter for ESML is the **Analysis Interface**, which assists in integrating third-party analysis tools.

In the approach described in this dissertation, a specific family of ESML interpreters are created to invasively modify a very large code base from properties specified in an ESML model. Such an interpreter enables the ideas of MDAA.

#### *4.2.3 Applying MDAA to Bold Stroke*

The objective of MDAA is evolution of legacy source from properties described in high-level aspect models. As model changes are made to certain properties of a system, those changes must have a corresponding effect at the implementation level. A common way to achieve this correspondence is through load-time configuration of

property files that are generated from the models (e.g., the XML configuration file deployed by the **Configuration Interface** interpreter). There are two key problems with the load-time configuration file technique, however:

1. The load-time configuration mechanism must be built into the existing implementation. The source implementation must know how to interpret the configuration file and make the necessary adaptations at all of the potential extension points. For example, in Bold Stroke the locking strategy used for each component is specified in an XML configuration file, which is loaded at run-time during initial startup. The component developer must know about the extension points and how they interact with the configuration file at load-time.
2. A typical approach to support this load-time extension is macro tailorability, as seen in Figure 4.7. At each location in the source where variation may occur, a macro is added that can be configured from the properties specified in the XML configuration file. However, this forces the introduction of macro tags in multiple locations of the source that may not be affected under many configurations. The instrumentation of the source to include such tailoring is often performed by manual adaptation of the source (see lines 3 and 13 of Figure 4.7). This approach also requires the ability to anticipate future points of extension, which is not always possible for a system with millions of lines of code and changing requirements.

These problems provide a major hurdle to the transfer of model-driven and load-time configuration approaches into large legacy systems. As an example, consider the two hundred billion lines of COBOL code that are estimated to exist in production systems [181]. To adopt the load-time configuration file approach to such systems will require large manual modifications to adjust to the new type of configuration.

The following two sub-sections represent how the MDAA approach can facilitate legacy evolution from the model properties without manual instrumentation to the source code. Two examples are illustrated, representing crosscutting concerns related to concurrency control and recording of flight data information.

### **Weaving Concurrency into Bold Stroke**

Recall the concurrency mechanism supported within Bold Stroke, as described in Section 4.2.1. In particular, consider the source code presented in Figure 4.7. There are a few problems with the macro tailorability approach, as used in this example code fragment:

1. Whenever a new component is created, the developer must remember to explicitly add the macros in the correct place for all future components (a large source of human error).
2. Because a component may be used in several contexts, it is typical that different locking strategies are used in various usage scenarios. For example, the very existence of a Null Lock type is a direct consequence of the fact that a component is forced to process the macro even in those cases when locking may not be needed for a particular instantiation of the component. The result is that additional compile-time (or, even run-time overhead, if the chosen C++ compiler does not provide intelligent optimizations) is incurred to process the macro in unnecessary cases.

As an alternative, MDAA provides a solution that does not require the locking to be explicitly added by the developer to all components. The approach only adds locking to those components that specify the need in a high-level model, which is based on the requirements of the specific application scenario that is being modeled.



Figure 4.11 shows the ESML concurrency metamodel, which contains the concurrency model specification as well as its association to the ESML base model element (i.e., `ComponentType` in this case), indicating that the `Concurrency` atom is to be inserted as a part of the `ComponentType` model in ESML. The `Concurrency` atom captures the configurable concurrency requirements of a component. The attributes of this atom (i.e., `Enable`, `LockType`, `LockStrategy` and `UpdateRate`) capture the specifics of the concurrency mechanism. `LockType` denotes the three locking types, i.e., Null Lock, Thread Mutex, and Recursive Thread Mutex. `LockStrategy` is an enumeration attribute that specifies the three kinds of locking strategies (i.e., Internal Locking, External Locking, or Synchronous Proxy) that can be used.

An aspect element named as `ConcurrencyAspect` is also attached to the `ComponentType` model (as shown in Figure 4.12), indicating that the `Concurrency` atom belongs to the `ConcurrencyAspect` and is visible from the `ConcurrencyAspect` point of view <sup>7</sup>. In our approach, we leverage the concept of aspect in GME and link it to the well-known definition of aspect in terms of AOSD [2]. Therefore, each aspect along with its group members in the GME metamodel corresponds to an aspect specification that represents an evolutionary change requirement and can be translated into the low-level aspect code to perform source-level aspect weaving. By utilizing the concept of aspect in GME, the base model and the evolutionary changes that are captured in the aspect models can be separated in terms of different viewpoints, which offers a clear separation of concerns for the purpose of understandability and analyzability.

The concurrency aspect metamodel can be integrated with the ESML base model via the metamodel composer, which produces the augmented version of the ESML component metamodel as shown in Figure 4.13. The metamodel composer is implemented as a plug-in component in the GME environment and can be invoked upon

---

<sup>7</sup>Please note that aspect has a special meaning in GME, which is used to partition the models into different visibility groups. When a model is displayed, it is always viewed from one particular aspect at a time. Some model elements may be visible in more than one aspect while others may be visible only in a single aspect [68].

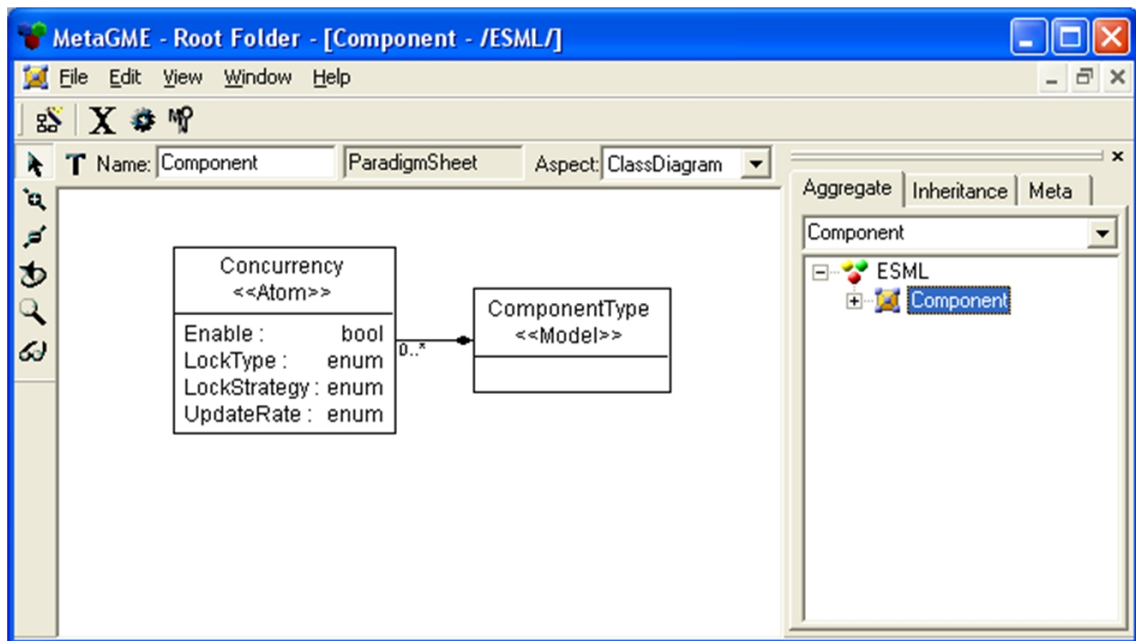


Figure 4.11: ESML Concurrency Aspect Metamodel

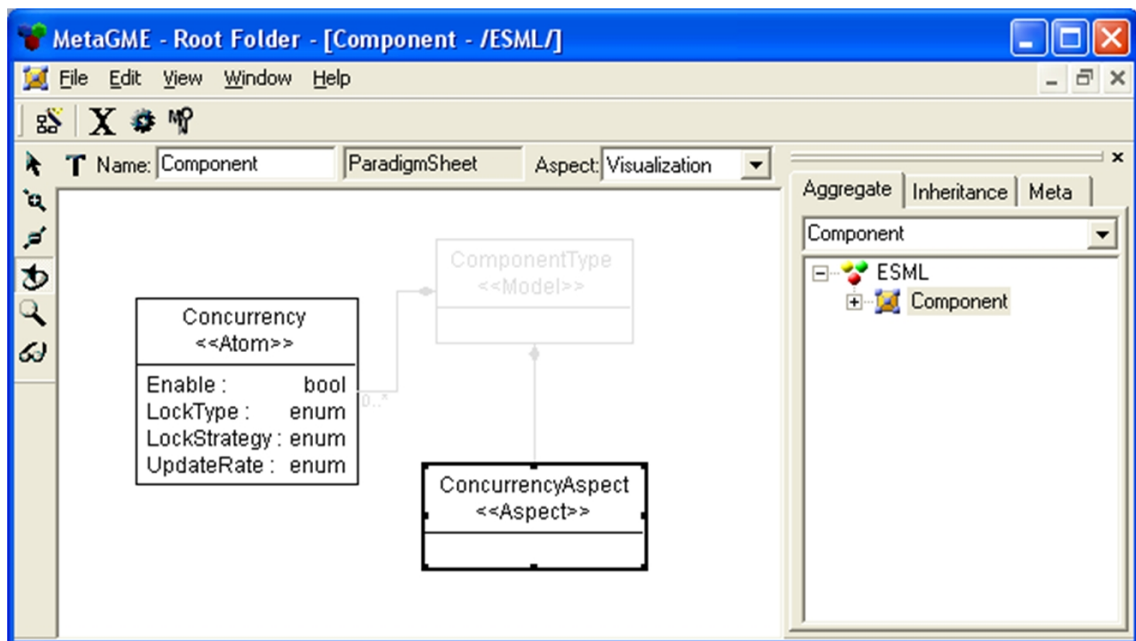


Figure 4.12: ESML Concurrency Metamodel Aspect View

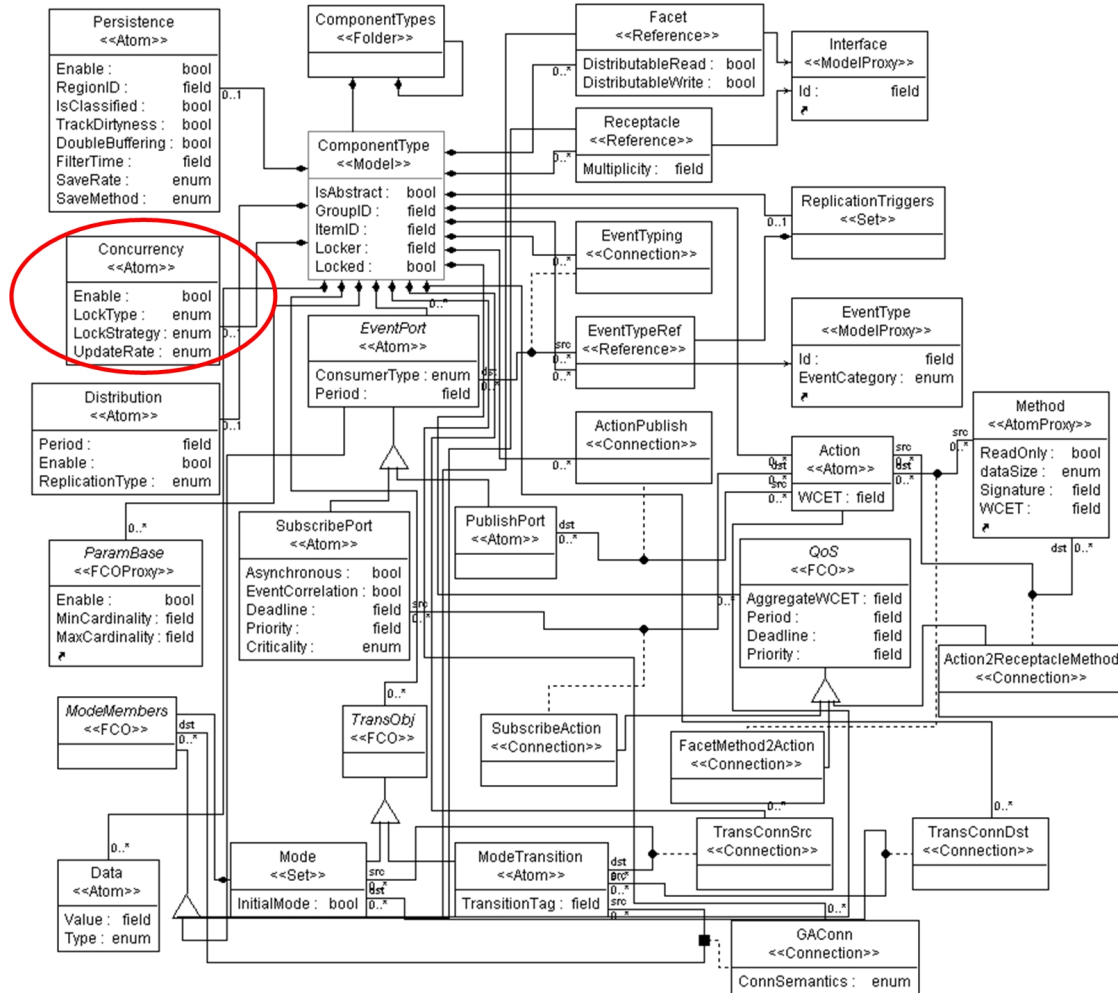


Figure 4.13: ESM Component Metamodel (Augmented with the Concurrency Metamodel)

any metamodel. The composition process starts with loading the aspect metamodel and locating the aspect element in it. For each model element that belongs to this aspect and does not exist in the base metamodel, it will be inserted in turn to the base metamodel, along with its attributes and relevant associations. The algorithm to perform the metamodel composition is described in Figure 4.14.

The updated ESM metamodel as shown in Figure 4.13 now contains the Concurrency atom that is associated to the ComponentType model, indicating that the concurrency concern is now added to the domain. The new metamodel thus drives

```

1  open BaseMetamodel
2  load AspectMetamodel
3  select Aspect in the AspectMetamodel
4  for (each model element m in the Aspect) do
5      if (m does not exist in the BaseMetamodel)
6          for (each m's connection c) do
7              e = get_connection_end(m, c)
8              insert m to the BaseMetamodel
9              add m's attributes to m
10             insert c to the BaseMetamodel connecting m and e
11             add c's attributes to c
12         end for
13     end if
14 end for
15 insert the Aspect to the BaseMetamodel
16
17 function get_connection_end(model m, connection c)
18     e = the model element at the other end of the connection c of model m
19     if (e does not exist in the BaseMetamodel)
20         for (each e's connection cc) do
21             ee = get_connection_end(e, cc)
22             insert e to the BaseMetamodel
23             add e's attributes to e
24             insert cc to the BaseMetamodel connecting e and ee
25             add cc's attributes to cc
26         end for
27     end if
28     return e
29 end function

```

Figure 4.14: Metamodel Composition Algorithm

the generation of a new ESML model editing environment that is able to configure the concurrency requirements for each component. Figure 4.15 illustrates the upgraded version of the `BM_ClosedEDComponent` model as was shown in Figure 4.10. In this new environment, the `Concurrency` atom is the only model element that is visible within the `Concurrency` aspect (see the bottom part of the figure). Two types of concurrency strategies are created in this particular example, i.e., Internal Locking and External Locking.

The concurrency model element can be added to the ESML component model manually or automatically via the use of C-SAW when the same concurrency strategy is applied to multiple component models. Sample code of the ECL specification to enable External Locking for an ESML component is shown in Figure 4.16. The aspect specification finds all of the component models whose name ends with `Impl` (Lines 19 to 20). For each resulting component model, the weaving strategy prescribes that one `Concurrency` atom be inserted. The `External Locking Concurrency` is set to be enabled, with the type `Recursive Thread Mutex` and an update rate at `40Hz`.

So far, the model-level aspect weaving has been accomplished by leveraging the metamodel composer and C-SAW. The next step is to generate the source-level aspect representation in order to enforce the transformational changes to the source code. This is automated by the MDAA interpreter that is customized for each specific aspect metamodel. The MDAA interpreter is a special model interpreter that is intended to generate a transformation specification in the form of an aspect representation, which is unlike the other model interpreters that are often used to synthesize executable code in the system's execution environment. In the Bold Stroke experimental case study, for instance, the concurrency aspect metamodel is associated with a specific MDAA interpreter that will be used to generate a concurrency transformation specification to transform the Bold Stoke base source from configuration of the ESML models.

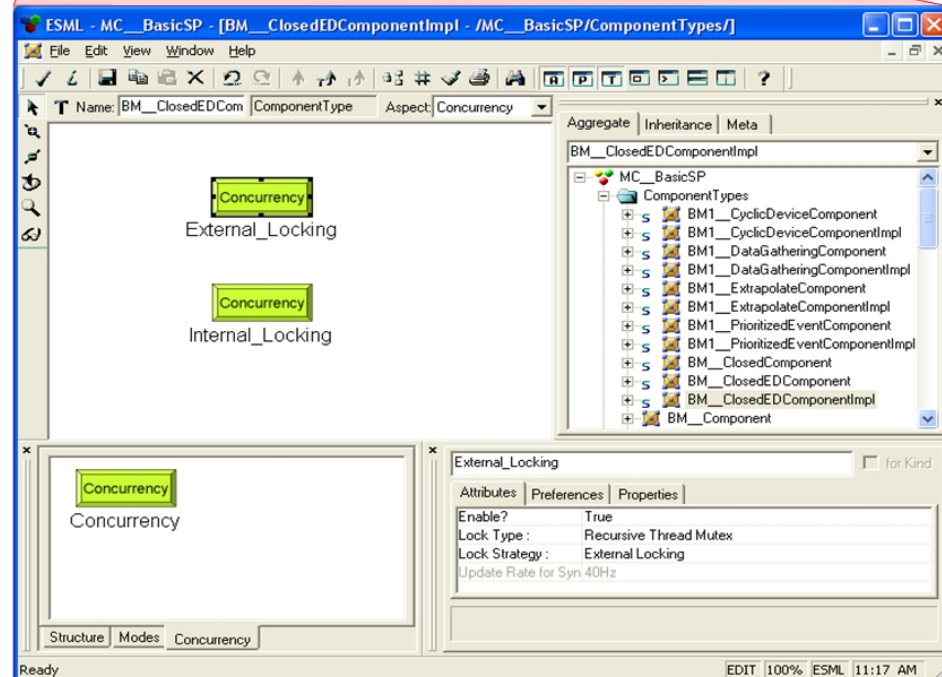
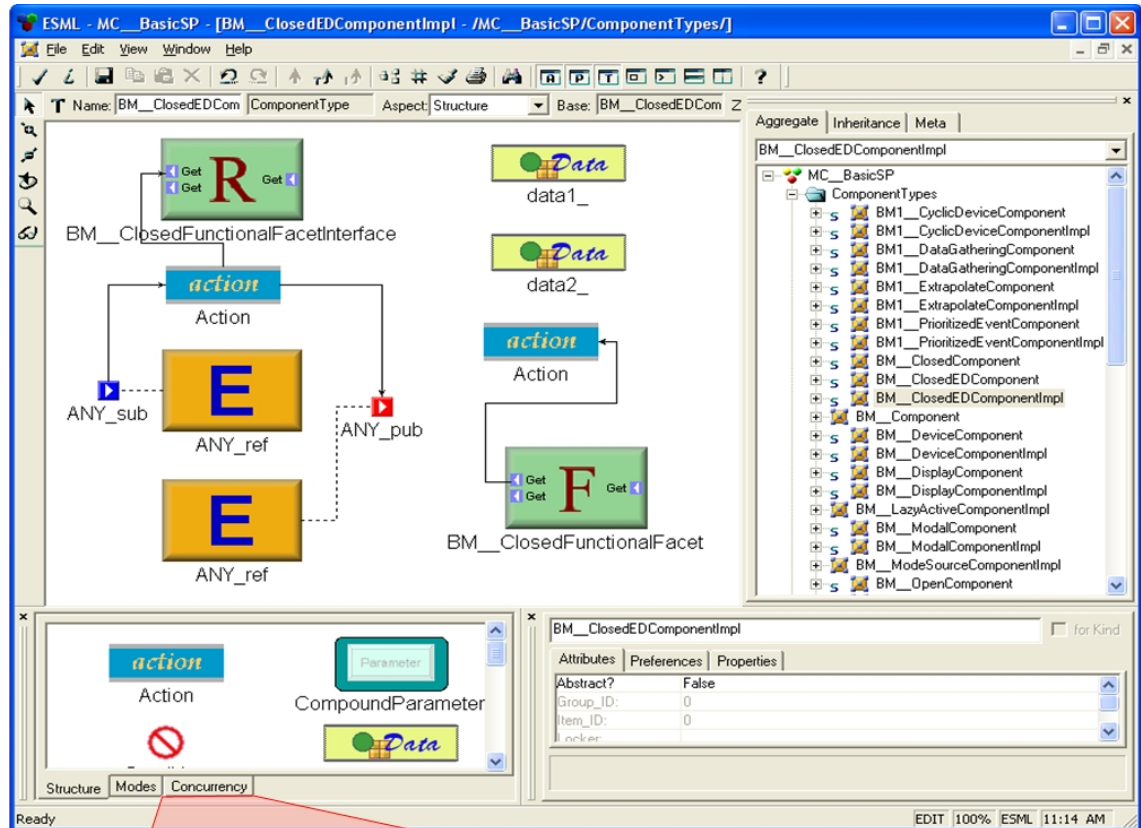


Figure 4.15: Internal Representation of the `BM_ClosedEDComponent` in ESML (Augmented with the Concurrency Aspect)

```

1  defines Start, AddExternalLocking;
2
3  strategy AddExternalLocking()
4  {
5      declare parentModel : model;
6      declare concurrencyAtom : atom;
7
8      parentModel := parent();
9      concurrencyAtom := parentModel.addAtom("Concurrency",
10                                         "External_Locking");
11      concurrencyAtom.setAttribute("Enable", "true");
12      concurrencyAtom.setAttribute("LockType", "Recursive Thread Mutex");
13      concurrencyAtom.setAttribute("LockStrategy", "External Locking");
14      concurrencyAtom.setAttribute("UpdateRate", "40Hz");
15  }
16
17  aspect Start()
18  {
19      rootFolder().findFolder("ComponentTypes").models().
20          select(m|m.name().endsWith("Impl"))->AddExternalLocking();
21  }

```

Figure 4.16: ECL Specification for Adding External Locking to ESML Component Models (Adapted from [122])

The transformation specification generation is based on the mappings between the aspect metamodel definition and the source representation. For instance, in ESML, the attribute value **Internal Locking** for the locking strategy of the concurrency atom corresponds to the C++ macro expression `UM__GUARD_INTERNAL_REGION`; in Bold Stroke, as listed previously in Table 4.1. In addition, the MDAA interpreter also encodes and maintains the locations or join points where the source statement is to be inserted (e.g., `UM__GUARD_INTERNAL_REGION`; needs to be added just before any update to any data member).

Figure 4.17 is an example of a generated transformation specification from the ESML models via the MDAA interpreter. The transformation specification is written in the Rule Specification Language (RSL) provided by the DMS transformation engine. RSL provides basic primitives for describing numerous transformations that are to be performed across the entire code base of an application. It consists of declarations of patterns, rules, conditions, and rule sets using the external form (i.e., concrete syntax) defined by a language domain. Typically, a large collection of RSL files are needed to describe the full set of transformations. The patterns and rules can have associated conditions that describe restrictions on when a pattern legally matches a syntax tree, or when a rule is applicable on a syntax tree.

Figure 4.17 shows the RSL specification for incorporating two kinds of concurrency strategies: insertion of an External Locking Statement and an Internal Locking Statement. The first line of the figure establishes the default language domain to which the DMS rules are applied (in this case, it is the implementation environment for Bold Stroke - Visual Studio C++ 6.0). Eight patterns are defined from line 3 to line 26, followed by two transformation rules. The patterns on lines 3, 6, 9, 13, 26 - along with the rule on line 28 - define the external locking transformation. Likewise, the patterns on lines 16, 19, 22 - and the rule on line 36 - specify the internal locking transformation.



```

1  default base domain Cpp~VisualCpp6.
2
3  pattern UM_GUARD_EXTERNAL_REGION_as_identifier_or_template_id():
4    identifier_or_template_id = "UM__GUARD_EXTERNAL_REGION".
5
6  pattern GetExternPushLock_as_identifier_or_template_id():
7    identifier_or_template_id = "GetExternalPushLock".
8
9  pattern ExternLockStmt(): expression_statement =
10    "\UM_GUARD_EXTERNAL_REGION_as_identifier_or_template_id\(\)
11    (\GetExternPushLock_as_identifier_or_template_id\(\)());".
12
13  pattern ExternLockAspect(s: statement_seq): compound_statement =
14    "{\ExternLockStmt\(\) {\s}}".
15
16  pattern InternLockStmt(): expression_statement =
17    "UM__GUARD_INTERNAL_REGION;".
18
19  pattern InternLockJoinPoint(expr:logical_or_expression): statement =
20    "data1_ = \expr;".
21
22  pattern InternLockAspect(expr:logical_or_expression, s:statement_seq):
23    statement_seq = "\s {\InternLockStmt\(\)
24    \InternLockJoinPoint\(\expr\)}".
25
26  pattern JoinPoint(id:identifier): qualified_id = "\id :: Update".
27
28  rule insert_extern_lock(id:identifier, s: statement_seq,
29    p:parameter_declaration_clause):
30    function_definition -> function_definition =
31      "void \JoinPoint\(\id\)(\p) {\s} " ->
32      "void \JoinPoint\(\id\)(\p) {\ExternLockAspect\(\s\)}"
33  if ~[modsList:statement_seq. s matches
34    "\:statement_seq \ExternLockAspect\(\modsList\)"].
35
36  rule insert_intern_lock(expr:logical_or_expression, s:statement_seq):
37    statement_seq -> statement_seq =
38      "\s \InternLockJoinPoint\(\expr\)" ->
39      "\InternLockAspect\(\expr\,\s\)"
40  if s ~= "\:statement_seq \InternalLockStmt\(\)".
41
42  public ruleset applyrules={insert_extern_lock, insert_intern_lock}.

```

Figure 4.17: A Set of Generated Locking Transformation Patterns and Rules in the DMS Rule Specification Language

Patterns describe the form of a syntax tree. They are used for matching purposes to find a syntax tree having a specified structure. Patterns are often used on the right-hand side (target) of a rule to describe the resulting syntax tree after the rule is applied. In the first pattern (line 3, Figure 4.17), a very simple pattern is described. This pattern matches the inserted macro (named `UM_GUARD_EXTERNAL_REGION`) to the syntax tree expression that is defined as `identifier_or_template_id` in the grammar definition of the DMS VC++ 6.0 domain. The third pattern (line 9) is used to combine the first and second pattern into a larger one, in order to represent the full macro statement along with its parameters. The target rule that describes the form of the resulting syntax tree is specified in the fourth pattern (line 13). This fourth pattern scopes the protected region and places the external locking statement as the first statement within the scope. Similarly, the pattern on line 22 describes the form of the resulting syntax tree after inserting an internal locking statement in front of any update of `data1_`. The last pattern (line 26) provides the context in which the transformation rules will be applied. Here, the rules will be applied to all of the components containing an `Update` method. This pattern is similar to a join point in AspectJ [108]. Although this last pattern is very simple, it quantifies over the entire code base and selects all of those syntax trees matching the pattern.

The RSL rules describe a directed pair of corresponding syntax trees. A rule is typically used as a rewrite specification that maps from a left-hand side (source) syntax tree expression to a right-hand side (target) syntax tree expression. As an example, the rule specified on line 28 of Figure 4.17 represents a transformation on all `Update` methods (specified by the `JoinPoint` pattern). The effect of this rule is to add an external locking statement to all `Updates`, regardless of the various parameters of each `Update` method. Notice that there is a condition associated with this rule (line 33). This condition describes a constraint that this rule should be applied only when there already does not exist an external locking statement. That is, the

transformation rule will be applied only once. Without this condition, the rules would be applied iteratively and fall into an infinite loop. The rule on line 36 applies the transformations associated with inserting an internal locking statement just before modification of the internal field named `data1_`. Rules can be combined into sets of rules that together form a transformation strategy by defining a collection of transformations that can be applied to a syntax tree. In the ruleset defined on line 42, the two locking rules are aggregated to perform a sequence of transformations (i.e., External/Internal Locking).

### Supporting a Black Box Data Recorder

In avionics systems, an essential diagnostic tool for failure analysis is a “black box” that records important flight information. This device can be recovered during a failure, and can reveal valuable information even in the event of a total system loss. There are several factors that make development of such a data recording device difficult:

1. During ground testing and simulation of the complete aircraft system, it is often useful to have a liberal strategy for collecting data points. The information that is collected may come from a large group of events and invocations generated during testing of a specific configuration of Bold Stroke.
2. However, an actual deployed system has very limited storage space to record data. In a deployed system, data may be collected from a small subset of the points that were logged during simulation. For example, only a few components may be of interest during specific phases of a mission. Also, only a subset of events may be recorded in an operational fighter jet.

It is a desirable feature to support the various types of recording policies that may be observed throughout development, testing, and deployment. Currently, the

development tools associated with Bold Stroke do not support a capability to plug recording policies easily into the code base. The manual effort that would be required to plug/unplug different data recording policies throughout all components would be unfeasible in general practice. It is possible to transform existing Bold Stroke code by adding the black box flight recorder concern. The recorder information is specified by a logging policy (as can be seen in the **Log** modeling element in the logging aspect metamodel as shown in Figure 4.18). Within the logging policy, a model engineer can specify policies such as “Record the values upon <entry/exit> of <a set of named methods>” or “Record the value upon every update to the <data variable>.”

Figure 4.19 contains the ECL aspect specification to connect **Log** atoms (of type On Method Exit) to **Data** atoms in ESML models (see the resulting model in Figure 4.20). The aspect specification finds all of the **Data** atoms (line 3 to line 6) in every component whose name ends with **Impl** (line 21 to line 25). For each **Data** atom, a

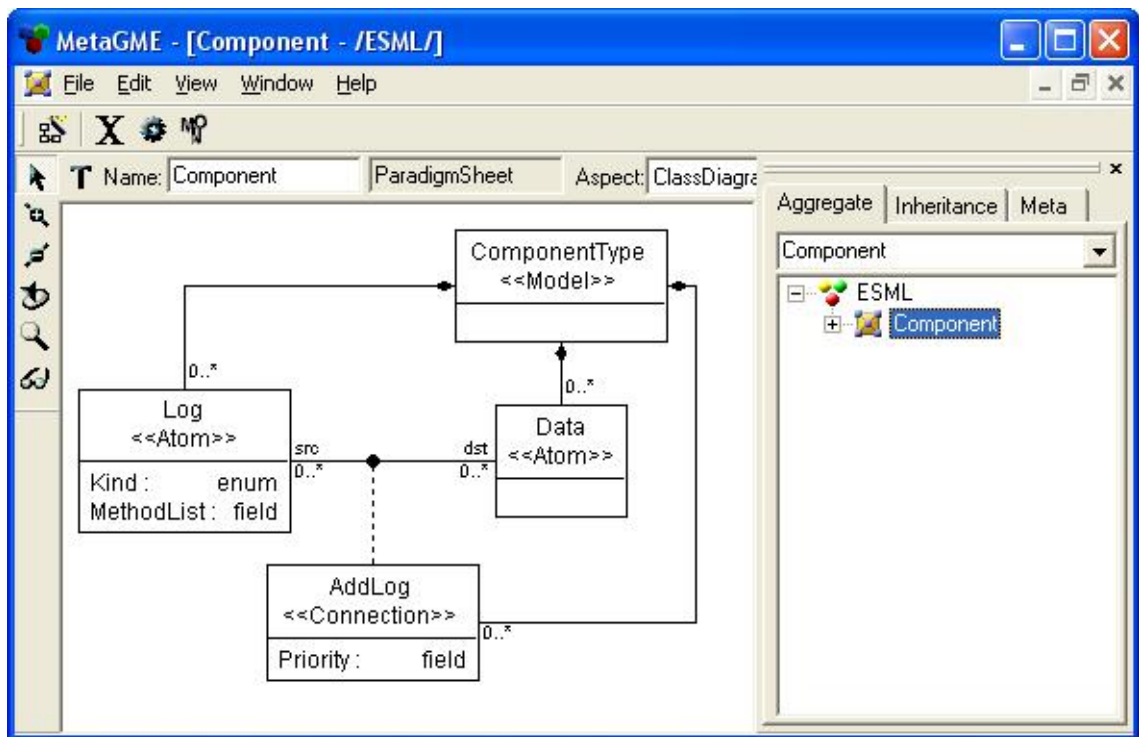


Figure 4.18: ESML Logging Metamodel

```

1  defines Start, logDataAtoms, AddLog;
2
3  strategy logDataAtoms()
4  {
5      atoms()->select(a | a.kindOf() == "Data")->AddLog();
6  }
7
8  strategy AddLog()
9  {
10     declare parentModel : model;
11     declare dataAtom, logAtom : atom;
12
13     dataAtom := self;
14     parentModel := parent();
15     logAtom := parentModel.addAtom("Log", "LogOnMethodExit");
16     logAtom.setAttribute("Kind", "On Method Exit");
17     logAtom.setAttribute("MethodList", "Update");
18     parentModel.addConnection("AddLog", logAtom, dataAtom);
19 }
20
21 aspect Start()
22 {
23     rootFolder().findFolder("ComponentTypes").models().
24         select(m|m.name().endsWith("Impl"))->logDataAtoms();
25 }

```

Figure 4.19: ECL Specification for Adding `LogOnMethodExit` in ESML Models (From [83])

new `Log` atom is created. Finally, it connects this new `Log` atom to its corresponding `Data` atom (line 18). As a result, after using C-SAW to apply this ECL specification, `LogOnMethodExit` atoms will be inserted into each component that has a `Data` atom. As a front-end design capability, the model weaving drives the automatic generation of the DMS RSL rules to transform the underlying Bold Stroke C++ source program.

The generated logging transformation represented in RSL can be found in Figure 4.21. In this example, the `LogOnMethodExit` logging policy is illustrated (this is specified as an attribute in the `Log` modeling element of Figure 4.20). The patterns on lines 3, 5, 8 - with the rule on line 10 - denote the update logging transformation. The

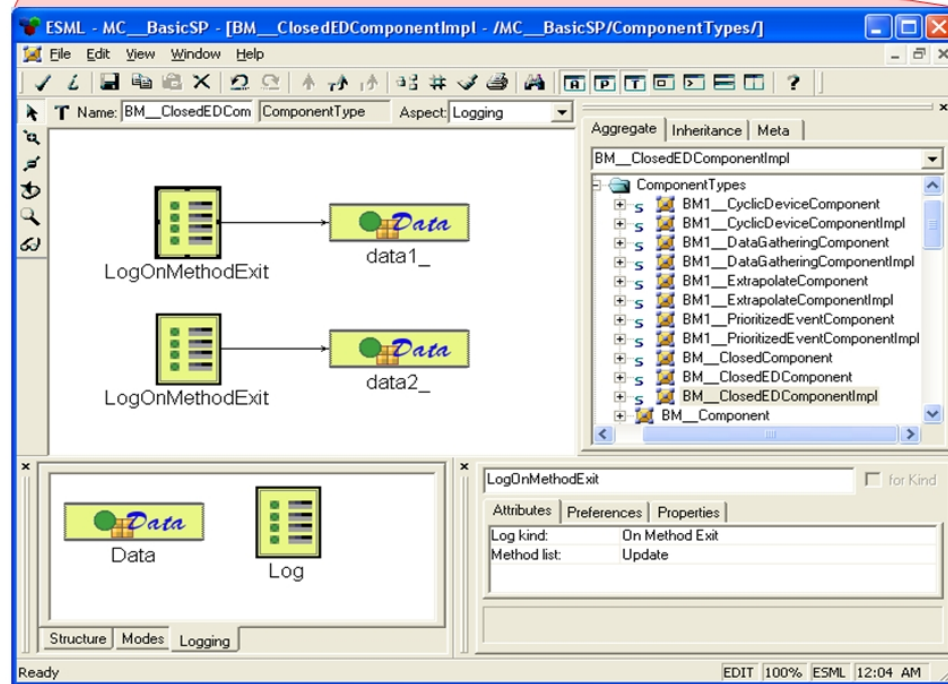
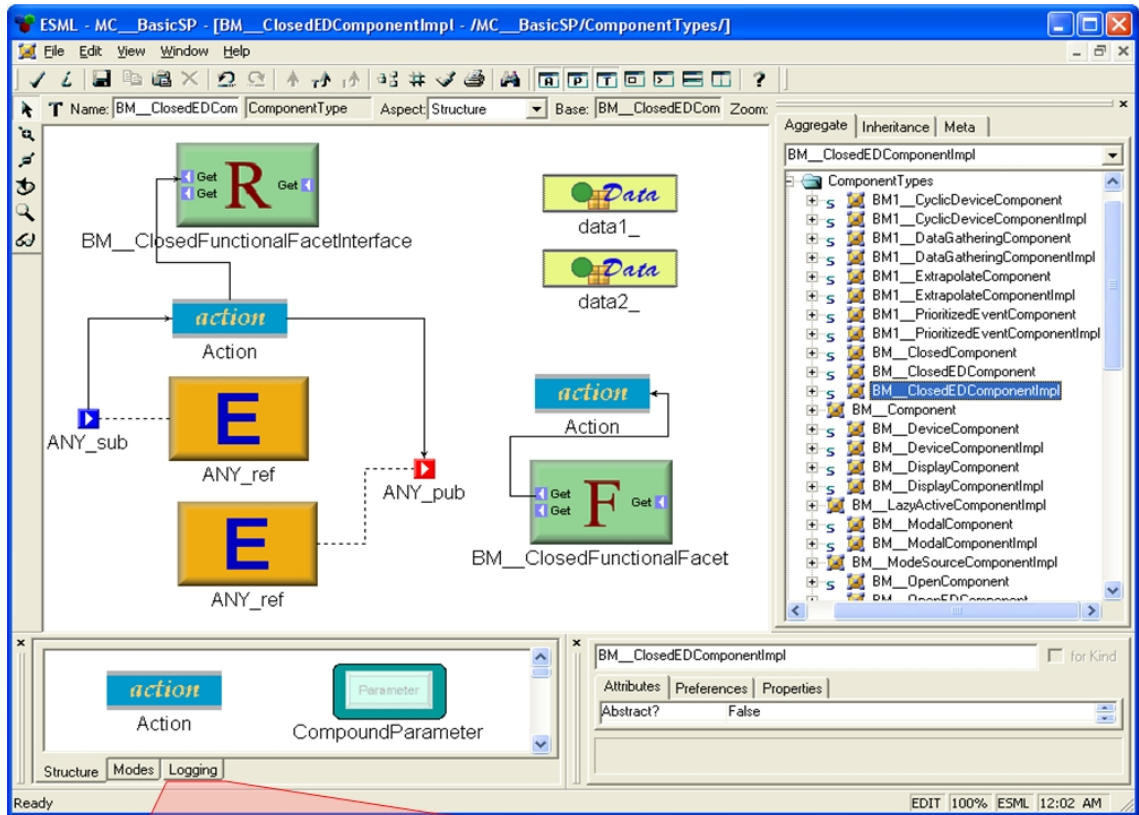


Figure 4.20: Internal Representation of the `BM_ClosedEDComponent` in ESML (Augmented with the Logging Aspect)

```

1  default base domain Cpp~VisualCpp6.
2
3  pattern LogStmt(): statement = "log.add(\"data1_=\" + data1_);".
4
5  pattern LogOnMethodExitAspect(s: statement_seq): statement_seq =
6      "\s \LogStmt\(\)".
7
8  pattern JoinPoint(id:identifier): qualified_id = "\id :: Update".
9
10 rule insert_log_on_method_exit(id:identifier, s:statement_seq,
11     p:parameter_declaration_clause):
12     function_definition -> function_definition =
13         "void \JoinPoint\(\id\)(\p) {\s} " ->
14         "void \JoinPoint\(\id\)(\p) {\LogOnMethodExitAspect\(\s\)}"
15 if ~[modsList:statement_seq. s matches
16     "\:statement_seq \LogOnMethodExitAspect\(\modsList\)"].
17
18 public ruleset applyrules={insert_log_on_method_exit}.

```

Figure 4.21: A Set of Generated Logging Transformation Patterns and Rules in the DMS Rule Specification Language

pattern on line 5 shows the resulting form after inserting a log statement on all exits of the **Update** method. The corresponding rule on line 10 inserts the logging statement upon the exit of every **Update** method of every component. Figure 4.22 presents the resulting Bold Stoke source code that is enhanced with the logging statement inserted at the end of the method **Update**.

It is important to reiterate that the domain model engineer does not create (or even see) the RSL transformation rules. These are created by the MDAA interpreter and directly applied toward the aspect-oriented transformation of Bold Stroke code using the DMS transformation engine.

#### 4.2.4 Experimental Results

A set of experimental case studies have been performed to validate the feasibility and advantages of the MDAA approach.

```

1 void BM__ClosedEDComponentImpl::Update (const UUEventSet& events)
2 {
3     BM_CompInstrumentation::EventConsumer(GetId(), "Update", events);
4     unsigned int tempData1 = GetId().GetGroupId();
5     unsigned int tempData2 = GetId().GetItemId();
6
7     /* REMOVED: code for implementing Real-time Event Channel
8
9     data1_ = tempData1;  /* REMOVED: actual var names (proprietary)
10    data2_ = tempData2;
11
12    log.add("data1_" + data1_);
13    log.add("data2_" + data2_);
14 }

```

Figure 4.22: The Update Method in Bold Stroke BM\_\_ClosedEDComponentImpl.cpp (Enhanced with the Logging Aspect)

Table 4.2 summarizes the comparison of the manual effort versus the MDAA approach to perform Bold Stroke source evolution. The experiment was conducted on 25 components contained in the Bold Stroke avionics system. Three specific evolutionary changes were applied, i.e., concurrency requirement, logging support and assertion checking. Assertion checking is a requirement that adds the Design-by-Contract (DBC) [134] paradigm to certain methods in the system. Details can be found in [195].

The comparison is based on the number of places that need to be changed and

Table 4.2: Comparison of Manual Modification to the MDAA Approach

	Manual Changes (No. of Places)	Aspect Model Interpreter (LOC)	ECL (LOC)	Generated RSL (LOC)
<b>Concurrency</b>	50	257	25	1250
<b>Logging</b>	200	305	37	3750
<b>Assertion</b>	100	270	33	2500



the lines of code that are involved in different MDAA components. As listed in Table 4.2, for instance, the full support of logging (including Log on Read/Write/Method Exit/Entry) would require manual instrumentation of the logging statements at 200 different locations that span over 25 different source files. One requirement change would result in the system maintainer revisiting all of these 200 different places and making corresponding modifications one by one, which is very time-consuming and error-prone. However, by adopting the MDAA approach, the change requirement is encapsulated within an ECL aspect specification module in a much more concise and localized manner (only 37 lines of ECL code in one file). Furthermore, the logging aspect interpreter only needs to be written once and is able to generate the low-level DMS RSL rules from various configurations of logging properties as specified in the logging models (which are also automated from the ECL specification by C-SAW).

The table provides some insights into how much effort the MDAA approach would need in order to conduct the same evolutionary changes as manual modification. Initially, the MDAA approach seems to involve more work (i.e., define the aspect metamodel, create the aspect interpreter and specify the ECL aspect specification) than simple source editing. However, the aspect metamodel and interpreter are only defined once by the aspect developer at the beginning and the rest of the evolution tasks are conducted by domain experts by using the modeling constructs and ECL specifications, as well as by invoking the interpreter to derive low-level rules. This is called “Write Once, Derive N” (adapted from the “Write Once, Deploy N” pattern from [75]), which provides a systematic solution to legacy evolution by reducing the accidental complexity in the manual and ad-hoc approach.

#### *4.2.5 Discussion*

This section presents an instantiation of the MDAA framework to support legacy evolution in terms of domain-specific models. The primary benefit of adopting DSM

is to allow domain experts to adapt the underlying source system from domain concepts that are specified in customized models. Domain experts do not have to understand programming concepts because the models are automatically synthesized by the MDAA interpreters.

This approach, however, requires each new aspect metamodel to be associated with a new MDAA interpreter. This is because each specific new aspect metamodel introduces different types of modeling elements, syntax and semantics that are unique to that particular aspect domain, which require different interpretation for generating different sets of aspect code for evolving the legacy system. As Figure 4.23 illustrates, the Concurrency aspect model requires a different interpreter than the Logging aspect model.

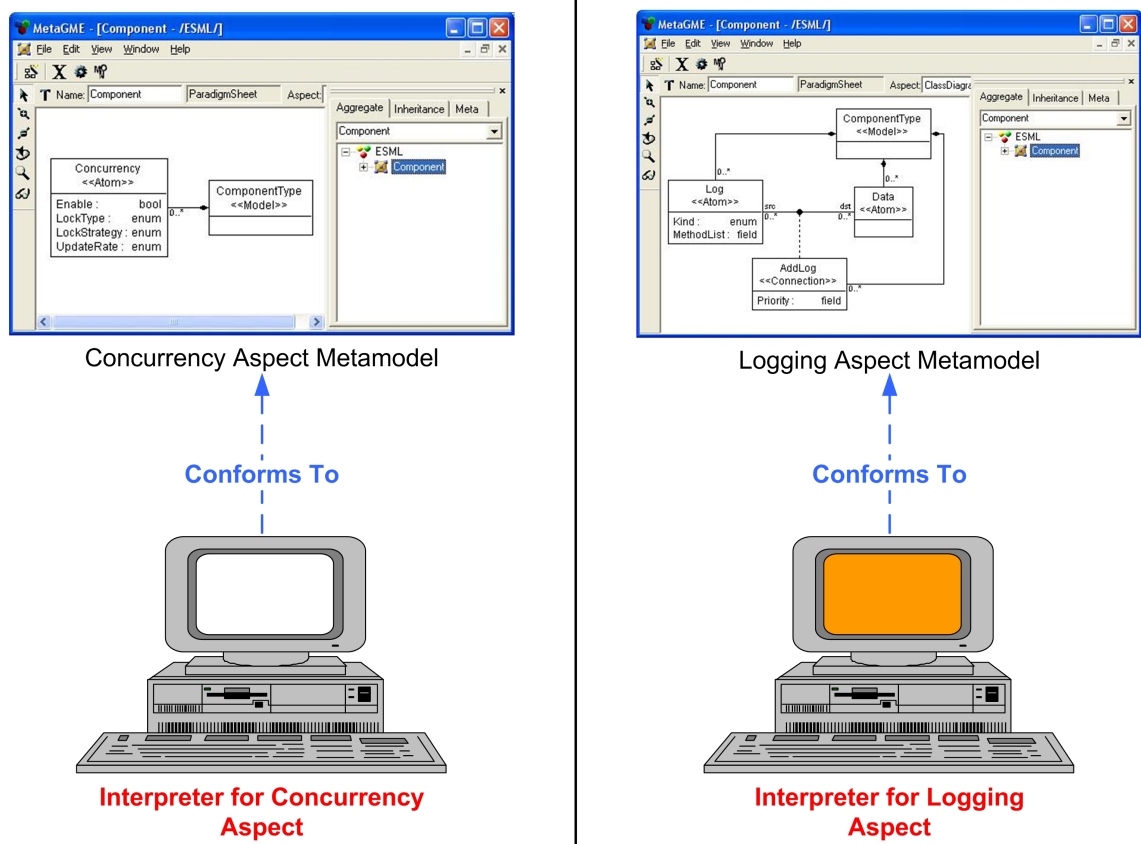


Figure 4.23: MDAA Interpreters for Different Aspect Models

The next section will introduce a more generic MDAA approach based on the mechanism of activity modeling, which only requires one aspect metamodel and one interpreter for aspect code generation.

### 4.3 Activity-Based System Evolution through MDAA

This section presents a UML (more specifically, UML activity modeling) based approach to realizing the MDAA framework. The approach is illustrated through a case study on adding failure handling activities to the Intelligent Network Fault Management (INFM) system [123] (see the INFM background introduction in Section 3.4.1). The experimental case study is implemented using the AOAM mechanism that was previously presented in Chapter 3 in combination with AOP [109].

Figure 4.24 shows the flow chart for the approach. Similar to Figures 4.1 and 4.6, the evolution process starts with model extraction. At this time, an activity model that abstracts the control flow logic of a system is constructed using the well-defined activity constructs (as described previously in Figure 3.1). Then, an aspect-activity model is specified to capture the new requirement changes that are to be applied to the system. The aspect-activity model is constrained by the definition of the aspect-activity profile (as shown in Figure 3.3). Subsequently, the aspect code generator navigates and parses the aspect model and produces the corresponding aspect code (e.g., written in AspectJ), which is in turn fed into the underlying AOP weaver (e.g., AspectJ weaver), along with the original base code. The generated enhanced version of the source code can be simulated and validated against the enhanced model that results from the model-level aspect weaving via the AOAM weaver (as introduced in Chapter 3).

The next two subsections will present the adoption of the activity-based MDAA approach on the INFM case study. The failure handler aspect models are introduced in Section 4.3.1, and Section 4.3.2 will demonstrate the generated aspect code.

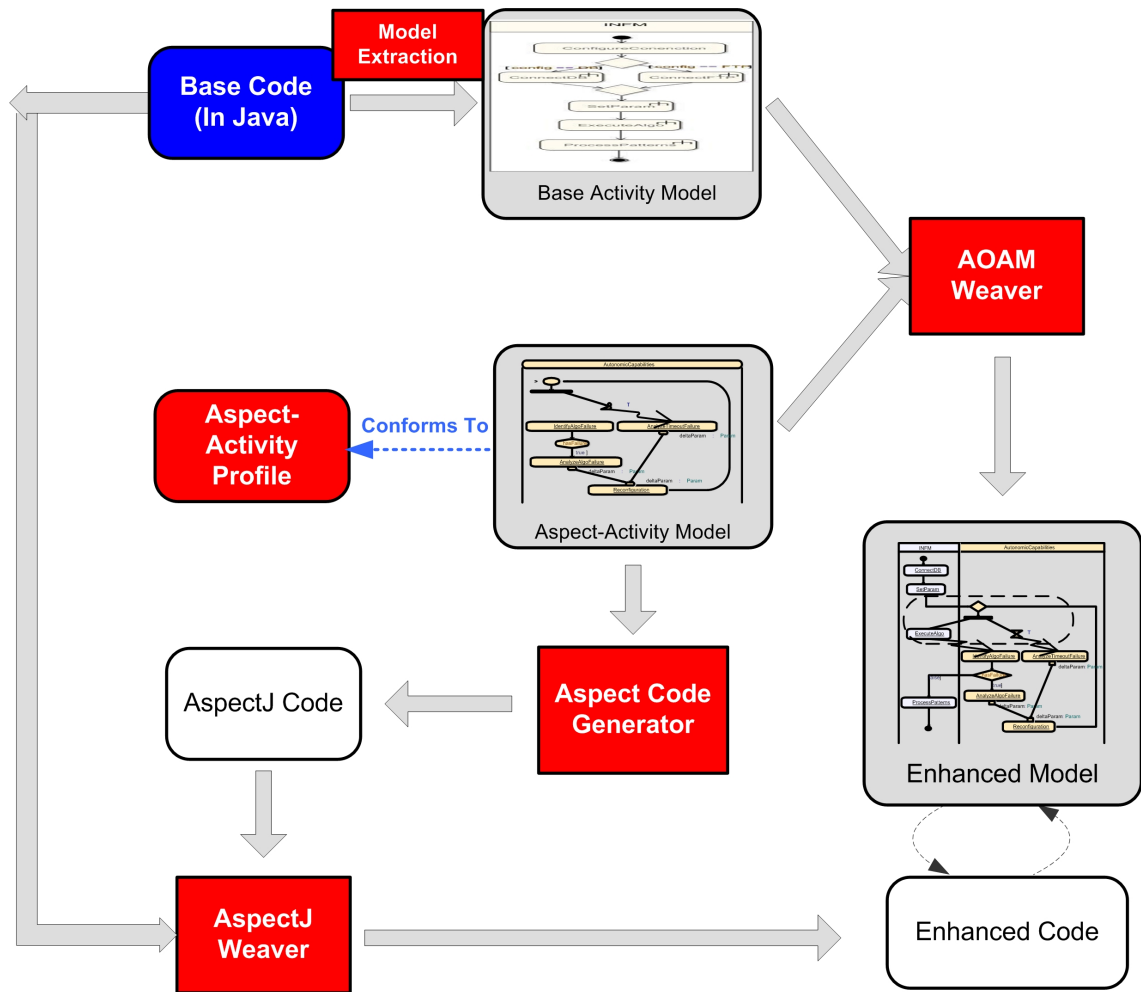


Figure 4.24: Activity-Based System Evolution through MDAA

#### 4.3.1 Modeling Failure Handler Aspects

During our experiment, we first studied the INFM system as well as the potential failures that might occur during the system execution and their corresponding resolutions. Then, a fragment of the INFM system (i.e., alarm correlation activity) was manually extracted for applying failure handling features (as previously shown in Figure 3.11 in Chapter 3). The performance of the alarm correlation algorithms is very sensitive to the settings of both parameters of the pattern discovery algorithms, namely the minimum support value and the minimum confidence value, denoted by *supp* and *conf* in the rest of this section. Table 4.3 lists the run-time failures that have occurred during operations that were not covered in the original system design, the possible causes of these failures, and the intended resolutions for the failures. These three major failure conditions listed in the table are considered dependability critical because:

- It is imperative not to violate the real-time constraints on operations for the on-line correlation algorithm as well as to avoid the possible operational problems caused by resource exhaustion for the off-line correlation algorithm.
- Relatively small values of *supp* and *conf* might cause the algorithms to return a large number of patterns. Although small values of both parameters would provide a more comprehensive list of candidate patterns, it raises issues in handling the large quantity of patterns. Furthermore, low *supp* and *conf* values increase the likelihood of applying false positive patterns to the succeeding correlation procedures.
- Relatively large values of *supp* and *conf*, on the other hand, might cause the algorithms to return too few patterns. This can negate the effectiveness of the correlation algorithms and increase the false negative ratio in pattern validation.

Table 4.3: INFM Software Failures and Resolution Table for Alarm Correlation

Failure	Possible Causes	Resolution
Timeout	<i>supp</i> set too low	Increase <i>supp</i> and re-run
Too many( $> U$ ) patterns returned	<i>supp</i> or <i>conf</i> set too low	1) Increase <i>conf</i> and recheck patterns; if too many( $> U$ ) patterns returned; 2) Increase <i>supp</i> and re-run.
Too few( $< L$ ) patterns returned	<i>supp</i> or <i>conf</i> set too high	1) Decrease <i>conf</i> and recheck patterns; if too few( $< L$ ) patterns returned; 2) Decrease <i>supp</i> and re-run.

The timeout failure is considered critical for on-line execution. Since it is less hazardous for off-line alarm correlation, the allowed times for re-run could be adjusted to a larger number. Similarly, too few patterns and too many patterns might have different risks under different operational modes. The adjustment of parameters in both situations needs further analysis, whereby a failure analyzer is required to carry out the predictive reasoning based on current environmental conditions and constraints, as well as the history performance of the algorithms. In other words, the failure analyzer is indeed an adaptive component that fine-tunes the behavior and strategy of the self-healing actions.

In Chapter 3, Figure 3.12 shows an aspect-activity model for managing timeout failure. Similarly, the pattern failure management aspect is modeled as in Figure 4.25. In contrast to the timeout failure, the pattern failure is identified by a special component after the *proceed* algorithm is completed. If a failure is detected, *RecheckPatterns* will filter the returned patterns by adjusting the value of *conf*. If

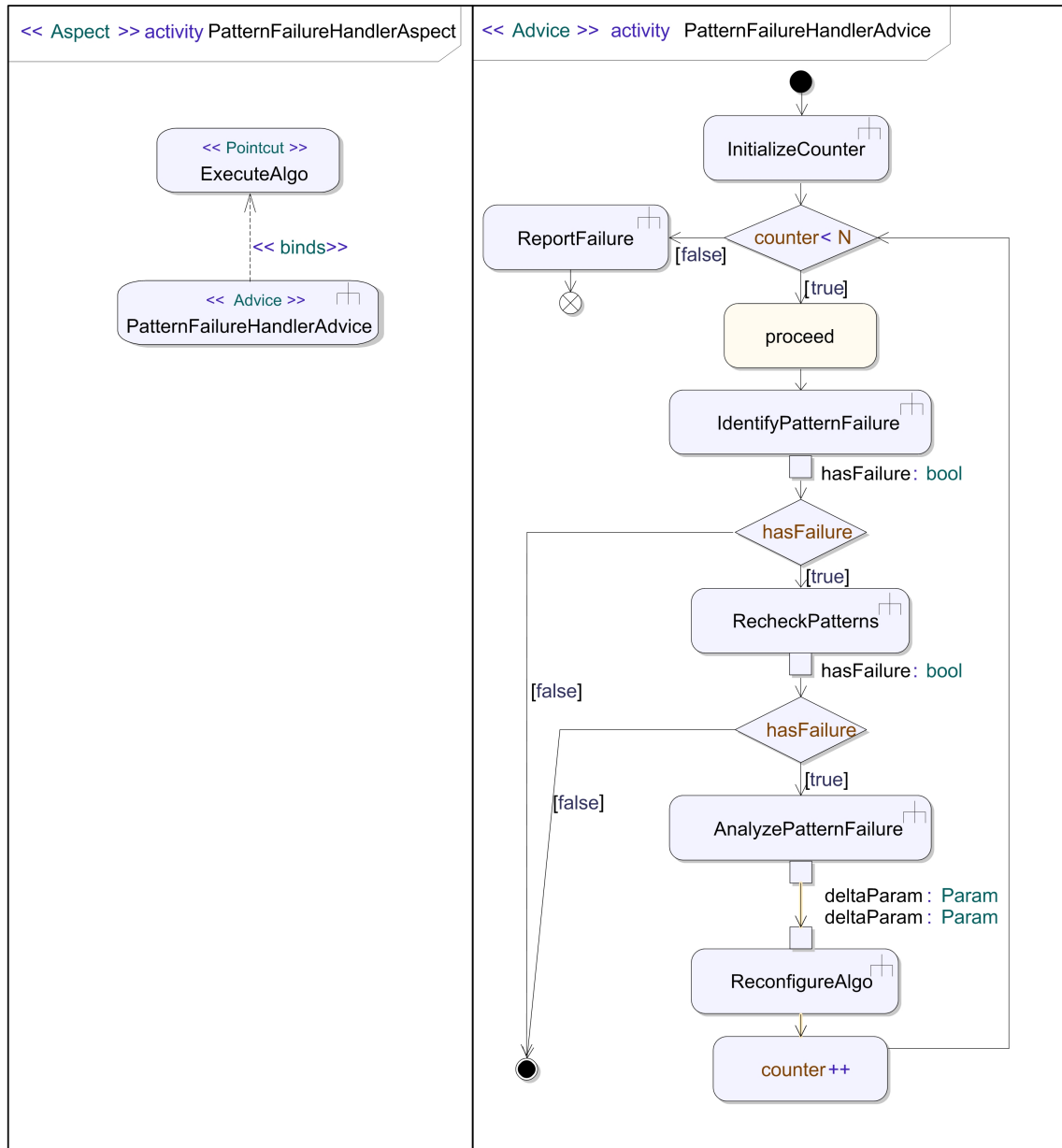


Figure 4.25: Pattern Failure Management Aspect Model

the failure still exists, the control flow will be passed to the pattern failure analyzer/mitigator, which will adjust the algorithm parameters based on the reasoning of different failure types, as shown in Figure 4.26.

#### 4.3.2 Aspect Code Generation

AOP [109] is applied to support source code composition. The failure handling aspect models are translated into an aspect program in AspectJ [108] by a specialized aspect-activity code generator. The *pointcut* in this case refers to the operation `ExecuteAlgo`. The failure detection and resolution activity is captured in an *advice*. The base code for the alarm correlation process is augmented with the failure handling instrumentation by a weaver. (In this particular case study, AspectJ [108] is chosen as the underlying weaver, because INFM is implemented in Java and AspectJ is a mature aspect weaver for Java.) As a result, a failure handling enabled INFM system is constructed from the high-level model specifications. An example of the generated AspectJ code is depicted in Figure 4.27. The `PatternFailureHandlerAspect` introduces four fields (from Line 3 to Line 6) that are necessary to implement the pattern failure

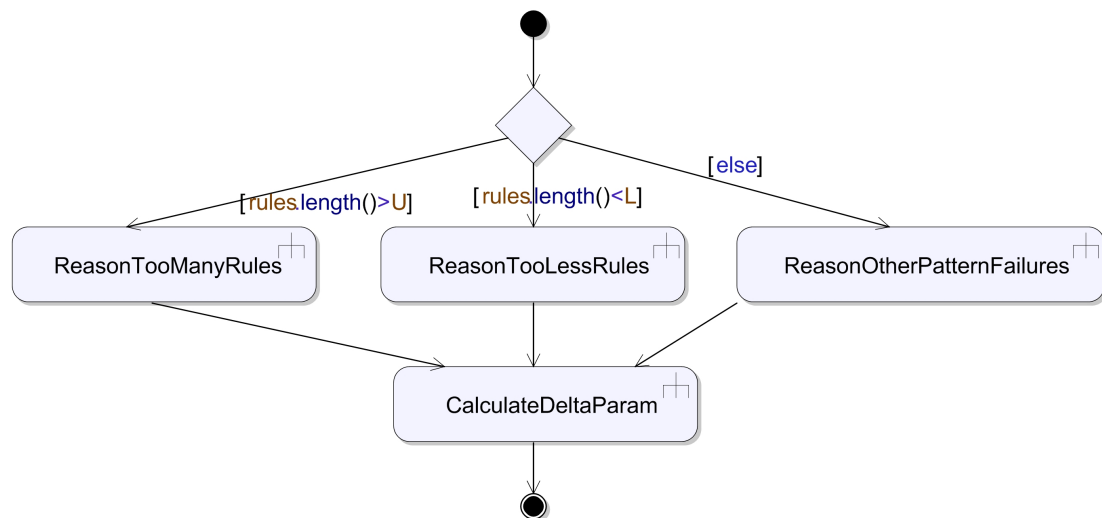


Figure 4.26: Pattern Failure Analyzer Model



```

1  aspect PatternFailureHandlerAspect
2  {
3      int N = 10;
4      int counter = 0;
5      boolean hasFailure;
6      Param deltaParam;
7
8      pointcut pct() : call(ExecuteAlgo(...));
9
10     around() : pct() {
11
12         while (counter < N) {
13             proceed();
14             hasFailure = IdentifyPatternFailure();
15             if (hasFailure) {
16                 hasFailure = RecheckPatterns();
17                 if (hasFailure) {
18                     deltaParam = AnalyzePatternFailure();
19                     ReconfigureAlgo(deltaParam);
20                     counter ++;
21                 }
22                 else { break; }
23             }
24             else { break; }
25         }
26     }
27 }

```

Figure 4.27: The Generated AspectJ Program for the Pattern Failure Handler Aspect

handling capability. The pointcut `pct` picks out each join point that is a call to the `ExecuteAlgo` method regardless of the method's parameters. The `around` advice (Line 10) runs in place of its join point picked out by the pointcut `pct`. Within the advice, four methods (i.e., `IdentifyPatternFailure`, `RecheckPatterns`, `AnalyzePatternFailure`, and `ReconfigureAlgo`) will be invoked given different conditions. The body/implementation of these methods can be either denoted using another activity model (e.g., `AnalyzePatternFailure` is specified further in Figure 4.26), or embedded with a source code fragment written in a GPL (e.g., Java).

### 4.3.3 Discussion

This section presents an activity model-driven approach to realize the MDAA framework. In contrast to the DSM-based MDAA as described in Section 4.2, activity-based MDAA only requires one aspect metamodel because the interpretation for the activity modeling elements is well-defined and fixed. Activity modeling, however, is only well-suited to specify process and control flow in a standardized fashion. It does not possess the expressive power for elaborating other domain-specific concepts when compared to the DSM approach.

## 4.4 Aspect Mining from a Modeling Perspective

For legacy software to benefit from AOSD, it is necessary to analyze the existing implementation to discover the crosscutting concerns and refactor them into aspects. The research on aspect mining refers to the identification and analysis of non-localized crosscutting concerns throughout an existing legacy software system [43]. The ultimate goal of aspect mining is to support aspect-oriented refactoring [113] to improve software comprehensibility, reusability and maintainability.

The challenges of aspect mining are focused along three separate phases:

- **Aspect Identification**

This phase is concerned with an analysis task that leads to identification of a suggested set of candidate aspects. This phase may require user interaction to provide initial seed information, or to assist in sifting through false positive noise (i.e., suggested aspects that are not really representative of a crosscutting concern).

- **Aspect Extraction**

After a set of candidate aspects has been identified, the crosscutting concern

must be extracted from the existing representation (i.e., all of the locations in the legacy software where the aspect appears must be removed).

- **Aspect Refactoring**

After extracting the aspects from the base representation, an equivalent aspect must be specified in an aspect language in order to preserve the initial functionality. The result is improved modularization as captured in the newly created aspect.

Most of the current research on aspect mining focuses solely on the implementation as applied to source code. However, an aspect-oriented approach can be beneficial at various levels of abstraction and at different stages of the software lifecycle. For instance, research in AOM [1] has the potential to help define common characteristics (which are encapsulated within aspects) from a perspective that is at a more abstract level. For existing models to benefit from AOSD, it is indispensable to perform reengineering techniques, such as aspect mining, at many different stages throughout the software development lifecycle.

This section presents our initial investigation into raising the benefits of aspect mining to a higher level of abstraction through application of aspect mining algorithms to domain-specific models. Specifically, the section describes an approach to tackle the aspect identification problem at the modeling level, rather than at the source code level. A clone detection technique is applied to identify crosscutting concerns that are represented as similar or duplicated (clone) modeling elements, which assists in modularizing a design through aspects before proceeding to the implementation level. Furthermore, our experience has led us to believe that aspects are easier to identify at the modeling level, because much of the accidental complexities of implementation concerns have been removed in the corresponding modeling abstractions.

The remainder of this section is structured as follows. Section 4.4.1 presents a clone detection-based approach aspect identification. Section 4.4.2 offers a case study

by applying clone detection to identify crosscutting concerns in ESML models.

#### *4.4.1 Clone Detection for Aspect Modeling Mining*

Pattern matching and clone detection are the two main approaches that can be applied for aspect identification. The pattern matching process is conducted by a human designer who suspects the existence of aspects in a model. The designer has to comprehend the domain information contained in a model and provide a pattern (“seed”) to indicate properties of potential aspects. Such a seed serves as the starting point for discovering all matched concerns. The seed can be represented based on a textual expression or on graphical models, as described in [196].

The pattern matching techniques assist users in efficiently locating predefined crosscutting concerns. However, users of pattern matching are required to have a considerable amount of knowledge about the domain and overall model structure. The users must input a particular format of seed so that the aspect mining process can be partially automated. Moreover, pattern matching cannot explore unknown classes of crosscutting concerns (i.e., those for which no seed is known) and will often result in missing some desirable aspects. In order to overcome the deficiencies of pattern matching, a clone detection technique has been developed for aspect mining that is applied to models.

Various clone detection techniques (as referenced in Section 5.3 of Chapter 5) have been investigated to detect duplicated source code. The intention of applying clone detection for aspect mining is to reveal the unknown crosscutting concerns through full automation of the aspect mining process. In terms of modeling, clone detection identifies the similar (clone) model fragments throughout the model hierarchy. The similarity of elements of sub-models are determined based on one of the three levels of similarity described below in terms of metamodeling concepts.

In the context of metamodeling, an atomic modeling element (e.g., an atom in

GME) is defined by a combination of its type, name, and set of attributes. Correspondingly, a model consists of a set of elements, including atoms, sub-models or connections. Three levels of similarity can be defined based on the type, name, and attribute (see Table 4.4) of the model elements:

- Level 1 indicates the most liberal policy (i.e., two atoms are considered clones as long as they have the same type; two models are clones if they own the same type and all of their elements are correspondingly Level 1 clones).
- Level 2 represents a moderate clone detection philosophy (e.g., two connections are considered clones if their source and targets are Level 2 clones, in addition that each connection has the same type and name).
- Level 3 defines the most stringent rule (i.e., two models are considered clones only when they hold the same type, name, and attribute set; furthermore, all of their elements should be correspondingly recognized as Level 3 clones.)

Table 4.4: Three Levels of Similarity

	<b>Atom</b>	<b>Model</b>	<b>Connection</b>
<b>Level 1</b>	Type	Type Elements	Type Source Target
<b>Level 2</b>	Type Name	Type Name Elements	Type Name Source Target
<b>Level 3</b>	Type Name Attributes	Type Name Attributes Elements	Type Name Attributes Source Target

Based on the above levels of similarity, the four steps of the clone detection algorithm for models are:

## 1. Metamodel preprocessing

The first step involves the partition of the metamodel entities into different groups that need to be compared. Each group includes a set of the type pairs, such as  $\{\text{Type-model}\} : \{\text{Type-element}\}$ , where  $\{\text{Type-model}\}$  is a collection of types whose model instances comprise some common elements, and  $\{\text{Type-element}\}$  is the collection of model elements that  $\{\text{Type-model}\}$  share. Because  $\{\text{Type-element}\}$  is contained by more than one model, it has the potential to become one of the selected crosscutting concerns.

As shown in Figure 4.28, the type **ModelA** and **ModelB** share the element **AtomAB**. **ModelB** and **ModelC** both contain **AtomBC**. So the partition of the illustrated metamodel would be:

$$\{\text{ModelA, ModelB}\} : \{\text{AtomAB}\}$$

$$\{\text{ModelB, ModelC}\} : \{\text{AtomBC}\}$$

The preprocessing of the metamodel partition facilitates the desired steps of the algorithm, because only those models that have the same type or fall into the same group will be compared. Furthermore, only the shared elements of

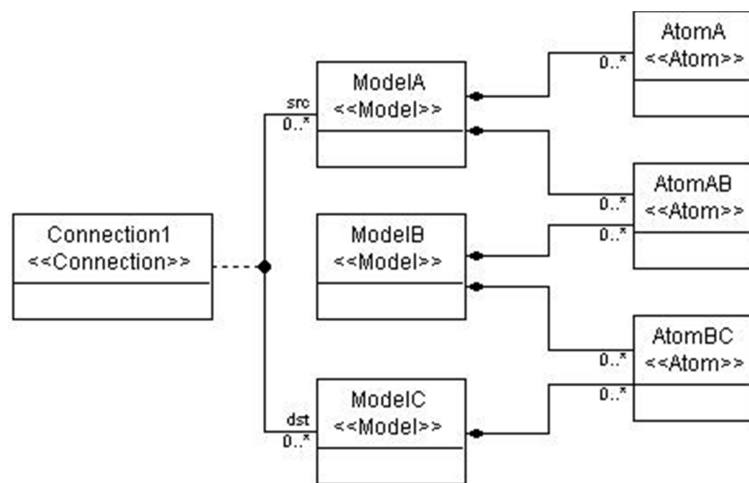


Figure 4.28: A Metamodel Example

the two models should be compared. For example, imagine that there is one instance of **ModelA** and one instance of **ModelB**. In such a case, we only need to consider whether their shared atoms (instances of **AtomAB**) are clones. Any other irrelevant elements will not be considered.

## 2. Model fragments comparison

The second step of clone detection in models determines if the elements of a sub-model pair are clones by comparison. From the root of the model hierarchy, each sub-model is compared with the other sub-models that either have the same type or fall into the same group in step 1. As an example, suppose a comparison is to be made between sub-model instance X and sub-model instance Y:

- If X and Y are of the same type, every element inside should be compared correspondingly. The comparison is based on the choice of the level of similarity as defined in Table 4.4. Each time, the atoms are compared first, then the models, followed by the connections.
- If X and Y are in the same group, only their shared elements need to be compared.
- If X and Y do not have the same type, and do not fall into the same group, it means that they cannot have an intersection; thus, further comparison is not necessary.

## 3. Maximally similar model fragments grouping

For all of the clone elements that sub-model instance X and Y share, we group them together as a common property named  $P$ , which is considered as the maximally similar model fragments of X and Y. If  $P$  is not null, the next task is to find out whether  $P$  is already stored in the list of maximally similar fragments. An efficient way to search for commonalities on a list is to construct a hash function  $h(P)$ , which computes the number of a bucket (hash value)

based on  $P$  [28]. The hash function will always return the same bucket number given the same  $P$ . If  $P$  is not in the bucket  $h(P)$ , then  $X$ ,  $Y$ , and  $P$  will be added to this bucket. If  $P$  is already in such a bucket, only  $X$  or  $Y$  will be added into the collection of the sub-models that share the same property  $P$ .

#### 4. Aspect filtering

The maximally similar model fragments generated from the above steps (i.e., the initial result of the clones) may contain too much noise and need to be refined further (i.e., many false positives could be suggested, which can be removed on further analysis). For instance, based on our experimentation we found that if one model entity in a maximally similar model fragment group has a connection (in or out) that does not fall into the same group, then this model entity is seldom considered as an aspect and can be filtered out.

##### 4.4.2 Case Study: Aspect Mining in ESML

This section presents a case study that applies clone detection for aspect mining on the ESML [107] models (see the introduction of ESML in Section 4.2.2). As previously described in Figure 2.13 of Chapter 2, an ESML model has a tree-like hierarchical structure. The model on the first layer is the root of `InteractionModel`, which specifies a particular scenario that involves certain configurations of various sub-models. These sub-models belong to the second layer. In this figure, only two component sub-models are depicted on the second layer (e.g., `BM_UserInputComponentImpl` and `BM_OpenEDComponentImpl`). Several models and atoms representing the containments of the second layer models are depicted separately on the third layer (e.g., `BM_OpenFunctionalFacetInterface` represents an interface for the component `BM_UserInputComponentImpl`). The fourth layer is the last layer shown in Figure 2.13 (e.g., the `SetData1` atom denotes a method object that is contained by the corresponding component interface model `BM_OpenFunctionalFacetInterface` and `BM_`



OpenFunctionalFacet). A solid line between any two layers represents containment, and a dotted line with an arrow represents connections that may occur on the same layer or across layers.

In the case where users have no knowledge of the system (or, they have some knowledge, but not enough to express textual or graphical patterns), the clone detection technique for aspect mining may be applied to suggest possible aspects within an ESML model. The level of similarity is set to Level 2 (i.e., only compare the type and the name, without considering the attributes) for this particular case study. After applying the algorithm, the maximal similar model fragments of `BM_UserInputComponentImpl` and `BM_OpenEDComponentImpl` are:

```
{data2_, Data2Cond, AddCondition}
{data1_, LogOnRead, AddLog}
{InternalLock}
{ANY_sub, ANY_ref, EventTyping}
{ANY_pub, ANY_ref, EventTyping}
```

The last two groups both contain model entities that carry connections out of the group (e.g., `ANY_sub` in `BM_UserInputComponentImpl` and `ANY_pub` in `BM_OpenEDComponentImpl`). Therefore, these two elements (as well as their relationships in the group) should be filtered out. Thus, the algorithm identifies the resulting aspect candidates for the three component models as:

```
{data2_, Data2Cond, AddCondition}
{data1_, LogOnRead, AddLog}
{InternalLock}
```

As the additional concerns that were identified automatically by our algorithm, consider the interface models `BM_OpenFunctionalFacetInterface` and `BM_OpenFunctionalFacet`, whose maximal similar model fragments are:

```

{SetData1}
{SetData2}
{operator new}
{operator delete}
{SetMemoryStore}
{GetMemoryStore}

```

In this example, **SetData1** will be removed in the filtering process because it has connections coming into or going out from the group. Consequently, the rest of the five atoms indicate the clone methods in the interface models and can be regarded as the potential aspect candidates (as a matter of fact, these five atoms appear in 7 different interface models).

## 4.5 Summary

In this chapter, two different modeling techniques, i.e., DSM and UML activity modeling, have been adopted to instantiate the MDAA framework for evolving legacy software systems. By applying evolutionary changes to the aspect models at a higher level of abstraction, low-level aspect code can be derived and woven into the legacy source to perform large scale adaptation.

There has always been an active debate between DSM and UML since the emergence of MDE. Table 4.5 compares these two approaches in terms of the MDAA realization. As stated by Booch et al. [37], the full value of MDE “is only achieved when the modeling concepts map directly to domain concepts rather than computer technology concepts.” DSM is best known for achieving this goal by allowing domain experts to work directly using the concepts that are familiar to them. Therefore, one domain is accommodated with one specific well-defined metamodel. On the contrary, UML is a general-purpose modeling language that uses graphical notations to create abstract models of a system from different perspectives. It is often considered as a

visualization of the low-level implementation concepts in terms of source code and it contains a set of fixed metamodels that are generic to all kinds of different domains.

Table 4.5: Comparison of DSM-based and UML-based MDAA

	<b>DSM-based MDAA</b>	<b>UML-based MDAA</b>
Base Metamodel	Domain-specific	Domain-generic
Model	Close to the domain concepts	Close to the implementation concepts
Aspect Metamodel	Multiple for one base metamodel	Only one for one base metamodel
Aspect Code Generator	Multiple for one base metamodel	Only one for one base metamodel

Although it is claimed that DSM is 5 to 10 times more productive than the UML approach [5], the benefit does not come without compromise. In DSM, code generators have to be customized for each domain in order to produce quality and efficient code, which requires more tooling effort when compared to UML, which offers a standardized way to interpret the fixed metamodels with well-defined semantics. When DSM is applied to the MDAA framework to support legacy evolution, each new evolutionary change requirement must be addressed by one aspect metamodel together with one aspect code generator, because each one of the change requirements introduces new domain concepts that necessitate different mapping mechanisms to the underlying aspect code. This can be considered as an *augmented* modeling approach as previously introduced in Section 2.3.1. In the UML-based approach, only one aspect metamodel (defined by a special UML profile) and one aspect code generator is required, which falls into the lightweight *constrained* modeling approach (as introduced in Section 2.3.1).

The MDAA framework, including the Bold Stroke experimental case study, has been published in [83, 190, 193, 195]. The mechanism and case study for the activity-based system evolution, as presented in this Chapter and Chapter 3, have been covered

in [123, 191, 192, 198]. Other contents related to MDAA have been presented in [196] (aspect mining on models), [102, 125] (a modeling framework for constructing self-healing software systems) and [194] (model-driven interpreter evolution).

## CHAPTER 5

### RELATED WORK

In this chapter, related works from different perspectives are presented. Section 5.1 discusses different techniques that relate to aspect and activity models. Section 5.2 compares state-of-the-art approaches for legacy evolution through model-driven techniques. Section 5.3 gives an overview of the existing aspect mining techniques.

#### 5.1 Research on Aspect-Oriented Activity Modeling

Although aspect-orientation originated at the programming language level [109], it now extends to other software lifecycle stages and is applied to different levels of software abstractions. For example, there is a growing community investigating Aspect-Oriented Modeling (AOM) [1] techniques, providing various concepts, notations and mechanisms to handle crosscutting concerns at the modeling level. This section summarizes some of the existing research that relates to aspect and activity models.

Barros et al. [23] propose a graphical composition operation supporting the addition of crosscutting requirements in activity models through node fusion, addition, and subtraction. In contrast to our current implementation of the aspect-activity extension, their approach considers all types of activity nodes as potential join points (i.e., not only action nodes, but also object and control nodes). However, as their approach is based on pure graph composition theory, it lacks semantic support for non-

graphical activity node specification, such as reflective APIs and regular expression-based operation pattern matching.

Charfi et al. [45] introduce an aspect-oriented extension to Business Process Execution Language for Web Services (BPEL4WS) - a variation and application of the activity modeling language. With web service composition captured in aspects, dynamic adaptation of composition logic can be supported. In their model, each BPEL activity is a possible join point during the execution of processes. Their aspect language is similar to ours except that they use XPath (a query language for XML documents) as the pointcut designator language, whereas our aspect-activity models are based on MOF/UML specification, which is more generic and can be applied to any activity model based on UML.

Solberg et al. [163] present a Model-Driven Engineering (MDE) framework that uses aspect-orientation to facilitate separation of concerns. The primary and aspect models defined in a platform-independent manner are transformed to platform-specific models through separate mappings. The resulting models are in turn composed to obtain an integrated design view. Unlike our approach, they do not provide an explicit support for modeling aspect constructs. Instead, the weaving mechanism is controlled by using extra composition directives that instantiate aspect models and bind them to the primary models.

Grassi et al. [76] propose a UML-based graphical notation for specifying aspects for the static and dynamic structure of the system model. Similarly, Cazzola et al. [44] present a high-level join point selection mechanism, which decouples the aspect definition from the base program structure and syntax. Both approaches focus on using activity models to represent pointcut patterns, rather than provide aspect support to activity models.

## 5.2 State-of-the-Art on Model-Driven Legacy Evolution

This section summarizes some related research in the area of software evolution in terms of model-driven techniques.

### 5.2.1 *Architecture-Driven Modernization*

The MDAA approach falls into the concept of Architecture-Driven Modernization (ADM) [88], which is the process of understanding and evolving existing software assets by leveraging some existing Object Management Group (OMG) modeling standards and its Model Driven Architecture (MDA) [164] initiative<sup>8</sup>. ADM aims at various purposes, such as software improvement, modification, interoperability, refactoring, restructuring, reuse, porting and migration.

Software modernization (i.e., “the process of understanding and evolving existing software assets” [69]) is driven by the need to capture and retool various architectural aspects of existing application systems. Three major architectural perspectives have been identified within two domains. The business domain corresponds to “the business architecture (i.e., models and correspondent diagrammatic views of organizational governance, business semantics, business rules and business processes), whereas the IT domain is focused on application, data and technical architectures” [70]. The software evolution at each architecture level requires three elements [70]:

- **Knowledge discovery of the existing solution**

The knowledge obtained from existing software is usually presented in the form of models to which various queries can be made when necessary. ADM comprises a standard called the Knowledge Discovery Metamodel (KDM) [92], which serves as one of the foundations to provide a common repository structure to facilitate the exchange of software assets. KDM is a metamodel that covers a large

---

<sup>8</sup>It is interesting to note that the acronym ADM looks like MDA in reverse, and in reality ADM does involve reverse engineering in that it incorporates an extraction of architectural models followed by applying the MDA forward engineering process for legacy system modernization.

and diverse set of applications, platforms, and programming languages. Therefore, it is a uniform language and platform-independent representation. However, it also allows addition of domain and implementation-specific knowledge with its extensibility mechanism. Although KDM provides a comprehensive view of application structure and data, it does not consider software elements below the procedure level, which is instead captured by another foundation of ADM – the Abstract Syntax Tree Metamodel (ASTM) [91]. ASTM unifies all syntactical language constructs into a common abstract syntax metamodel. The KDM and ASTM are complementary to each other, with the intent to enable a comprehensive representation of applications at different abstraction levels and facilitate the exchange of various metadata across multiple software artifacts.

- **Target architecture definition**

In order for the system to be modernized, a clear definition of the target architecture is required. This phase is the pivotal and most demanding one in ADM [30], because the architecture should be formulated based on the requirements from various perspectives as well as the existing models of business domains and IT technologies.

- **Transformative steps that move from the source architecture to the target architecture**

This phase completes the ADM process in providing a transformational bridge between existing systems and target architectures. The MDA model transformation techniques (e.g., MOF QVT standard [94]) can be leveraged to enable the mappings and transformation between the source architecture models and target architecture models. Consistency between different abstraction levels of the models must be maintained during each transformation step.



### 5.2.2 *Model-Driven Legacy Migration*

Research on model-driven legacy migration [112] proposes to use a variety of models to define, analyze, and execute legacy migration procedures. By leveraging the traditional migration process with the modeling approach, the migration models streamline the process of legacy migration through a set of models. The application aspect model acts as a container to define a software application in terms of source code, domain concepts, architecture, and life cycle processes. The data layer model captures the persistence aspect of the system, such as the storage mechanism and schema. The platform model specifies the environment within which the system is operating. Finally, the migration mapping model describes a set of activities, methods, and tools that transforms a source model into the target model.

The model-driven legacy migration research claims that the use of migration models can accommodate the migration requirements from all different types of stakeholders (e.g., users, vendors, and administrators). Furthermore, with the clearly-defined migration activities through the migration mapping models, the migration project can be well-planned and estimated. The approach, however, does not take into account the mapping problems between the existing systems and the corresponding high-level models, nor does it provide any tool support or real-world implementation.

### 5.2.3 *Model-Driven Modernization of Complex Systems*

Similar to the OMG's ADM, the project "MOdel driven MOdernization of Complex Systems (MOMOCS)" [9] aims at investigating the methodology and related tools for fast re-engineering of complex systems (i.e., systems that are composed by an interconnection of hardware, software, user interfaces, firmware, business and production processes). It leverages OMG's modeling standards, and provides a model-based solution to help abstract away unnecessary details and concentrate on the characteristics of interest of the systems that need to be modernized [20].

Figure 5.1 illustrates an overview of the modernization process that is adopted by MOMOCS. The process model is called XIRUP (eXtreme end-User dRiven Process). It starts with a preliminary evaluation on the underlying system to assess whether the modernization is feasible. If so, knowledge of the system will be acquired during the system understanding phase for identifying and quantifying the components (features) that need to be improved or added. Then, transformations are applied to the existing components in order to build the new adapted ones that are deployable. After the transformations are validated, the new components are ready for migration that involves deployment on specific platforms. The whole modernization process is thus carried out in an incremental and iterative manner.

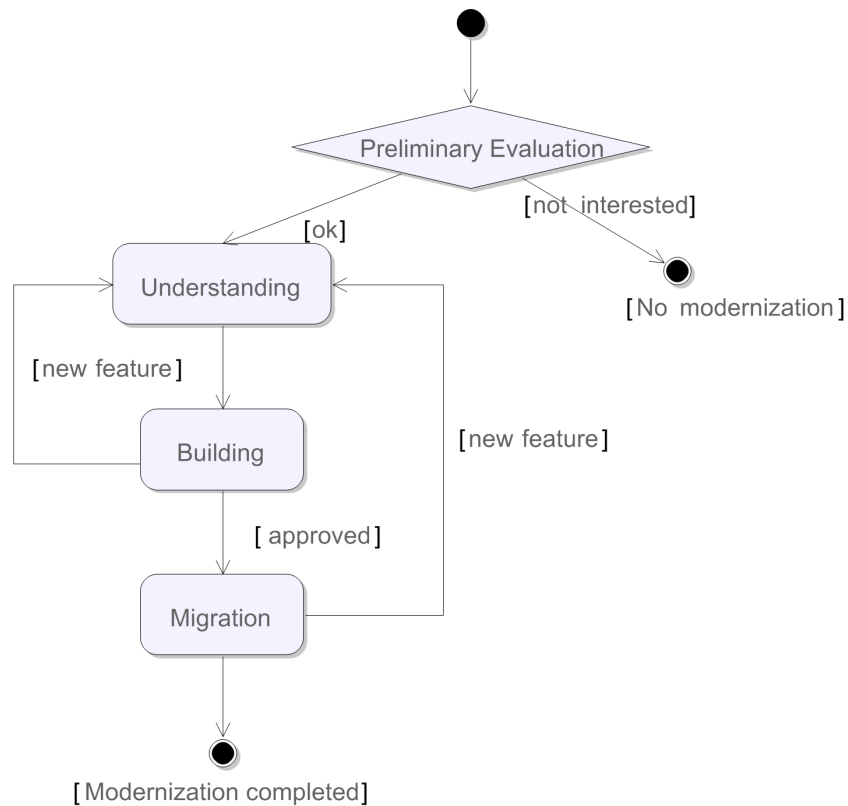


Figure 5.1: The XIRUP Process Model (Adapted from [20])

#### 5.2.4 *Model-Driven Software Evolution*

Another relevant research work is being investigated as a European research project named Model-Driven Software Evolution (MoDSE) [10]. Instead of using model-driven techniques to support legacy system evolution, the goal of this project is focused on developing a systematic approach to supporting evolution for software systems that are particularly constructed via MDE techniques.

As previously introduced in Section 2.3.4 of Chapter 2, the state-of-the-art MDE techniques are well-equipped to support application evolution, which is accomplished through modification of the existing models, from which a brand-new application can be regenerated completely. However, less support exists for environment evolution, which includes metamodel evolution as well as platform evolution (i.e., model interpreter/code generators and application framework changes that reflect new requirements on the target platform). With the intent to fill in this gap, the MoDSE project is dedicated to developing a prototype programming environment that assists software engineers with development and maintenance of MDE systems.

In their recent research on metamodel evolution [184], a heterogeneous approach is developed to support coupled evolution [115] for any scenario of software language evolution. Similar to the model transformation techniques as introduced in Section 2.4, this research provides a generic architecture from which the model-level transformation can be automatically generated from the metamodel-level evolution specification that is defined in a domain-specific transformation language (DSTL). In contrast to the traditional model transformation approaches that rely on a single fixed transformation language, DSTL is tailored for each specific domain (e.g., programming languages, modeling or data modeling) and can be automatically derived from a given meta-metamodel definition.

### 5.2.5 Model-Driven Engineering for Software Migration

Legacy evolution is not only an academic research topic, but also a real problem and key issue in industry and business. Numerous industrial solutions have been proposed and developed to facilitate the evolution of existing software. Sodifrance Inc. [11] is such a company dedicated to providing efficient solutions to modernize large industrial IT systems.

Since 1994, Sodifrance has adopted MDE approaches for modernization projects [67]. A tool suite called Model-In-Action (MIA) [12] has been developed for automating the software migration process through model manipulation techniques. As illustrated in Figure 5.2, the model-driven process is divided into four steps. The first step is to parse the legacy source code into the code model that conforms to the metamodel ( $L$ ) of the legacy application implementation language. Then, reverse engineering is performed to abstract a high-level view and build a platform-independent model (PIM) from the code model. This step is realized by model transformation from the legacy language meta-model ( $L$ ) to a pivot metamodel ( $ANT$ ) that contains built-in packages to represent different viewpoints of the system, such as generic static data structures, actions, algorithms, and graphical user interfaces (GUI). The obtained PIM thus needs to be transformed to a platform-specific model (PSM) of the application in order to fit the target platform requirements. The UML metamodel is adopted as the paradigm to describe a PSM, which is finally translated into the source code of the new application using template-based text generation techniques.

Similar to Sodifrance, Interactive Objects' [8] MDA-based legacy transformation process also consists of four steps, i.e., source parsing, grammar model building, generic model transformation and UML model/code generation [150]. Their solutions are completely compliant to OMG's MDA standard, based on MOF, UML and M2M transformations.

In contrast to these industrial approaches, the MDAA framework is not constrained

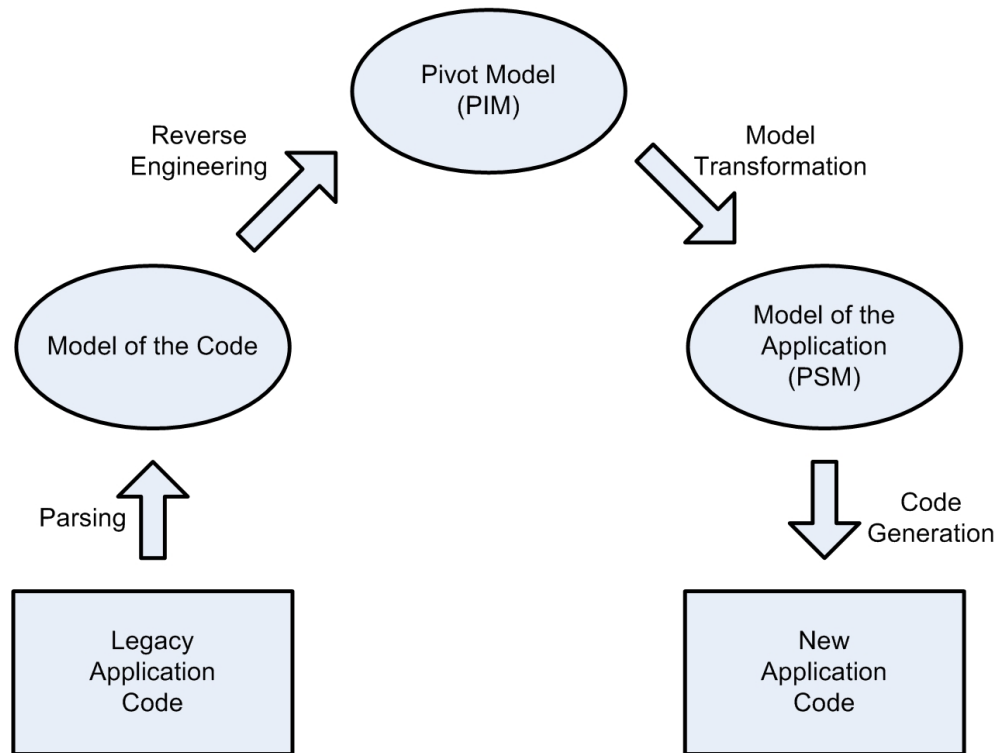


Figure 5.2: Model-driven Migration Process in Sodifrance (Adapted from [67])

by any particular modeling paradigm – there does not exist a single generic meta-model to cover all the perspectives of all the distinct systems. We believe that different application systems and different evolution tasks require disparate modeling solutions. For example, Domain-Specific Modeling (DSM) might be beneficial in describing real-time embedded systems by using the concepts that are directly related to the problem domain, whereas UML-based modeling might be more appropriate in representing application logic that is closer to the implementation level. Furthermore, with the aspect-orientation capability, the application source code does not have to be regenerated as a whole (which is inefficient and sometimes impossible). Instead, only the transformation logic that is represented in the aspect code needs to be generated from the separate aspect models.

### 5.3 Approaches on Aspect Mining

The topics of clone detection and aspect mining have received considerable attention in the research literature. Various clone detection techniques have been developed and implemented to detect duplicated source code. Baker [22] applies a token-based analysis to locate the duplication in large software systems. CCFinder [105] is a tool that also uses a token-based representation of source code to find clones. Baxter et al. [28] use the abstract syntax tree (AST) representation of a source program to find clones through the discovery of similar sub-trees. Our approach is similar to Baxter’s technique. However, since these two approaches are working at different levels of abstraction, they differ on what are compared. Baxter’s approach determines the similarity of sub-trees based on the number of shared and different nodes of the sub-trees. Our approach determines the similarity of elements of sub-models based on one of the three levels of similarity for the model elements. Mayrand et al. [127] use metrics that are calculated from the source fragment to find clones.

As for aspect mining, the current state-of-the-art is represented by the collection of tools described below. All of these tools are focused on source code analysis.

The Aspect Browser [87] enables users to enter regular expressions as patterns to identify aspects. An early contribution of Aspect Browser was an aspect visualizer that graphically conveyed a visual overview of the crosscutting effect of a specific aspect. The Aspect Mining Tool (AMT) [98] augments the Aspect Browser with type-based mining.

In the Prism tool [189], users define a fingerprint that captures a certain property of a crosscutting concern in code. The Prism advisor autonomously computes the crosscutting property of the mining target and returns all of the matches, which are called footprints.

FEAT [151] introduces the concept of a concern graph that localizes an abstracted representation of program elements contributing to the implementation of the concern.

FEAT enables users to perform maintenance tasks that involve non-localized changes. Users initiate the search process by providing a seed, which is expressed through a text file using a declarative language to describe a concern. FEAT generates the concern graph automatically according to the declared concern. Users can visit the source file corresponding to each class in the concern graph.

Ophir [161] is a fully automatic mining and refactoring tool based on the combination of a program dependence graph (PDG) and abstract syntax tree (AST). Ophir’s aspect identification algorithm starts only at specific points of each method in order to speed up the processing time. However, this approach may overlook some potential aspects.

ER-Miner [154] provides automated support for identifying crosscutting concerns within the requirement documents by using natural language processing techniques. While it is intended to be applied at the requirements level, our approach is performed on the design of domain-specific models.

Breu and Zimmermann [41] use version history to mine aspect candidates. Their approach yields a high precision for big projects with a long history but suffers from the much fewer available data for small projects.

Aspect Browser, AMT, Prism, and FEAT all require user interaction. Users must understand the application domain and provide the pattern seed from their knowledge of the code. This limitation is in addition to the fact that these tools only look for source code level aspects. To our knowledge, no other research has been presented that focuses on the implications of aspect mining from a modeling perspective. Our approach can be distinguished from all of the related work summarized above by the observation that we have applied clone detection to search for aspects at the model level. The primary benefit our approach offers over the existing techniques is that modularization of a design through aspects is done even before proceeding to the implementation level.

## CHAPTER 6

### FUTURE WORK

This chapter puts forth some ideas about the future work based on the contents presented in Chapters 3 and 4. In the area of Aspect-Oriented Modeling (AOM), future research includes introduction of new types of join points, extensions to aspect mining from models and aspect interference handling. As for the Model-Driven Aspect Adaptation (MDAA) framework, the potential future work may support model simulation, version control of models as well as the alignment with the Model-Driven Architecture (MDA) and Architecture-Driven Modernization (ADM) standards.

#### 6.1 Aspect-Oriented Modeling

In this dissertation research, an aspect-oriented approach is applied to the modeling level to capture evolutionary changes at the requirements/design phase. AOM is a key technique to support modular evolution in that evolutionary requirements are encapsulated in a localized fashion. During our experiments, two different aspect modeling paradigms were investigated and leveraged, i.e., Aspect-Oriented Activity Modeling (AOAM) as well as aspect-oriented domain-specific modeling. This section will discuss some of the potential future works that fall into this category.

- **More types of join points**

In AOM terminology, a join point represents a point of interest in the modeling hierarchy through which evolutionary requirement concerns may be woven.



Therefore, the type of chosen join point is based on the unique modeling constructs in a specific modeling language. In our current implementation of the AOAM approach, join points are limited to the action nodes (see Figure 3.3 for the aspect-activity modeling profile definition), because actions are considered the most meaningful constituents in the activity model. In fact, it is hypothetically valid that join points can be all types of the modeling elements. For example, additional behavioral activities might be required at a certain decision point that is represented by a control node (see the activity metamodel definition as shown previously in Figure 3.1). Future work will include investigations of other types of join points as well as more advanced pointcut expressions.

- **Aspect mining**

The research on aspect mining of models is still in its infancy. Very few techniques and supporting tools are currently available to support aspect identification, let alone aspect extraction and refactoring. There are several areas that need additional investigation to further the maturity of model-driven aspect mining:

- Noise Filtering: The result of the clone detection is usually adulterated with too much undesired noise. Currently, we only use one filter layer that is based on model connections. Other metrics can be taken into account for the filtering analysis (e.g., model containment or other types of relationships).
- Visualization of Modeling Aspects: An aspect mining tool enables identification of the potential aspects and often provides the capability to visualize the various locations affected by an aspect. Traditional aspect mining techniques work on the source code level, thus their corresponding visualization tools are based on a graphical notation that is particular for line-oriented

software statistics [87]. Because a model is a containment hierarchy of entities, it is necessary to develop a specific means to visualize the cross-cutting aspects over different levels of models. Our future visualization tool will use a tree structure to display the model hierarchy natively with potential aspects highlighted across the whole structure. Users will have the option to expand or collapse any level of a specific model.

- Aspect Extraction and Refactoring: With respect to general model refactoring, we have already implemented a model refactoring browser in GME by means of a model transformation engine [197]. The research on aspect-oriented refactoring is still under investigation, which aims to extract the mined crosscutting concerns into the separately described aspects. For instance, these aspects can be represented by aspect-oriented model transformation rules written in the Embedded Constraint Language (ECL).

- **Aspect interference analysis**

A key point when dealing with aspects is the notion of aspect interference (i.e., when multiple aspectual behaviors are superimposed at the same join point, different composition orders may reveal various inconsistency problems). In such circumstances, the aspects interact with each other in a potentially undesired manner, due to the side-effects caused by the aspects. The problem of aspect interference is intrinsic to every AOSD technique. As a preliminary investigation that addressed this issue, the AOAM approach as described in Chapter 3 allows precedence relationships to be specified at the modeling level to prevent undesirable interference (i.e., the `<<follows>>` relationship between advice and aspects as well as the `<<hidden_by>>` and `<<dependent_on>>` relationship between aspects). Model engineers make design decisions explicitly based on the dependencies between aspectual behaviors, from which the underlying composition mechanism in the AOAM weaver derives a proper composition

order automatically. The current implementation has not taken into account pointcut-to-pointcut interference. However, in recent AOSD literature, the so-called “fragile pointcut” problem [170] has been studied as an important aspect interference issue. The future work will include investigation on the topic of pointcut interference. In addition, the aspect precedence specification only provides a preventive solution. Formal evaluation and analysis is thus needed to detect the aspect interference on the specification of the aspect models as well as during the run-time execution of the composed activity models. Future work also contains the integration of the activity-aspect composition mechanism with the debugging and simulation feature provided by Telelogic TAU [13]. A component called a model verifier in TAU allows model engineers to simulate the UML models in a similar way to the debugging capability provided by most programming language IDE tools (e.g., Eclipse and Microsoft Visual Studio). For instance, one can simulate the activity models automatically or can manually step through actions and control flows. By integrating the AOAM weaver with the built-in model verifier, the activity-aspect models can be simulated with the base activity model. This way, model engineers can verify the impacts that are caused by each applied aspect.

## 6.2 Model-Driven Aspect-Adaptation for Software Evolution

Chapter 4 presented a MDAA framework to facilitate legacy software evolution by incorporating AOSD and MDE techniques. This section will cover the future works that need to be investigated in this area.

- **System validation through model simulation**

The enhanced system resulting from the MDAA framework needs to be validated to ensure that it meets the evolution requirements. According to the Capability Maturity Model (CMM) [178], “validation confirms that the product,

as provided, will fulfill its intended use. In other words, validation ensures that ‘you built the right thing’.” Within the context of MDE, the validation process can be facilitated by integrated model simulation, which is “simulation-based validation” [148]. By providing the traceability between the high-level design models and the corresponding implementation, not only the enhanced system can be validated as a whole by simulating the composed models (see Figure 4.1), each individual evolutionary change to the system can also be validated as a separate part by simulating the corresponding aspect models.

- **Version control support**

Along with the system evolution, various versions of the base system, modeling artifacts (e.g., base metamodels, aspect metamodels, base models, and aspect models), and aspect code generators would co-exist. It becomes extremely difficult to track and manage dependencies, consistencies and variations among all these artifacts over time. Thus, a well-managed traceability mechanism between different artifacts needs to be in place to support software evolution. Our future work is to design and develop a robust version control system that is customized for the MDAA framework to maintain consistency between the different versions of a variety of software assets.

- **MDA/ADM alignment**

As introduced in Chapter 5, ADM [88] provides a set of standards to facilitate the evolution of legacy software. A future work of the MDAA framework would be to leverage the ADM resources to make the approach more applicable. For example, extracting models that conform to the KDM (Knowledge Discovery Metamodel) [92] or ASTM (Abstract Syntax Tree Metamodel) [91] would result in more reusable models that are able to fit into other ADM standard packages (e.g., analysis package, visualization package or refactoring package). Future

work can also involve the discovery of the architecture-related implementation in the legacy system. The Platform Definition Models (PDM) can be extracted to denote the non-functional architectural aspects of the system. The Platform-Independent Models (PIM) can also be constructed to represent the functional design decisions in the system. This way, the legacy system can be refactored and further evolved by leveraging the existing MDA technologies.

## CHAPTER 7

### CONCLUSION

Software is subject to change in order to adapt to the altering and evolving requirements. Therefore, a desired result is to achieve modularization such that an evolutionary change in a design decision is isolated to one location [143]. This dissertation research is focused on providing a solution to modular software evolution by leveraging Model-Driven Engineering (MDE) and Aspect-Oriented Software Development (AOSD) techniques. This chapter concludes the contributions of the dissertation research, and provides a summary of a few lessons learned during the research investigation.

#### 7.1 Aspect-Oriented Activity Modeling

Activity modeling has a long history of being adopted as a popular means for specifying behavioral aspects of a system. As the described system becomes more complicated, a single concern may cut across multiple places that spread over the complete set of activity models. In order to better encapsulate crosscutting concerns and support the evolution of activity models in a modular fashion, this dissertation research investigated the Aspect-Oriented Activity Modeling (AOAM) approach, which brings the AOSD solution to the activity modeling space. Aspect-specific constructs (i.e., aspects, pointcuts and advice) have been introduced as an extension to the activity models by use of the UML profile technique. Three kinds of composition

mechanisms (i.e., pointcut composition, advice composition and aspect composition) have been developed for reducing aspect interference as well as facilitating aspect reuse. The prototype of the AOAM approach has been implemented as a part of the industrial strength Aspect-Oriented Modeling (AOM) weaver that was initiated at Motorola. Experimental studies have been performed on a real-world use case. From our experience, compositions at different granularity levels improve the aspect reusability to a greater extent. In addition, the capability of separating different concerns as provided by AOAM allows each concern to be changed and maintained in a modular fashion.

## 7.2 Model-Driven Aspect-Adaptation

The MDAA approach presented in this dissertation supports the modular evolution of legacy software systems by uniting model-driven and aspect-oriented techniques. The modularity is realized by encapsulating evolutionary changes in the high-level aspect models that are translated into the low-level aspect code to be woven into the base system. The components involved in the MDAA framework are: 1) model extractor (which reverse engineers the legacy source to base models that conforms to a pre-defined metamodel); 2) aspect model specification environment (which is used for the domain engineer or requirements engineer to specify evolutionary changes in the high-level aspect model); 3) model composer (which refers to the aspect model weaver that composes the base models and the aspect models); 4) aspect code generator (which takes the aspect models and produces the aspect source code); and 5) program composer (which generates the composed system source by taking the base legacy source code and the generated aspect code). The research conducted experimental studies on different systems (e.g., a mission computing avionics system and a network fault management system) based on various modeling approaches (e.g., UML-based and DSM-based). The comparison of the two approaches are discussed

in Chapter 4. Each approach has its own advantages and disadvantages. It is worth noting that there does not exist a general standard on which modeling approach should be chosen. Such a decision depends on the specific use case and the environments/tools available. In addition, the initial stages of the MDAA solution (i.e., model extraction, aspect metamodel definition and aspect code generator implementation) may require a large amount of development effort. But once these resources are in place, MDAA can provide useful facilities to the domain experts or business users in terms of system evolution.

### 7.3 Lessons Learned

This section lists a few lessons that were learned during the dissertation research:

#### 1. **Reducing aspect interference**

During the design of the AOAM aspect modeling constructs, the aspect interference problem was not taken into account at the beginning. We soon ran into the problem of having multiple aspects imposed at the same join point. Furthermore, the ordering of these aspects were unpredictable because it is randomly determined by the underlying weaver. We then studied the area of aspect interference and designed a solution to reduce this issue by introducing precedence relationships in the aspect definition. Although the interference problem is not completely resolved (which requires a formal validation mechanism), it can be prevented by specifying precedence and dependence explicitly.

#### 2. **Constructing a metamodel composer**

The DSM-based MDAA framework implementation relies on the C-SAW model transformation engine to realize the model-level aspect weaving. However, C-SAW can only be used for composing models that are under the definition of the same metamodel. Due to the evolving nature of systems, each type of



evolutionary change might correspond to a new aspect metamodel. In order to apply C-SAW, a metamodel-level composer is needed to integrate the base metamodel and aspect metamodel. The new composed metamodel thus serves as the input to the C-SAW engine.

### 3. Utilizing a robust program transformation engine

During the experimental study with the MDAA approach, our earlier investigation with OpenC++ [46] and AspectC++ [165] suggests that the parsers in these tools are not adequate to handle the complexities that exist in the million lines of C++ code in Bold Stroke. We then turned to the commercial tool DMS, which was able to parse and transform the Bold Stroke component source. The only pitfall with DMS is the complexities of its rule specification language (RSL). However, the RSL is supposed to be hidden from the domain engineer and generated from the high-level models that are augmented with aspect specification.

## Bibliography

- [1] Aspect-Oriented Modeling. <http://www.aspect-modeling.org/>.
- [2] Aspect-Oriented Software Development. <http://www.aosd.net/>.
- [3] AspectJ Website. <http://www.eclipse.org/aspectj/>.
- [4] Domain-Driven Design. <http://www.domaindrivendesign.org/>.
- [5] Domain-Specific Modeling. <http://www.dsmforum.org/>.
- [6] Dynamic Aspects Workshop. <http://www.aosd.net/workshops/daw/>.
- [7] Eclipse Model-to-Model Transformation Project. <http://www.eclipse.org/m2m/>.
- [8] Interactive Objects Software - The Legacy Modernization Solutions Provider. <http://www.arcstyler.com/>.
- [9] MObel driven MObernisation of Complex Systems (MOMOCS). <http://www.momocs.org/>.
- [10] Model-Driven Software Evolution (MoDSE). <http://swerl.tudelft.nl/bin/view/MoDSE/>.
- [11] Sodifrance Inc. <http://www.sodifrance.fr/en/index.php>.
- [12] Sodifrance: Model-In-Action Tool Suite. <http://www.mia-software.com/>.
- [13] Telelogic TAU G2. <http://www.telelogic.com/corp/products/tau/g2/index.cfm>.
- [14] The DMS Software Reengineering Toolkit. <http://www.semdesigns.com/Products/DMS/DMSToolkit.html>.
- [15] Xerox Palo Alto Research Center. <http://www.parc.com/research/projects/aspectj/default.html>.
- [16] Aditya Agrawal, Gabor Karsai, and Ákos Lédeczi. An End-to-End Domain-Driven Software Development Framework. In *Proceedings of the 18th ACM SIG-PLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Domain-Driven Development Track*, pages 8–15, Anaheim, CA, October 2003.

- [17] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley, 2006.
- [18] Ed Arranga. In Cobol's Defense. *IEEE Software*, 17(2):70–75, March/April 2000.
- [19] Uwe Aßmann. *Invasive Software Composition*. Springer-Verlag, 2003.
- [20] Alessandra Bagnato, Luciano Baresi, Francisco Garijo, Jesus Gorrongoitia, Matteo Miraz, Juan Pavón Mestras, Giorgio Pezzuto, Luis Quijada, Andrey Sadovykh, Marco Serina, and Jan Vollmar. MOMOCS: MDE for the Modernization of Complex Systems. In *Model-Driven Engineering: Processes, Coherence, Traceability and Trustworthy Components (NEPTUNE DAYS)*, Paris, France, April 2008.
- [21] Arpad Bakay and Endre Magyari. The UDM framework. Technical Report, Institute of Software Integrated Systems, Vanderbilt University. <http://www.escherinstitute.org/Plone/tools/suites/mic/udm/UDMAPI.pdf>, October 2004.
- [22] Brenda S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proceedings of the 2nd Working Conference on Reverse Engineering*, pages 86–95, Toronto, ON, Canada, July 1995.
- [23] Joao Paulo Barros and Luis Gomes. Towards the Support for Crosscutting Concerns in Activity Diagrams: a Graphical Approach. In *the 4th International Workshop on Aspect-Oriented Modeling (AOM), 6th International Conference on the Unified Modeling Language (UML)*, San Francisco, CA, October 2003.
- [24] Don Batory. Multilevel Models in Model-Driven Development, Product-Lines, and Metaprogramming. *IBM Systems Journal*, 45(3):527–539, July 2006.
- [25] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 187–197, Portland, OR, May 2003.
- [26] Ira D. Baxter. Design Maintenance Systems. *Communications of the ACM*, 35(4):73–89, April 1992.
- [27] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS: Program Transformations for Practical Scalable Software Evolution. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 625–634, Edinburgh, Scotland, May 2004.
- [28] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 368–377, Bethesda, MD, November 1998.

- [29] Keith H. Bennett and Václav T. Rajlich. Software Maintenance and Evolution: a Roadmap. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE), Future of Software Engineering Track*, pages 73–87, Limerick, Ireland, June 2000.
- [30] Tony Beveridge and Col Perks. Blueprint for the Flexible Enterprise. *Intelligent Enterprise*, 3(4):46, March 2000.
- [31] Sami Beydeda, Matthias Book, and Volker Gruhn. *Model-Driven Software Development*. Springer, 2005.
- [32] Jean Bézivin. From Object Composition to Model Transformation with the MDA. In *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS)*, page 350, Santa Barbara, CA, July - August 2001.
- [33] Jean Bézivin, Mikaël Barbero, Hugo Bruneliere, Jeff Gray, and Frédéric Jouault. Reverse Engineering in Eclipse with the MoDisco Project. In *EclipseCon*, Santa Clara, CA, March 2007.
- [34] Jean Bézivin, Hugo Brunelière, and Mikaël Barbero. The Model Discovery (MoDisco) Component: A Proposal for a New Eclipse/GMT Component. [http://www.eclipse.org/gmt/modisco/doc/MoDisco\\_Proposal\\_1.0.pdf](http://www.eclipse.org/gmt/modisco/doc/MoDisco_Proposal_1.0.pdf), October 2006.
- [35] Jean Bézivin, Frédéric Jouault, and David Touzet. An Introduction to the ATLAS Model Management Architecture. Technical Report 05-01, Laboratoire d’Informatique de Nantes-Atlantique (LINA). <http://www.sciences.univ-nantes.fr/lina/atl/www/papers/RR-LINA2005-01.pdf>, February 2005.
- [36] Dorothea Blostein and Andy Schürr. Computing with Graphs and Graph Transformations. *Software Practice and Experience*, 29(3):197–217, March 1999.
- [37] Grady Booch, Alan W. Brown, Sridhar Iyengar, James Rumbaugh, and Bran Selic. An MDA Manifesto. *Business Process Trends/MDA Journal*, pages 133–143, May 2004.
- [38] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
- [39] Don Box, Charlie Kindel, and Grady Booch. *Essential COM: The Component Object Model*. Addison-Wesley Longman, 1997.
- [40] Mark G. J. van den Brand, Jan Heering, Paul Klint, and Pieter A Olivier. Compiling Language Definitions: The ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, July 2002.

- [41] Silvia Breu and Thomas Zimmermann. Mining Aspects from Version History. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 221–230, Tokyo, Japan, September 2006.
- [42] Frederick P. Brooks. No Silver Bullet Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, April 1987.
- [43] Magiel Bruntink, Arie van Deursen, Remco van Engelen, and Tom Tourwé. On the Use of Clone Detection for Identifying Crosscutting Concern Code. *IEEE Transactions on Software Engineering*, 31(10):804–818, October 2005.
- [44] Walter Cazzola and Sonia Pini. Join Point Patterns: A High Level Join Point Selection Mechanism. In *the 9th International Workshop on Aspect-Oriented Modeling (AOM), 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Genova, Italy, October 2006.
- [45] Anis Charfi and Mira Mezini. Aspect-Oriented Web Service Composition with AO4BPEL. In *Proceedings of the European Conference on Web Services (ECOWS)*, Springer-Verlag LNCS 3250, pages 168–182, Erfurt, Germany, September 2004.
- [46] Shigeru Chiba. A Metaobject Protocol for C++. In *Proceedings of the 10th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 285–299, Austin, Texas, October 1995.
- [47] Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.
- [48] Siobhán Clarke and Elisa Banaissad. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley, 2005.
- [49] James R. Cordy. Source Transformation, Analysis and Generation in TXL. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 1–11, Charleston, SC, January 2006.
- [50] Thomas Cottenier, Aswin Van Den Berg, and Tzilla Elrad. An Add-in for Aspect-Oriented Modeling in Telelogic TAU G2. In *Telelogic User Group Conference (UGC)*, Denver, CO, October 2006.
- [51] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad. Joinpoint Inference from Behavioral Specification to Implementation. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP)*, Springer-Verlag LNCS 4609, pages 476–500, Berlin, Germany, July - August 2007.
- [52] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., 2000.

- [53] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *the 2nd OOPSLA Workshop on Generative Techniques in the Context of Model-Driven Architecture*, Anaheim, CA, October 2003.
- [54] Edsger W. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1(2):115–138, June 1971.
- [55] Rémi Douence, Pascal Fradet, and Mario Südholt. A Framework for the Detection and Resolution of Aspect Interactions. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE)*, Springer-Verlag LNCS 2487, pages 173–188, Pittsburgh, PA, October 2002.
- [56] Stéphane Ducasse, Tom Mens, and Radu Marinescu. International ERCIM-ESF Workshop on Challenges in Software Evolution (ChaSE), Berne, Switzerland. <http://w3.umh.ac.be/evol/meetings/evol2005.html>, April 2005.
- [57] Pascal Durr, Tom Staijen, Lodewijk Bergmans, and Mehmet Aksit. Reasoning About Semantic Conflicts Between Aspects. In *the 2nd European Interactive Workshop on Aspects in Software (EIWAS)*, Brussels, Belgium, September 2005.
- [58] Thomas Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall PTR, 2005.
- [59] Len Erlikh. Leveraging Legacy System Dollars for E-Business. *IT Professional*, 2(3):17–23, May/June 2000.
- [60] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.
- [61] Marcos Didonet Del Fabro, Jean Bézivin, Frédéric Jouault, Erwan Breton, and Guillaume Gueltas. AMW: a Generic Model Weaver. In *Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM)*, pages 105–114, Paris, France, June - July 2005.
- [62] Jean-Marie Favre. Meta-Model and Model Co-evolution within the 3D Software Space. In *International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA)*, Amsterdam, The Netherlands, September 2003.
- [63] Jean-Marie Favre. CacOphoNy: Metamodel-Driven Architecture Recovery. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE)*, pages 204–213, Delft, The Netherlands, November 2004.
- [64] Jean-Marie Favre. Foundations of Meta-Pyramids: Languages vs. Metamodels - Episode II: Story of Thotus the Baboon1. In *Language Engineering for Model-Driven Software Development*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, February - March 2004.

- [65] Jean-Marie Favre. Foundations of Model (Driven) (Reverse) Engineering: Models - Episode I: Stories of The Fidus Papyrus and of The Solarus. In *Language Engineering for Model-Driven Software Development*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, February - March 2004.
- [66] Robert Filman and Dan Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *OOPSLA Workshop on Advanced Separation of Concerns*, Minneapolis, MN, October 2000.
- [67] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas, and Jean-Marc Jézéquel. Model-Driven Engineering for Software Migration in a Large Industrial Context. In *Proceedings of the 10th ACM/IEEE Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Springer-Verlag LNCS 4735, pages 482–497, Nashville, TN, September - October 2007.
- [68] Institute for Software Integrated Systems. The Generic Modeling Environment: GME 5 User’s Manual. <http://www.isis.vanderbilt.edu/Projects/gme/>, 2005.
- [69] OMG ADM Task Force. Why do We Need Standards for the Modernization of Existing Systems. <http://adm.omg.org/legacy/ADM.whitepaper.pdf>, 2003.
- [70] OMG ADM Task Force. Architecture-Driven Modernization: Transforming the Enterprise. <http://www.omg.org/cgi-bin/doc?admtf/2007-12-01>, 2007.
- [71] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [72] Robert France, Indrakshi Ray, Geri Georg, and Sudipto Ghosh. An Aspect-Oriented Approach to Early Design Modeling. *IEE Proceedings – Software*, 151(4):173–185, August 2004.
- [73] David S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley and Sons, 2003.
- [74] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [75] Pieter Van Gorp, Dirk Janssens, and Tracy Gardner. Write Once, Deploy N: A Performance Oriented MDA Case Study. In *Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, pages 123–134, Monterey, CA, September 2004.
- [76] Vincenzo Grassi and Andrea Sindico. UML Modeling of Static and Dynamic Aspects. In *the 9th International Workshop on Aspect-Oriented Modeling (AOM), 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Genova, Italy, October 2006.

- [77] Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck. Handling Crosscutting Constraints in Domain-Specific Modeling. *Communications of the ACM*, 44(10):87–93, October 2001.
- [78] Jeff Gray, Yuehua Lin, and Jing Zhang. Degrees of Independence in Aspect-Oriented Modeling to Support Two-Level Weaving. In *Real-Time CORBA Component Model Workshop*, St. Louis, MO, March 2003.
- [79] Jeff Gray, Yuehua Lin, and Jing Zhang. Automating Change Evolution in Model-Driven Engineering. *IEEE Computer*, 39(2):51–58, February 2006.
- [80] Jeff Gray, Matti Rossi, and Juha-Pekka Tolvanen. Special Issue: Domain-Specific Modeling with Visual Languages. *Journal of Visual Languages and Computing*, 15(3–4):207–209, June-August 2004.
- [81] Jeff Gray, Janos Sztipanovits, Douglas C. Schmidt, Ted Bapty, Sandeep Neema, and Aniruddha Gokhale. Two-level Aspect Weaving to Support Evolution of Model-Driven Synthesis. In *Aspect-Oriented Software Development*, pages 681–709. Addison-Wesley, 2004.
- [82] Jeff Gray, Juha-Pekka Tolvanen, Steven Kelly, Aniruddha Gokhale, Sandeep Neema, and Jonathan Sprinkle. Domain-Specific Modeling. In *Handbook of Dynamic System Modeling*, (Paul Fishwick, ed.), pages 7–1 through 7–20. CRC Press, 2007.
- [83] Jeff Gray, Jing Zhang, Yuehua Lin, Hui Wu, Suman Roychoudhury, Rajesh Sudarsan, Aniruddha Gokhale, Sandeep Neema, Feng Shi, and Ted Bapty. Model-Driven Program Transformation of a Large Avionics Framework. In *Proceedings of the 3rd ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE)*, Springer-Verlag LNCS 3286, pages 361–378, Vancouver, BC, October 2004.
- [84] Jeffrey G. Gray. *Aspect-Oriented Domain-Specific Modeling: A Generative Approach Using a Meta-weaver Framework*. PhD Thesis, Vanderbilt University. <http://www.cis.uab.edu/gray/Pubs/Dissertation.pdf>, 2002.
- [85] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools*. Wiley, 2004.
- [86] William G. Griswold and David Notkin. Automated Assistance for Program Restructuring. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3):228–269, July 1993.
- [87] William G. Griswold, Yoshikiyo Kato Y, and Jimmy J. Yuan. Aspect Browser: Tool Support for Managing Dispersed Aspects. In *the 1st Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems*, Denver, CO, November 1999.



- [88] Object Management Group. Architecture Driven Modernization. <http://adm.omg.org/>.
- [89] Object Management Group. UML Semantics version 1.1. <ftp://ftp.omg.org/pub/docs/ad/97-08-04.pdf>, 1997.
- [90] Object Management Group. UML Specification v1.5. <http://www.omg.org/cgi-bin/doc?formal/03-03-01>, 2003.
- [91] Object Management Group. Abstract Syntax Tree Metamodel ASTM (ASTM) final RFP. <http://www.omg.org/docs/admtf/05-02-02.pdf>, 2005.
- [92] Object Management Group. ADM: Knowledge Discovery Meta-Model (KDM), v1.0. <http://www.omg.org/docs/formal/08-01-01.pdf>, 2005.
- [93] Object Management Group. MOF Core Specification, v2.0. <http://www.omg.org/cgi-bin/doc?formal/2006-01-01>, 2006.
- [94] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification. <http://www.omg.org/docs/ptc/07-07-07.pdf>, 2007.
- [95] Object Management Group. UML 2.1.1 Superstructure Specification. <http://www.omg.org/cgi-bin/doc?formal/07-02-03>, 2007.
- [96] Michael Gruninger and Jintae Lee. Ontology: Applications and Design. *Communications of ACM*, 45(2):39–41, 2002.
- [97] Patrick A. V. Hall. *Software Reuse and Reverse Engineering in Practice*. Chapman & Hall, 1992.
- [98] Jan Hannemann and Gregor Kiczales. Overcoming the Prevalent Decomposition in Legacy Code. In *ICSE Workshop on Advanced Separation of Concerns*, Toronto, ON, Canada, May 2001.
- [99] Tim Harrison, David Levine, and Douglas C. Schmidt. The Design and Performance of a Hard Real-Time Object Event Service. In *Proceedings of the 12th ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 184–200, Atlanta, GA, October 1997.
- [100] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.
- [101] Honeywell Inc. DOME (The DObmain Modeling Environment) Guide. <http://www.htc.honeywell.com/dome/index.htm/>, 1999.
- [102] Michael Jiang, Jing Zhang, David Raymer, and John Strassner. A Modeling Framework for Self-Healing Software Systems. In *MoDELS Workshop on Models@run.time*, Nashville, TN, September - October 2007.

- [103] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming - Special Issue on Second Issue of Experimental Software and Toolkits (EST)*, 72(1–2):31–39, June 2008.
- [104] Frédéric Jouault and Ivan Kurtev. On the Architectural Alignment of ATL and QVT. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC), Model Transformation Track*, pages 1188–1195, Dijon, France, April 2006.
- [105] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002.
- [106] Gabor Karsai, Aditya Agrawal, Feng Shi, and Jonathan Sprinkle. On the Use of Graph Transformation in the Formal Specification of Model Interpreters. *Journal of Universal Computer Science*, 9(11):1296–1321, November 2003.
- [107] Gabor Karsai, Sandeep Neema, and David Sharp. Model-Driven Architecture for Embedded Software: A Synopsis and an Example. *Science of Computer Programming*, 73(1):26–38, 2008.
- [108] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting Started with AspectJ. *Communications of ACM*, 44(10):59–65, October 2001.
- [109] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, Springer-Verlag LNCS 1241, pages 220–242, Jyväskylä, Finland, June 1997.
- [110] Jörg Kienzle and Samuel Gélineau. AO Challenge - Implementing the ACID Properties for Transactional Objects. In *Proceedings of 5th International Conference on Aspect-oriented Software Development (AOSD)*, pages 202–213, Bonn, Germany, March 2006.
- [111] Jörg Kienzle, Yang Yu, and Jie Xiong. On Composition and Reuse of Aspects. In *the 2nd Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, Boston, MA, March 2003.
- [112] Patrick DJ Kulandaisamy. Model Driven Legacy Migration. In *Architecture-Driven Modernization Workshop*, Chicago, IL, March 2004.
- [113] Ramnivas Laddad. *Aspect Oriented Refactoring*. Addison-Wesley Professional, 2008.

- [114] Bert Lagaisse, Wouter Joosen, and Bart De Win. Managing Semantic Interference with Aspect Integration Contracts. In *International Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT)*, Lancaster, UK, March 2004.
- [115] Ralf Lämmel. Coupled Software Transformations. In *the 1st International Workshop on Software Evolution Transformations*, pages 31–35, Delft, the Netherlands, November 2004.
- [116] Ralf Lämmel and Chris Verhoef. Cracking the 500-Language Problem. *IEEE Software*, 18(6):78–88, 2001.
- [117] Ákos Lédeczi, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gabor Karsai. Composing Domain-Specific Design Environments. *IEEE Computer*, 34(11):44–51, November 2001.
- [118] Ákos Lédeczi, James Davis, Sandeep Neema, and Aditya Agrawal. Modeling Methodology for Integrated Simulation of Embedded Systems. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 13(1):82–103, January 2003.
- [119] Manny M. Lehman. Laws of Software Evolution Revisited. In *Proceedings of the 5th European Workshop on Software Process Technology (EWSPT)*, pages 108–124, Nancy, France, October 1996.
- [120] Xue Li. A Survey of Schema Evolution in Object-Oriented Databases. In *Proceedings of the 31st International Conference on Technology of Object-Oriented Language and Systems (TOOLS)*, pages 362–371, Nanjing, China, September 1999.
- [121] Bennet P. Lientz and E. Burton Swanson. Problems in Application Software Maintenance. *Communications of ACM*, 24(11):763–769, November 1981.
- [122] Yuehua Lin. *A Model Transformation Approach to Automated Model Evolution*. PhD Thesis, University of Alabama at Birmingham. <http://www.cis.uab.edu/softcom/dissertations/LinYuehua.pdf>, 2007.
- [123] Yan Liu, Jing Zhang, Michael Jiang, David Raymer, and John Strassner. A Model-based Approach to Adding Autonomic Capabilities to Network Fault Management System. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*, pages 859–862, Salvador, Brazil, April 2008.
- [124] Yan Liu, Jing Zhang, Xin Meng, and John Strassner. Sequential Proximity-Based Clustering for Telecommunication Network Alarm Correlation. In *Proceedings of the 5th International Symposium on Neural Networks (ISNN)*, Springer-Verlag LNCS 5264, pages 30–39, Beijing, China, September 2008.

- [125] Yan Liu, Jing Zhang, and John Strassner. Model-Driven Adaptive Self-Healing for Autonomic Computing. In *the 3rd IEEE International Workshop on Modeling Autonomic Communication Environments (MACE)*, Springer-Verlag LNCS 5276, pages 62–73, Samos Island, Greece, September 2008.
- [126] Roberto Lopez-Herrejon, Don Batory, and Christian Lengauer. A Disciplined Approach to Aspect Composition. In *Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 68–77, Charleston, SC, January 2006.
- [127] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 244–253, Monterey, CA, November 1996.
- [128] George H. Mealy. A Method to Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34:1045–1079, September 1955.
- [129] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. A Taxonomy of Model Transformations. In *Language Engineering for Model-Driven Software Development*, Schloss Dagstuhl, Germany, February - March 2004.
- [130] Tom Mens and Serge Demeyer. *Software Evolution*. Springer, 2008.
- [131] Tom Mens and Michel Wermelinger. Separation of Concerns for Software Evolution. *Journal of Software Maintenance*, 14(5):311–315, September - October 2002.
- [132] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in Software Evolution. In *Proceedings of the 8th International Workshop on Principles of Software Evolution (IWPSE)*, pages 13–22, Lisbon, Portugal, September 2005.
- [133] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.
- [134] Bertrand Meyer. Applying “Design by Contract”. *IEEE Computer*, 25(10):40–51, October 1992.
- [135] Jeffrey Moad. Maintaining the Competitive Edge. *Datamation*, 36(4):61–66, February 1990.
- [136] Edward F. Moore. Gedanken-experiments on Sequential Machines. *Automata Studies, Annals of Mathematical Studies*, 34:129–153, 1956.
- [137] Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. Composing Aspects at Shared Joinpoints. In *Proceedings of International Conference NetObjectDays (NODE)*, pages 19–38, Erfurt, Germany, September 2005.

- [138] Sandeep Neema, Arpad Bakay, and Gabor Karsai. Embedded Systems Modeling Language. <http://w3.isis.vanderbilt.edu/Projects/mobies/Downloads/ESML.pdf>, 2004.
- [139] Sandeep Neema, Ted Bapty, Jeff Gray, and Aniruddha S. Gokhale. Generators for Synthesis of QoS Adaptation in Distributed Real-Time Embedded Systems. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE)*, Springer-Verlag LNCS 2487, pages 236–251, Pittsburgh, PA, 2002.
- [140] Greg Nordstrom, Janos Sztipanovits, Gabor Karsai, and Ákos Lédeczi. Meta-modeling - Rapid Design and Evolution of Domain-Specific Modeling Environments. In *the 6th Symposium on Engineering of Computer-Based Systems (ECBS)*, pages 68–74, Nashville, TN, March 1999.
- [141] Adeniyi Onabajo, Iryna Bilykh, and Jens Jahnke. Wrapping Legacy Medical Systems for Integrated Health Network. In *the 4th International Conference NetObjectDays Workshop on Migration and Evolvability of Long-life Software Systems (MELLS)*, Erfurt, Germany, September 2003.
- [142] William F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD Thesis, University of Illinois at Urbana-Champaign. <ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z>, 1992.
- [143] David Lorge Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
- [144] David Lorge Parnas. Software Aging. In *Proceedings of the 16th International Conference on Software Engineering (ICSE)*, pages 279–287, Sorrento, Italy, 1994.
- [145] Renaud Pawlak, Laurence Duchien, and Lionel Seinturier. CompAr: Ensuring Safe Around Advice Composition. In *Proceedings of the 7th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 163–178, Athens, Greece, June 2005.
- [146] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 1–24, Kyoto, Japan, September 2001.
- [147] James L. Peterson. Petri Nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.
- [148] K. Ravindran, K. A. Kwiat, and G. Ding. Simulation-Based Validation of Protocols for Distributed Systems. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS)*, page 318, Big Island, HI, January 2005.

- [149] Raghu Reddy, Sudipto Ghosh, Robert France, Greg Straw, James M. Bieman, N. McEachen, Eunjee Song, and Geri Georg. Directives for Composing Aspect-Oriented Design Class Models. In *Transactions on Aspect-Oriented Software Development I*, Springer-Verlag LNCS 3880, pages 75–105, February 2006.
- [150] Thijs Reus, Hans Geers, and Arie van Deursen. Harvesting Software Systems for MDA-Based Reengineering. In *Proceedings of the 2nd European Conference on Model-Driven Architectures: Foundations and Applications (ECMDA-FA)*, Springer-Verlag LNCS 4066, pages 213–225, July 2006.
- [151] Martin P. Robillard and Gail C. Murphy. Concern Graphs: Finding and Describing Concerns. Using Structural Program Dependencies. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pages 406–416, Orlando, FL, May 2002.
- [152] Winston Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *Proceedings of the 9th International Conference on Software Engineering (ICSE)*, pages 328–338, Monterey, CA, March - April 1987.
- [153] Grzegorz Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific Publishing Co. Inc., 1997.
- [154] Américo Sampaio, Ruzanna Chitchyan, Awais Rashid, and Paul Rayson. EA-Miner: A Tool for Automating Aspect-Oriented Requirements Identification. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 352–355, Long Beach, CA, November 2005.
- [155] Stephen R. Schach. *Object-Oriented and Classical Software Engineering*. McGraw-Hill, 2006.
- [156] Andrea Schauerhuber, Wieland Schwinger, Elisabeth Kapsammer, Werner Retschitzegger, Manuel Wimmer, and Gerti Kappel. A Survey on Aspect-Oriented Modeling Approaches. Technical Report, Vienna University of Technology. [http://www.wit.at/people/schauerhuber/publications/aomSurvey/AOMSurvey\\_Schauerhuber\\_Oct2007.pdf](http://www.wit.at/people/schauerhuber/publications/aomSurvey/AOMSurvey_Schauerhuber_Oct2007.pdf), October 2007.
- [157] Hermann Schichl. Models and History of Modeling. In *Modeling Languages in Mathematical Optimization*, pages 25–36. 2004.
- [158] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley and Sons, 2000.
- [159] Shane Sendall and Wojtek Kozaczynski. Model Transformation - the Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, September/October 2003.

- [160] David Sharp. Component-Based Product Line Development of Avionics Software. In *the 1st Software Product Lines Conference (SPLC)*, pages 353–369, Denver, CO, August 2000.
- [161] David Shepherd, Emily Gibson, and Lori Pollock. Design and Evaluation of an Automated Aspect Mining Tool. In *Proceedings of International Conference on Software Engineering Research and Practice*, pages 601–607, Las Vegas, NV, June 2004.
- [162] Marcelo Sihman and Shmuel Katz. Superimpositions and Aspect-Oriented Programming. *The Computer Journal*, 46(5):529–541, September 2003.
- [163] Arnor Solberg, Devon Simmonds, Raghu Reddy, Sudipto Ghosh, and Robert France. Using Aspect Oriented Techniques to Support Separation of Concerns in Model Driven Development. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC)*, pages 121–126, Edinburgh, Scotland, UK, July 2005.
- [164] Richard Soley and the OMG Staff Strategy Group. Model-Driven Architecture. <ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf>, 2000.
- [165] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to the C++ Programming Language. In *Proceedings of the 40th International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*, pages 53–60, Sydney, Australia, February 2002.
- [166] Jonathan Sprinkle. *Metamodel Driven Model Migration*. PhD Thesis, Vanderbilt University. [http://www.isis.vanderbilt.edu/publications/archive/Sprinkle\\_JM\\_8\\_0\\_2003\\_Metamodel\\_.pdf](http://www.isis.vanderbilt.edu/publications/archive/Sprinkle_JM_8_0_2003_Metamodel_.pdf), 2003.
- [167] Jonathan Sprinkle and Gabor Karsai. A Domain-Specific Visual Language For Domain Model Evolution. *Journal of Visual Languages and Computing*, 15(3-4):291–307, June - August 2004.
- [168] Thomas Stahl, Markus Völter, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-Driven Software Development*. Wiley, 2006.
- [169] Dominik Stein, Stefan Hanenberg, and Rainer Unland. A UML-based Aspect-Oriented Design Notation for AspectJ. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD)*, pages 106–112, Enschede, The Netherlands, March 2002.
- [170] Maximilian Stoerzer and Christian Koppen. PCDiff: Attacking the Fragile Pointcut Problem. In *Proceedings of the 1st European Interactive Workshop on Aspects in Software (EIWAS)*, Berlin, Germany, July 2004.

- [171] Maximilian Stoerzer, Robin Sterr, and Florian Forster. Detecting Precedence-Related Advice Interference. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE)*, pages 317–322, Tokyo, Japan, September 2006.
- [172] John Strassner. *Policy-Based Network Management: Solutions for the Next Generation (The Morgan Kaufmann Series in Networking)*. Morgan Kaufmann Publishers Inc., 2003.
- [173] Alistair G Sutcliffe and Nikolay D Mehandjiev. End-User Development: Tools that Empower Users to Create their Own Software Solutions. *Communications of the ACM*, 47(9):31–32, September 2004.
- [174] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. JAsCo: An Aspect-Oriented Approach Tailored for Component Based Software. In *Proceedings of the 2nd International Conference on Aspect-oriented Software Development (AOSD)*, pages 21–29, Boston, MA, March 2003.
- [175] Janos Sztipanovits and Gabor Karsai. Model-Integrated Computing. *IEEE Computer*, 30(4):110–111, April 1997.
- [176] Janos Sztipanovits and Gabor Karsai. Generative Programming for Embedded Systems. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE)*, Springer-Verlag LNCS 2487, pages 32–49, Pittsburgh, PA, October 2002.
- [177] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, pages 107–119, Los Angeles, CA, May 1999.
- [178] CMMI Product Team. Capability Maturity Model Integration (CMMI) for Software Engineering (CMMI-SW), v1.1. Technical Report, Software Engineering Institute, Carnegie Mellon University. <http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr029.pdf>, 2002.
- [179] Juha-Pekka Tolvanen and Steven Kelly. Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences. In *Proceedings of the 9th International Software Product Line Conference (SPLC)*, Springer-Verlag LNCS 3714, pages 198–209, Rennes, France, September 2005.
- [180] William Ulrich. The Evolutionary Growth of Software Engineering and the Decade Ahead. *American Programmer*, 3(10):12–20, 1990.
- [181] William Ulrich. *Legacy Systems: Transformation Strategies*. Prentice-Hall, 2002.
- [182] Hans van Vliet. *Software Engineering: Principles and Practice, 2nd Edition*. Wiley, 2000.



- [183] Jonne van Wijngaarden and Eelco Visser. Program Transformation Mechanics. A Classification of Mechanisms for Program Transformation with a Survey of Existing Transformation Systems. Technical Report UU-CS-2003-048, Department of Information and Computing Sciences, Utrecht University. <http://www.cs.uu.nl/research/techreps/repo/CS-2003/2003-048.pdf>, 2003.
- [184] Sander Vermolen and Eelco Visser. Heterogeneous Coupled Evolution of Software Languages. In *Proceedings of the 11th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Springer-Verlag LNCS 5301, pages 630–644, Toulouse, France, September - October 2008.
- [185] Eelco Visser. A Survey of Rewriting Strategies in Program Transformation Systems. In *International Workshop on Reduction Strategies in Rewriting and Programming (WRS)*, Utrecht, The Netherlands, May 2001.
- [186] Eelco Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA)*, pages 357–362, Utrecht, The Netherlands, May 2001.
- [187] Nanbor Wang, Douglas C. Schmidt, and Carlos O’Ryan. Overview of the CORBA Component Model. In *Component-Based Software Engineering: Putting the Pieces Together*, pages 557–571. Addison-Wesley Longman Publishing Co., Inc., 2001.
- [188] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2004.
- [189] Charles Zhang and Hans-Arno Jacobsen. PRISM is Research In aSpect Mining. In *Companion Proceedings of the 19th ACM SIGPLAN International Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 20–21, Vancouver, BC, Canada, October 2004.
- [190] Jing Zhang. On the Use of Model-Driven Program Transformation in a Large Avionics Product Line. In *Conference of the Mid-southeast Chapter of the ACM*, Gatlinburg, TN, November 2003.
- [191] Jing Zhang, Thomas Cottenier, Aswin van den Berg, and Jeff Gray. Aspect Interference and Composition in the Motorola Aspect-Oriented Modeling Weaver. In *Model Driven Engineering Languages and Systems (MoDELS) Workshop on Aspect-Oriented Modeling (AOM)*, Genova, Italy, October 2006.
- [192] Jing Zhang, Thomas Cottenier, Aswin van den Berg, and Jeff Gray. Aspect Composition in the Motorola Aspect-Oriented Modeling Weaver. *Journal of Object Technology - Special Issue on Aspect-Oriented Modeling*, 6(7):89–108, August 2007.

- [193] Jing Zhang and Jeff Gray. Legacy System Evolution through Model-Driven Program Transformation. In *the 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC) Workshop on Model-Driven Evolution of Legacy Systems (MELS)*, pages 625–634, Monterey, CA, September 2004.
- [194] Jing Zhang, Jeff Gray, and Yuehua Lin. A Generative Approach to Model Interpreter Evolution. In *the 4th OOPSLA Workshop on Domain-Specific Modeling (DSM)*, pages 121–129, Vancouver, BC, Canada, October 2004.
- [195] Jing Zhang, Jeff Gray, and Yuehua Lin. A Model-Driven Approach to Enforce Crosscutting Assertion Checking. In *ICSE Workshop on the Modeling and Analysis of Concerns in Software (MACS)*, St. Louis, MO, May 2005.
- [196] Jing Zhang, Jeff Gray, Yuehua Lin, and Robert Tairas. Aspect Mining from a Modeling Perspective. *International Journal of Computer Applications in Technology*, 31(1-2):74–82, 2008.
- [197] Jing Zhang, Yuehua Lin, and Jeff Gray. Generic and Domain-Specific Model Refactoring using a Model Transformation Engine. In *Model-driven Software Development*, pages 199–218. Springer, 2005.
- [198] Jing Zhang, Yan Liu, Michael Jiang, and John Strassner. An Aspect-Oriented Approach to Handling Crosscutting Concerns in Activity Modeling. In *IAENG International Conference on Software Engineering*, pages 885–890, Hong Kong, March 2008.