GENAWEAVE: A GENERIC ASPECT WEAVER FRAMEWORK BASED ON MODEL-DRIVEN PROGRAM TRANSFORMATION

by

SUMAN ROYCHOUDHURY

JEFF GRAY, COMMITTEE CHAIR
PURUSHOTHAM BANGALORE
BARRETT BRYANT
MARJAN MERNIK
ANTHONY SKJELLUM
RANDY SMITH

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

2008

GENAWEAVE: A GENERIC ASPECT WEAVER FRAMEWORK BASED ON MODEL-DRIVEN PROGRAM TRANSFORMATION

SUMAN ROYCHOUDHURY

COMPUTER AND INFORMATION SCIENCES

ABSTRACT

Legacy software affects critical functions of our daily lives (e.g., general commercial transactions, scientific applications and military defense systems), and represents a significant investment by government, scientific and corporate institutions. As a consequence of the longevity of such systems, existing legacy software is subject to decay over a period of time, making it increasingly difficult to address changing stakeholder requirements. Modern research approaches for software engineering and programming language design, such as aspect-oriented software development (AOSD), have been investigated as effective techniques for improving modularization of software. However, a general trend in research for supporting aspects has focused primarily on Java as the target programming language, neglecting the multiple billions of lines of existing code written in other languages. Rather than bringing the legacy code to existing Java-based weavers, a viable alternative is to take Aspect-Oriented Programming (AOP) principles to the legacy languages and tool environments.

Given the large number of programming languages currently in use, a solution that mitigates the effort needed to create each new aspect weaver is more desirable than an approach that manually recreates a weaver from scratch for each legacy language. The research presented in this dissertation utilizes Program Transformation Engines (PTEs) to construct aspect weavers for legacy languages. A core focus of the research is a generic platform that permits reusability of software artifacts among aspect weavers constructed

for various General-Purpose Languages (GPLs). In addition, the research described in this dissertation aims to eliminate the accidental complexities that are typically associated with using PTEs. In order to fulfill these two objectives, the research utilizes a model-driven front-end that is layered on top of the program transformation based back-end. Specifically, the research makes a contribution by combining Model-Driven Engineering (MDE) with PTE to construct aspect weavers for GPLs through models and program transformations. The approach described in the dissertation uses models to capture the essence of various AOP language constructs at a higher-level of abstraction. These models are then mapped to concrete weavers for GPLs through a combination of higher-order model transformation and program transformation rules. A generic extension to the framework further supports reusability of artifacts among weavers during the construction process. In addition, the framework allows experimentation with new AOP constructs (e.g., loops) and helps to evolve commercial and scientific software (e.g., Blitz++, HPL) maintained in legacy languages like Object Pascal, C and FORTRAN. The research presented in this dissertation outlines several challenges that were identified in providing a generic platform to create aspect weavers and demonstrates how each of those challenges was mitigated during the course of this research.

DEDICATION

This work is dedicated to my beloved dad − I miss you in every moment of my life and I wish you were here to see my dreams come true.

My mom − Without your blessings and unconditional love, I would have never reached my goal.

My wife, Mohua – You supported me in every little thing that came my way and sacrificed a part of your life for the sake of mine. I hope I can make your dreams come true through mine.

My brother and sister-in-law − I was never short of encouragement and advice whenever I needed some, thank you for being there always.

ACKNOWLEDGEMENTS

Foremost, I would like to thank my advisor, Dr. Jeff Gray, for providing me with an opportunity to complete my Ph.D. thesis at the University of Alabama at Birmingham. I especially want to thank him for his continuous support and able guidance that made this work possible. Dr. Gray has been actively interested in my work and has always been available to advise me. I am very grateful for his patience, motivation, enthusiasm, and immense knowledge in the field of Computer Science, in particular, Software Engineering and Aspect-Oriented Programming that, taken together, make him a great advisor.

I would like to thank Dr. Anthony Skjellum and Dr. Purushotham Bangalore for providing me with an opportunity to collaborate and conduct research in the field of high-performance computing. Throughout the several research meetings we held together, I gained immense knowledge that contributed to my own research, especially how advanced software engineering principles could be applied to applications belonging to high-performance computing. I am indebted to their valuable insights and thoughtful advice during the course of my research.

I would like to thank Dr. Barrett Bryant for his course on Compiler Design that helped me to understand the various technologies that I used in my research. I should also mention that Dr. Bryant and Dr. Skjellum have often performed the role as my co-advisors and I would regularly seek their kind advice to fulfill my research and career goals.

I would like to extend my appreciation to Dr. Marjan Mernik and Dr. Randy Smith for accepting my invitation to serve on my thesis committee and thereby providing me with valuable feedback to improve my research work.

I must sincerely thank Dr. Frédéric Jouault for his time and effort in assisting me to overcome some of the difficult challenges that I was facing during the later stages of my research. To Dr. Ira Baxter, I greatly appreciate your guidance on DMS and thank you for the speedy response to all of my intricate questions that you answered with minute detail and care.

I am also thankful to all the faculty members of the Computer and Information Sciences Department at UAB, for the various courses they offered, which helped me to broaden my knowledge and decide upon my research topic.

I would like to thank Rosanne Brill, Jim Sahaj, Sudeep Sabnis, Nguyen Long, David Barron, Ken Lidster, Roger Andrews and all my fellow colleagues at Synergex who provided me with valuable internship experience. The internship was a great learning experience and I gained valuable knowledge working with their core compiler team.

To my fellow coworkers at UAB, Robert Tairas, Shairaj Sheikh, Rajesh Sudarsan, Nikhil Garge, Premkumar Somasundaram, Jing Zhang, Dr. Faizan Javed, Dr. Hui Wu, Dr. Yuehua Lin, Dr. Shih-hsi Liu, Dr. Carl Wu, Dr. Francisco Hernandez and Ritu Arora, I thoroughly enjoyed working with you all and will carry the wonderful memories of the time we spent together for the rest of my life.

Finally, this dissertation research would not have been completed without the caring and able support from the Computer and Information Sciences Department,

especially Dr. John Johnstone, Janet Tatum, Kathy Baier and Fran Fabrizio, I am grateful for your care and help.

TABLE OF CONTENTS

Page

ABSTRACT iii

DEDICATION v

ACKNOWLEDGEMENTS vi

LIST OF FIGURES xiii

LIST OF ABBREVIATIONS xvii

CHAPTER

1 INTRODUCTION 1

1.1 Tools and Techniques to Support Modularization of Legacy Software 3
1.2 Challenges of Language-Independent Legacy Modernization 7
1.3 Research Objectives and Contributions 10
1.4 Overview of the Research 13

2 BACKGROUND 17

2.1 Aspect-Oriented Programming 17
2.1.1 Separation of Concerns 18
2.1.2 Crosscutting Concerns and the Join Point Model 20
2.2 Aspect Weaving 22
2.2.1 Current State-of-the-Art in Legacy AOP Modernization 23
2.2.2 Comparative Discussion of AOP Tools to Support Legacy Languages 25
2.3 Program Transformation - Design Maintenance System 28
2.3.1 DMS Key Features and Support for Abstract Syntax Trees 29
2.3.2 Specifying Rewrites using DMS Rule Specification Language 31
2.3.3 Other Program Transformation Engines 33
2.4 Model-Driven Engineering - AMMA 35
2.4.1 Kernel Meta-Meta Model 35
2.4.2 Textual Concrete Syntax 36
2.4.3 ATLAS Transformation Language 37
2.4.4 Related Work 39

TABLE OF CONTENTS (Continued)

Page

CHAPTER

3 PROGRAM TRANSFORMATION BASED ASPECT WEAVER CONSTRUCTION....41

3.1 An Aspect Weaver for Object Pascal....42
3.1.1 Crosscutting Concerns in Object Pascal Application....42
3.1.2 Weaver Transformation Rules for the Object Pascal Case Studies....47
3.2 Weaving into C++ Template Libraries....56
3.2.1 Simple Pointcut Expressions for C++ Templates....57
3.2.2 Advanced Pointcut for C++ Templates....60
3.2.3 Template Weaving using Program Transformation....66
3.3 Adaptation and Specialization of Scientific Libraries....71
3.3.1 Aspects in Blitz++....71
3.3.2 Specializing HPL using Program Transformation....79
3.4 Related Work....86
3.5 Limitations of Program Transformation Engines....89

4 GENERIC ASPECT WEAVER FRAMEWORK BASED ON MODEL-DRIVEN PROGRAM TRANSFORMATION....92

4.1 Role of MDE in Aspect Weaver Construction....93
4.1.1 Challenges and Overview of GenAWeave Framework....94
4.1.2 Program Transformation Back-End....96
4.1.3 Challenges of Program Transformation Engine Usage....97
4.2 Model-Driven Front-End....99
4.2.1 Metamodel for Front-End Aspect Language....99
4.2.2 Implementing the Front-End Aspect Language with AMMA....102
4.3 Model Transformation....105
4.3.1 Program Transformation Rule Generator....105
4.3.2 Target Metamodel for RSL....106
4.3.3 Model Transformation using ATL....108
4.3.4 Remaining Challenges to be Addressed by the Framework....111
4.4 Extending to a Generic Framework....112
4.4.1 Support for a Generic Aspect Front-End....114
4.4.2 Generalizing the Rule Generator Design....120
4.4.3 Support for a Reusable Back-End....126
4.5 Experimental Evaluation – Object Pascal and FORTRAN Weavers....128
4.5.1 Object Pascal Weaver....129
4.5.2 FORTRAN Weaver....133
4.5.3 Join Point for Loops....135
4.5.4 Discussion of Experimental Results....138

TABLE OF CONTENTS (Continued)

Page

CHAPTER

4.6 Integrating the GenAWeave Framework within Eclipse 141
4.7 Related Work 143

5 FUTURE WORK 147

5.1 Improving Reusability of the Generic Aspect Metamodel 148
5.2 Improving Reusability of the ATL Rule Generator 149
5.3 Constructing Weavers for other GPLs 151
5.4 Applying the approach to DSALs 152
5.5 Applying the approach to Scientific Computing Applications 152
5.6 Generic Refactoring and Generic Aspect-Mining Engines based on Model-Driven Program Transformation 153

6 CONCLUSION 155

6.1 Challenges addressed by the GenAWeave Framework 156
6.2 Lessons Learned 158

LIST OF REFERENCES 161

APPENDIX

A ASPECT PASCAL METAMODEL SPECIFICATIONS 174

A.1 Generic Aspect Metamodel KM3 Specification 175

A.2 Aspect Pascal KM3 Specification 177

A.3 Aspect Pascal TCS Specification 178

B ASPECT FORTRAN METAMODEL SPECIFICATIONS 183

B.1 Aspect FORTRAN KM3 Specification 184

B.2 Aspect FORTRAN TCS Specification 185

C RSL METAMODEL SPECIFICATION FOR BACK-END PTE 189

C.1 KM3 Specification for RSL 190

C.2 TCS Specification for RSL 192

TABLE OF CONTENTS (Continued)

Page

APPENDIX

D MODEL TRANSFORMATION RULES FOR ASPECT PASCAL AND ASPECT FORTRAN WEAVER 196

D.1 ATL Rule for Translating Call Expression Join Point in Aspect Pascal 197

D.2 ATL Rule for Translating Loop Expression Join Point in Aspect Pascal 202

D.3 ATL Rule for Translating Loop Expression Join Point in Aspect FORTRAN 208

E BACK-END WEAVER TRANSFORMATION FUNCTIONS 215

E.1 PARLANSE Reusable External Functions 216

E.2 PARLANSE External Function for Loop Execution Join Point 219

F DMS PARLANSE FUNCTIONS TO SPECIALIZE HPL 225

F.1 PARLANSE External Function to Remove Macro Definitions from HPL 226

F.2 PARLANSE External Function to Specialize HPL 228

LIST OF FIGURES

Figure *Page*

1-1 The current state of legacy software....2

1-2 Overview of model-driven aspect weaver framework....13

1-3 Overview of topics discussed in dissertation....15

2-1 AOP and separation of concerns....19

2-2 AspectJ specification to capture logging in BizObject's methods....22

2-3 Overview of back-end transformation process....31

2-4 A simple example of a program transformation rule that illustrate aspect weaving of *before* advice....32

2-5 An example of ATL transformation....38

3-1 Progress meter updating....43

3-2 Exception handling code for processing dialog....43

3-3 Logging of SQL query Data Definition Language (DDL) statements....44

3-4 Preamble for widget button clicks....45

3-5 Synchronization in a database error handler....47

3-6 Transformation rule for updating progress meter....48

3-7 Transformation rule for SQL logging....50

3-8 Visitor function written in PARLANSE....51

3-9 RSL rule for weaving dirty bits....53

3-10 PARLANSE external condition function *func_sig_has_click*....54

3-11 RSL rule for modularizing synchronization 55

3-12 An example class with multiple template instantiations 58

3-13 STL vector class and its usage 60

3-14 STL vector$1 class with updated references in Application instances 62

3-15 Scope designators in pointcut expressions 63

3-16 Aspect specification for inserting the push_back log to all vectors of ANY type in ANY class 64

3-17 Pointcut specification for weaving into all vectors of type int in ANY class 65

3-18 Pointcut specification for weaving into all vectors of type int in class A 65

3-19 Pointcut specification for weaving into all vectors of type int in method foo of class A 65

3-20 Pointcut specification for weaving into vectors of type int and referenced by variable ai in method foo of class A 65

3-21 Overview of template weaving process 66

3-22 DMS transformation rules for weaving log statement into push_back method 68

3-23 DMS transformation rules to update the application program 70

3-24 Precondition check and setupStorage in Blitz++ array implementation 75

3-25 Aspect specification for precondition and memory allocation in templates 75

3-26 Redundant assertion check on base template specification 76

3-27 Aspect specification for redundant assertion checks 76

3-28 Subset of base pattern used to generate the vector operation template 78

3-29 Rules used to generate mathematical operations using a base template definition 79

3-30 Preprocessor directives in a HPL software package 81

3-31 Specialization overview of HPL software package....82

3-32 Transformation rule for specializing macro definitions in HPL 83

3-33 Comparing size of HPL BLAS library before and after specialization....84

3-34 Time and performance analysis between HPL-ALL vs. HPL-CBLAS 85

4-1 Overview of our model-driven aspect weaver framework....95

4-2 Subset of Aspect Pascal metamodel represented as a class diagram 100

4-3 KM3 specification (snippet) for Aspect Pascal....103

4-4 TCS specification (snippet) for Aspect Pascal....104

4-5 Subset of the RSL metamodel (as a class diagram) 107

4-6 Model transformation scenario for generating RSL rules from aspects....108

4-7 ATL transformation (snippet) from Aspect Pascal to RSL....110

4-8 Class Diagram (snippet) of Aspect Pascal extending from a common Generic Aspect metamodel....116

4-9 Metamodel (snippet) of Aspect FORTRAN conforming to a common Generic Aspect metamodel....118

4-10 TCS specification showing differences in concrete syntax for Aspect FORTRAN (top) and Aspect Pascal metamodel (bottom) 118

4-11 ATL specification used to generate lower-order transformation rules (RSL) for weaving Object Pascal source program 123

4-12 ATL specification used to generate lower-order transformation rules (RSL) for weaving FORTRAN source program....124

4-13 Reusable external function in the GenAWeave framework....127

4-14 Aspect to capture progress meter updating 130

4-15 Aspect Pascal model generated from Aspect Pascal source program 131

4-16 Aspect to capture synchronization in a database error handler 132
4-17 Generated Aspect Pascal model from Aspect Pascal source program 133
4-18 Encryption of messages during MPI_SEND 134
4-19 Aspect to enable encryption during MPI calls 135
4-20 Adding timer around do loops 136
4-21 Aspect to add timer around do loops 137
4-22 Aspect FORTRAN model generated from source aspect program 137
4-23 A comparative analysis of model transformation rules 139
4-24 Reusability summary for FORTRAN and Object Pascal weavers 141
4-25 GenAWeave framework within Eclipse 142
4-26 Syntax errors displayed within the editor 143
5-1 Improving the front-end metamodel design 149
5-2 An example showing ATL rule inheritance 150

LIST OF ABBREVIATIONS

ANT	Another Neat Tool
AMMA	ATLAS Model Management Architecture
AMW	Atlas Model Weaver
ANTLR	ANother Tool for Language Recognition
AOL	Aspect-Oriented Language
AOM	Aspect-Oriented Modeling
AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
API	Application Programming Interface
ASF	Algebraic Specification Formalism
AST	Abstract Syntax Tree
ASTM	Abstract Syntax Tree Metamodel
ATL	Atlas Transformation Language
BLAS	Basic Linear Algebraic Subprograms
BNF	Backus-Naur Form
CBLAS	C implementation of BLAS
CLI	Common Language Infrastructure
CMOF	Complete Meta-Object Facility
DDL	Data Definition Language
DDMM	Domain Definition Metamodel

DLL	Dynamic Link Library
DMS	Design Maintenance System
DOM	Document Object Model
DSAL	Domain-Specific Aspect Language
DSL	Domain-Specific Language
DSM	Domain-Specific Modeling
EMF	Eclipse Modeling Framework
EMOF	Essential Meta-Object Facility
FBLAS	Fortran implementation of BLAS
GME	Generic Modeling Environment
GMF	Graphical Modeling Framework
GPAL	General-Purpose Aspect Language
GPL	General-Purpose Language
GUI	Graphical User Interface
HPL	High Performance Linpack
IDE	Integrated Development Environment
IL	Intermediate Language
JPM	Join Point Model
JPS	Join Point Shadows
JTS	Jakarta Tool Suite
KM3	Kernel Meta-Meta Model
LOC	Lines of Code
MDA	Model-Driven Architecture

MDE	Model-Driven Engineering
MDR	Metadata Repository
MOF	Meta-Object Facility
MPI	Message Passing Interface
OCL	Object Constraint Language
OMG	Object Management Group
PARLANSE	PARallel LANguage for Symbolic Expressions
PTE	Program Transformation Engine
QVT	Query View Transformations
RSL	Rule Specification Language
SDF	Syntax Definition Formalism
SOC	Separation of Concerns
SQL	Structured Query Language
STL	Standard Template Library
TCS	Textual Concrete Syntax
TS	Technical Space
TXL	Tree TRANSformation Language
UML	Unified Modeling Language
VSIPL	Vector Signal Image Processing Library

CHAPTER 1

INTRODUCTION

A long-standing goal of software engineering is to construct software that is easily modified and extended. The desired result is to achieve modularization such that a change in a design decision is isolated to one location of a program [Parnas, 1972]. The proliferation of software in everyday life (e.g., embedded systems in avionics software [Sharp, 2000], and software that enables commercial transactions [Arranga, 2000]) has augmented the conformity and invisibility of software. As demands for software increase, future requirements will necessitate new strategies for improved modularization and restructuring in order to support the requisite adaptations [Chaplin *et al*., 2001; Masuhara and Kiczales, 2003; Cazzola *et al*., 2005].

Moreover, software maintenance is a very costly and time consuming part of the software life cycle [Schach and Tomer, 2000]. The problems are even more compelling when maintaining legacy code [Ulrich, 2002], which adds a significant cost to many engineering organizations [Capra *et al*., 2007]. Recent IT market research [McKinsey, 2004; CHAOS, 2006] shows that customization, including legacy modernization and integration projects, represents more than 50% of the annual $230 billion spent on all software budgets. Average maintenance costs are higher than development costs. This is because organizations spend an enormous amount of time and money to understand the details of implementing a software system before deciding what changes were needed to

realize them. Over a period of twenty years of research in the field of software maintenance, Lehman et al. published the eight Laws of Evolution [Lehman *et al*., 1997]. Key findings of his research include that maintenance is really a series of evolutionary developments and that maintenance decisions are aided by understanding what happens to systems (and software) over time. Lehman et al. demonstrated that systems must be *continually adapted*, else they become progressively less satisfactory and are subject to decay over a period of time. Moreover, as they evolve, systems become more complex unless some action is taken to reduce the complexity.

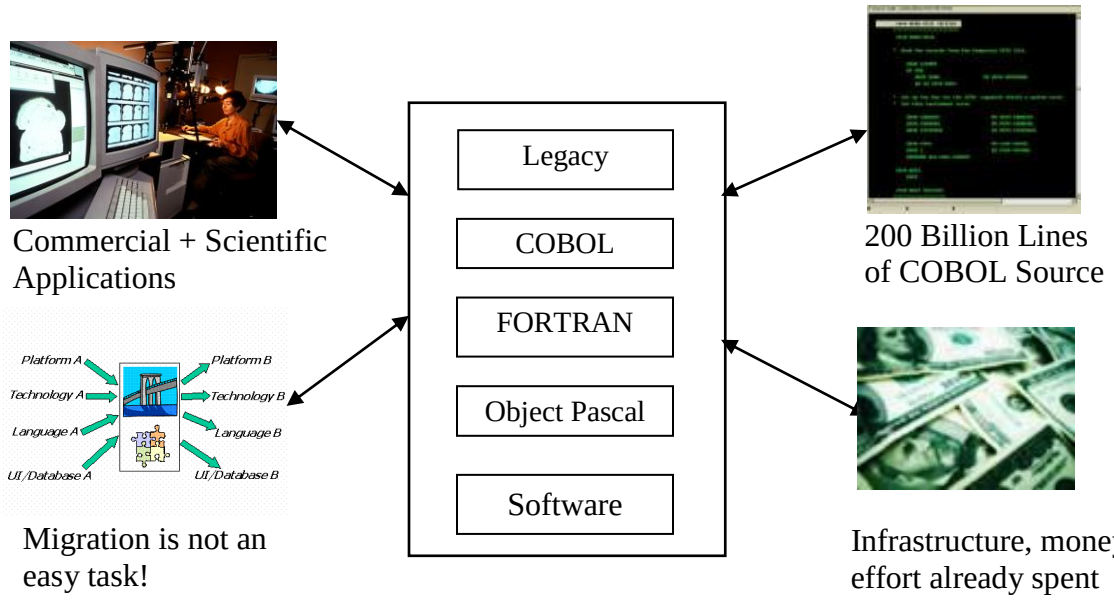


Figure 1-1 **–** The current state of legacy software

Figure 1-1 shows this current state of legacy software. Presently, there are literally multiple billions of lines of legacy code maintained in hundreds of disparate languages and paradigms [Lämmel and Verhoef, 2001]. Such an enormous amount of legacy code exists in large proportions in commercial, embedded and scientific domains. In fact, a

Gartner report, as cited in [Ulrich, 2002], estimates that there are at least 200-250 billion lines of existing COBOL code in production use (of course, the *total* amount of legacy code is much higher when other languages are considered). A considerable amount of time, effort and money have been invested in building large infrastructure to construct legacy software.

In spite of the significant investments and proliferation of legacy software systems, the majority of language researchers and tool vendors have focused their attention on just a few popular languages, such as C++ and Java. A naïve proposal would attempt to migrate the existing legacy code into a modern object-oriented language like Java. Such a proposition is often not possible due to cultural and political concerns within the institution that owns the legacy system [Ulrich, 2002]. There is a strong need for improving the process of software maintenance by using modern software engineering tools and advanced modularization techniques that may reduce development time, save valuable labor resources, and improve the quality of the software system [Schutter and Adams, 2007].

1.1 Tools and Techniques to Support Modularization of Legacy Software

To support software adaptation and evolution of legacy software [Baxter, 1992], new paradigms such as Aspect-Oriented Software Development (AOSD) (http://www.aosd.net), have shown initial promise in assisting a developer in isolating points of variation and configurability [Filman *et al.*, 2004]. It has been observed that some programming languages provide modularization mechanisms that force other non-orthogonal concerns to be scattered and tangled across a code base [Tarr *et al.*, 1999;

Giese and Vilbig, 2006]. For example, a credit card processing system's core concern would process payments, while its system-level concerns would handle logging, transaction integrity, authentication, security and performance. Many such concerns tend to affect multiple implementation modules. Using current object-oriented programming methodologies, these non-orthogonal concerns span over multiple modules, resulting in systems that are harder to design, understand, implement, and evolve.

Aspects are a new language construct for cleanly separating concerns that crosscut the modularization boundaries of an implementation [Kiczales *et al.*, 2001]. In a fundamentally new way, aspects permit a software developer to quantify, from a single location, the effect of a concern across a body of code [Filman and Friedman, 2004], thus improving overall modularization. A translator called a *weaver* is responsible for merging the separated aspects with the base code. The focus of the research presented in this dissertation is primarily based on the idea of constructing aspect weavers for multiple legacy languages using a generalized approach.

An additional approach for improving software modularization is refactoring, which has been defined as "the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure" [Fowler *et al.*, 1999]. Refactoring has transitioned from research into everyday software practice as an effective means for evolving software and improving modularization [Opdyke, 1992; Mens and Tourwe, 2003; Dig and Johnson, 2005]. Refactoring features are built into many modern development environments (e.g., Java refactoring support in Eclipse – http://www.eclipse.org). However, like AOSD, mature refactoring engines

support only a couple of prominent languages; consequently, many other languages are ignored, resulting in an inability to apply modern principles to legacy source.

Research into software transformation 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 [Griswold and Notkin, 1993]. Such software transformation techniques can be categorized as being either horizontal, or vertical [Czarnecki and Helsen, 2006]. The research into horizontal transformation pertains to modification of a software artifact at the same abstraction level. This is the typical connotation when one thinks of the term *transformation*, with examples being code refactoring at the implementation level [Fowler *et al.*, 1999], and model transformation at a higher design level [Agrawal, 2003]. Horizontal transformation systems often lead to invasive composition of the software artifact [Aßmann, 2003]. Vertical transformation is typically more appropriately called *translation* (or synthesis) because a new artifact is being synthesized from a different abstraction level (e.g., model-driven software synthesis [Neema *et at*., 2002], and reverse engineering). Vertical transformations are more generative in nature [Czarnecki and Eisenecker, 2000]. There has been prior research that combines both levels as demonstrated by [Gray *et al*., 2003], i.e., horizontal transformation from the investigation into aspect model weaving, and vertical translation from model-driven code generation.

Typically, Program Transformation Engines (PTEs) are well-suited for modifying artifacts at a lower-level of abstraction (i.e., program source code) [Baxter *et al.*, 2004]. Model-Driven Engineering (MDE) is useful for transforming artifacts at a higher-level of abstraction (i.e., software modeling level) [Schmidt, 2006]. An investigation of the

underlying science that combines these two technical spaces (i.e., source code and software models) may realize more benefits than each considered separately. Therefore, transformation techniques that operate jointly at both levels of software abstraction can benefit from the collective usage of both PTEs and MDE. This is exactly the vision of the research presented in this dissertation. The research utilizes the combined power of mature PTEs and MDE tools to modernize legacy software artifacts. Higher-order transforms at the modeling level are used to generate lower-order transforms operating at the source code level.

In particular, the research described in this dissertation provides a general framework to construct aspect weavers based on model-driven program transformation for legacy software artifacts. The term framework is both conceptual and practical in nature. The conceptual part of the framework refers to the preferred approach towards constructing aspect weavers in a generalized way. According to [Fayad and Schmidt, 1997], "A framework is a reusable, 'semi-complete' application that can be specialized to produce custom applications." In the context of the above definition, the practical nature of the GenAWeave framework refers to the reusable design that includes a generic metamodel and model transformation libraries that produce concrete program transformation rules for various General-Purpose Languages (GPLs). In addition, the framework provides extension points that allow new features to be added to existing weavers.

One of the aims of a general framework is to define a set of reusable components (i.e., generic transformations and generic metamodel) that can be suitably applied across

a wide of range of languages. However, there are several key challenges towards the construction of a generic framework and are summarized in the following section.

1.2 Challenges of Language-Independent Legacy Modernization

As a result of programming language research over the past fifty years, a veritable "Tower of Babel" exists with multiple billions of lines of legacy code maintained in hundreds of different languages [Lämmel and Verhoef, 2001]. In fact, legacy languages are estimated to account for a large percentage of existing production software [Ulrich, 2002]. Yet, the majority of Aspect-Oriented Programming (AOP) [Kiczales *et al*., 1997] research is focused on just a few modern languages, such as Java. A generalized approach that brings aspects to legacy software is still missing. An attempt to migrate legacy code into modern object-oriented languages is often resisted by organizations because of their prior investments to build such systems. Moreover, migration is a complex task and may induce new undesirable faults that were not earlier present in a stable legacy code base.

Rather than bringing the code to existing Java-based weavers, an alternative is to take AOP principles to the legacy languages and tool environments [Schutter and Adams, 2007; Gray and Roychoudhury, 2004]. Given the large number of languages in use, a solution that mitigates the effort needed to create each new weaver is more desirable than an approach that manually recreates a weaver from scratch for each legacy language. Programming languages can be clustered into classes with structural and/or semantic similarities and the knowledge that is gained while constructing a weaver for one specific language can be reused during another construction for a different language. However, there are several key challenges to providing an initial methodology that allows

experimentation with aspects in languages other than Java. Some of the research questions that immediately follow are:

- Is there a technique to construct aspect weavers for legacy languages without extending or inventing a new parser (or compiler) from scratch?
- Can such a construction be supported in a more generic or language-independent way?
- Can the knowledge of building a weaver from a previous construction be re-used in a different language and platform context?

This core set of questions lead to four major obstacles toward adoption of aspects for legacy software. These obstacles and challenges are summarized in the remainder of this subsection.

Challenge C1 - The Parser Construction Problem: Building a parser for a toy language, or a subset of an existing language, is not difficult. However, designing a parser that is capable of handling millions of lines of production legacy code is an onerous task. As observed in [Lämmel and Verhoef, 2001], "Measuring this and other projects, it became clear to us that the total effort of writing a grammar by hand is orders of magnitude larger than constructing the renovation tools themselves. *So the dominant factor in producing a renovation tool is constructing the parser*." Moreover, constructing analysis and modification tools for software assets is a laborious process. The first thing that is required to implement a new maintenance tool is the underlying parser for a specific programming language. Parser development for any of the legacy languages in use today implies a major up-front investment [Lämmel and Verhoef, 2001]. For assessment purposes, software developers who want to explore the capabilities of aspects

in legacy systems will require industrial-scale parsers to allow them to evaluate the feasibility of adoption within their organization. Incomplete parsers for small research prototypes will not scale and may leave a negative first impression of aspects.

Challenge C2 - The Weaver Construction Problem: When a new program restructuring or modularization idea is conceived (e.g., AOP), it is often desired to impart the idea to older legacy applications. In order to realize such an objective, a capability is needed to perform the underlying transformations and rewrites on a syntax tree or on an abstract model. This is not an easy task and requires considerable effort to provide a sound infrastructure for program transformation. However, it is often the case that the integration efforts to support a core set of transformations are repeated for each language to which the new idea is applied. Such repetition of effort is unfortunate and strongly suggests the need for further generalization of transformation objectives.

Challenge C1 and part of *Challenge C2* can be addressed by using program transformation techniques [Gray and Roychoudhury, 2004], where, full-fledged parsers available in program transformation frameworks can be reused to assist in constructing aspect weavers. PTEs generally have support for low-level rewriting (i.e., by using term-rewriting or graph-rewriting [Aßmann and Ludwig, 1999]) that can be used to construct aspect weavers for multiple GPLs. However, the abstraction level at which these transformation systems operate is too low for general-purpose software development. Moreover, the core set of transformations has to be customized for each language. These two additional challenges are discussed in the following paragraphs.

Challenge C3 – Accidental Complexities of Transformation Specifications: An inherent difficulty associated with using program transformation engines is the low-level

of abstraction at which a transformation rule is specified. Due to many accidental complexities, program transformations typically are at an improper level of abstraction for general use by programmers. It is therefore desirable to provide a higher-level of abstraction to the user. The research presented in this dissertation increases the level of abstraction of transformation rules in the form of a high-level aspect language that is used to specify the aspect program that facilitates software adaptation.

Challenge C4 – Language-Independent Generalization of Transformation Objectives: Although most program transformation engines provide a general toolkit with pre-existing parsers, the transformation rules that actually perform the desired restructuring are encoded to the productions of a specific concrete syntax (i.e., grammar of the base language; the term *base* language refers to the language for which the aspect weaver is constructed). Thus, all of the effort that is placed into creating the transformations to enable weaving cannot be reused in other language domains. A key research contribution of this dissertation is an approach that brings higher-order transformations at the modeling level to increase the level of reuse among concrete lower-order transformation rules across multiple languages.

1.3 Research Objectives and Contributions

The research described in this dissertation makes a contribution to evolving maintenance goals of legacy software systems by adopting an aspect-oriented approach. The direction of the research focuses on constructing a generalized aspect-weaving framework based on model-driven program transformation that is applicable to legacy software artifacts. The intellectual merit of this research is an investigation of the

underlying science to support reusable generic transformations to improve the modularity and adaptive maintenance of legacy software. The following objectives represent the core contributions of the research:

- An exploration of the underlying science to support an abstract source model, generic transformations, and high-level aspect language as a foundation for generic software transformation and analysis;
- A generative methodology to permit construction of aspect weavers for multiple languages; provide a language experimentation environment for investigating ideas in new paradigms without constructing all of the underlying parsing and transformation mechanisms from scratch.

The research will guide tool developers to extend support for AOP in legacy languages by adopting the generalized weaver construction framework described in this dissertation. The scalability of the framework is provided by using a powerful PTE, namely, the Design Maintenance System (DMS) [Baxter *et al.*, 2004], which represents the back-end of the framework. DMS provides support for mature language tools (e.g., lexers, parsers, and analyzers) for more than a dozen programming languages. It has been used to parse several million lines of code written in many of these languages. A background discussion of DMS is provided in Chapter 2.

The adoption of DMS as a back-end provides a solution to *Challenge C1* through immediate availability of industrial-scale parsers. DMS also provides functionality for transforming a program after it has been parsed. In particular, the Rule Specification Language (RSL, introduced in Chapter 2) available in DMS provides the low-level support required to modify the source program. RSL offers a partial solution to *Challenge*

C2. Thus, *Challenge C1* and part of *Challenge C2* as enumerated in Section 1.1 are gratuitously resolved through the adoption of a mature program transformation engine into the weaver construction framework [Gray and Roychoudhury, 2004].

However, there remain additional challenges that are typically associated with any PTE. For example, the RSL used in DMS works at a low-level of abstraction. That is, users of RSL need to understand the grammar of the base language and the underlying parsing algorithms in order to write transformation rules required to modify the source programs. This introduces unavoidable accidental complexities and makes it increasingly difficult for general developers to specify aspects (i.e., transformation objectives) at this low-level (*Challenge C3*). The research presented in this dissertation makes a contribution towards resolving this challenge by providing a high-level aspect language on top of RSL (i.e., a façade to RSL) that hides all of the accidental complexities and minute details associated with RSL.

Moreover, the transformation rules that are constructed to perform aspect weaving are generally tied to the grammar of the base language. Such tight coupling of RSL with the base language grammar impedes reusability because all of the previously constructed rules to enable weaving cannot be reused in other language domains (*Challenge C4*). The research presented here makes a key contribution towards resolving *Challenge C4* by providing a generative technique where higher-order transformation rules specified at the modeling level are used to generate concrete lower-order RSL transformation rules working at the source code level. These higher-order model transformation rules are simpler than RSL and can be reused in other language domains. The next section gives an

overview of the research and outlines how the remaining chapters are organized in this dissertation.

1.4 Overview of the Research

To provide a platform to explore a generalized mechanism for constructing aspect weavers for GPLs, the research combines two key techniques, namely model-driven engineering and program transformation. Figure 1-2 shows a high-level overview of the framework. A model-driven front-end (item 1 in Figure 1-2) is used to capture the syntax of an aspect program in the form of an aspect model. The aspect model in turn conforms to an aspect metamodel that is defined using the ATLAS Model Management Architecture (AMMA, discussed in Chapter 2) [Kurtev *et al*., 2006].

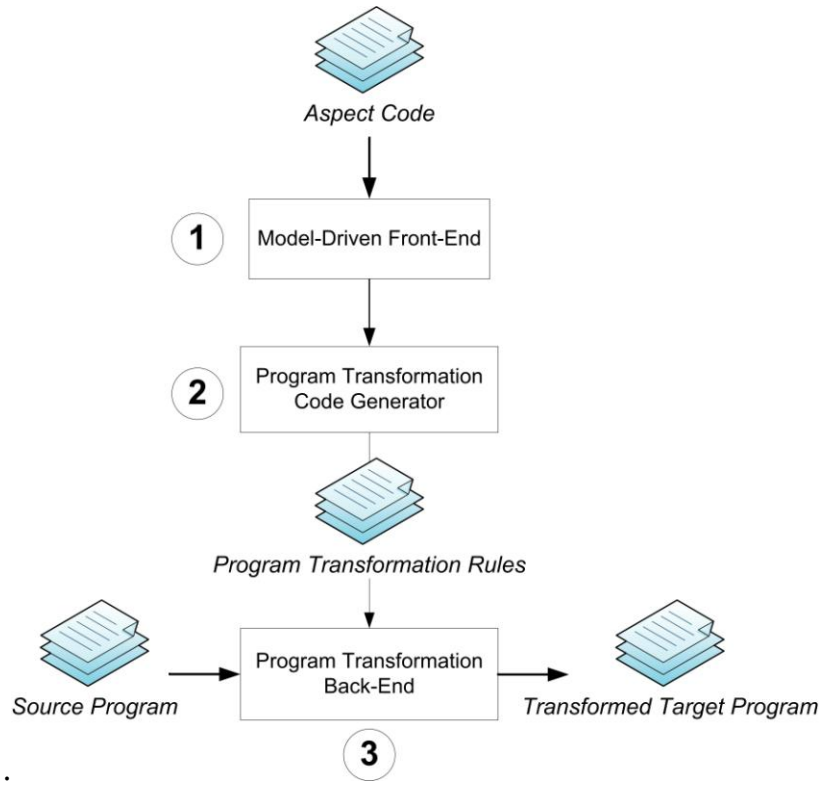


Figure 1-2 - Overview of model-driven aspect weaver framework

The high-level aspect language is used to raise the level of abstraction and hide the accidental complexities that are associated with the RSL program transformation rules processed by the back-end (item 3 in Figure 1-2) of the framework. As a direct benefit of using an aspect language, users can specify their weaving intent at a higher-level of abstraction instead of using low-level RSL code (i.e., a solution to *Challenge C3*).

The heart of the framework is a higher-order program transformation rule generator (item 2 in Figure 1-2) that produces program transformation rules (RSL) from an input aspect program (i.e., a solution to *Challenge C4*). Much of the program transformation rule generator can be reused from one GPL to another (discussed in detail in Chapter 4). The aspect code is initially parsed by the front-end and later analyzed by the rule generator. The result is a set of generated RSL rules that serve as input along with the source program to the back-end PTE to accomplish the desired weaving. A more detailed description of the framework will be discussed in Chapter 4 of the dissertation.

Each of the key components (shown as items 1, 2, and 3 in Figure1-2) of this framework will be discussed in detail in Chapters 2, 3 and 4, including their primary benefits and internal mechanisms. The future chapters will also outline the reasons behind choosing each of these core components and explain why it is desirable to follow a combined MDE / Program Transformation philosophy to construct aspect weavers for GPLs.

Figure 1-3 provides an overview of the topics discussed in the dissertation. Chapter 2 introduces the necessary background information to provide the reader with a better understanding of other sections of the dissertation. The first part of Chapter 2

introduces the concepts behind AOP and aspect weaving. The chapter also introduces the key components of a PTE, specifically, RSL and other key features of DMS that are useful in transforming source code. The last part of Chapter 2 introduces MDE, in general, and the AMMA toolsuite [Kurtev *et al*., 2006], in particular.

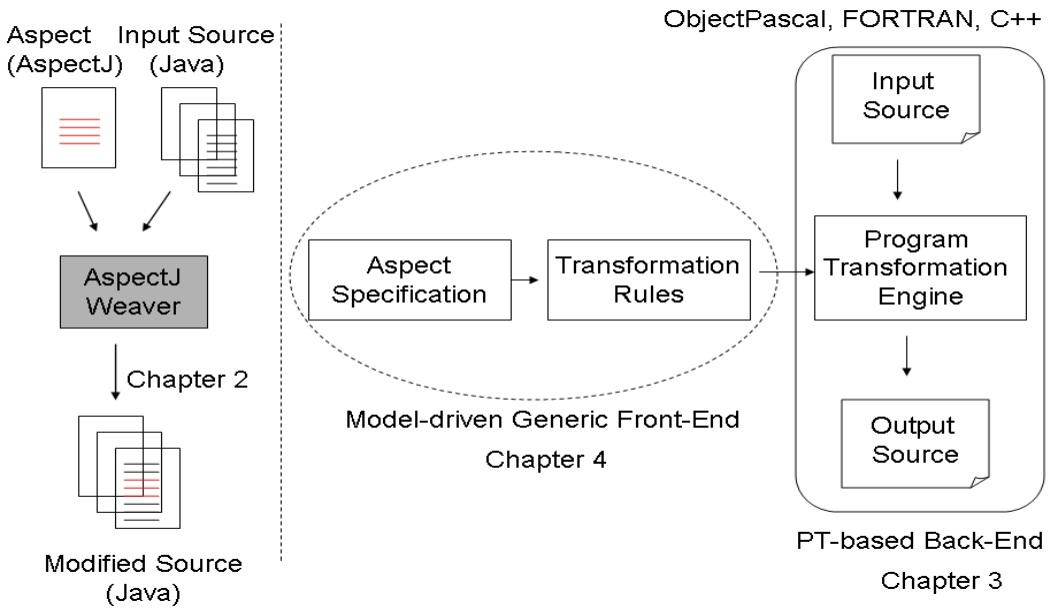


Figure 1-3 - Overview of topics discussed in dissertation

Chapter 3 describes how individual weavers for specific programming languages can be constructed by the use of program transformation techniques alone. The chapter describes in detail the construction of an aspect weaver for Object Pascal and provides several case study examples. The chapter also describes a technique to construct an aspect weaver for C++ templates and how scientific computing application such as Blitz++ [Veldhuizen, 1998] can benefit from this technique. Finally, the chapter presents a well-known scientific computing library - High Performance Linpack (HPL)

Benchmark [Petitet *et al*., 2004], and demonstrates how specialization of HPL can be accomplished using program transformation techniques. All of these examples give evidence for the usefulness of a program transformation based approach to modularize legacy software, ranging from commercial to high-performance computing domains.

Chapter 4 is the core of the research and describes how the model-driven approach that is layered on top of program transformation engines is used to construct reusable, extensible and partly generic aspect weavers for GPLs. The chapter describes in detail the front-end architecture, the back-end architecture and the program transformation rule generator used in the framework. The chapter also provides useful statistics used to evaluate the framework by constructing aspect weavers for Object Pascal and FORTRAN.

Chapter 5 outlines some of the existing limitations of this work and provides a roadmap for future extensions and possible areas of applicability. Chapter 6 offers a concluding summary of the research contributions. There are also several appendices: Appendix A-E provides the implementation-specific source code used to construct the front-end and back-end of the GenAWeave framework. Appendix F shows implementation details of the low-level DMS external functions required to specialize HPL [Petitet *et al*., 2004].

CHAPTER 2

BACKGROUND

The contribution of this dissertation represents research that unites MDE [Schmidt, 2006] and program transformation techniques [Baxter *et al.*, 2004; Gray and Roychoudhury, 2004] to support AOP [Kiczales *et al*., 1997; Filman *et al*., 2004] among modern and legacy programming languages. This chapter provides a background discussion of the three different technologies involved in this research. Section 2.1 presents the general concepts of AOP. Section 2.2 describes the underlying construction mechanism for different types of aspect weavers. Specifically, the section describes the current state-of-the-art techniques adopted by various researchers to construct aspect weavers. Section 2.3 introduces DMS, which is the low-level PTE used in this research. Section 2.4 introduces MDE and in particular the AMMA toolsuite, which forms the front-end of the generic aspect weaving framework.

2.1 Aspect-Oriented Programming

AOP is a programming technique that allows programmers to modularize crosscutting concerns (i.e., program features that cut across the typical divisions of modularity, such as logging [Laddad, 2003]). Such features often cannot be cleanly decomposed from the rest of the system in both the design and implementation, and result in either scattering or tangling of source code [Kiczales *et al*., 1997]. In AOP,

crosscutting concerns are encapsulated in separate reusable modules called aspects. The AOP paradigm was introduced to strengthen the notion of separation of concerns (SOC) [Tarr *et al.*, 1999; Giese and Vilbig, 2006], which necessitates breaking down a program into distinct parts with minimum overlap in functionality. The term SOC is believed to be coined by Edsger Dijkstra in his 1974 paper "On the Role of Scientific Thought," which was later published in [Dijkstra, 1982].

2.1.1 Separation of Concerns

The principle of SOC states that a model of an application should be organized as a set of distinct modular units where each unit encapsulates one particular concern or functionality of the application. A concern is any piece of interest or focus in a program. The advantage of this approach is that the description of a feature is localized and is therefore more easily adapted.

Many SOC approaches have been proposed during the past several years, such as Subject-Oriented Programming and design [Ossher *et al.*, 1994], Feature-Oriented Programming [Batory, 2003], Composition Filters [Bergmans and Aksit, 2001] and Multidimensional Separation of Concerns [Tarr *et al*., 1999]. However, the focus of the research in this dissertation is centered around AOP [Kiczales *et al*., 1997], in particular, by providing a generic framework to extend AOP support for legacy and modern programming languages.

The challenge addressed by AOP arises from the fact there are certain concerns in an application that are typically crosscutting and are difficult to separate. For instance, if writing an application for handling employee records, the bookkeeping and indexing of

such records is a core concern, while logging a history of changes to the employee database, or an authentication system, would be crosscutting since they overlap more parts of the program. These concerns are typically tangled or scattered across the system and are difficult to separate from the core functionality. Figure 2-1 illustrates the undesired effect of crosscutting concerns and suggests a possible solution.

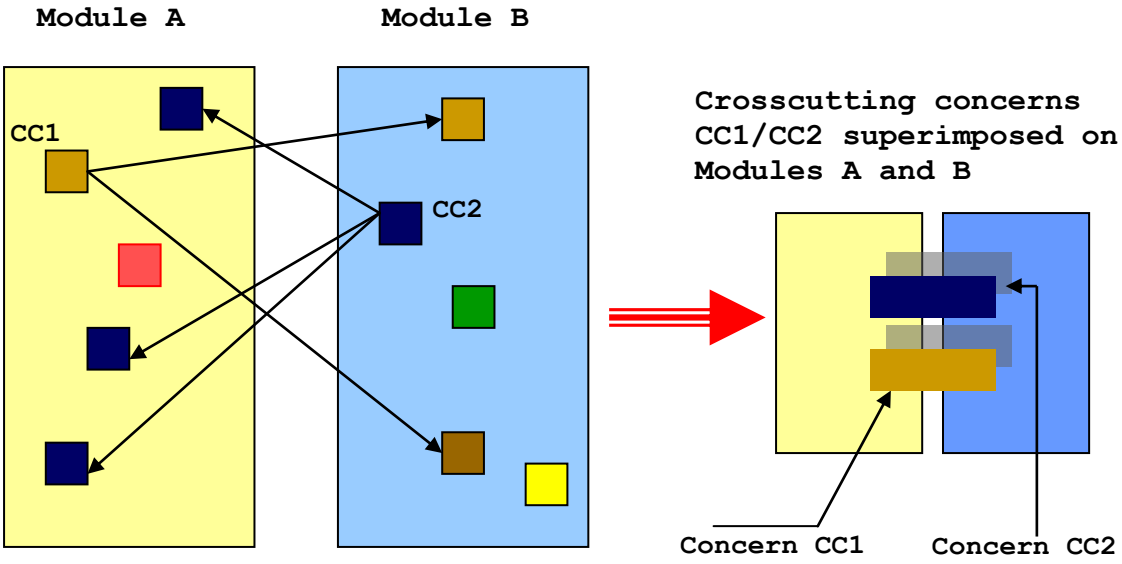


Figure 2-1 – AOP and separation of concerns

The above figure shows two modules of the same application. Each module addresses several concerns or functionality in the application. Some of these functionalities are cleanly captured in each module (i.e., the isolated square boxes) while others spread across the module boundaries. For example, the concern CC1 which exists in Module A is also scattered and tangled across Module B. Similarly, concern CC2 which exists in Module B is scattered and tangled across Module A. Ideally, it is desired that concerns (features) that are encapsulated in a single modular unit do not spread across features that are encapsulated in a different modular unit. Unfortunately, this is

usually not possible with crosscutting concerns using traditional programming constructs. A more common situation is the one shown in Figure 2-1 where a crosscutting concern CC1 in modular unit A is scattered across (crosscuts) the features of module B. A more desired solution is schematically shown in the right side of the figure. In this case, the crosscutting concerns CC1 and CC2 are captured in a separate aspect module and are then superimposed or weaved into modules A and B. Such separation of crosscutting concerns is the main motivation behind AOP.

2.1.2 Crosscutting Concerns and the Join Point Model

The undesired consequence that crosscutting concerns can cause to software systems can multiply with the size of the application. In fact, crosscutting concerns are treated as second class citizens in most languages and there is no explicit representation for their modularization. For example, logging the method entry and exit points in a very large system may lead to scattering of the logging concern across other useful features present in the code base [Eaddy *et al*., 2007]. This may introduce unnecessary cohesion in the system resulting in poor modularization [Schach, 1996]. In addition to logging, examples of other crosscutting concerns that are difficult to modularize using traditional object-oriented languages are security checks, transaction management, pre-fetching and disk quota operations [Coady and Kiczales, 2003]. It is often desirable to have a way to create a single separate module that describes all of the functionality of a crosscutting concern.

Aspect-oriented techniques provide new language constructs to cleanly separate concerns that crosscut the modularization boundaries of an implementation. In a

fundamentally new way, aspects permit a software developer to quantify, from a single location, the effect of a concern across a body of code, thus improving the modularization of crosscutting concerns. Some of the common constructs in AOP approaches, such as AspectJ (an aspect language for Java) [Kiczales *et al.*, 2001], include the following:

Join Point: Specific points of the execution in a program, such as a method invocation or a particular exception being thrown. Join points typically express the location of crosscutting concerns.

Pointcut: Means of identifying a set of join points through a predicate expression.

Advice: Defines actions to be performed at associated join points. Advice represents the behavior of crosscutting concerns.

Inter-type declarations: The ability of aspects to add methods, constructors and fields to existing types. They can also be used to implement interfaces and to declare super-types. However, inter-type declarations are not investigated in the research presented in this dissertation and may be explored in future.

Aspect: A modularization of a concern for which the implementation might cut across multiple boundaries; generally defined by pointcuts and advice.

Using the above terminology, an example as shown in Figure 2-2 depicts a simple logging aspect in AspectJ. The pointcut `log` captures all join points that correspond to calls on public `BizObject` methods (the '`*`' represents a wildcard, and '`..`' represents any number of parameters).

The `before` and the `after` advice binds the pointcut to specific actions to be performed just before and after each join point is reached (i.e., before and after BizObject's method call invocation). The entire crosscutting concern is captured in a

single aspect called `Logger`. The different constructs introduced in this example illustrate the benefit of AOP in modularizing systems that exhibit such crosscutting. Thus, instead of a scattered representation of a logging concern, aspects help to modularize concerns by capturing them in separate modules [Laddad, 2003]. This directly improves software maintainability by reducing the code size and improving the code comprehensibility. Separation of crosscutting concerns also makes the system easier to change and evolve. As an indirect effect, reduced code may also lead to a smaller memory footprint that can increase the performance of software.

```
1. aspect Logger {
2.   pointcut log(): call(public * BizObject.*(..));
3.   before(): log() {
4.      System.out.println("before calling BizObject methods");
5.    }
6.   after(): log() {
7.      System.out.println("after calling BizObject methods");
8.    }
10. }
```

Figure 2-2- AspectJ specification to capture logging in BizObject's methods

2.2 Aspect Weaving

The term "Aspect Weaving" refers to combining aspects (i.e., crosscutting features) with non-crosscutting pieces of source code that together perform the desired functionality as specified by the system requirement. The tool that is responsible for merging the separated aspects with the base code is called an aspect weaver. Since the inception of AOP, there has been significant research in weaver implementation and corresponding aspect weavers exist for several modern programming languages like C++ and Java (e.g., AspectJ [Kiczales *et al.*, 2001] – a tool for weaving aspects into Java-based applications). However, there has been less research focus in applying the benefits

of AOP to other legacy languages like FORTRAN, Pascal or COBOL. Therefore, the primary focus of the research described in this dissertation is to investigate appropriate techniques to extend AOP support to other legacy languages.

The different strategies that are often applied to construct aspect weavers vary with the underlying low-level support. Moreover, weaving could be performed during pre-processing time, during compilation, by a post-compile processor, at load or run-time or using a combination of these approaches. Runtime weaving is also known as dynamic weaving while weaving done pre, post or during compilation time is sometimes referred to as static weaving. Existing software transformation techniques to enable AOP for legacy languages can be classified as:

- *object-based transforms*, such as a visitor object applied to an object model
- *intermediate representations* that permit primitive transformations to be applied to a set of languages (e.g., .Net CodeDOM [Thai and Lam, 2003])
- *XML-based transforms* that use an XML DOM structure [Germon, 2001]
- *term rewriting*, such as a transformation rule [Klint *et al*., 2004]

2.2.1 Current State of the Art in Legacy AOP Modernization

This section discusses briefly the current state-of-the-art examples of the above techniques within the context of the enumerated challenges as introduced in Chapter 1.

SourceWeave.Net: SourceWeave.Net [Jackson and Clarke, 2004] is built on top of CodeDOM, which is the .NET standard for representing source code as abstract syntax trees (ASTs) [Thai and Lam, 2003]. Using SourceWeave.NET, a developer can write base and aspect components in standard C#, VB.NET and J#. An XML descriptor file is

used to specify the interaction between the aspects and representative components. The technique uses a mapping to identify Join Point Shadows (JPS) (areas in the source where one or more join points may emerge) and uses a "pointcut-to-join point binding" to isolate parts of the source.

Weave.Net: Weave.NET is a load-time weaver that allows aspects and components to be written in a variety of .Net based languages [Lafferty and Cahill, 2003]. It takes an existing .Net binary component as input with crosscutting specifications provided in an XML file. The behavior (implementation-specific advice code) of an aspect is provided separately in another .Net assembly. Weave.NET recreates the input assembly, but in this regenerated version, join points are bound to behavior in the aspect assembly as specified in the XML aspect file. Because all transformations are done at the intermediate language (IL), it serves as a language-independent weaver.

AspectCOBOL: An initial prototype that brings aspects to COBOL was developed through a collaboration of academic and industrial partners [Lämmel and Schutter, 2005]. The implementation reuses a pre-existing COBOL front-end to construct an AST that is persisted as XML. The AspectCOBOL weaver operates on the XML representation using a DOM-based approach. The weaver has similar semantics to AspectJ pointcuts [Colyer *et al*., 2004], but uses an imperative language that is closer to COBOL syntax. The weaver provides *ad hoc* type analysis (e.g., use-to-def site navigation) for more sophisticated data join points.

Aspicere combined with GCC 4.0: Aspicere is an aspect language for C, but future extensions aim to combine Aspicere with GCC4.0 to introduce two new intermediate representations: GENERIC and GIMPLE trees [Adams, 2005]. Each

different language front-end produces a forest of GENERIC trees, which are then turned into GIMPLE for optimizations and eventually fed to the back-ends. Aspicere aims at providing the weaver's semantics as expressed in terms of generic trees that can eventually lead to language-independent AOP.

The focus of the research described in this dissertation is primarily based on term rewriting systems or program transformation engines that offer scalable parsers for a variety of legacy languages and a powerful low-level infrastructure to modify source programs. The following section provides a comparative discussion between the current state-of-the-art AOP transformation tools and the term rewriting system primarily used in this research.

2.2.2 Comparative Discussion of AOP Tools to Support Legacy Languages

From a comparative discussion of these representative approaches, each provides a distinguishing set of strengths and weaknesses. For example, Weave.Net offers a strong solution to *Challenge C1* because of the availability of pre-existing industrial scale parsers (however, Weave.Net is limited to applications hosted within .Net). Comparatively, SourceWeave.Net is weak on *Challenge C1* due to the limited availability of CodeDOM providers beyond a handful of languages.

The representation of the underlying abstract source model contributes to several differences affecting the solutions to each challenge. Because of its reliance on CodeDOM, SourceWeave.Net has limitations in terms of expressiveness. C# constructs map reasonably well to CodeDOM, but that is not true of all .Net constructs. The proposed Aspicere project seems similar in respect to the SourceWeave.Net approach

with their GCC4.0 GENERIC trees closely related to the CodeDOM abstract source model. It remains to be shown if either CodeDOM or GCC4.0 GENERIC trees are applicable to a large class of legacy languages like COBOL, FORTRAN and Object Pascal. Moreover, a considerable engineering effort would be required if all programming languages were forced to conform to a generic source model or compiled to a common intermediate language. Further, such an approach would ignore all of the effort that has already been spent in constructing lexers, parsers, analyzers and other tools for these languages.

A related challenge emerging from the source model concerns the issue of scalability. The verbosity of an XML code representation may hamper the size of an application that can be weaved. It has been reported that an XML representation is up to 50 times larger than other internal representations and much slower to transform [Germon, 2001]. This may influence the ability of SourceWeave.Net and AspectCOBOL to handle very large applications. This is not a problem for Weave.Net, which weaves into the IL.

The research described in this dissertation provides a more general approach for mitigating the challenges described in Chapter 1 by constructing a generic framework to extend AOP in legacy languages using term rewriting and MDE [Roychoudhury, 2004; Roychoudhury *et al*., 2007]. The weaving is achieved during pre-processing time. Term rewriting is a paradigm that is used in fields such as program transformation and theorem proving [Klint *et al*., 2004]. In term rewriting, rules define a refinement to a structure by specifying a pattern to be matched and the resulting effect. The choice of term rewriting over object-based or XML-based transforms is supported by the following observations:

- A mature program transformation system that performs term rewriting (e.g., the Design Maintenance System (DMS) [Baxter *et al.*, 2004]) can ease the construction effort for weavers of legacy languages by offering a direct solution to *Challenges C1* and *C2*. Such systems provide availability of industrial scale parsers for multiple legacy languages, as well as an underlying low-level transformation engine to restructure source code.
- The term rewriting model offers a complex JPS (e.g., nested conditional statements, as discussed in [Sullivan *et al*., 2005]) and a rich pointcut to join point binding that is informed by join point context information. However, PTEs don't provide explicit representation of JPS, rather such join points can be located using external functions written for the transformation engine.
- Term rewriting offers powerful pattern matching and efficient tree traversal strategies (e.g., using visitors over ASTs) that can scale to several million lines of code.
- With term rewriting, internal Application Programming Interfaces (APIs) are available to modify ASTs in an arbitrary manner, thereby allowing more complex and flexible transforms required by legacy applications (e.g., aspects and loops [Harbulot and Gurd, 2005]).
- In contrast to the verbose AST representation in XML-based approaches, DMS provides internal data structures (e.g., hypergraphs) to represent the underlying AST. This offers an improved level of optimization to support parsing and transforming large legacy applications.

In the following section, the PTE used in this research is introduced. This provides the necessary background information required to understand the inner workings of the underlying PTE as it is used throughout the dissertation.

2.3 Program Transformation - Design Maintenance System

The Design Maintenance System (DMS) [Baxter *et al.*, 2004] is a program transformation system and re-engineering toolkit developed by Semantic Designs (www.semdesigns.com). The core component of DMS is a term rewriting engine that provides powerful pattern matching and source translation capabilities. 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. DMS provides pre-constructed domains for several dozen languages.

The DMS Rule Specification Language (RSL) provides basic primitives for describing numerous transformations that are to be performed across the entire code base of an application. The RSL consists of declarations of patterns, rules, conditions, and rule sets using the external form (concrete syntax) defined by a language domain. 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 of a rule to describe the resulting syntax tree after a transformation rule is applied. 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. 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. 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. Typically, a large collection of RSL files are needed to describe the full set of transformations on a base program.

In addition to the RSL, a language called PARLANSE (PARallel LANguage for Symbolic Expressions) is available in DMS. Transformation functions can be written in PARLANSE to traverse and manipulate the parse tree at a finer level of granularity than provided by RSL. PARLANSE is a functional language for writing transformation rules as external patterns to provide deeper structural transformation. The DMS rules, along with the corresponding PARLANSE code, represent the core transformations required for aspect weaving. However, due to the very low-level nature of the rewrite rules, it is not desirable that programmers be required to write their specifications using term-rewriting or PARLANSE-specific functions. Instead, a high-level aspect language (similar to AspectJ) that hides the accidental complexities of RSL and PARLANSE from the programmer can be used to specify the weaving (please see Chapter 4). Some of the key features of DMS are discussed in the following sub-section.

2.3.1 DMS Key Features and Support for Abstract Syntax Trees

DMS supports full UNICODE-based parser and lexer generation with automatic error recovery. The parser technology is based on Generalized LR parsing [López, 1992], and can handle any context-free language, even with ambiguities. Pretty-printer generation converts Abstract Syntax Trees (ASTs) back to a properly formatted legal source file, according to specified layout information, including source comments. DMS also provides a multi-pass attribute-evaluator [Paakki, 1995; Alblas, 1991] generation

from the language grammar, to allow arbitrary analysis (including name/type analysis procedures) to be specified in terms of the concrete grammar provided. In addition, DMS provides sophisticated symbol-table construction facilities for global, local, inherited, overloaded and other language-dependent name lookup and namespace management rules.

DMS is designed to work on large-scale source systems with up to several million lines of source code across tens of thousands of source files having multiple languages at the same time. It runs either on a single processor system or on symmetric multiple processor workstations for enhanced performance.

For DMS, each domain may have several representations. Each representation is a set of explicitly declared hypergraph nodes that can be composed to represent domain instance fragments. The hypergraph interface provides primitives for arbitrary manipulation of such graphs, including crossing boundaries between graph nodes from different representations. The AST Interface is designed to support trees that represent string-based languages, as definable via the `DMSStringGrammar` domain.

The `DMSStringGrammar` domain, given a particular grammar, automatically defines the AST representation for that string language, essentially based on the terminals and non-terminals of the supplied grammar. Some of the important operations on ASTs provided by DMS are `ScanTreeNodes`, `FindChildWithProperty`, `GetNthChild`, `CopyTree` and `FindParentWithProperty`. The AST function `FindChildWithProperty` applies the property function (i.e., a visitor function) to every node in an in-order walk of the tree. If the function returns true after any call, the rest of the tree walk is skipped and the node is returned as a result; otherwise, a void node

is returned as a result. The property function will only be called once for each node, even if the tree shares substructure. This function is very useful for pattern matching and is used in several places in the low-level implementation of the aspect weaver.

2.3.2 Specifying Rewrites using DMS Rule Specification Language

Figure 2-3 presents an overview of the back-end transformation process. The program transformation rule (shown in general in Figure 2-3 with a specific example in Figure 2-4) is written in the RSL and processed by the back-end transformation engine to perform the actual rewriting.

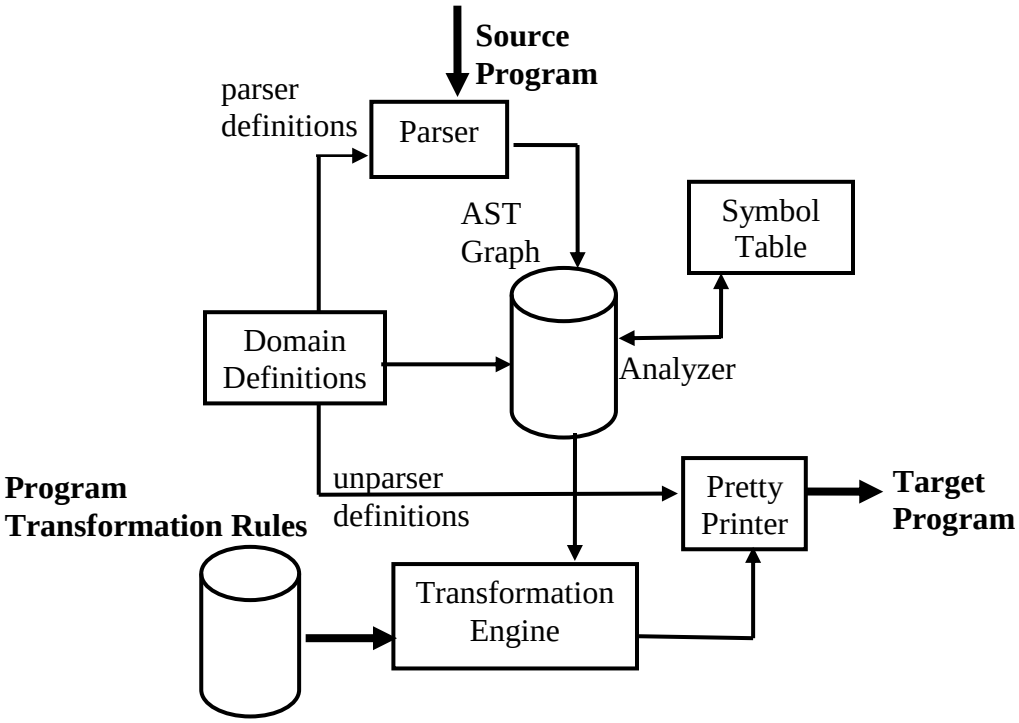


Figure 2-3- Overview of back-end transformation process

Although term rewriting has several application domains (e.g., code migration, code refactoring or program refinement), the particular example in Figure 2-4 highlights

an aspect-oriented style. As stated earlier, an RSL transformation rule consists of declarations of patterns, rules, conditions, and rule sets using the concrete syntax defined by a language domain. The first line of this transformation rule resolves the *domain* (i.e., language) to which the rule can be applied. In this case, a tracing probe is inserted before the execution of all functions written in Object Pascal. The statement list (`stmt_list`) that appears inside of a function body is passed as a parameter to this rule (Line 2). Note that 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 (syntactically denoted by "→" in RSL). The `insert_probe` rule matches all function body declarations in the source program and adds a `ShowMesssage` dialog box before the execution of the original statement list (i.e., `\stmt_list`, Line 4). In this way, a simple tracing probe is added at the beginning of all function execution. Rules can be combined into rule sets that form a transformation strategy by defining a collection of transformations that can be applied to a syntax tree.

```
1. default base domain ObjectPascal.
2. private rule insert_probe(stmt_list: statement_list):
3. function_body  →  function_body
4. = "begin \stmt_list end"  →
5.   "begin ShowMessage(\"Entering Method\"); \stmt_list end".
6. public ruleset TraceAllFunctions = {insert_probe}
```

Figure 2-4- A simple example of a program transformation rule that illustrates aspect weaving of *before* advice

These transforms along with the source program are syntactically checked and statically analyzed to ensure the expected weaving behavior. However, RSL rules are

typically hardcoded and dependent on the grammar of the base language (i.e., all text highlighted in bold in Figure 2-4 corresponds either to a terminal or non-terminal symbol in the Object Pascal grammar). The dependence of RSL on the base language grammar hinders rule reusability (i.e., in case the above functionality needs to be added for a different programming language, the RSL shown in Figure 2-4 cannot be reused directly).

2.3.3 Other Program Transformation Engines

In addition to DMS, there are a few other program transformation systems, namely ASF+SDF [van den Brand et al., 2002], TXL [Cordy *et al*., 2002] and Stratego [Visser, 2001] that offer similar capabilities to restructure the underlying source program.

The ASF+SDF Meta-Environment is an Integrated Development Environment (IDE) and toolset for interactive program analysis and transformation. It combines SDF (Syntax Definition Formalism), ASF (Algebraic Specification Formalism) and other technologies. Some of the features available in ASF+SDF are program analysis, program transformation, generation of IDEs, visualization of parse trees and pretty printer generation.

Tree TRANSformation Language (TXL) is a unique programming language specifically designed to support computer software analysis and source transformation tasks [Cordy *et al*., 2002]. It is a hybrid functional/rule-based language with unification, implied iteration and deep pattern matching. Each TXL program has two components: A description of the structures to be transformed that are specified as a directly interpreted Backus-Naur Form (BNF) grammar, in context-free ambiguous form; and a set of structural transformation rules specified by example as pattern/replacement pairs

combined using functional programming. The formal semantics and implementation of TXL are based on formal tree rewriting, but the trees are largely hidden from the user due to the by-example style of rule specification.

Stratego is a language and toolset for program transformation [Visser, 2001]. The Stratego language provides rewrite rules for expressing basic transformations, programmable rewriting strategies for controlling the application of rules, concrete syntax for expressing the patterns of rules in the syntax of the object language, and dynamic rewrite rules for expressing context-sensitive transformations, thus supporting the development of transformation components at a high-level of abstraction.

DMS was chosen for this research because of the maturity of the tool, as compared to other transformation engines, and the immediate availability of a large collection of pre-constructed domains. We have the confidence that many of the pre-existing parsers that have been defined in DMS are capable of parsing large-scale industrial legacy software. There are many well-defined language definitions provided within DMS that have been used to parse multiple-millions of lines of commercial code. It is possible that the technique described in this paper could also apply to other transformation systems. For the technique described in this thesis to have a real impact, the ability to parse large code bases in multiple languages is paramount toward providing a framework for injecting aspects into legacy systems.

In the following section we introduce model-driven engineering as the basis for providing a suitable front-end to program transformation systems. This particularly addresses *Challenges C3* and *C4* that were introduced in Chapter 1.

2.4 Model-Driven Engineering - AMMA

A large body of research in the area of Aspect-Oriented Modeling (AOM, http://www.aspect-modeling.org) has focused on new notations and weaving tools that improve the ability to express a design within a model through composition of separate concerns. In the research presented in this dissertation, we examine the converse – that is, how modeling can improve aspect orientation. Specifically, the research makes a contribution by showing how MDE [Schmidt, 2006] is used to construct new aspect weavers for General-Purpose Languages (GPLs) through models and transformations. To provide the necessary background information of our desired choice of MDE toolkit, in this section, we introduce the Atlas Model Management Architecture platform (AMMA) [Kurtev *et al*., 2006]. AMMA is a suite of MDE tools that can be used to implement domain-specific languages (DSLs) [Mernik *et al*., 2005] as well as high-level aspect languages useful for constructing the front-end of an aspect weaver. AMMA is designed and developed by the ATLAS Team (INRIA and LINA), and is composed of several elements: Kernel Meta-Meta Model (KM3) [Jouault and Bézivin, 2006], Textual Concrete Syntax (TCS) [Jouault *et al*., 2006], Atlas Transformation Language (ATL) [Jouault and Kurtev, 2005] and a few other tools (e.g., Atlas Model Weaver (AMW) [Jossic *et al*., 2007], Model Discovery [MoDisco, 2008]).

2.4.1 Kernel Meta-Meta Model

KM3 [Jouault and Bézivin, 2006] is a platform-independent language (metamodeling platform) to write metamodels, and thus to define the abstract syntax of DSLs. The purpose of KM3 is to give a relatively simple solution to define the Domain

Definition MetaModel (DDMM) of a DSL. The DDMM of KM3 is a meta-metamodel, and is defined in KM3 itself, just like the grammar of Backus-Naur Form (BNF) may be described in BNF itself.

KM3 uses concepts like *class*, *attribute*, and *references* and is structurally close to the Object Management Group's Meta-Object Facility [OMG MOF, 2003] and Eclipse's Ecore [Budinsky *et al*., 2003]. Mappings to and from MOF 1.4 and Ecore have been defined in ATL, making KM3 usable with tools like the Eclipse Modeling Framework (EMF) [Budinsky *et al*., 2003] and Netbean's Metadata Repository (MDR). As a meta-metamodel, KM3 is simpler than MOF 1.4, MOF 2.0 [OMG MOF, 2003] and Ecore. It contains only 14 classes whereas, for instance, Ecore has 18 classes and MOF 1.4 has 28 classes. Only the core concepts of these other meta-metamodels are available in KM3. The KM3 specification for the high-level aspect language used in the front-end of this framework is shown in Chapter 4 (also, please see Appendices A and B).

2.4.2 Textual Concrete Syntax

TCS enables the specification of textual concrete syntax for DSLs by attaching syntactic information to metamodels [Jouault *et al*., 2006]. Unlike KM3, which uses an object-oriented syntax to specify the abstract definition of metamodels, TCS uses a grammar-like notation to specify the corresponding concrete syntax for metamodels. Thus, KM3 along with TCS provides the complete language description for a DSL.

Two of the basic TCS constructs are Primitive Templates and Class Templates. Primitive Templates specify the lexer token corresponding to a given metamodel DataType, identified by its name. More than one primitive template may be defined for a

single data type. This is typically the case for strings where one template is used to represent identifiers and a second one is used to represent string literals. Exactly one primitive template may be declared as default for each data type. Class Templates specify how classes are represented. This specification consists of a sequence of syntactic elements that are: keywords and special symbols. A Class Template has the same name as its corresponding Class. Exactly one class template must be declared as *main*. The main class template corresponds to the root of the model. In contrast to primitive templates, only one class template can be defined for each class in the metamodel. An example of TCS specification is shown in Appendices A and B.

2.4.3 ATLAS Transformation Language

ATL is a model transformation language and toolkit developed by the ATLAS Group (INRIA & LINA) [Jouault and Kurtev, 2005]. ATL provides ways to produce a set of target models from a set of source models. ATL is developed on top of the Eclipse platform; the ATL Integrated Environment (IDE) provides a number of standard development tools (e.g., syntax highlighting, debugger) that aims to ease development of ATL transformations.

ATL is applied in a transformational pattern as shown in Figure 2-5. In this figure, the Grammarware [Klint *et al.*, 2005] technical space (TS) refers to the text-based languages and tools, and the MDE or modelware TS refers to the class of model-based languages and tools used in software engineering. As shown in Figure 2-5, transition from one TS to another is achieved by means of TCS Injection or TCS Extraction. During TCS injection, a source program is converted to a source model, while, during TCS

extraction, a target program is extracted from a target model. M1, M2 and M3 are the three modeling levels present at these two TS. In the transformational pattern shown in Figure 2-5, a source model M_s is transformed into a target model M_t. The transformation is driven by a transformation specification (or a transformation program) ms2mt.atl written in the ATL language. The source and target models and the transformation specification (ms2mt.atl) conform to their metamodels MM_s, MM_t and ATL, respectively. Finally, all the respective metamodels conform to the common MOF meta-metamodel. Detail description about the model transformation process applicable to the current GenAWeave framework is described in Chapter 4. An example of a model transformation rule is presented in Appendix D.

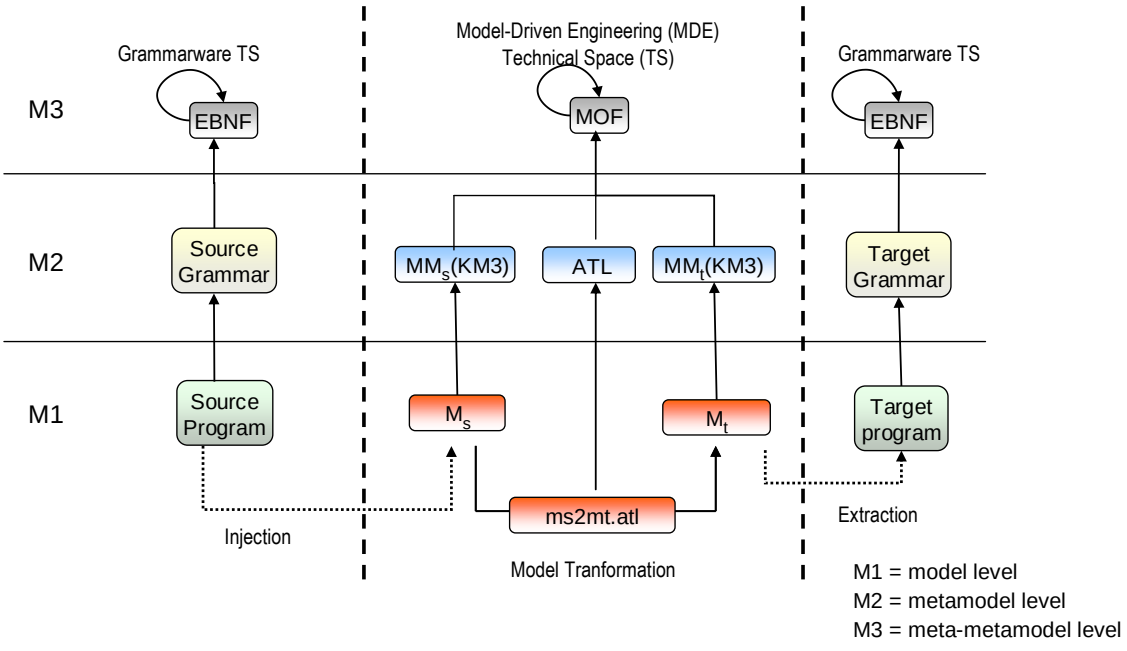


Figure 2-5- An example of ATL transformation

ATL transformations are unidirectional, operating on read-only source models and producing a write-only target model. During the execution of a transformation the

source model may be navigated but changes are not allowed. The target model cannot be navigated. A bidirectional transformation is implemented as a combination of transformations, one for each direction.

2.4.4 Related Work

MDE refers to the systematic use of models as primary engineering artifacts throughout the engineering lifecycle of a software system. As a related idea, Model-driven Architecture (MDA) is a conceptual software design approach launched by the Object Management Group (OMG, http://www.omg.org) that supports MDE. MDA provides a set of guidelines for structuring specifications expressed as models. It also provides standards for engineering model-driven artifacts that are expressed using the Meta-Object Facility (MOF). There are two versions of MOF, namely Essential MOF (EMOF) and Complete MOF (CMOF). In addition, models can be expressed in the Unified Modeling Language (UML, http://www.uml.org/) and transformations on models can be expressed using Query View Transformations (QVT) [OMG QVT, 2001].

In addition to OMG's MDA initiative, the Eclipse Modeling Framework (EMF) is an Eclipse Modeling project that provides a Java implementation of the basic principles of model engineering. EMF is based on the Ecore meta-metamodel. The Graphical Modeling Framework (GMF, http://www.eclipse.org/gmf/) is an Eclipse Modeling project, which can be used to specify graphical syntax for Ecore metamodels.

The Generic Modeling Environment (GME) [Lédeczi *et al*., 2001] is a configurable toolkit for creating domain-specific modeling (DSM) [Gray *et al*., 2007] and program synthesis environments. The configuration is accomplished through metamodels

specifying the modeling language of the application domain. The metamodel contains all the syntactic, semantic, and presentation information regarding the domain. The metamodeling language is based on the UML class diagram notation and Object Constraint Language (OCL) constraints [OMG OCL, 2001]. Although the metamodeling language in GME is visual (UML classes), AMMA based metamodeling is textual and is well-suited for describing the high-level aspect language presented in this dissertation.

Another interesting toolkit that can be used for developing DSLs is LISA [Mernik *et al.*, 2002], which provides a generic interactive environment for programming language development. Using formal language specifications of a particular programming language, LISA can produce a language-specific environment that includes an editor, a compiler/interpreter and other graphical tools.

Our choice of AMMA was influenced by the fact that the different tools available in AMMA can be applied equally to both OMG's MOF and the open source EMF (i.e., independent of the underlying metamodeling platform). In addition, AMMA provides a powerful model transformation language (ATL) that is useful for translating models specified in a high-level aspect language to models specified in the low-level RSL language. Moreover, KM3 and TCS help with the design of our textual based high-level aspect language, which is used in the front-end of our framework.

In the following chapter we introduce program transformation based aspect weaving using low-level rewrite specifications that are applicable to several GPLs (e.g., Object Pascal, C and C++ Templates). The technique introduced in Chapter 3 forms the basis of constructing aspect weavers using program transformation rules and is evaluated across commercial and high-performance scientific computing domains.

CHAPTER 3

PROGRAM TRANSFORMATION BASED ASPECT WEAVER CONSTRUCTION

A systematic approach towards realizing a generic framework for language-independent program transformation requires the ability to capture the knowledge for language-specific transforms. This chapter discusses in detail the construction technique for language-specific aspect weavers. In particular, the chapter introduces aspect weavers specific to programming languages like Object Pascal and C++ template libraries, which are applicable to commercial and scientific computing domains [Gray and Roychoudhury, 2004]. In addition, initial results about a slightly different construction technique (i.e., scientific library construction) for composing High Performance Linpack (HPL) libraries are also presented [Petitet *et al.*, 2004]. All of the above mentioned techniques use DMS as a low-level transformation engine [Baxter *et al.*, 2004].

Section 3.1 provides a detailed description of our PTE based technique to construct an aspect weaver for Object Pascal, which is validated against a commercial case study application. Section 3.2 discusses the design of an aspect language for C++ templates and also presents the underlying technique for weaving into C++ templates. Section 3.3 offers several examples of crosscutting concerns identified in a scientific library, namely Blitz++ [Veldhuizen, 1998] and demonstrates improved modularization

of such libraries using AOP and generative programming techniques [Czarnecki and Eisenecker, 2000]. The section also presents a PTE-based specialization technique used to improve the construction of a high-performance scientific library, namely HPL.

3.1 An Aspect Weaver for Object Pascal

This section describes the crosscutting concerns that were identified in a distributed application implemented in Object Pascal. Three different utility applications within this suite each had their share of problems with respect to scattered and tangled code. The utilities that serve as the case study for this research were implemented in 42K source lines of Object Pascal. This section provides a general discussion of several crosscutting concerns that were identified. Other crosscutting concerns exist in these utilities (e.g., database access control logic that is spread over a dozen classes), but this section focuses only on a subset of all identified aspects. For each concern, the number of times that the implementation redundantly appears is provided, which implies the amount of code that can be removed when modularized as aspects (e.g., the code in Figure 3-1 appears 62 times and contains about 5 lines of code per case).

3.1.1 Crosscutting Concerns in Object Pascal Application

Four specific examples of crosscutting concerns as identified in an Object Pascal application are described below:

Progress Dialog Meter: The Database Manager is a utility that assists customers in upgrading to a new database schema after installing an update to the application software. It manages the schema evolution problem by converting a database instance to

a new schema [Bounif and Pottinger, 2006]. Utilities like the Database Manager often provide feedback to the user in the form of a processing dialog, or meter, which indicates the progress of the overall task. The updating of the progress meter represents a crosscutting concern because the code to increment the meter is spread across the methods that perform much of the functionality (e.g., deleting database triggers, compiling new stored procedures, and other evolution tasks). Figure 3-1 contains a redundant code fragment that appears in 62 different places of the Database Manager. This code is necessary to update the processing dialog after each database evolution task is completed.

```
1.  Inc(TotalInserts);
2.  if not ProcDlg.Process(TotalInserts/TotalCalc) then
3.     begin
4.       ProcDlg.Canceled := True;
5.       Result := True;
6.       exit;
7.     end;     // if not Process
```

Figure 3-1 – Progress meter updating

Replication of exception handling code can have negative consequences [Lippert and Lopes, 2000]. With respect to error handling in the Database Manager, the code fragment in Figure 3-2 appears 33 times in various methods in order to stop the processing dialog after an exception. However, it would be desirable to create a single separate module that describes all of the functionality of updating the progress meter.

```
1.  on E : Exception do
2.    begin
3.      dmSERVERS.HandleException(E);
4.      dmSERVERS.ProcDlg.Canceled := True;
5.    end;
```

Figure 3-2 – Exception handling code for processing dialog

Logging of SQL Query Statements: Another crosscutting concern that is scattered throughout the Database Manager is the logging of SQL code. As the Database Manager utility upgrades the customer's database to a new schema, all of the SQL commands that are generated to perform the upgrade are logged to a file so that they can be examined later in the event of a problem (please see the code fragment in Figure 3-3). Although a special logging object was created, the numerous places and contexts where the object is called may vary. In fact, the methods of the logging object are invoked in over 50 different places in the Database Manager. Please notice that the logging call is also context-dependent and parameterized by the name of the query object. The ability to collect the logging actions in a single module would aid in better separation of this canonical logging concern. Unfortunately, for Object Pascal and most other programming languages, there are no language constructs to provide these desired capabilities.

```
1. with dmSERVERS.qryCreateTriggers do
2. begin
3.     <statements that build a SQL Create Trigger>
4.
5.     LogSQL.AddSQL(dmSERVERS.qryCreateTriggers, True);
6.     ExecSQL;
7. end;
```

Figure 3-3 – Logging of SQL query Data Definition Language (DDL) statements

Language Internationalization Utility: There are several tasks involved in internationalizing software. One technique is to represent all translations of each text string in a resource Dynamic Link Library (DLL). The creation of this library, however, requires a tool that assists in the management of all of the different strings for all of the supported written languages. The Language Internationalization tool supports such a task. The implementation of the Language Internationalization tool resulted in 24 classes.

Several of the classes interact with all of the controls within a Graphical User Interface (GUI) and update a database during any modification to GUI widgets. Among all of the events that are processed in the application, a dirty bit is used to keep track of whether a modification is made to a widget.

```
1.     // The user wants to perform another search
       // using the same search criteria
2.     procedure TLangInt.SearchAgainClick(Sender: TObject);
3.     begin

4.       // Perform an update if an edit occurred that might
5.       // change the focus of the listview
6.       if EditMadeDirtyBit then
7.           SaveDBControls;
8.     ...
9.     end;
```

Figure 3-4 – Preamble for widget button clicks

There are 29 unique places in the source code of the Language Internationalization utility where access to the Boolean variable `EditMadeDirtyBit` is made. There were only two different contexts in which the `EditMadeDirtyBit` was accessed. One context simply dealt with setting the value to true or false, based upon a particular situation, and involved lazy-writing of the edit. This was spread across several diverse classes and represented 9 of the places where this concern occurred. The other context in which access to the dirty bit appeared dealt with performing a specific action based upon the value of `EditMadeDirtyBit`. The code for deciding the next action, based on the value of the bit, was identical in each source code location and always occurred as the first statement in a widget-click event handler (see Figure 3-4). Thus, this other redundant code was found in 20 different places in the Language Internationalization tool. Any modification or change to the way in which a text string

was stored often required a change to the way in which this concern was implemented. This required adaptations to many locations in the code in order to make the change. Forgetting to update the change in any one of these places could result in a loss, or corruption, of data during the modification of a language string.

Database Error Handler Synchronization: Often, a commercial application must work with databases from several different vendors (e.g., Oracle, Interbase, and SQL Server). In such a situation, exception handling of database errors is a major difficulty because each database has its own way of raising exceptions. The same conceptual error (e.g., a null in a required field) may be raised in completely different ways with dissimilar error codes. The application, however, must make this transparent while interpreting the exception to provide a meaningful message back to the end-user. To accomplish this transparency, a database error handling DLL was created and integrated into the main application. This library contained 23 classes. The majority of these classes were responsible for handling specific types of exceptions using the Chain of Responsibility pattern [Gamma *et al.*, 1994]. After the code was created for the error handlers, a new requirement was added. It was determined that the exception handling code must be thread-safe because numerous clients would be accessing the database at the same time. The addition of this concurrency concern resulted in a manual invasive change to over 20 classes. An example error handler is shown in Figure 3-5. In that figure, lines 4-5 and 7-9 represent this single synchronization concern. Furthermore, this exact code is replicated in all of the entry and exit points of each type of error handler.

```
1. function TExNullField.Handle(ServerType: TServerType;
2.                              E : EDBEngineError) : Integer;
3.  begin
4.    TExHandleCollection(Collection).LockHandle;
5.    try
6.      <database error handling code omitted here>
7.    finally
8.      TExHandleCollection(Collection).UnLockHandle;
9.    end;
10. end;
```

Figure 3-5 – Synchronization in a database error handler

3.1.2 Weaver Transformation Rules for the Object Pascal Case Studies

The purpose of this section is to introduce the low-level DMS program transformation rules that will drive the aspect weaving process for the Object Pascal weaver. Each of the crosscutting concerns identified in Section 3.1.1 is revisited to demonstrate the use of RSL to weave in each concern. In each case, there are two key parts to the weaving process: 1) the identification of the join points in the source AST that match a given pattern; and, 2) specifying rewrite rules to operate on those points to derive a new representation (i.e., adding advice).

Weaving the Progress Meter Dialog: Figure 3-6 presents the complete RSL transformation rule for weaving the processing meter concern described in Figure 3-1. On the first line of this transformation rule, the domain to which the rule can be applied is identified (in this case Object Pascal). Patterns describe the form of a syntax tree. Often, they are used for matching purposes to find a syntax tree having a specified structure (as such, they provide a type of quantification across a code base [Filman and Friedman, 2004]). Additionally, patterns can appear on the right-hand side (target) of a rule to describe the resulting syntax tree after the rule is applied. Patterns can be combined to form larger patterns.

The `advice` pattern in Figure 3-6 specifies the statement associated with the advice of the processing dialog concern (here, advice is just a user-defined name for a pattern – the word "advice" has no formal semantics within RSL, but is so named because it conceptually represents the concern that is to be weaved). The code that is associated with the advice pattern is the same conditional statement from Figure 3-1. The Object Pascal grammar defines the `if_statement` and `statement_list` production rules that are evident in the pattern and rule specifications. Throughout the paper, parts of the Object Pascal grammar are italicized and RSL reserved words are boldfaced in order to highlight the differences. No visual adornments are given to the regular Object Pascal source code.

```
1. default base domain Object Pascal.
2.
3. pattern advice(): if_statement =
4.
5.    "if not ProcDlg1.Process (TotalInserts/TotalCalc) then
6.       begin
7.         ProcDlg1.Canceled := True;
8.         Result := True;
9.         exit;
10.      end;".
11.
12.rule probe_progress_meter(): statement_list -> statement_list =
13.
14.   "Inc(TotalInserts);"
15.  ->
16.   "Inc(TotalInserts); \advice\(\);".
17.
18.public ruleset applyrules =  { probe_progress_meter }.
```

Figure 3-6 – Transformation rule for updating progress meter

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. The rule `probe_progress_meter` isolates each node (call to function `Inc`) that increments

the database insertion counter. At each join point, the advice pattern is weaved into the progress meter. In this case, the former `statement_list` associated in the increment statement is rewritten (syntactically denoted by "→" in RSL) to a new `statement_list` that appends the advice to the increment. Rules can be combined into rule sets that form a transformation strategy by defining a collection of transformations that can be applied to a syntax tree.

Meta-variables are used as placeholders for sub-trees, and specified using an escape syntax (i.e., "`\identifier`"). An RSL meta-variable can represent the tree defined by a pattern or a parameter to a rule. In Figure 3-6, the meta-variable reference "`\advice\(\)`" names the tree that is associated with the advice pattern and appended to the increment statement. In the next subsection, Figure 3-7 contains parameters (e.g., `\id1`, `\id2`, and `\slist`) to the `probe_logging` rule that serve as placeholders to holes that are filled during the term rewrite process.

SQL Logging Transformations: Surprisingly, separating the logging of the SQL data definition commands, as shown in Figure 3-3, was the most difficult aspect to represent in the RSL. The difficulty stemmed from the `with` construct in Object Pascal, which is a shorthand notation for referencing fields within an object by setting a context block.

The `with` statement of Figure 3-3 (Line 1) provides a context for accessing the fields of the query object (e.g., `dmSERVERS.qryCreateTriggers`) without having to prefix each reference in the block with the object name. Yet, the logging call that was embedded in this context required the name of the bounded query object. The trick for the RSL logging rules, as shown in Figure 3-7, is to trace back to the `with` statement that

contains the query object. This is accomplished with an external pattern called `add_log_stmt`.

```
1.  default base domain Object Pascal.
2.
3.  external pattern add_log_stmt (slist1:statement_list,
            slist2:statement_list,
            id1:IDENTIFIER,
            id2:IDENTIFIER): statement_list = 'add_log_statement'
            in domain Object Pascal.

4.  pattern advice(id1:IDENTIFIER, id2:IDENTIFIER):
5.          statement_list = "LogSQL.AddSQL (\id1.\id2 , True);".
6.
7.  pattern func_call_sig(): "ExecSQL".
8.
9.  rule probe_logging(id1:IDENTIFIER,id2:IDENTIFIER,
            slist:statement_list): with_statement -> with_statement =

10.  "with \id1 . \id2 do
11.    begin
12.       \slist
13.    end"
14.  →
15.  "with \id1 . \id2 do
16.    begin
17.       \add_log_stmt\(\slist \, \advice\(\id1 \,\id2\) \,
                        \id2 \,\func_call_sig\(\)\)
18.    end".
19.
20. public ruleset applyrules =  { probe_logging }.
```

Figure 3-7 – Transformation rule for SQL logging

There are certain things that cannot be specified in the RSL, such as tree-walking strategies. In such cases, it is possible to write external functions that escape from the RSL and return a value. In DMS, exit functions are written in a functional language called PARLANSE, which is a parallel language for symbolic expression that provides an enriched set of interfaces for performing operations on ASTs. The special parallel constructs provided by PARLANSE can offer performance improvements while traversing the hierarchical tree structure during pattern search [Baxter *et al.*, 2004]. Within the AOSD community, there has been extensive research in adaptive and strategic

programming to address traversal strategies [Lämmel *et al*., 2003; Lieberherr *et al*., 2001], but there was no mechanism to apply these ideas directly to RSL.

```
1.(lambda (function boolean AST:Node )function
2.   (value (local (;; );;
3.      (;;
4.          (ifthen(== ~t (AST:ContainsString ?))
5.            (;;
6.              (= search_string (AST:GetString ?))
7.              (ifthen (== (@ search_string)
8.               arguments:4)  (return ~t)) ifthen
9.             );;
10.          )ifthen
11.          (return ~f)
12.        );;
13.      )local
14.      ~f
15.   )value
16.)lambda
…
```

Figure 3-8 – Visitor function written in PARLANSE

The objective of the `add_log_stmt` external pattern is to insert a new log statement before every call to the `ExecSQL` statement. However, the parameters to be logged come from the variable access definition that is attached to the `with` statement.

The visitor function used to find child nodes that match the pattern `func_call_sig` can be found in Figure 3-8. Note that the fourth argument that is passed to the external pattern is the function call identifier `ExecSQL`. The visitor returns true whenever it finds a match to this call statement in the syntax tree. From the external pattern, all matching placeholders are returned and the right-hand side (RHS) of the rule weaves in the advice to transform the original syntax tree.

Transforming Dirty Bits: Recall from Section 3.1.1 that a dirty bit was used to determine if a lazy-write was needed to update the state of a database as a result of an edit to a language string. That concern required a simple conditional statement to be attached

to the beginning of all widget "`Click`" event handlers. Figure 3-9 is an RSL transformation that provides support for weaving this aspect into the source code of the Language Internationalization utility.

The `advice` pattern in Figure 3-9 represents the simple conditional statement that is to be prefixed to the widget `Click` methods. The patterns `isClick` and `click` are used to identify the placeholder in the source AST. The left-hand side (LHS) of the rule `probe_dirty_bit` transforms the source syntax tree to its new representation depending on the external condition `func_sig_has_click`, which is invoked from the `isClick` pattern. Note that external conditional functions are coded in PARLANSE. This external condition is needed to match the wildcard "`*Click`" specification. It is not possible within RSL to look into the contents of a syntax-tree node, but this can be accomplished in an external condition.

The function `func_sig_has_click` (line 2, Figure 3-9) takes two arguments. The first argument is the identifier node that denotes places of interest in the search process. The second argument is a constant identifier string that is used to match the place holders in the source tree. The DMS transformation engine will continue to apply all sets of rules until no rules can be fired. It is possible to have an infinite set of rewrites if the transformations are not monotonically decreasing (i.e., when one stage of transformation continuously introduces new trees that can also be the source of further pattern matches). Notice that there is a condition specified at the bottom of Figure 3-9 (line 29). This condition describes a constraint stating that the set of rules should be applied only to those join points where a transformation has not occurred already. Specifically, it states that the rules should be applied when it is not the case that the

advice is already prefixed to a statement list. The transformation rule will be applied only once to each `Click` event handler. Without this condition, the rules would be applied iteratively and fall into an infinite loop.

```
1.  default base domain Object Pascal.

2.  external condition func_sig_has_click(id1:IDENTIFIER,
3.                                        id2:IDENTIFIER)
4.                    = 'func_sig_has_click'.

5.  pattern advice(slist:statement_list) : statement_list =

6.    "if EditMadeDirtyBit then
7.        SaveDBControls;
8.     \slist".

9.  pattern isClick(id:IDENTIFIER): IDENTIFIER
10.               = id if func_sig_has_click(click(), id).
11.
12. pattern click (): I    DENTIFIER = "Click".
13.
14. rule probe_dirty_bit (id1:IDENTIFIER, id2:IDENTIFIER,
15.                       fps:formal_parameters,
16.                       slist:statement_list):
17.   implementation_decl -> implementation_decl =
18.
19.  "procedure \id1 . \isClick\(\id2\)  \fps ;
20.   begin
21.     \slist
22.   end;"
23. ->
24.  "procedure \id1 . \id2   \fps ;
25.   begin
26.     \advice\(\slist\)
27.   end;".
28.
29. if ~[modslist:statement_list .slist matches
30.            "\:statement_list \advice\(\modslist\)"].
31.
32. public ruleset applyrules =  { probe_dirty_bit }.
```

Figure 3-9 – RSL rule for weaving dirty bits

Figure 3-10 shows the PARLANSE function that is used to perform the wildcard pattern matching. It utilizes the pre-defined DMS `StringScan` and `ASTInterface` libraries to perform the scanning operation over the placeholders. The function returns

true if it finds a slot as specified by the pattern `click`. The advice is applied to all placeholders that match this given pattern.

```
1.  (define func_sig_has_click
2.   (lambda Registry:MatchingCondition
3.    (let (;;(= [const_string (reference string)]
4.                             (AST:GetString arguments:1))
5.            (= [search_string (reference string)]
6.                             (AST:GetString arguments:2))
7.            (= [scanner StringScan:Scan ]
8.            (StringScan:MakeScan search_string))
9.         );;
10.
11.       (value
12.         (while (== (StringScan:End? (. scanner)) ~f)
13.           (ifthenelse
14.                (StringScan:MatchString? (. scanner)
15.                   const_string)
16.                   (return ~t)
17.               (StringScan:Advance (. scanner))
18.           )ifthenelse
19.          )while
20.         ~f
21.        )value
22.    )let
23.   )lambda
24.  )define
```

Figure 3-10 – PARLANSE external condition function `func_sig_has_click`

Error Handling Transformation: The concurrency control concern from Figure 3-5 can be weaved using RSL in a style similar to those transformations already shown and is illustrated in Figure 3-11.

In the transformation of Figure 3-11, the `try/finally` block (lines 14-18) that implements the concurrency control is wrapped around the critical section of the source code. The original critical section is denoted by the `statement_list` that is represented by the `slist` meta-variable (line 15) in the transformation. The pattern `probe_handle` (line 3) identifies the slot from the function signature (line 27) where the advice needs to be applied (i.e., function name with a signature like *.`Handle`). The

`lock` (line 9) and the `unlock` (line 6) patterns are then inserted before and after the critical section of the source code. The RHS of the rule `probe_synchronize` (line 20) rewrites the entire function definition and inserts the RSL pattern `advice` (line 12) in the function body. The conditional constraint in line 37 denotes the condition when this rule can be applied.

```
1.  default base domain Object Pascal.
2.
3.  pattern probe_handle(id:IDENTIFIER):
4.          qualified_identifier = "\id.Handle".
5
6.  pattern unlock():statement =
7.          "TExHandleCollection(Collection).UnLockHandle".
8.
9.  pattern lock():  statement =
10.         "TExHandleCollection(Collection).LockHandle".
11.
12. pattern advice(slist:statement_list): statement_list =
13.   "\lock\(\);
14.     try
15.        \slist
16.     finally
17.        \unlock\(\);
18.     end;".
19.
20. rule probe_synchronize(slist:statement_list,
21.                        id:IDENTIFIER,
22.                        fps:formal_parameters,
23.                        frt:function_result_type):
24.
25.    implementation_decl -> implementation_decl =
26.
27.   "function \probe_handle\(\id\) \fps : \frt ;
28.    begin
29.      \slist
30.    end;"
31. ->
32.   "function \probe_handle\(\id\) \fps : \frt ;
33.    begin
34.      \advice\(\slist\)
35.    end;".
36.
37. if ~[modsList:statement_list .slist matches
38.            "\:statement_list  \advice\(\modsList\)"].
39.
40. public ruleset applyrules =  { probe_synchronize }.
```

Figure 3-11 – RSL rule for modularizing synchronization

In another context of using a different GPL, Section 3.2 discusses the techniques to modularize C++ templates using AOP and program transformation.

3.2 Weaving into C++ Template Libraries

Aspects have the potential to interact with many different kinds of language constructs in order to modularize crosscutting concerns. Although several aspect languages have demonstrated advantages in applying aspects to traditional modularization boundaries (e.g., object-oriented hierarchies), additional language concepts such as parametric polymorphism can also benefit from aspects. Many popular programming languages support parametric polymorphism (e.g., C++ templates), but with the emergence of generics in Java 5, the combination of aspects and generics is a topic in need of further investigation. The section enumerates the general challenges of uniting aspects with C++ templates. It also emphasizes the need for new language constructs to extend AOP support to C++ templates and provides an initial solution to realize this goal.

The most detailed discussion of aspects and C++ templates is described in [Lohmann *et al*., 2004], within the context of AspectC++ (an aspect language for C++) [Spinczyk *et al*., 2002]. The effort to add aspects to templates in AspectC++ has been partitioned along two complimentary dimensions:

- Weaving advice into template bodies
- Using templates in the bodies of aspects

Where as the AspectC++ work has focused along the second dimension (i.e., using templates in the aspect body), the key contribution of this section is a deeper

investigation along the first dimension (i.e., weaving advice in the template body). In addition, the research enumerates a key challenge pertaining to C++ templates. Although a template is instantiated in multiple places, it may be the case that the crosscutting feature is required in only a subset of those instances. For example, it may be required to weave in vector templates of type `int` only (i.e., `vector<int>`), leaving vectors of all other types unchanged. Additional language features are required to describe such specific intentions and is explained in detail in the following sections.

3.2.1 Simple Pointcut Expressions for C++ Templates

This section introduces several essential concepts of AOP for C++ templates. An application of the Standard Template Library (STL) [Josuttis, 1999] vector class is presented, along with a description of a program transformation technique for modularizing a crosscutting concern among vector instances. Initially, some of the elementary pointcut language constructs for C++ templates are introduced in Section 3.2.1. Section 3.2.2 motivates the need for advanced pointcuts for C++ templates and presents our approach to support this technique.

Figure 3-12 shows a simple implementation of class `Foo` that uses several instances of the STL vector class. The join point model and pointcut language are explained in terms of actual template definitions. The listing is purposely simplified so that the concepts are not complicated by peripheral details. There are three fields defined in `Foo`, either of type `vector<int>` or `vector<float>`. The methods `getMyInts` and `getMyFloats` return the corresponding vector field, and the method `addFloats` adds a new floating point number to a given floating point vector

```
1.  #include <vector>
2.  using namespace std;
3.
4.  class Foo {
5.    public:
6.       vector<int> getMyInts();
7.       vector<float> getMyFloats();
8.       void addMyFloats(vector<float>,float);
9.    protected:
10.      vector<int> myInts;
11.      vector<float>  myFloats;
12.      vector<float>  someOtherFloats;
13.   };
14.
15.   vector<int> Foo::getMyInts() {
16.        return myInts;
17.   }
18.   vector<float> Foo::getMyFloats(){
19.        return myFloats;
20.   }
21.   void Foo::addFloats(vector<float> any,float aFloat) {
22.        any.push_back(aFloat);
23.   }
24.   ...
```

Figure 3-12 – An example class with multiple template instantiations

Using `Foo` as a reference for discussion, some of the primitive pointcut expressions defined in our aspect language for C++ templates are explained below:

- A primitive `get` for the field `myFloats` is captured by the following pointcut expression:

```
get(vector<*> Foo::myFloats) or
get(vector<float> Foo::myFloats)
```

Note the wildcard "*" refers to any vector type.

- The *`execution`* of *all* "getters" (i.e., `getMyInts` and `getMyFloats`) is matched by the pointcut expression:

```
execution(vector<*> Foo::get*(..)).
```

The expression "`get*`" matches all "`get`" methods.

- However, to match the `execution` of a *specific* get method (e.g., `getMyInts`), the above pointcut expression can be rewritten as:

```
execution(vector<int> Foo::getMyInts(..)).
```

Here, instead of using a wildcard, we specify the exact method signature.

- Similarly, a *`call`* to the method `addFloats` is matched by the pointcut expression:

```
call(void Foo::addFloats(vector<float>,float)).
```

A key challenge that is addressed in the design of the pointcut language occurs from the realization that a template can be instantiated in multiple places, yet it may be the case that the crosscutting feature is required in only a subset of those instances. A generalized pointcut expression that quantifies over specific types may capture several unintended instantiations. For example, if there are multiple `vector<float>` fields defined in class `Foo`, it may be required to log a call only to the `push_back` method for the field `myFloats`, and leave other `vector<float>` fields (e.g., `someOtherFloats`) unaltered.

The flexibility to quantify over specific template instances provides additional power towards AOP in C++ templates that is not limited to specific types. However, a language mechanism is needed to define the quantification scope of a pointcut with respect to the semantics of C++ templates. The following section motivates the need for advanced pointcut expressions for C++ templates through a preliminary example.

3.2.2 Advanced Pointcut for C++ Templates

A fragment of the actual STL vector class definition is presented in Figure 3-13a, which shows the implementation of two vector-specific operations, `push_back` and `pop_back`. The sample code in Figure 3-13b illustrates the use of a vector in an application program. In this simple application, three different types of vector instances are declared (i.e., vectors of type `int`, `char`, and `float`). The `push_back` method is invoked on each vector instance to insert an element of a different type.

```
1. template <class T>
2. class vector{
3. //...
4.
5. public:
6. void push_back
7.        (const T& x ) {
8. // insert element at end
9.   if (finish !=
10.     end_of_storage){
11.  construct(finish, x);
12.       finish++;
13.     } else
14.   insert_aux(end(), x);
15.    }
16. }
17.void pop_back() {
18.// erase element at end
19.  if (!empty())
20.   erase(end() - 1);
21.}
22.// ...
23.// other implementation
24.// details omitted here
25.};
```

a - STL vector implementation

```
1. class A {
2. vector<int> fi1;
3. vector<float> fi2;
4. void foo() {
5.    vector<int> ai;
6.    //...
7.    ai.push_back(1);
8.    fi1.push_back(2);
9.    fi2.push_back(3.0);
10.   //...
11.  }
12.};

1. class B {
2. vector<char> bc;
3. vector<int> fi;
4. void bar() {
5.    vector<int> bi;
6.    vector<float> bf;
7.    //...
8.    bc.push_back('a');
9.    bi.push_back(1);
10.   bf.push_back(2.0);
11.   //...
12.  }
13.   };
```

b - Application using STL vectors

Figure 3-13- STL vector class and its usage

Considering the canonical logging example, suppose that important data in specific vector instances needs to be recorded whenever the contents of the vector are changed. That is, within the context of an STL vector class, a requirement may state that

logging is to occur for all items added to each execution of the `push_back` method, *but only for specific instantiations*. For example, it may be desired to log only vector fields of type `<int>` in class `A` (e.g., field `fi1` in class `A`) without affecting other local vector instantiations of type `int` in class `A` or `B` (e.g., those appearing in the local scope of method `foo` in class `A` or method `bar` in class `B`).

In order to record or log the contents of a given vector instance, the `push_back` method as defined in the original vector template (Figure 3-13a) must be adapted. However, any change to this base template definition will affect all instantiations that reference the original vector template. For example, if logging support is added to the `push_back` method in the original vector template, all instantiations of vector (e.g., fields `fi1`, `fi2` in class `A`, fields `bc`, `fi` in class `B`, or method variables `bi` or `bf` in class `B`) will automatically implement support for logging. But, according to the requirement, it is only desired to capture logging to specific instances of the vector (e.g., fields of type `vector <int>`) and not to all its instances. This challenge is addressed using Template Subtyping.

Template Subtyping: In order to affect only `int` instances of the given vector template in fields of class `A` (or fields of class `B`) and leave other types (e.g., `float`, `char`) of vector instances unaltered, a new subtype `vector$1` is constructed, which inherits from the original vector template. The log statement is then added to the over-written `push_back` method of the `vector$1` template. The top-half of Figure 3-14 shows the adapted definition of the `push_back` method in this `vector$1` template. Note that the method call `log.add(x)` is added at the beginning of the `push_back` method in Figure 3-14. Finally, all field references in class `A` and `B` of type

`vector<int>` are updated with this new `vector$1` template (shown in the middle of Figure 3-14). However, all other references to the original vector template are left unaltered (e.g., field `fi2` or method variable `ai` in class `A`).

Figure 3-14- STL `vector$1` class with updated references in Application instances

Although template specialization seems related to template subtyping, there could be instances where specialization may fail. For example, if only a particular instance of a specific type needs to be adapted (i.e., *only* the field `fi1` in class `A`), specialization techniques would fail as any specialization will be universally applied to *all* references of type `vector<int>` (e.g., method variable `ai` in class `A`). However, using template subtyping, *only the functions that need to be adapted are transformed with respect to the*

new aspect semantics, but the rest of the class template remains unchanged. The scoping rules required to describe the context of a given pointcut expression with respect to C++ templates are discussed further.

Scoping Rules for Templates: Figure 3-15 illustrates the scoping rules for templates.

Scope Designator	Description
C::*	All global template instantiations of class C (fields)
* C.*(..)::*	All local template instantiations within all methods of class C
(C::* || * C.*(..)::*)	All template instantiations (both global and local) within class C
C.M(..)::*	All local template instantiations within method M of class C
* C.*(..)::V	Any template instantiation that is referenced by a variable V in all methods of class C
* C.M(..)::V	Template instantiation that is referenced by a variable V in method M of class C

Figure 3-15- Scope designators in pointcut expressions

From the categorization of scope designators shown in Figure 3-15, the example from Figure 3-14 can be re-visited to observe the scoping rules for classes `A` and `B` in the application program. At the bottom of Figure 3-14, two pointcut specifications are shown that capture the logging concern for specific `vector` instances depending on the scoping rule applied to the base class template. The pointcut in the bottom-left of Figure 3-14 can be read as, "select all fields of type `vector<int>` in *class* `A` that lead to an execution of the `push_back` method." Similarly, the pointcut in the bottom-right of Figure 3-14

can be read as, "select all fields of type `vector<int>` in *class* `B` that lead to an execution of the `push_back` method."

To illustrate this scoping rule further, additional examples are provided in Figures 3-16 through 3-20. Each pointcut definition is progressively more focused in limiting the scope of the join points that are captured (i.e., from a pointcut that captures all vectors of any type in any class, down to a pointcut that specifies a specific instance in a distinct method). Figure 3-16 offers an example of the aspect language to add the logging statement to the `push_back` method in *all* vectors of any type from *any* class. The pointcut `push_back_method` represents the points of `execution` where the advice is to be applied. In the pointcut expression, `vector<*>` denotes all types of vector instances.

```
1.  template <class T>
2.  aspect InsertPushBackLogToAllVector {
3.    pointcut push_back_method(const T& x):
4.       execution(vector<*>::push_back(..)) &&  args(x);
5.    before(const T& x):push_back_method(x) {
6.      log.add(x);
7.     }
8. }
```

Figure 3-16 - Aspect specification for inserting the `push_back` log to all vectors of ANY type in ANY class

Figure 3-17 defines a pointcut that specifies the `execution` join point for the `push_back` method of all vectors of type `int`. The low-level implementation details involving the program transformation rules to automate the required changes to the template class and application program will be shown in Section 3.2.3.

```
1. pointcut push_back_method():
2.   execution(vector<int>::push_back(..));
```

Figure 3-17- Pointcut specification for weaving into all vectors of type `int` in ANY class

To add finer granularity, Figure 3-18 describes the pointcut specification for `execution` of all vectors of type `int` in class `A`. To be more specific in limiting the scope of a pointcut, Figure 3-19 defines a pointcut capturing all `int` vectors in method `foo` that are defined in class `A`.

```
1. pointcut push_back_method():
2.   execution((A::*  || * A.*(..)::*)<-
3.              vector<int>::push_back(..));
```

Figure 3-18- Pointcut specification for weaving into all vectors of type `int` in class `A`

```
1. pointcut push_back_method():
2.   execution(* A.foo(..)::*<-
3.              vector<int>::push_back(..));
```

Figure 3-19- Pointcut specification for weaving into all vectors of type `int` in method `foo` of class `A`

Figure 3-20 is the most specific pointcut expression. It will only match a particular template instance `ai` whose type is of `vector<int>` and is defined within the scope of method `foo` of class `A`.

```
1. pointcut push_back_method():
2.   execution(* A.foo(..))::ai<-
3.              vector<int>::push_back(..));
```

Figure 3-20- Pointcut specification for weaving into vectors of type `int` and referenced by variable `ai` in method `foo` of class `A`

3.2.3 Template Weaving using Program Transformation

The aspect language shown in the previous section illustrates the high-level language specifically constructed to handle C++ templates. In this section, emphasis is placed on the low-level implementation details used to automate the weaving process through a program transformation engine.

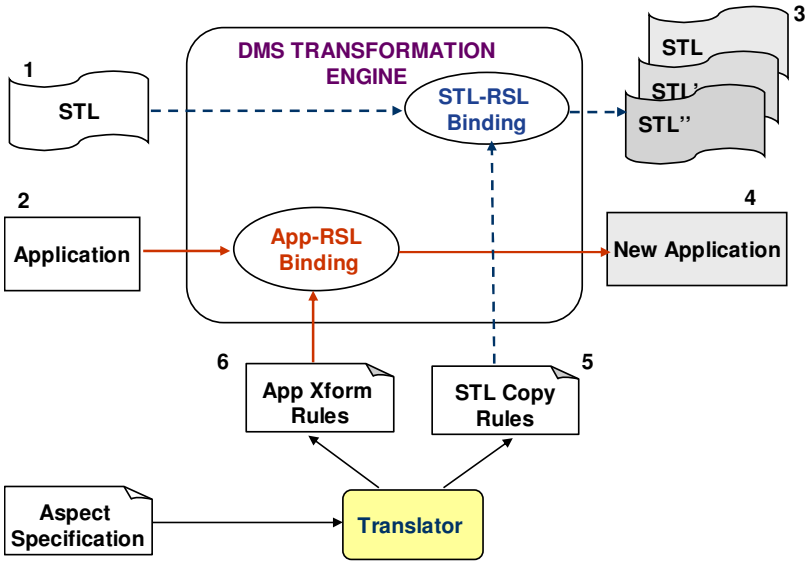


Figure 3-21- Overview of template weaving process

Figure 3-21 presents an overview of the automated transformation process that uses the DMS program transformation system as its underlying engine. One of the major components involved in the implementation of the weaver is the translator (bottom of figure), which parses and translates a high-level aspect language into low-level rewrite rules (i.e., referenced as items #5 and #6). This facilitates the application programmers to specify their intent using a high-level aspect language and remain oblivious to the existence of a low-level transformation engine.

The heart of the weaving process (core infrastructure) is the DMS transformation engine, which takes the source files and the generated rules as input. The user provides three different source files as input to the transformation process: the original STL source code (shown as item #1 in Figure 3-21), an application program based on the STL instances (shown as item #2), and a high-level aspect language specification (examples shown in Section 3.2.2) used to describe the specific crosscutting concern with respect to template instantiations.

The translator includes a lexer, parser, and pattern evaluator (i.e., pattern parser and attribute evaluator) that takes the aspect specification and instantiates two different sets of parameterized transformation rules (i.e., STL copy rules and App transformation rules, shown separately as #5 and #6 in Figure 3-21). The pointcut expressions are bound to the corresponding transformation rules that are instantiated for matching patterns. The STL copy rules generate a subtype copy of the original STL class template by inheriting from the base template. The crosscutting concerns are weaved into this new subtype by overwriting appropriate methods as defined in the STL-RSL Binding. Note that each subtype copy rule encapsulates only one crosscutting concern for each specific template type (e.g., `vector<float>`). Therefore, it is desired to generate only one subtype copy for every type, each of which has one specific concern weaved into its base definition (shown as #3). However, if multiple concerns crosscut a specific type, then the corresponding subtype copy should also replicate this behavior by encapsulating multiple crosscutting concerns weaved into one copy. Similar to the STL-RSL Binding, the App-RSL Binding transformation modifies the user application program (shown as #2) based on the App transformation rules, and generates the new application (shown as #4) that is

able to be compiled as a pre-processing phase and executed along with the generated subtype STL copies. The remainder of this section provides a discussion of the transformation rules that implement these ideas.

Transformation Rules for Template Weaving: Figure 3-22 (STL template subtype copy rule, also shown as #5 in Figure 3-21) shows the low-level RSL specification for weaving a logging concern into the `push_back` method in an STL `vector` class. Two steps are involved in the weaving process: 1) make a subtype copy of the original vector template class, and 2) insert the logging statement into appropriate places in the abstract syntax tree. The first line of the rule establishes the default base language domain (e.g., C++) to which the transformations are applied.

```
1.    default base domain Cpp.
2.    pattern log_statement(): statement_seq = "log.add(x);".
3.    pattern weaved_method_name(): identifier = "push_back".
4.    pattern new_template_name(): identifier = "vector$1".

5.    external pattern copy_template
6.       ( td : template_declaration,
7.         st : statement_seq,
8.         method_name : identifier,
9.         template_name : identifier ):
10.   template_declaration = 'copy_template' in domain Cpp.
11.
12.rule insert_log_to_template
13.   ( td : template_declaration ):
14.     template_declaration -> template_declaration
15.   = td ->
16.     copy_template (td, log_statement(),
17.                    weaved_method_name(),
                          new_template_name()).

18.public ruleset applyrules = { insert_log_to_template }.
```

Figure 3-22- DMS transformation rules for weaving log statement into `push_back` method

Pattern `log_statement` in line 2 represents the log statement that will be inserted before the execution of the `push_back` method. Pattern

`weaved_method_name` in line 3 defines the name of the method that will be transformed (i.e., `push_back` in this case). Pattern `new_template_name` in line 4 specifies the new name for the vector (i.e., `vector$1`).

As stated earlier, exit functions (i.e., external patterns and functions) in DMS are written in PARLANSE, which use internal APIs for performing various traversal and tree operations on the parsed AST. In this example, the external pattern `copy_template` (line 5 of Figure 3-22) is a PARLANSE function that performs the actual process of subtyping, naming, and weaving.

This external pattern takes four input parameters: 1) a template declaration to be operated on, 2) a statement sequence representing the advice, 3) a method name where the advice is to be weaved, and 4) a new name for the template subtype. The rule `insert_log_to_template` on line 12 triggers the transformation on the vector class by invoking the specified external pattern.

After applying this rule to the code fragment shown in Figure 3-23, a new template class named `vector$1` (inherited from `vector`) will be generated with the logging statement inserted at the beginning of the `push_back` method (i.e., the automated result is the same as found in Figure 3-14). At this stage, the weaving process is still not complete because the application program also needs to be updated to reference the new `vector$1` instance.

```
1.    default base domain Cpp.
2.    pattern pointcut( id : identifier ):
3.         declaration_statement = "vector<int> \id;".
4.
5.    pattern advice( id : identifier ):
6.         declaration_statement = "vector$1<int> \id;".
7.
8.    external pattern replace_vector_instance
9.          ( cd  : class_declaration,
10.        ds1 : declaration_statement,
11.        ds2 : declaration_statement ):
12.        class_declaration = 'replace_vector_instance'
13.                                        in domain Cpp.
14.
15.rule replace_template_instance
16.    ( cd : class_declaration, id : identifier):
17.
18.      class_declaration -> class_declaration
19.    = cd -> replace_vector_instance
20.           (cd,pointcut(id),advice(id)).
21.public ruleset applyrules = {replace_template_instance}.
```

Figure 3-23- DMS transformation rules to update the application program

The DMS transformation rule to update the corresponding application program (App transformation rule, also shown as #6 in Figure 3-21) is specified in Figure 3-23. Pattern `pointcut` (lines 2 and 3) identifies the condition under which the rule will be applied (i.e., in this case, all `int` vector declarations). Pattern `advice` (lines 5 and 6) defines the name of the new transformed type (`vector$1<int>`). After applying this particular rule (line 21) to a given user application, the external pattern `replace_vector_instance` replaces the type of every template instantiation declared as type `vector<int>` into an instance of type `vector$1<int>`.

Sections 3.1 and 3.2 primarily focused on constructing aspect weavers by applying program transformation rules on Object Pascal and C++ Templates respectively. These sections also described some of the AOP language extensions required to support aspect weaving in the case of C++ templates. In the following section, application of program transformation systems to improve the modularization and construction of

scientific libraries will be discussed. First, we introduce Blitz++ [Veldhuizen, 1998], which is a C++ template based library used for numerical computing applications and shows how AOP can improve the modularity of such libraries. Second, we discuss a technique to construct or compose scientific libraries based on specialization for a given architecture. The latter technique is not directly related to AOP, but nevertheless, it shows the usefulness of program transformation for constructing such systems.

3.3 Adaptation and Specialization of Scientific Libraries

Scientific computing was an initial application domain for the early examples of AOP [Irwin *et al*., 1997]. However, aside from an application of AspectJ [Kiczales *et al.*, 2001] to an implementation of JavaMPI [Harbulot and Gurd, 2004], AOP has not been applied or investigated deeply within the area of scientific computing. This is primarily due to the fact that such applications are typically written in FORTRAN, C, or C++, but the center of AOP research has largely remained focused on Java-based implementations. Nevertheless, there is a strong potential for impact if aspects can be used to improve the modularization of scientific computing applications written in languages other than Java.

3.3.1 Aspects in Blitz++

Optimizing performance, while preserving the benefits of programming language abstractions, is a major hurdle faced in scientific computing [Skjellum *et al*., 2004; Quinlan *et al*. 2004; Veldhuizen and Dennis Gannon, 1998]. Object-Oriented Programming Languages (OOPLs) have popularized useful features (e.g., inheritance and polymorphism) in the development of complex scientific problems. However, the

performance bottleneck associated with OOPLs has been a major concern among High-Performance Computing (HPC) researchers. Alternatively, languages such as FORTRAN have dominated the numerical computing domain, even though the primitive programming constructs in such languages make applications difficult to maintain and evolve.

Compiler extensions (e.g., High Performance C++ [Johnson and Gannon, 1997] and High Performance Java [Getov *et al*., 1998]) and scientific libraries (e.g., POOMA [Reynders *et al*., 1996], MTL [Siek and Lumsdaine, 1998], and Blitz++ [Veldhuizen, 1998]) have been developed to extend the benefits of object-oriented programming to the scientific domain. In particular, Blitz++ is a popular scientific package that has specific abstractions (e.g., arrays, matrices, and tensors) that support parametric polymorphism through C++ templates. The goal of the Blitz++ project was to develop techniques that enable C++ to compete or exceed the speed of FORTRAN for numerical computing. Blitz++ arrays offer functionality and efficiency, but without any language extensions. The Blitz++ library is able to parse and analyze array expressions at compile-time and perform loop transformations. Blitz++ currently provides dense vectors and multidimensional arrays, in addition to matrices, random number generators, and tiny vectors. The overall size of the Blitz++ library is approximately 115K source lines of code (SLOCs). Moreover, there are several additional source code directories that serve as benchmarks and test cases.

Although Blitz++ makes extensive use of templates for array and vector implementation, the issue addressed in this paper is the ability to apply AOP concepts to large scientific template libraries like Blitz++. This section contains a description of some

of the array and vector implementation templates in Blitz++, and identifies several crosscutting features in the current Blitz++ implementation. The general approach could be applied to other libraries that use parametric polymorphism implemented in languages such as Ada or Java.

The first example in this section represents the common case of a debugging precondition that appears in array-impl.h and resize.cc. These files primarily represent arrays whose dimensions are unknown at compile-time and require resizing during runtime. In addition, there are several methods that perform block reduction operations and conversions to and from a matrix and vector.

A second crosscutting feature in array-impl.h is `setupStorage`, which is used for initial memory allocation for arrays and appears in both array-impl.h and resize.cc.

The third example is based on redundant assertion checks on the lower and upper bounds of an array during instantiation. It appears 46 times in array-impl.h and is similar in concept to redundant assertion checking described by Lippert and Lopes [Lippert and Lopes, 2000].

Precondition and `setupStorage` Aspects: The Blitz++ library has a debugging mode that is enabled by defining the preprocessor directive `BZ_DEBUG`. In this mode, an application executes slowly because Blitz++ performs precondition and bounds checking on the array index. Under this condition, if an error or fault is detected by the system, the program halts and displays an error message. Figure 3-24 shows a sample precondition check for an array implementation. The rank of the vector influences the precondition to be checked.

Another aspect that crosscuts the array implementation boundaries is `setupStorage`. The method is called to allocate memory for any new array. However, any missing length arguments will have their value taken from the last argument in the parameter list. For example, `Array<int,3> A(32,64)` will create a 32x64x64 array, which is handled by the routine `setupStorage`. Both the `BZPRECONDITION` (lines 7 and 15 of Figure 3-24) and `setupStorage` (lines 9 and 17) can be individually considered as two different pieces of advice applied to the same pointcut (i.e., the former as `before` advice and the latter as `after` advice).

Figure 3-25 presents the corresponding aspect specification for the crosscutting concern identified in Figure 3-24. This allows the separation of crosscutting concerns from the base code (Figure 3-24) and encapsulates them as aspects (Figure 3-25) to be woven using a low-level translator and weaver.

The `BZPRECONDITION` statement (line 4 in Figure 3-25) and the `setupStorage` statement (line 7 in Figure 3-25) form part of the `before` and the `after` advice. The pointcut `ArrayConstructor` refers to execution of all `Array` constructors defined in any `Array` type (denoted by the wildcard *). However, if it is desired to match only arrays of type `Array<int>`, more selective pointcuts can be used. The function call `thisJoinPoint.getArgs().length` will return the length of the parameter list in the `Array` constructor.

```
1.  template<typename T_expr>
2.    _bz_explicit Array (_bz_ArrayExpr<T_expr> expr);

3.  Array(int length0, int length1,
4.        GeneralArrayStorage<N_rank> storage = GeneralArrayStorage<N_rank>())
5.   : storage_(storage)
6.   {
7.     BZPRECONDITION(N_rank >= 2);
8.     // implementation code omitted
9.     setupStorage(1);
10.  }
11.  Array(int length0, int length1, int length2,
12.       GeneralArrayStorage<N_rank> storage = GeneralArrayStorage<N_rank>())
13.  : storage_(storage)
14.  {
15.    BZPRECONDITION(N_rank >= 3);
16.    // implementation code omitted
17.    setupStorage(2);
18.  }
```

Figure 3-24- Precondition check and `setupStorage` in Blitz++ array implementation

```
1. aspect InsertBZPreCon_MemAllocation {
2.  pointcut ArrayConstructor(): execution(Array<*>::Array(..));

3.  before(): ArrayConstructor() {
4.     BZPRECONDITION(N_rank  >= thisJoinPoint.getArgs().length());
5.  }
6.  after(): ArrayConstructor() {
7.     setupStorage(thisJoinPoint.getArgs().length()-1);
8.  }
9. }
```

Figure 3-25- Aspect specification for precondition and memory allocation in templates

Redundant Assertion Checking: Another crosscutting feature present in Blitz++ is assertion checking, which is used to evaluate the size or range of array instances. To detect errors in ranges, each array allocation makes an implicit call to `assertInRange`, which checks the lower and upper bounds of an array instance.

This particular assertion is defined in all array template specifications, according to a general pattern as shown in Figure 3-26 (e.g., `assertInRange` in lines 3 and 7). However, note that the number of index parameters passed to the `assertInRange` routine implicitly depends on the size of the `TinyVector`. For example, as presented in

Figure 3-26, to allocate a `TinyVector` of size 1 requires only one parameter (i.e., `index`[0]) to be passed to `assertInRange`. Similarly, for a different allocation size of N, the range will be checked on `index`[0], `index`[1], ... ,`index`[N-1]. This kind of assertion is repeated 46 times in array-impl.h and is context-dependent on the size of each template container.

```
1.  template<int N_rank2> T_numtype operator()
2.                        (TinyVector<int,1> index) const {
3.      assertInRange(index[0]);
4.      return data_[index[0] * stride_[0]];
5.  }
6.  T_numtype operator() (TinyVector<int,2> index) const {
7.      assertInRange(index[0], index[1]);
8.      return data_[index[0] * stride_[0] +  index[1] * stride_[1]];
9.  }
```

Figure 3-26- Redundant assertion check on base template specification

To avoid the crosscutting assertion checking in every definition of an array implementation, the aspect specification (as defined in Figure 3-27) will weave this concern into the template code. The `operator` pointcut refers to all operator methods in the array implementation class. The `getParamList` special construct (line 7 of Figure 3-27) returns the list of index parameters for each call to `assertInRange`.

```
1. aspect AssertInRange {
2.
3.    pointcut operator ():
4.     execution(Array<*>::operator()(..));
5.
6.    before(): operator() {
7.      assertInRange(thisJoinPoint.getParamList());
8.    }
9. }
```

Figure 3-27- Aspect specification for redundant assertion checks

Crosscutting Generic Functions: This subsection discusses the combination of AOP with other generative programming techniques [Czarnecki and Eisenecker, 2000]. In Blitz++, templates such as binary and unary operations for arrays and vectors are synthesized from a code generator implemented in several C++ routines. For consideration in this section, attention is focused on a specific set of unary vector (mathematical) operations in a template specification, which are generated to the vecuops.cc source file in the Blitz++ library containing approximately 12K SLOCs. Most of these mathematical operations (e.g., `log`, `sqrt`, `sin`, `floor`, `fmod`) have the same syntactic structure and can be specified concisely in the form of a pattern. An analysis of the generation process revealed that the entire template specification is essentially a cross-product between the set of defined mathematical operations (λ) and a base template (β) that represents the general pattern structure. As observed, the set of mathematical functions crosscut the entire unary vector general pattern.

If $\lambda_1, \lambda_2,...\lambda_n$ represent the set of mathematical operations (e.g., `log`, `sqrt`, `sin`) that crosscut the partial base template structure β (whole of Figure 3-28), then the code generated as the cross-product of λ and β can be represented as $\lambda_1\beta + \lambda_2\beta +... + \lambda_n\beta$. The partial string identifier `OPERATION` (highlighted in bold in Figure 3-28) identifies the locations in the partial base template structure where the mathematical operations must be woven to generate the whole template structure (i.e., $\sum \lambda \times \beta$ = 12k SLOCs). The concept here is somewhat different than standard AOP practice and more analogous to generative programming, but the idea of a cross-product between a set of mathematical operations and a base pattern is germane to the overall process of template weaving. Although this example is based on vector operations using mathematical functions, similar situations

(e.g., operations on Blitz++ arrays) exist in several other generated template specifications in the Blitz++ library.

```
1.    template<class P_numtype1>
2.    inline _bz_VecExpr   <_bz_VecExprUnaryOp <VectorIterConst<P_numtype1>,
3.                           _bz_OPERATION<P_numtype1>>>
4.
5.    OPERATION(Vector<P_numtype1>& d1)
6.    {
7.      typedef bz_VecExprUnaryOp <VectorIterConst<P_numtype1>,
8.                                 _bz_OPERATION<P_numtype1>> T_expr;
9.      return _bz_VecExpr <T_expr> (T_expr (d1.begin()));
10.   }
```

Figure 3-28- Subset of base pattern used to generate the vector operation template

The transformation rule describing the weaving of the mathematical functions with the base pattern is shown in Figure 3-29. The first line of the rule identifies the programming language (base domain) of the transformed source, which is C++ in Blitz++. Lines 3-8 use patterns for matching a syntax tree with a specified structure. The rule as shown in Line 11 describes a directed pair of corresponding syntax trees. The right-hand side of the rule specification uses an external function (i.e., `generate_template_code` in Line 27) to generate code. The first parameter to this external function is a template definition (β). The second and third parameters are the two annotated markers in the source AST that need to be replaced with the enumerated mathematical operations. The fourth and subsequent parameters are the set of generic mathematical operations (e.g., log, sin, sqrt) to be woven into the base pattern during code generation. Using the above rule specification and the base pattern as shown in Listing 6, nearly 12K source lines of code are generated which resembles the entire set of unary mathematical operations present in the Blitz++ library.

```
1    default base domain Cpp.
2
3    pattern aspect_op():    identifier = "OPERATION".
4    pattern aspect_bz_op(): identifier = "_bz_OPERATION".
5
6    pattern op1(): identifier = "log".
7    pattern op2(): identifier = "sin".
8    pattern op3(): identifier = "sqrt".
9    ...
10
11   rule generate_vec_template (td:template_declaration):
12       declaration_seq -> declaration_seq
13       =
14       td -> generate_template_code (td, aspect_op(),
15      aspect_bz_op(), op1(), op2(), op3(),...)
```

Figure 3-29- Rules used to generate mathematical operations using a base template definition

In the following section we analyze a popular scientific computing library, namely, High Performance Linpack (HPL) Benchmark [Petitet *et al*., 2004] and discuss how program transformation techniques can improve the modularity and construction effort of such libraries.

3.3.2 Specializing HPL using Program Transformation

Given the abundance of legacy code available in the scientific computing domain, there have been several constructive efforts made within the software engineering community to reduce the cost of development and the life-cycle maintenance of such systems. This section discusses how software product line architecture can be adopted and generative component engineering [Czarnecki and Eisenecker, 2000] can be applied during the development and maintenance of scientific computing libraries. In particular, we investigate a well-known scientific computing library - High Performance Linpack Benchmark [Petitet *et al*., 2004], and demonstrate how specialization of HPL for a given

architecture can improve its comprehensibility, reduce its memory footprint and thereby improve its overall performance, portability and maintenance costs.

HPL is a software package that solves a random dense linear system (LU factorization) on distributed-memory architectures [Petitet *et al*., 2004]. The HPL software package requires the availability of an implementation of either the Basic Linear Algebra Subprograms (BLAS) [Dongarra, 2002] or the Vector Signal Image Processing Library (VSIPL, http://www.vsipl.org/). Machine-specific as well as generic implementations of the BLAS and VSIPL are available for a large variety of systems. Furthermore, BLAS can be categorized into FBLAS (a FORTRAN implementation of BLAS) or CBLAS (a C implementation of BLAS) [Dongarra, 2002]. Depending on the machine architecture and the availability of the type of BLAS (either FBLAS or CBLAS) or VSIPL, the software package mostly relies on preprocessor directives to make specific calls to appropriate linear algebra subroutines.

A mechanism that can automatically specialize (i.e., deconstruct and reconstruct) the library based on the underlying machine architecture and the availability of the type of linear algebra package can greatly reduce the lines of code in HPL, thereby improve its comprehensibility and reduce its overall memory footprint. DMS is used as the underlying transformation engine to achieve this goal.

The current macro-based implementation of HPL showing its dependence on the specific linear algebra library is presented in Figure 3-30. The figure shows a specific code snippet of one of the BLAS libraries that uses preprocessor directives to make appropriate calls to BLAS or VSIPL subroutines.

```
1.  #ifdef HPL_CALL_CBLAS
2.      cblas_dgemm( ORDER, TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB,
                     BETA, C, LDC );
3.  #endif
4.  #ifdef HPL_CALL_VSIPL
5.      if( ORDER == HplColumnMajor )
6.      {
7.           HPL_dgemm0( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB,
8.                       BETA, C, LDC );
9.      }
10.     else
11.     {
12.          HPL_dgemm0( TRANSB, TRANSA, N, M, K, ALPHA, B, LDB, A, LDA,
13.                      BETA,C, LDC );
14.     }
15. #endif
16. #ifdef HPL_CALL_FBLAS
17.      double          alpha = ALPHA, beta = BETA;
18. #ifdef StringSunStyle
19. #ifdef HPL_USE_F77_INTEGER_DEF
20.      F77_INTEGER     IONE = 1;
21. #else
22.      int             IONE = 1;
23. #endif
24. ...
```

Figure 3-30- Preprocessor directives in a HPL software package

The various preprocessor directives that conditionally check for the specific type of linear algebra subroutine make the code increasingly difficult to maintain. Any new functionality that needs to be added later on must include such macro-based conditional checks. This significantly increases the overall size of the software although a considerable part of it is never processed. Therefore, a technique than can automatically specialize the library depending on the available architecture is better suited for future maintenance. In addition, such specialization reduces the overall size of the library (Figure 3-33) that may lead to better understanding of the core functionalities of the software. Moreover, a smaller memory footprint can also result in general improvement in performance of HPL (Figure 3-34).

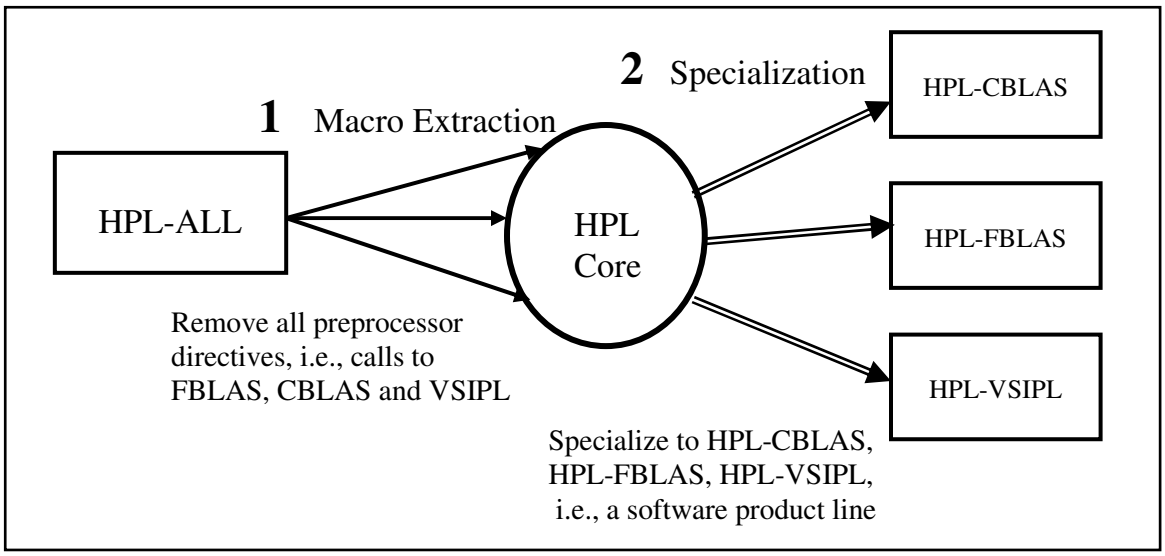


Figure 3-31- Specialization overview of HPL software package

Figure 3-31 provides an overview of the specialization technique applied to HPL. Step one denotes the deconstruction process that removes all preprocessor directives (i.e., relating to the type of linear algebra subroutine) from the existing HPL package. The resultant HPL core is then specialized (i.e., reconstructed) depending on the underlying hardware and software architecture. This produces a family of HPL packages (i.e., a software product line) with appropriate calls to specific linear algebra subroutines.

In order to realize this goal, the program transformation rule as shown in Figure 3-32 serves as the initial solution.

```
1.  external pattern remove_macro(tran_unit:translation_unit,id:identifier):
2.            translation_unit = 'remove_macro' in domain Cpp~ISO14882c1998.

3.  external pattern add_macro(tran_unit:translation_unit,id:identifier):
4.            translation_unit = 'add_macro' in domain Cpp~ISO14882c1998.

5.  pattern FBLAS(): identifier = "HPL_CALL_FBLAS".
6.  pattern CBLAS(): identifier = "HPL_CALL_CBLAS".
7.  pattern VSIPL(): identifier = "HPL_CALL_VSIPL".

8.  rule del_cblas(t_u: translation_unit): translation_unit -> translation_unit
9.  = t_u -> remove_macro(t_u,CBLAS())
10. if tran_unit ~= remove_macro(t_u,CBLAS()).

11. rule del_vsip(t_u: translation_unit): translation_unit -> translation_unit
12.  = t_u -> remove_macro(t_u,VSIPL())
13. if t_u ~= remove_macro(t_u,VSIPL()).

14. rule del_fblas(t_u: translation_unit): translation_unit -> translation_unit
15.  = t_u -> remove_macro(t_u,FBLAS())
16. if t_u ~= remove_macro(t_u,FBLAS()).

17. rule add_cblas(t_u: translation_unit): translation_unit -> translation_unit
18.  = t_u -> add_macro(t_u,CBLAS())
19. if t_u ~= add_macro(t_u,CBLAS()).

20. rule add_vsip(t_u: translation_unit): translation_unit -> translation_unit
21.  = t_u -> add_macro(t_u,VSIPL())
22. if t_u ~= add_macro(t_u,VSIPL()).

23. rule add_fblas(t_u: translation_unit): translation_unit -> translation_unit
24.  = t_u -> add_macro(t_u,FBLAS())
25. if t_u ~= add_macro(t_u,FBLAS()).
```

Figure 3-32- Transformation rule for specializing macro definitions in HPL

The transformation uses two external patterns (`remove_macro` and `add_macro`) to substitute appropriate preprocessor directives (i.e., `HPL_CALL_FBLAS`, `HPL_CALL_CBLAS` or `HPL_CALL_VSIPL`) from the HPL software package. The external patterns are implemented as PARLANSE functions and are shown in Appendix F. Step one involves the execution of the rules `del_cblas`, `del_fblas` and `del_vsipl` and step two involves the execution of the architecture-specific rule (i.e., either of `add_cblas` or `add_fblas` or `add_vsipl`). The rules are subsequently processed by DMS to achieve the desired specialization.

Figure 3-33 shows the overall size of the HPL BLAS library before and after specialization with CBLAS. The CBLAS version HPL comparatively contains very few lines of code compared to the original version.

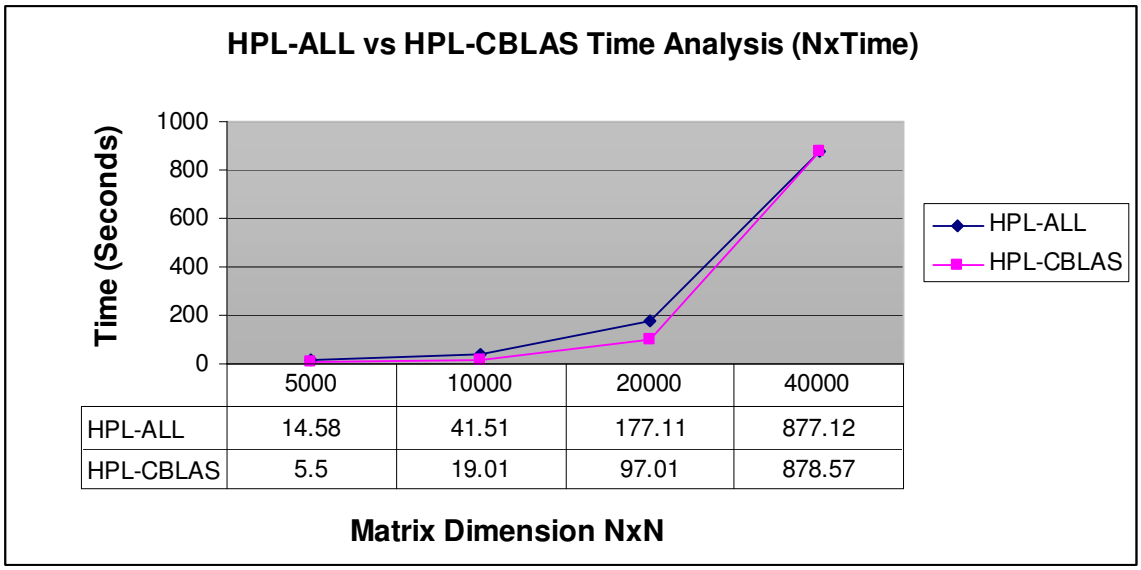
Size of HPL BLAS (*before*)		**Size of HPL BLAS (*after*)**	
Number of lines of code:	1719	Number of lines of code:	40
Number of directive lines:	390	Number of directive lines:	50
Number of empty lines:	172	Number of empty lines:	10
Number of comment lines:	731	Number of comment lines:	731
Number of empty comment lines:	327	Number of empty comment lines:	327
Total number of lines:	3339	Total number of lines:	1158

Figure 3-33- Comparing size of HPL BLAS library before and after specialization

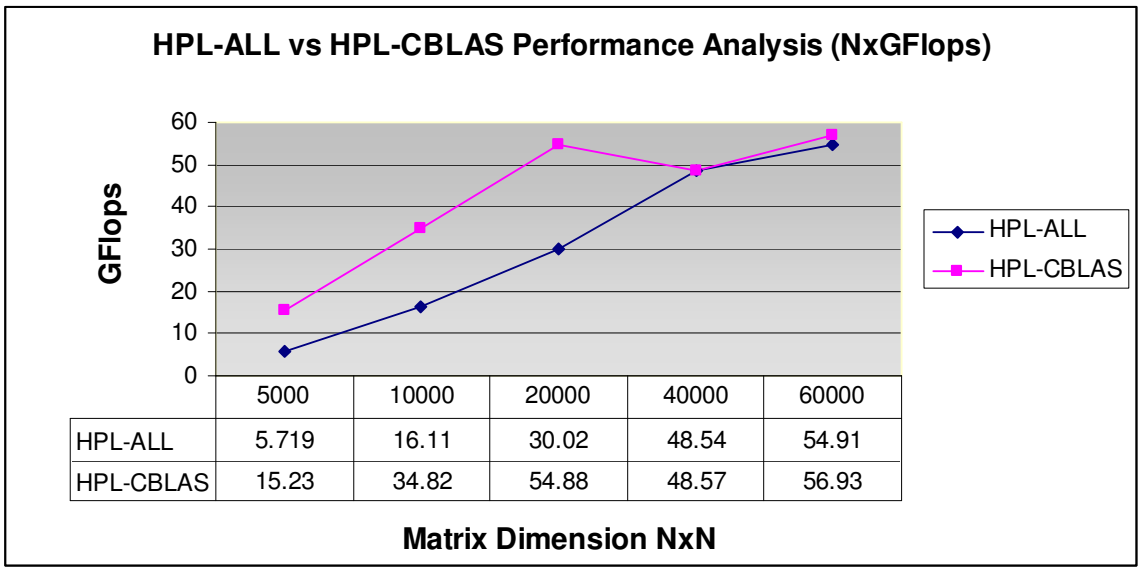
Figure-3-34 shows the time analysis and performance analysis graphs between HPL-ALL and HPL-CBLAS for a fixed block size of 112, row-major process mapping of 4x8 (PxQ) and variable matrix dimension of 10000 to 60000 (NxN) square matrix. It may be noted that HPL-CBLAS is a specialized version of HPL that only contains calls to the HPL-CBLAS linear algebra package.

From the graph shown in Figure 3-34, it is observed that for small matrix dimensions (10000 – 20000), HPL-CBLAS gives an improved performance over HPL-ALL. However, as the dimension increases, the performance eventually evens out. The increased performance for HPL-CBLAS for small matrix dimensions may be attributed to the reduced complexity of the code due to removal of preprocessor directives from the base HPL package. This allows the compiler (preprocessor) and the runtime system to process fewer tasks (conditional checks) that results in an overall increase of initial

performance. Obviously as the matrix size increases, these gains are weighted out as compared to other tasks that lead to both versions performing similarly.



	5000	10000	20000	40000
HPL-ALL	14.58	41.51	177.11	877.12
HPL-CBLAS	5.5	19.01	97.01	878.57



	5000	10000	20000	40000	60000
HPL-ALL	5.719	16.11	30.02	48.54	54.91
HPL-CBLAS	15.23	34.82	54.88	48.57	56.93

Figure 3-34- Time and performance analysis between HPL-ALL vs. HPL-CBLAS

Although the performance increase is a beneficial side-effect, the main advantage of HPL specialization is the improved comprehensibility of the software package (i.e.,

better understanding of the complex scientific library). In addition, the reduced size of the library may make it easier to maintain and evolve the software package.

3.4 Related Work

It is commonly known within the AOSD community that aspect weaving can be performed using a general transformation framework for a specific programming language. This observation was first made by Fradet and Südholt as an early position paper [Fradet and Südholt, 1998]. In similar work, a detailed description of a weaver for a declarative language was provided by Lämmel [Lämmel, 1999], which used functional meta-programs to weave aspects. The ideas described in this chapter are not focused on foundations of transformation systems, but rather the scalability to which legacy languages can be supported by existing transformation engines.

Several researchers have contributed valuable results in the area of language extension frameworks. As an example, the Jakarta Tool Suite (JTS) contains the basic tools to support the addition of new programming features to Java [Batory *et al.*, 1998]. JTS assists in the construction of new pre-processors for DSLs that are transformed into Java. Another tool, called JastAdd, is a weaver and compiler construction system for Java based on AST transformation using JavaCC [Hedin and Magnusson, 2003]. Although all of these tools have advanced technologies for extension and analysis, these efforts are still bound to a specific language (i.e., Java). In the GENOA system, Devanbu has observed that many program analysis tools offer a fixed-point solution such that their internal structure is unusable in other similar contexts. For example, the parser, type checker, and parse-tree analysis algorithms for a C++ metrics tool are often not reused in

other C++ static analysis tools [Devanbu, 1999]. GENOA claims to have support for re-targetable front-ends, but it is not evident that GENOA provides a diverse set of commercial-grade parsers to realize this claim.

At the AOSD web site (http://www.aosd.net/), several weaver research prototypes are described for various languages. Aside from AspectJ [Kiczales *et al.*, 2001], perhaps the most mature of these is AspectC++ [Spinczyk *et al*., 2002]. As noted in the introduction of Section 3.2, a discussion of templates and aspects in AspectC++ within the context of generative programming is discussed in [Lohmann *et al*., 2004]. The focus of the AspectC++ work is on the interesting notion of incorporating parametric polymorphism into the bodies of advice. In contrast, the focus of C++ template weaving is a deeper discussion of the complimentary idea of weaving crosscutting features into the implementation of template libraries.

In addition to AspectJ and AspectC++, the following weaver prototypes have been implemented and available for download:

- Apostle for Smalltalk [Apostle, 2008]
- JBoss-AOP: Java based AOP for the JBoss application server [JBoss, 2008]
- PostSharp for Microsoft .NET [PostSharp, 2008]
- AspectR for Ruby [AspectR, 2008]

There are numerous efforts that support construction of aspect-specific weavers for a specific programming language. The capabilities offered by these tools and frameworks permit new aspect languages to be developed to extend a specific base programming language. An aspect-specific framework is described in [Constantinides *et al*., 2002], which is concerned primarily with issues of concurrent programming (e.g.,

synchronization, scheduling). Associated with the goals of the concurrency framework, the concept of composing multiple aspect-specific languages is explored in [Brichau, 2002]. Related to this aspect-specific language area, the XAspects effort provides a capability for exploration of new domain-specific aspect languages (DSALs) [Shonle *et al*., 2003]. The XAspects work, however, also is limited to Java development.

In Chapter 2, we discussed the work of several researchers who have identified the benefits of language-independent aspect weaving and applicable to modern programming languages within the .Net framework. Although similar to intent to our goals, the approach described in this research is applicable toward large legacy systems that were developed prior to the existence of .Net.

Within the scientific computing domain, ROSE provides optimizations using source-to-source transformation of ASTs for C++ applications [Veldhuizen and Dennis Gannon, 1998]. The transformations are expressed using a DSL [Schordan and Quinlan, 2003]. The type of transformations performed by ROSE are focused solely on optimization issues of scientific libraries and are not applicable to the kinds of transformations we advocate in this paper to improve the modularization of crosscutting concerns in scientific code bases. Another interesting notion of crosscutting concerns that may apply within the scientific computing domain is to identify parallelism within blocks of sequential code. In [Chalabine and Kessler, 2006], Chalabine and Kessler have suggested seven different forms of interdependent concerns that are necessary to introduce parallelism within sequential programs. In addition to Blitz++ and HPL, future work will also explore the idea of improving the modularity and comprehensibility of

other scientific libraries like POOMA [Reynders *et al*., 1996] and MTL [Siek and Lumsdaine, 1998].

3.5 Limitations of Program Transformation Engines

This chapter showed the benefits of using program transformation engines (PTEs) in modularizing large software systems written in a variety of programming languages. Generally, the PTEs are used to construct language-specific aspect weavers and applied to software systems in both commercial and scientific domains. In addition to aspect weaver construction, certain specialization techniques could also be applied using PTEs to reduce the complexities of scientific libraries (e.g., HPL) that help to improve comprehensibility of such complex systems. Program transformation engines like DMS or ASF+SDF provide direct availability of scalable parsers and an underlying low-level transformation framework (e.g., term rewriting, RSL, PARLANSE and AST Interface) to modify the underlying source programs (i.e., by modifying the AST). This low-level transformation framework can form the basis of constructing aspect weavers.

However, a PTE-based weaver construction process raises new challenges and faces inherent accidental complexities (*Challenge C3*); i.e., the rewrite rules used to modify base programs are difficult to compose, which makes it accessible to only language researchers and is generally hard to comprehend by average software developers. Moreover, the rewrite rules are often tied to the grammar of the source language (e.g., Object Pascal), which impedes reusability when this language changes (*Challenge C4*). In addition, the entire weaver is rendered unusable if one switches to a new transformation engine during weaver evolution. Furthermore, to provide advanced

aspect weaving capabilities (like that of AspectJ), the underlying rewrite rules can become significantly complex. For example, to provide reflective capabilities like `thisJoinPoint` or to perform signature matching with wildcards, more complicated transformation rules are required. Such rules generally use exit functions (written in PARLANSE) to do static analysis on the underlying AST. This requires a thorough understanding of the various term rewriting semantics specific to a particular PTE. Thus, in order to use a tool like DMS to construct aspect weavers requires knowledge of the base language grammar, and of the core machinery provided by DMS. These additional challenges make program transformation systems typically accessible only to language researchers with less penetration in mainstream software development. To summarize, these limitations are listed as follows:

- The rewrite rules used to modify base programs are difficult to compose, which makes it accessible to only language researchers (accidental complexities)
- The transformation rules are tied to the grammar of a specific language (language-specific)
- The entire weaver is rendered unusable if the base transformation engine is replaced with another one (interoperability problem)
- The PTE may be proprietary, i.e., may not be available for use by all desired parties (e.g., DMS).

Therefore, although program transformation engines (as illustrated in this chapter) provide solutions to *Challenges C1* and *C2*, they fail to provide a reasonable solution to challenges like *C3* and *C4* (please refer to all the challenges *C1 - C4* in Chapter 1).

To eliminate some of the accidental complexities associated with PTEs, but still leverage the power of such systems, the next chapter provides a generic approach towards constructing aspect weavers for GPLs. Specifically, the approach uses a layered architecture and combines MDE with program transformation techniques to construct aspect weavers. The introduction of a model-driven front-end has several benefits in the overall context of providing a generic framework for aspect weaver construction, as discussed in detail in Chapter 4.

CHAPTER 4

GENERIC ASPECT WEAVER FRAMEWORK BASED ON MODEL-DRIVEN PROGRAM TRANSFORMATION

Aspect orientation has been used to improve modeling through modularization of crosscutting concerns that emerge at higher-levels of abstraction. A large body of research in this area of AOM has focused on new notations [Clarke and Baniassad, 2005] and weaving tools [Gray *et al.*, 2003] that improve the ability to express a design within a model through composition of separate concerns. In this chapter, we examine the converse – that is, how modeling can improve aspect orientation. Specifically, this chapter makes a contribution by showing how MDE in combination with PTE is used to construct new aspect weavers for GPLs through models and transformations. The approach described in the chapter uses models to capture the essence of various AOP constructs at an abstract level. These models are then mapped to concrete weavers for GPLs through a combination of higher-order model transformation and lower-level program transformation rules. A generic extension to the framework further supports reusability of artifacts among weavers during the construction process. Aspect weavers for FORTRAN and Object Pascal were constructed to evaluate the framework, and their features were assessed against several case study applications.

4.1 Role of MDE in Aspect Weaver Construction

The history of software development paradigms reveals that a new paradigm often has its genesis in programming languages and then moves up to design and analysis (e.g., structured programming preceded structured design and analysis, and object-oriented programming predated object-oriented design and modeling). This same progression can also be observed with respect to aspect orientation. Most of the early work on aspects was heavily concentrated on issues at the coding phase of the software lifecycle [Kiczales *et al*., 1997]. There were, however, initial efforts that focused on applying advanced separation of concerns in earlier phases of the software lifecycle. One of the first examples of this type of work was described by [Clarke *et al*., 1999], where a new way to carve a system into a set of elemental parts in order to support crosscutting concerns was applied at the design level. Since then, the area of AOM has grown to support several workshops [AOM, 2008; Models and Aspects, 2008], journal special issues [Object Technology, 2007; IJSEKE, 2006], and books [Clarke and Baniassad, 2005; Jacobson and Ng, 2005] on the topic. A broad range of contributions in this area have emerged, such as extensions to UML to support aspects [Ho *et al*., 2002; Stein *et al*., 2002], new notations for aspect-oriented design [France *et al*., 2004], model composition rules that define weaving semantics [Reddy *et al*., 2006], and tool support for aspect modeling [Gray *et al*., 2001; Cottenier *et al*., 2007, Lahire *et al*., 2007, Ubayashi *et al*., 2006]. Among all of the AOM contributions, a general observation is that it can be advantageous to apply aspects at levels closer to the problem space (e.g., analysis, design, and modeling), in addition to the solution space (e.g., implementation and coding).

Similar to AOM's focus on the benefits that aspects offer to modeling, we believe there are also advantages that MDE [Schmidt, 2006] can provide for aspect orientation. Specifically, in Chapter 3 of the dissertation, it was demonstrated how aspect weavers for various programming languages can be constructed using a program transformation approach. However, the use of a PTE raises new challenges that could be best realized by adopting a MDE based approach. In particular, MDE provides a capability to isolate the dependence on specific transformation engines by decoupling the source aspect language from the target PTE language and enabling the construction of aspect weavers from high-level aspect specifications and metamodels. The decoupling ensures that the source aspect metamodel does not need to be altered even if one chooses to opt for a different target PTE, only a new PTE metamodel needs to be developed. Conversely, for every new language, one needs to add the appropriate metamodel extensions to the base aspect metamodel, but no change to the target metamodel is needed. Another advantage is that both the aspect language (source) and rules language (target) can evolve independent of each other. This leads to new features being added to the weaver with minimum cost on maintenance (i.e., only new mappings are added). The next sub-section summarizes all the challenges that were raised in Chapter 1 and provides a solution by means of the GenAWeave framework.

4.1.1 Challenges and Overview of GenAWeave Framework

There were four major challenges that were identified in Chapter 1 that led toward adoption of aspects for legacy languages. They were *Challenge C1* – the parser construction problem**,** *Challenge C2* **–** the weaver construction problem**,** *Challenge C3* **–**

accidental complexities of transformation specifications and *Challenge C4* – language-independent generalization of transformation objectives.

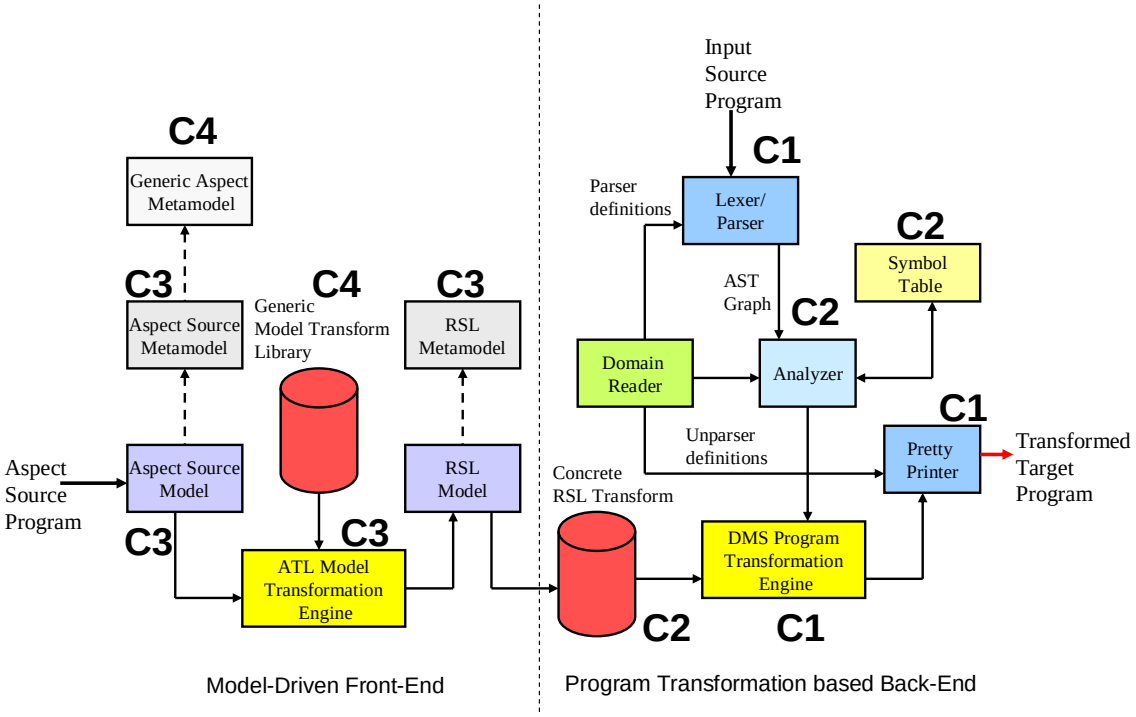


Figure 4-1 **–** Overview of our model-driven aspect weaver framework

Figure 4-1 represents the high-level architecture of the GenAWeave framework that offers solutions to these challenges. Each of these challenges (*C1-C4*) shown in the figure maps to a key process in the GenAWeave framework. For example, the adoption of DMS as a back-end transformation engine provides a solution to *Challenge C1* (*parser construction problem*) through immediate availability of industrial-scale parsers. Through transformation rules and a rich API of transformation functions, DMS also offers a partial solution to *Challenge C2 (weaver construction problem)*. However, the low-level representation of transformation rules introduces accidental complexities (*Challenge C3*) that make it difficult for programmers to specify aspects at this level. Moreover, the

rewrite rules are often tied to the grammar of the source language (e.g., Object Pascal), which impedes reusability when this language changes (*Challenge C4*).

To address these challenges, a model-driven approach is used as part of the front-end of the framework. In particular, a high-level aspect language that conforms to an aspect metamodel is used to raise the abstraction level of transformation rules. A model transformation library that translates higher-order aspect models to lower-order RSL models constitutes the heart of the framework. The generality of the framework is provided by a generic aspect metamodel that captures the commonalities of different aspect languages for various GPLs. The differences are captured using metamodel extensions. Moreover, the model transformation library provides additional generality that allows them to be reused among multiple aspect weavers. This generality is accomplished by enforcing the higher-order model transformation rules to conform to a generic interface (i.e., abstract structure). Thus, the MDE-based approach offers a solution to *Challenge C3 (accidental complexities)* and *Challenge C4 (generalization of transformation objectives)*.

4.1.2 Program Transformation Back-End

It was first observed in [Fradet and Südholt, 1998] that aspect weaving can be performed using a general transformation framework. Aßmann and Ludwig provided an early demonstration of aspect weaving using graph rewriting [Aßmann and Ludwig, 1999]. Most PTEs support a term-rewriting or graph-rewriting engine such that transformation rules can be constructed that realize the weaving of aspects into a source program [Baxter *et al.*, 2004; Cordy *et al.*, 2002; van den Brand *et al.*, 2002; Visser,

2001]. In Chapter 3, we demonstrated how a PTE can be used to construct an aspect weaver for Object Pascal [Gray and Roychoudhury, 2004] and C++ templates [Roychoudhury *et al*., 2008].

The technique from Chapter 3 uses program transformation rewrite rules to locate crosscutting concerns and weave aspects into source code. As shown in the right-hand side of Figure 4-1, the input source is initially tokenized (i.e., using a lexer) and parsed (i.e., using a parser) to produce an AST. The AST is then statically analyzed and concrete program transformation rules (i.e., RSL) are used to identify points of interest in the AST that represents a particular crosscutting concern. As mentioned earlier in Chapter 2, RSL typically consist of patterns, conditions, rules and rulesets that together perform the desired pattern matching on the source AST. Frequently, exit functions in the form of PARLANSE external functions [Baxter *et al.*, 2004; Gray and Roychoudhury, 2004] are used in conjunction with RSL to perform complex pattern matching and weaving. After a desired match is found, the AST is accordingly modified (i.e., weaved) and prettyprinted to produce the transformed target program.

4.1.3 Challenges of Program Transformation Engine Usage

In spite of source code modification capabilities, PTEs are often difficult to use and require sufficient knowledge of the underlying parsing techniques, language grammar, and proprietary languages (e.g., RSL and PARLANSE). Thus, PTEs generally tend to operate at a level that is not appropriate for general software development. Moreover, to provide advanced aspect weaving capabilities (e.g., like that of AspectJ), the underlying rewrite rules can become significantly complex. For example, to provide

reflective capabilities like `thisJoinPoint` or to perform signature matching with wildcards, complex transformation rules are required. Such rules generally use exit functions to do static analysis on the underlying AST [Gray and Roychoudhury, 2004]. This requires a thorough understanding of the various term rewriting semantics specific to a particular PTE. Moreover, the rewrite rules are often tied to the grammar of the base language (as highlighted in bold in Figure 2-4), which impedes reusability when the base language changes. Thus, using a tool like DMS to construct aspect weavers requires knowledge of the base language grammar (concrete syntax), and of the core machinery provided by DMS. These additional complexities make program transformation systems typically accessible only to language researchers and hampers PTE penetration in mainstream software development.

In our initial research in constructing an aspect weaver for Object Pascal using DMS [Gray and Roychoudhury, 2004], we observed these broader challenges and recognized that an appropriate front-end support alongside a systematic code generator was needed to bring program transformation systems closer to mainstream software development. The proper selection of an appropriate front-end and program transformation rule generator can hide the accidental complexities associated with PTEs. Nevertheless, aspect weavers can still leverage the power of PTEs to perform the complex code transformation. In the following section, we introduce our investigation into a model-driven front-end and discuss the primary benefits offered by MDE in the overall context of the framework.

4.2 Model-Driven Front-End

There are many ways to design the front-end of an aspect language. In some examples, the language format is expressed in raw XML [Lafferty and Cahill, 2003], but in other cases it is expressed in a more sophisticated declarative language [Lämmel, 1999]. Through our investigation in the design of various aspect languages, we realized that the declarative nature of expressing aspects (e.g., as popularized by pointcuts in AspectJ) has a common language-independent characteristic. For example, the concepts of join points, pointcuts and advice can be adapted to many aspect language designs within the same language paradigm. Metamodels can precisely capture these concepts and their relations.

In addition, a model-driven front-end is well-suited for abstracting the various semantics associated with PTEs. MDE provides an abstraction layer that can be mapped down to program transformation rules. Combining the technical spaces of MDE and program transformation offers more possibilities than each considered separately.

4.2.1 Metamodel for Front-End Aspect Language

Figure 4-2 shows an excerpt of the abstract syntax of an aspect language in the form of a metamodel represented as a collection of three class diagrams. This metamodel illustrates the specification of Aspect Pascal, which is an aspect language we defined for Object Pascal. An aspect described in this language consists of `Pointcuts` and `Advice`. They together constitute the fundamental elements for defining an aspect-oriented language (influenced by the asymmetric AspectJ style). As evident in Figure 4-2, an aspect can have multiple `pointcuts` and multiple `advice`.

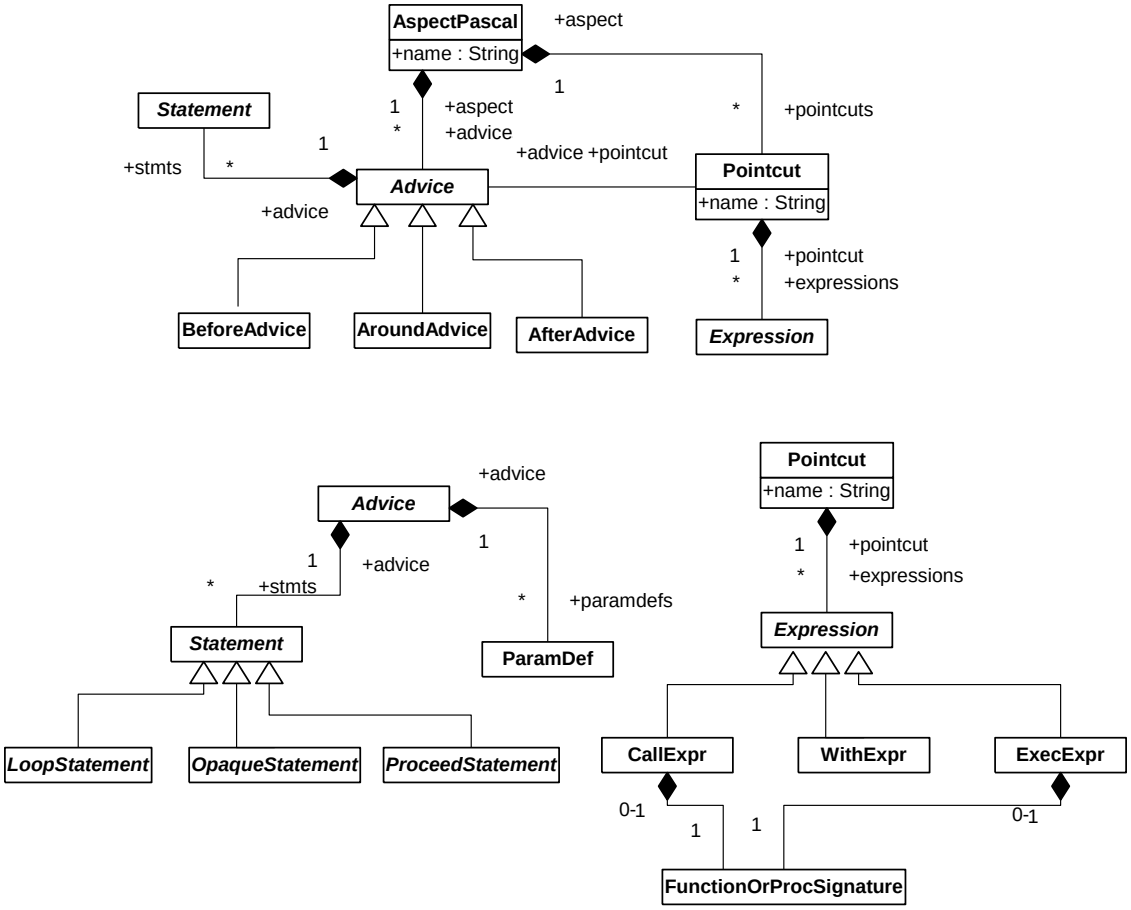


Figure 4-2 **–** Subset of Aspect Pascal metamodel represented as a class diagram

An `Advice`, defined as an abstract class in the metamodel, can be further categorized as `BeforeAdvice`, `AfterAdvice` or `AroundAdvice`. An advice can have advice parameters and an advice body (i.e., a list of statements). Every advice parameter has a type and a name associated with them and is used for passing the context information (e.g., passing the parameter type to pointcut expressions). Every advice statement conforms to the grammar of the base language. Because the back-end program transformation engine already has the parser/analyzer available for managing the base language, the body of the advice is typically delegated to the back-end for further

processing. Such late binding of an advice body reduces the complexity of the metamodel by not including every possible program construct that belongs to the base GPL. These program fragments are referenced in the front-end metamodel as `OpaqueStatements` (i.e., statements that are not handled by the front-end). In addition to `OpaqueStatement`, there are other special statements: `loop` statement and `proceed` statement. `Proceed` statement is generally used in bodies of `around` advice. An example of a loop statement is given in Section 4.5.

`Pointcuts` consist of pointcut expressions, which can, for instance, be further expressed as *call* expressions, *with* expressions or *execution* expressions. Pointcut expressions form the key for pattern matching. All pointcut expressions are derived from the abstract `Expression` class. As seen in Figure 4-2, both the `CallExpr` and `ExecExpr` expressions are derived from `Expression` and both reference the type pattern `FunctionOrProcSignature`, which identifies the prototype declaration (i.e., signature) for a function or procedure defined in Object Pascal. This is particularly useful for pattern matching. Although `call` and `exec` may be the two most common forms of pointcut expressions, new expressions can be experimented with and derived from the base `Expression` class template. For example, Object Pascal allows the definition of `with` expressions that are used to pass the context information from parent to child objects. Other pointcut expressions available in the join point model of the base language can be similarly added to the metamodel of the front-end aspect language. Wildcards are also allowed and examples are given in Section 4.5.

The pointcut expressions are translated to RSL patterns or rules that do the actual pattern matching on the source code. The front-end AOP layer is simply a façade to the

back-end program transformation engine. It helps to hide the accidental complexities associated with PTEs (*Challenge C3*) and also provides a platform to experiment with new AOP language constructs that can be suitably translated to back-end rewrite rules. The translation mechanism that generates the back-end RSL rules from the front-end aspect language is explained in detail in Section 4.3 and Section 4.4.

4.2.2 Implementing the Front-End Aspect Language with AMMA

As explained in Chapter 2, AMMA [Kurtev *et al.*, 2006] is a suite of MDE tools that can be used to implement the aspect language described in Figure 4-2. The first step in creating a front-end is to create a metamodel that defines the abstract syntax of the aspect language. The KM3 [Jouault and Bézivin, 2006] language within AMMA may be used for this purpose. Although other MDE tools can be used to define the metamodel, we chose KM3 because it has the added advantage of being independent of the concrete MDE technology (e.g., the Eclipse Modeling Framework – EMF, or OMG's Meta-Object Facility - MOF). In addition to technology independence, KM3 also provides a simple textual syntax that is well-suited for defining the metamodel described in Figure 4-2. The example in Figure 4-3 demonstrates how KM3 is used to define the Aspect Pascal metamodel.

```
class AspectPascal extends LocatedElement {
   attribute name : String;
   reference domain container : Domain;
   reference pointcuts[1-*] container : Pointcut oppositeOf aspect;
   reference advice[1-*] container : Advice oppositeOf aspect;
}
class Pointcut extends Element {
   attribute name : String;
   reference aspect : AspectPascal oppositeOf pointcut;
   reference paramdefs[*] container : ParameterDef;
   reference exprs[1-*] container : Expression oppositeOf pointcut;
}
abstract class Advice extends LocatedElement {
   reference aspect : AspectPascal oppositeOf advice;
   reference pointcut : Pointcut;
   reference paramdefs[*] container : ParameterDef;
   reference stmts[1-*] container : Statement;
}
```

Figure 4-3 - KM3 specification (snippet) for Aspect Pascal

Figure 4-3 shows a snippet of the KM3 code used to implement the metamodel specification introduced in Figure 4-2. The `AspectPascal` class contains references to other classes; namely, the core elements `Pointcut` and `Advice`. The `oppositeOf` construct is used to maintain reverse navigational links for efficient traversal purposes required during model transformation (Section 4.3). Thus, instead of representing the model as a tree, the `oppositeOf` reference helps to maintain the metamodel as a graph that can be traversed in the opposite direction, if necessary. Generally, `advice` and `pointcuts` can be traversed in both directions (i.e., from the parent `AspectPascal` class to the child `Pointcut` or `Advice` class, and vice versa). Similarly, pointcut `exprs` can be traversed in the reverse direction. The complete KM3 specification for the Aspect Pascal metamodel is available in Appendix A.

In addition to the abstract syntax shown as a metamodel in KM3, the concrete syntax of the aspect language is specified in a separate model. To express this model,

AMMA offers TCS [Jouault *et al*., 2006], which uses a grammar-like notation to describe the syntax of a language. Figure 4-4 shows an excerpt of the Aspect Pascal concrete syntax defined in TCS. This figure illustrates how the concrete syntax of different metamodel elements (e.g., *Aspects, Pointcuts,* and *Advice*) is expressed in TCS. In TCS, every class represented in the KM3 specification has its corresponding template definition. It also introduces other terminal tokens like separators, brackets and semicolons that are required to describe the concrete syntax of the aspect language but are not captured in the abstract syntax of the metamodel. Thus, TCS gives the structure of the source aspect language. In addition, context information can also be passed and stored in the symbol table for further analysis. The complete TCS specification for the Aspect Pascal metamodel is shown in Appendix A.

```
Template AspectPascal main
      :  "aspect" name "{" pointcut advice "}"
      ;
template Pointcut context addToContext
      :  "pointcut" name "(" paramdefs{separator = ","} ")"
         ":" exprs {separator = "&&"} ";"
      ;
template Advice abstract;

template BeforeAdvice
      :  "before" "(" paramdefs {separator = ","} ")" ":"
         ...
      ;
template AfterAdvice
      :  "after"  "(" paramdefs {separator = ","} ")" ":"
          ...
      ;
```

Figure 4-4 - TCS specification (snippet) for Aspect Pascal

The front-end would be incomplete without appropriate code generators that transform the front-end aspect language to its corresponding target language. In our

model-driven GenAWeave framework, the back-end is the transformation language of the PTE; specifically, the RSL. The following section demonstrates how RSL transformation rules are generated from the front-end aspect specification.

4.3 Model Transformation

This section describes the model transformations rules that appropriately translate a given aspect specification to its corresponding rule specification. Both the source (high-level aspect language) and target (low-level RSL) languages are defined using a metamodel based approach. The translated RSL rules are subsequently processed by DMS to perform the actual weaving for a source program written in a specified GPL. The following sub-sections discuss in detail the model transformation approach and begin with an introduction of the program transformation rule generator.

4.3.1 Program Transformation Rule Generator

The program transformation rule generator (shown as item 2 in Figure1-2 and also shown as Generic Model Transform Library in Figure 4-1) represents the core of the framework and embodies a higher-order transformation (i.e., a model transformation rule is used to generate program transformation rules). As mentioned earlier, the front-end aspect language is only a façade to the back-end PTE and all pointcut declarations and advice code present in the source aspect language are eventually translated to target RSL code that consists of RSL patterns, external conditions and rewrite rules. Therefore, the goal of the program transformation rule generator is to synthesize transformation engine

specific weaving code (RSL) from the front-end representation defined by a higher-order aspect specification.

4.3.2 Target Metamodel for RSL

In order to realize a systematic translation from a high-level aspect language to a low-level transformation language, it is necessary to define a metamodel for the back-end program transformation engine. The target RSL metamodel serves two basic purposes. Firstly, it allows experimenting with new aspect languages (e.g., Aspect Ruby or Aspect FORTRAN) and new aspect constructs (e.g., loops) without changing the model for the back-end PTE. In this case, the commonalities of different aspect languages for various GPLs can be captured in a generic aspect metamodel. The differences can be captured using metamodel extensions; *however, no change is required for the target metamodel*. This helps to improve the generality of the framework.

Secondly, instead of an ad hoc technique, a metamodel allows more sophisticated translations where complex pointcut expressions and join point shadows (areas in the source where join points may emerge) from the front-end aspect language could be correspondingly mapped to patterns and rules in the back-end RSL language. The presence of a target metamodel provides an internal representation of the back-end transformation language (RSL) that can be used to validate the generated lower-order transforms. For future experimental purposes, the presence of a RSL metamodel may also permit bidirectional mappings (currently, the mapping is unidirectional, from Aspect-to-RSL). In such a scenario, given a generated RSL program as input, the corresponding

aspect specification for a different GPL may be recovered, provided a mapping exists between RSL and the GPL.

To capture the essential concepts of RSL, an RSL metamodel has been created in KM3, illustrated by a class diagram in Figure 4-5. As noted earlier, RSL consists of elements like patterns, rules, conditions, and rule-sets, which are captured in this metamodel. The complete KM3 and TCS specification for the RSL metamodel is available in Appendix C. It should be noted that the target metamodel defines the essence (i.e., concepts and relations) of a domain without concern for semantics. In our case, the semantics of the various components of the source aspect metamodel are captured in the mapping to RSL defined as an ATL transformation. ATL is the model transformation language of AMMA [Jouault and Kurtev, 2005]. The semantics of the aspect language is thus captured in terms of the semantics of RSL, which is in turn processed by DMS. Case studies are presented in Section 4.5, where complete scenarios describing this model to model transformation are explained with concrete examples.

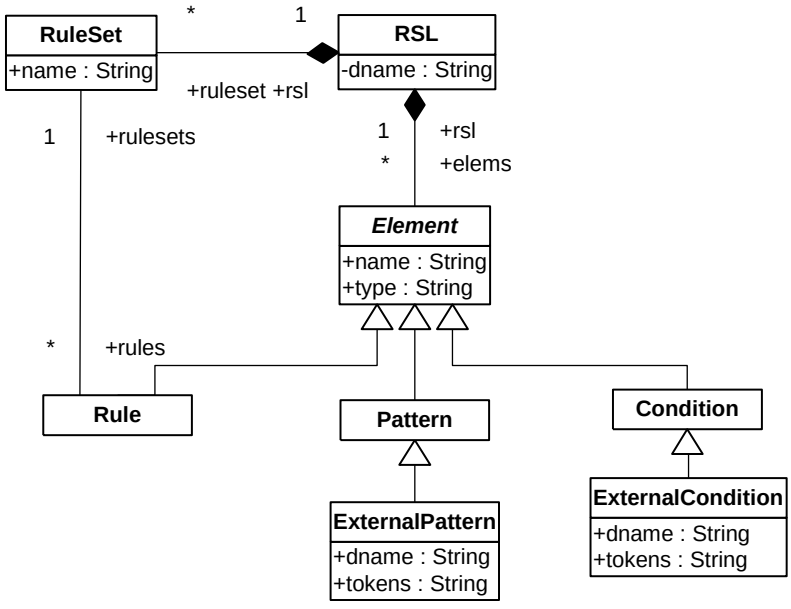


Figure 4-5 - Subset of the RSL metamodel (as a class diagram)

The following section describes the model transformation technique that uses ATL as the core artifact to transform the source aspect model to the target RSL model.

4.3.3 Model Transformation using ATL

Given the definition of the source and target metamodels, it is possible to generate RSL program transformation rules from an aspect program using model transformations.

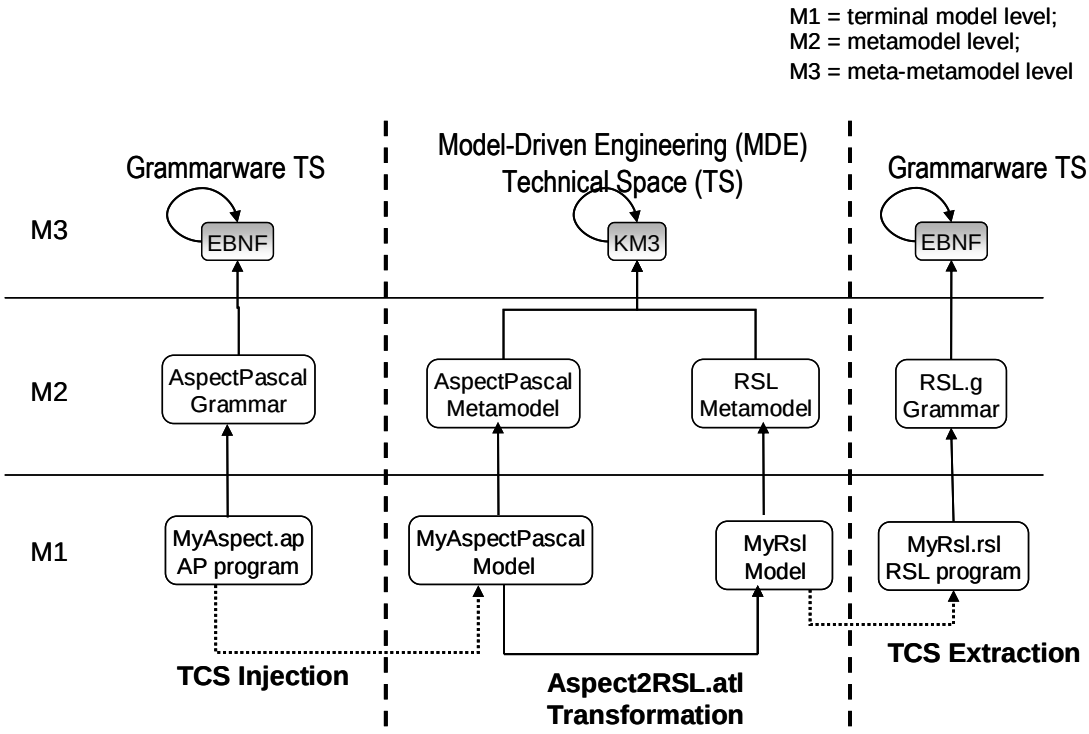


Figure 4-6 **-** Model transformation scenario for generating RSL rules from aspects

Figure 4-6 explains the complete model transformation scenario in the GenAWeave framework. In this figure, M1, M2, and M3 are the three modeling levels in the Grammarware [Klint *et al.*, 2005] and MDE technical spaces (TS). From the Grammarware TS, the front-end aspect source file is initially injected into a source aspect model using TCS. The aspect model is then transformed into a target RSL model using a

model transformation defined in ATL. This ATL transformation forms the core of the program transformation rule generation process. After translation, the generated RSL model belonging to the MDE TS is extracted (using TCS) into the target RSL program in the Grammarware TS.

To modularize the RSL generation process, the framework defines a library of ATL transformations with each transformation corresponding to a primitive pointcut specification (e.g., `call`, `execution`). For a given aspect, the corresponding ATL transformation rule is automatically invoked depending on the pointcut specification used in the aspect. The higher-order ATL transformation generates the lower-order RSL transformation that eventually performs the aspect weaving. The collective set of all model transformation rules is assembled in a transformation library that implements the semantics of the source aspect language.

Figure 4-7 depicts a snippet of a sample ATL transformation from the core transformation library. This particular transformation evaluates a `call` expression in the source aspect, and generates the corresponding RSL transformation rule. The ATL helper function `EvalCallExpr` is used for this purpose. The transformation maps individual elements from the source aspect metamodel to the target RSL metamodel. For example, Aspect Pascal model elements like advice (Line 10, Figure 4-7) and pointcuts (Line 11, Figure 4-7) are mapped to RSL elements like patterns, conditions and rules (i.e., RSL elements in Figure 4-5). Similarly, *before* advice statements (Line 25, Figure 4-7) from the source aspect language are mapped to RSL patterns. The relationships between the source aspect model elements to the target RSL model elements can be one-to-one, one-

to-many, many-to-one or many-to-many. This depends on the type of pointcut expression used in the source aspect program.

```
1. module AspectPascal2RSL;
2. create OUT : RSL from IN : APascal;
3. rule APascal2RSL {
4.     from
5.         s : APascal!APascal
6.     to
7.         t : RSL!RSL (
8.             dname <- 'ObjectPascal',
9.             elems <- Sequence {
10.                         s.advice,
11.                         s.pointcut->collect(e |
12.                            thisModule.EvalCallExpr(e)
13.                         ),
14.                        ...
15.             },
16.             ruleset <- rs
17.         ),
18.       rs : RSL!RuleSet (
19.             name <- s.name,
20.             rules <- s.pointcut->collect(e|e.name)
21.         )
22. }
23. rule BeforeAdvice2Pattern {
24.     from
25.         s : APascal!BeforeAdvice
26.     to
27.         t : RSL!Pattern (
28.             name  <- 'before_advice_stmt_list'
29.             ptype <- 'statement_list',
30.             ptext <-  spt
31.         ),
32.       spt : RSL!SimplePatternText (
33.             ptext <- s.stmts->iterate(...)
34.         )
35. }
-- [original code omitted for brevity]
```

Figure 4-7 - ATL transformation (snippet) from Aspect Pascal to RSL

It should be noted that the source aspect metamodel to describe these pointcut expressions is completely independent of the target RSL language. Additionally, the aspect metamodel is structurally and semantically similar to a traditional AOP language,

like AspectJ. This metamodel captures many of the essential concepts of AOP (influenced by the asymmetric AspectJ style) - join points, pointcuts and advice. The actual transformation on the source code is performed using RSL rules that are generated from the higher-order aspect language using ATL. These ATL transformations implement the semantics of the source aspect language and all corresponding mapping information from source to target are embedded in the ATL specifications.

The generated RSL is not shown here because it is internal to the framework (i.e., users of the framework do not see any of the intermediate transformation rules); however, interested readers who want to view the generated artifacts may refer to the GenAWeave website [GenAWeave, 2008], which represents the project webpage for the framework and includes video demonstrations, papers, and all of the source. In addition to the website reference, the experimental case studies presented in Section 4.5 also serve as specific examples for describing the complete transformation scenario illustrated in this section.

4.3.4 Remaining Challenges to be Addressed by the Framework

The model-driven weaver generation framework presented in this section offers a solution to the challenge of using a program transformation engine to implement an aspect weaver. The previous sections provided a discussion of the key parts of the framework, including the front-end aspect language, the transformation rule generator and the back-end weaving engine. The context of the discussion was centered on the creation of a weaver for a single base language, such as Object Pascal, to address *Challenge C3 (accidental complexities)*. However, an additional challenge remains. As

mentioned in the beginning of this chapter, a program written in RSL or any other term-rewriting engine is typically tied to the grammar of the source program (i.e., the RSL examples presented in Chapter 3 have grammar productions appearing throughout the transformation rule). Moreover, there are variations in design from one aspect language to another, even if a common generic part is shared. Unless carefully designed, the front-end, the core transformation libraries, and the back-end modules are rendered unusable when constructing a new weaver in another context (i.e., a new aspect language for a new base programming language). The goal of any extensible framework is to avoid constructing a single fixed-point solution (i.e., constructing each new weaver from scratch) after enough knowledge, time, and effort have been spent. The next section discusses how this framework was made more generic to support reuse in new contexts. Thus, instead of building a new weaver from scratch, the benefit from the experience gained in a previous construction can be reused and applied toward the construction of a new weaver for a different programming language.

4.4 Extending to a Generic Framework

Generalizing the framework presented in Section 4.3 to accommodate a broad range of GPLs is challenging due to the dissimilarities among various programming languages. Yet, many languages in the same paradigm (e.g., structured or object-oriented) may share common concepts at an abstract level such that parts of the framework can be reused. Unfortunately, most aspect weavers are built from scratch with little emphasis on reusing the existing knowledge or framework already available for constructing a weaver for a particular GPL.

In Section 2.2.1 of Chapter 2, several techniques toward language-independent legacy modernization were discussed. Section 2.2.2 presented a comparative study about the strengths and weaknesses of these techniques. From this comparative study, we realized that although the prevailing .Net/CLI based techniques serves well for modern programming languages, they fail to address legacy languages like FORTRAN, COBOL and Object Pascal due to their non-conformance with .Net specification. Moreover, such an approach would ignore all available artifacts that are already available for these languages. While investigating a generic aspect weaving framework, we understood these challenges and discovered a solution whereby the model-driven weaver framework uses the existing parsers of DMS, but extracts out the commonalities among weavers constructed for various GPLs. Although our approach does not automate all the tasks involved in creating an aspect weaver (i.e., making it language-independent), GenAWeave can considerably reduce the weaver construction effort by reusing the shared or common parts among different aspect weavers through abstract models and corresponding model transformations.

Moreover, because DMS provides support for 23 different programming languages (including legacy languages like COBOL, FORTRAN, and C), a generic front-end with a reusable code generator that translates our front-end aspect language to RSL can make use of all the parsers and analyzers that are already available within each of the language domains supported by DMS. In addition, we may also consider changing the back-end if another PTE supports other languages that we would like to use. The solution approach introduced in this section addresses the obstacles toward weaver construction enumerated in *Challenge C4 (generalization of transformation objectives)*.

4.4.1 Support for a Generic Aspect Front-End

The first step toward a generalized model-driven weaver framework is to design a generic aspect front-end that can be shared among various GPLs. If the AspectJ definition of an aspect is used, every language that is integrated into the framework must define the meaning of a join point model (JPM), pointcuts, and advice within the language context. Such a notion can be defined abstractly such that each new aspect language inherits and extends this common definition. As such, a full generic source model is not needed when the notion of join points is considered to occur within limited boundaries of a program (e.g., method execution, method invocation, memory allocation), which is a concept shared by most languages. An abstract join point model for the model-driven weaver framework only considers a small subset of concepts shared across most languages in a common paradigm.

Reconsidering the Aspect Pascal metamodel of Figure 4-2, it can be observed that metamodel elements such as pointcut, advice, abstract expressions, and abstract statements are actually generic in the Aspect Pascal metamodel. Thus, instead of modeling these elements as part of the Aspect Pascal metamodel, they can be extracted to a common generic core. However, there may be differences in the concrete syntax of certain model elements. For example, concrete statements and expressions may vary from one GPL to another. In such cases, the differences can be captured in individual metamodel extensions [Barbero *et al*., 2007] and commonality can be shared using a general metamodel. To explain this concept, this section will summarize the construction of aspect weavers for two different GPLs (i.e., Object Pascal, and FORTRAN) using the

GenAWeave framework. The example shows how languages across different paradigms can even share AOP concepts through metamodel extension.

Figure 4-8 shows the class diagram representing the new Aspect Pascal metamodel that extends from the core GAspect metamodel. The latter captures all of the essential concepts that are intrinsic to most aspect-oriented languages (influenced by the asymmetric AspectJ style). For example, the core model elements such as pointcuts and advice belong to GAspect. There are also abstract placeholders for expressions and statements in GAspect. Although the figure does not show a metamodel for JPM, a further enhancement in this direction could be made in the future.

Every language-specific expression and statement must extend from these abstract definitions. For example, a concrete `execution` expression join point or a `call` expression join point for any Aspect-Oriented Language (AOL) must be derived from the abstract `expression` join point of GAspect. In Figure 4-8, the `CallExpr` expression and `ExecExpr` expression of Aspect Pascal inherits from `FuncOrProcDefExpr` (which itself is derived from the abstract `Expression` class) and references the `FuncOrProcSignature` type pattern. The type pattern captures the concrete syntax (i.e., signature) for expressing functions or procedures in Object Pascal and is dependent on the grammar of the base language. For every new language, the concrete syntax of the type pattern varies. The dotted rectangle in Figure 4-8 depicts all those points of variability that are specific to Aspect Pascal.

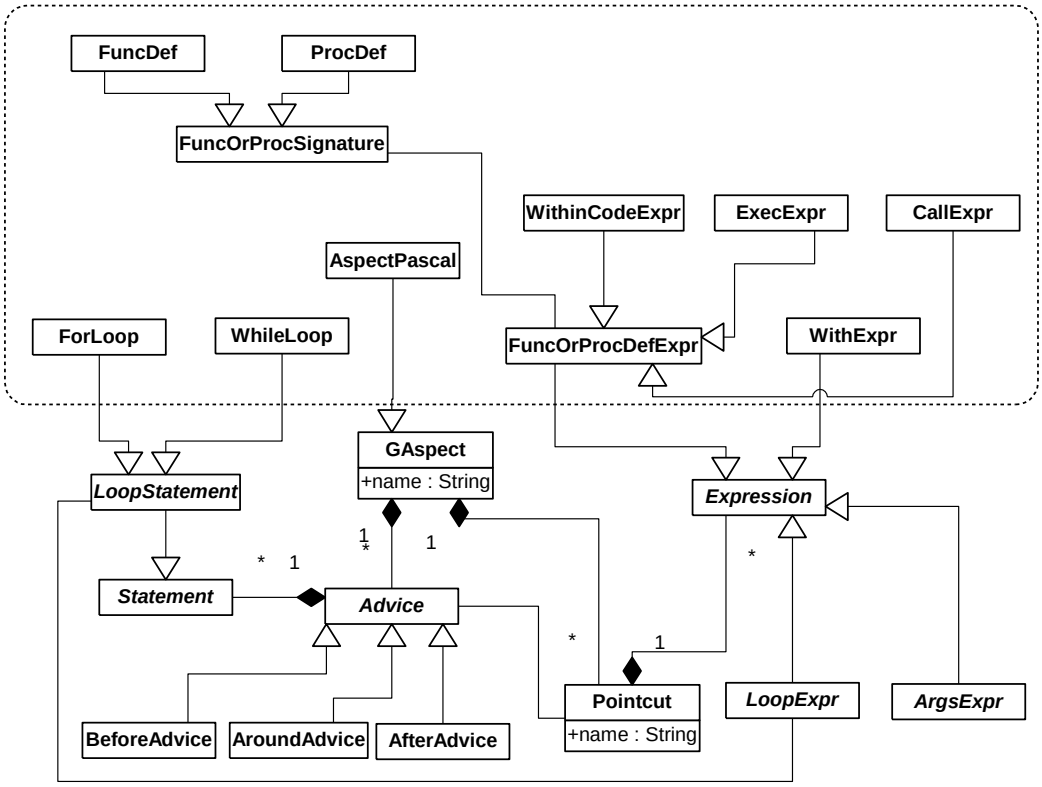


Figure 4-8 - Class Diagram (snippet) of Aspect Pascal extending from a common Generic Aspect metamodel

Because most programming languages have some form of support for loops, we have introduced the notion of a `loop execution join point` in the generic metamodel. Concrete loop statements belonging to the base AOL must be derived from the abstract `LoopStatement` of GAspect. The Aspect Pascal metamodel shows support for `while` loop and `for` loop join points that are extended from the abstract `loop execution join point` present in GAspect. The concept of a `loop execution join point` is not present in AspectJ, but has been found to be useful for monitoring high-performance scientific applications [Harbulot and Gurd, 2005].

Furthermore, a join point for capturing `with` expressions in Object Pascal is introduced in the Aspect Pascal metamodel. An example of a crosscutting concern based on a `with` expression join point is given in [Gray and Roychoudhury, 2004]. In a similar way, the entire join point model for an aspect-oriented language can be constructed by adding concrete extensions from the abstract GAspect metamodel. Moreover, the technique allows experimentation with new features (e.g., `loop execution join point`) to be added to an existing AOL. Such an addition is beneficial if the aspect language should evolve. The Aspect Pascal metamodel shown here is only a snippet of the original. The complete KM3 and TCS specification of Aspect Pascal is available in Appendix A.

Figure 4-9 shows the corresponding metamodel for Aspect FORTRAN. Similar to Aspect Pascal, the Aspect FORTRAN metamodel is extended from the generic core GAspect. However, the points of variability (shown by the enclosed dotted rectangle) for this metamodel exist in their concrete syntax. In the case of Aspect FORTRAN, the `call`, `exec` and `withincode` expressions reference subroutine/function definitions unlike the procedure/function definitions in the Aspect Pascal metamodel. Moreover, the concrete function definitions for Aspect FORTRAN and Aspect Pascal are different due to the dissimilarity in their underlying grammar. The TCS specification in Figure 4-10 shows this variability of concrete syntax for the two metamodels.

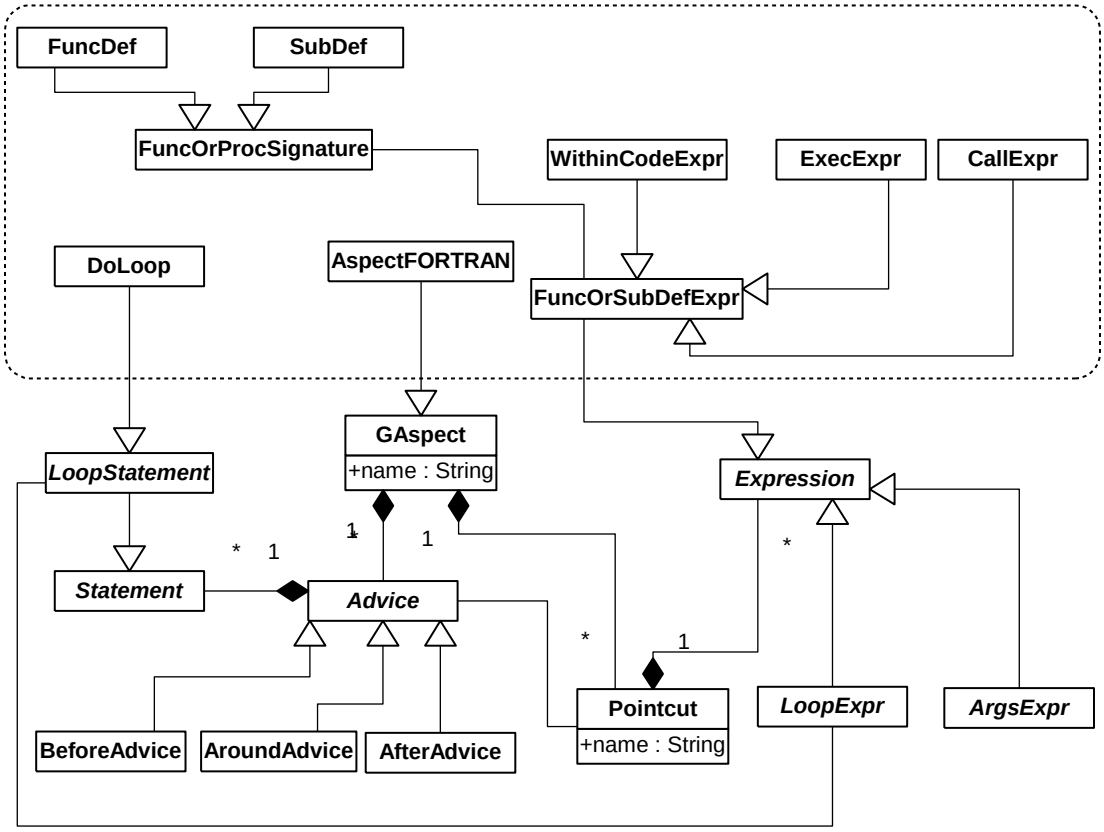


Figure 4-9 - Metamodel (snippet) of Aspect FORTRAN conforming to a common Generic Aspect metamodel

```
1. template FuncDef
2. :  "FUNCTION" name "(" paramdefs{separator = ","} ")"
3. ;
4. template SubDef
5. :  "SUBROUTINE" name "(" paramdefs{separator = ","} ")"
6. ;
```

```
1. template FuncDef
2. :  "function" (isDefined(classifier) ? classifier ".")
3.     name "(" paramdefs{separator = ";"} ")"
4. ;
5. template ProcDef
6. :  "procedure" (isDefined(classifier) ? classifier ".")
7.     name "(" paramdefs{separator = ";"} ")"
8. ;
```

Figure 4-10 - TCS specification showing differences in concrete syntax for Aspect FORTRAN (top) and Aspect Pascal metamodel (bottom)

The top-half of Figure 4-10 shows the concrete syntax of function/subroutine definitions for Aspect FORTRAN. The bottom-half shows the corresponding concrete syntax for Aspect Pascal. All points of variation between the two metamodels are captured in their corresponding extended metamodel (dotted rectangle), but the commonality is captured in the generic aspect metamodel. The complete KM3 and TCS specification of Aspect FORTRAN is available in Appendix B.

In addition, GAspect also captures certain program fragments belonging to a GPL that may not be analyzed or parsed by the front-end. Instead, these program fragments are delegated to the back-end PTE for parsing and analysis. Such fragments typically appear in the body of advice code and are referenced as `OpaqueStatement`s. This considerably reduces the complexity of the aspect metamodel as several language constructs of the base language need not be parsed or analyzed by the front-end. Instead, the back-end PTE that already has the capability (parser/analyzer) to process the base language (Object Pascal / FORTRAN) can handle such program fragments. An example of using `OpaqueStatement` is shown in the experimental case study of Section 4.5.

The construction of a generic aspect metamodel helps to generalize the commonalities among distinct aspect languages. Each common concept may be refined using language-specific metamodel extensions. Furthermore, an extension of GAspect may categorize commonalities within a paradigm that can be reused (e.g., a metamodel named Object-Oriented that extends GAspect with common OO concepts, which is then extended by concrete OO languages). This was one of the important lessons learned during the course of this research and can significantly improve the genericity of the metamodel.

4.4.2 Generalizing the Rule Generator Design

The goal of the program transformation rule generator is to translate a given aspect to a corresponding program transformation rule (e.g., RSL). This role is handled by an assembly of transformation libraries written in ATL. In the context of a generic framework, it is desirable to reuse as much of the transformation library code as possible when constructing an aspect weaver for a new GPL. To realize this objective, the transformation libraries must follow a general guideline (similar to a generic API) that ensures maximum reusability.

The guideline ensures that every transformation rule that captures the semantics of a particular weaving intent must conform to a generic interface. For example, an RSL rule that captures the semantics of a method invocation join point (i.e., to trap a particular method call and trigger advice) should conform to a generic method invocation interface that the back-end transformation engine expects. By conforming to this generic interface, model transformation libraries written for various GPLs share a generalized common pattern. For example, a method call join point for any GPL should conform to a generic method call interface named `generic_advice_call`, which accepts the following five named-parameters: `program_root_`, `method_id_`, `proceed_call_`, `before_advice_` and `after_advice_`. The parameter types to `generic_advice_call` function are determined by the concrete syntax (grammar) of the base GPL. For example, for a FORTRAN 90 program, this generic function should be encoded as: {`name` → *`type`*}

```
generic_advice_call  (
   { program_root_  → Fortran90_program },
   { method_id_     → Name},
   { proceed_       → Name },
   { before_advice_ → execution_part_construct_list },
   { after_advice_  → execution_part_construct_list },
) → Fortran90_program
```

One may note that although the types (shown in *italics*) are concrete, the interface is abstract. This generalization is necessary to address *Challenge C4* and facilitate the ATL rule generator to program to a common interface that can be reused among various GPLs. At this point, one may recollect from Chapter 3 how RSLs or any term-rewrite rules are tied to the concrete syntax of the base programming language. The `proceed_` is internally used to determine if the advice is an around advice that makes a call to `proceed`. Similarly, for an Object Pascal program, the `generic_advice_call` is encoded as follows:

```
generic_advice_call  (
   { program_root_  →  ObjectPascal },
   { method_id_     →  IDENTIFIER},
   { proceed_       →  IDENTIFIER },
   { before_advice_ →  statement_list },
   { after_advice_  →  statement_list },
) → ObjectPascal
```

For every join point in the AOP language model, a set of formal interfaces to which each corresponding ATL transformation must conform have been developed (i.e., there is a separate generic interface for `method execution` or `loop execution join point`). The generic interfaces not only enforce the code generators for different aspect weavers to adhere to a known abstract interface, but also considerably reduces the development time and effort to transfer knowledge from one rule generator to another (please see Section 4.5.4 for evidence that supports this claim).

Figure 4-11 and Figure 4-12 show comparative snippets of the model transformation rules (ATL specifications) for translating a method call join point written in Aspect Pascal or Aspect FORTRAN to a corresponding program transformation rule (RSL rewrite specification). Each of the ATL specifications (Figures 4-11 and 4-12) consist of several smaller ATL rules that together perform the actual transformation. For example, the rules `AfterAdvice2Pattern`, `BeforeAdvice2Pattern` and `PointCutToExternalPattern` (as shown in Figures 4-11 and 4-12) are used to construct the ATL specification for translating *method call join point*. However, this is only a subset; the complete ATL specification is available at Appendix D. The individual rules (e.g., `AfterAdvice2Pattern`, `BeforeAdvice2Pattern`) are fired whenever a corresponding model element (e.g., model elements like `BeforeAdvice`, `AfterAdvice` in the Aspect Pascal metamodel) in the source metamodel is reached.

Both of these higher-order model transformation rules conform to an abstract structure (generic interface) that drives the ATL rule generator. As a direct benefit of forcing the ATL transformations to conform to a common structure or interface, the model transformation rules presented in Figures 4-11 and 4-12 appear distinctly similar. For example, all of the three rules (i.e., `AfterAdvice2Pattern`, `BeforeAdvice2Pattern` and `PointCutToExternalPattern`) have the same Left-Hand Side (LHS), where as the main difference lies in their concrete syntax (i.e., the grammar of the two languages).

```
rule BeforeAdvice2Pattern {
	from
		s : APascal!BeforeAdvice
	to
		t : RSL!Pattern (
			phead  <- ph,
			ptoken <- 'statement_list',
			ptext  <- spt
		),
		ph : RSL!PatternHead (
			name <- 'before_advice_stmt'
		),
		...
}
rule AfterAdvice2Pattern {
	from
		s : APascal!AfterAdvice
	to
		t : RSL!Pattern (
			phead  <- ph,
			ptoken <- 'statement_list',
			ptext  <- spt
		),
		ph : RSL!PatternHead (
			name <- 'after_advice_stmt'
		),
		...
}
lazy rule PointCutToExternalPattern {
	from
		s : APascal!Pointcut
	to
		t : RSL!ExternalPattern (
			dname  <- 'ObjectPascal',
			eptext <- 'around_advice_call',
			ptoken <- 'ObjectPascal',
			phead  <-  ph
		),
		ph : RSL!PatternHead (
			name   <- 'around_advice_call',
			params <-  Sequence {pp1,pp2,pp3,pp4,pp5,pp6}
		),
		pp1 : RSL!PatternParameter (
			name    <- 'program',
			referTo <- 'ObjectPascal'
		),
		pp2 : RSL!PatternParameter (
			name    <- 'method_name',
			referTo <- 'IDENTIFIER'
		),
		pp3 : RSL!PatternParameter (
			name    <- 'proceed_call',
			referTo <- 'IDENTIFIER'
		),
		...
}
```

Figure 4-11 - ATL specification used to generate lower-order transformation rules (RSL) for weaving Object Pascal source program

```
rule BeforeAdvice2Pattern {
	from
		s : AFortran!BeforeAdvice
	to
		t : RSL!Pattern (
			phead  <- ph,
			ptoken <- 'execution_part_construct_list',
			ptext  <- spt
		),
		ph : RSL!PatternHead (
			name <- 'before_advice_stmt'
		),
		...
}
rule AfterAdvice2Pattern {
	from
		s : AFortran!AfterAdvice
	to
		t : RSL!Pattern (
			phead  <- ph,
			ptoken <- 'execution_part_construct_list',
			ptext  <- spt
		),
		ph : RSL!PatternHead (
			name <- 'after_advice_stmt'
		),
		...
}
lazy rule PointCutToExternalPattern {
	from
		s : AFortran!Pointcut
	to
		t : RSL!ExternalPattern (
			dname  <- 'FORTRAN',
			eptext <- 'around_advice_call',
			ptoken <- 'Fortran90_program',
			phead  <- ph
		),
		ph : RSL!PatternHead (
			name   <-   'around_advice_call',
			params <- Sequence {pp1,pp2,pp3,pp4,pp5,pp6}
		),
		pp1 : RSL!PatternParameter (
			name    <- 'program',
			referTo <- 'Fortran90_program'
		),
		pp2 : RSL!PatternParameter (
			name    <- 'method_name',
			referTo <- 'NAME'
		),
		pp3 : RSL!PatternParameter (
			name    <- 'proceed_call',
			referTo <- 'NAME'
		),
		…
}
```

Figure 4-12 - ATL specification used to generate lower-order transformation rules (RSL) for weaving FORTRAN source program

To further understand the mapping, in the ATL rule `BeforeAdvice2Pattern`, the `before advice` in the source aspect metamodel is mapped to a RSL `pattern` in the target RSL metamodel that consists of a pattern head (`phead`), a pattern token (`ptoken`) and the pattern text (`ptext`). Similarly, a RSL `external pattern` is translated from a source `pointcut` specification and has the same LHS signature (`dname`, `eptext`, `ptoken`, `phead`) for both Object Pascal and FORTRAN generators. The main difference lies in the concrete syntax (Right-Hand Side) of the base language grammar as referred in the transformation rules where an `execution_part_construct_list` in FORTRAN is mapped as a `statement_list` in Object Pascal. Obviously, there are other non-terminal and terminal tokens in both the Object Pascal and FORTRAN grammar that have similar structural representation and meaning but differ by name in their BNF form. The strategy is always to follow a common abstract structure (or substructure) to translate a particular join point from an aspect description to RSL. However, in certain cases, where the difference in signature or concrete syntax between two language grammars differs significantly, it may not be directly possible to map to a generic interface. Instead, the mapping can then conform to sub-structures or sub-interfaces.

Although the current construction technique requires the transformation library to be recreated manually by updating the concrete syntax, a possible extension could be to apply this mapping information automatically (e.g., by using model weaving [Jossic *et. al.*, 2007]) to generate part of this library. From our own experience in constructing aspect weavers using the generalized framework, we realized that a large part of the

generic front-end and program transformation rule generator could be reused across languages with little customization (please see Section 4.5.4).

In the following section we present some interesting case studies that use the model-driven aspect weaving framework to construct aspect weavers for two different GPLs. In particular, we construct aspect weavers for Object Pascal and FORTRAN and make comparative studies of reuse of their front-end, the rule generator and the back-end. The observations made in the case studies help to validate the techniques presented so far. They also reveal some of the limitations of the technique and the lessons learned during the process that can be applied for future improvements.

4.4.3 Support for a Reusable Back-End

In addition to the generic front-end support along with a reusable model transformation library, GenAWeave provides support for a partially reusable back-end. The back-end consists of various external (helper) functions that are called by the generated program transformation rules (RSL). These external functions are generally used for traversing the AST and locating join points in the AST to apply transformations. The external functions are written in PARLANSE and internally use several APIs present in the `DMSRuleSpecificationLanguage` and `DMSStringGrammar` domain [Baxter *et al.*, 2004]. Most of the pattern matching and transformation on the underlying AST is achieved by a combination of RSL and PARLANSE helper functions. Therefore, an important task in constructing a weaver for a particular GPL is to provide support for these helper functions that are internally called by the generated RSL.

To support back-end construction, GenAWeave provides a reusable library of external functions that can be used to construct part of the low-level weaving infrastructure for a given weaver. Figure 4-13 provides an example of a reusable external function written in PARLANSE, which can be used by any weaver within the GenAWeave framework. This particular function (`name_ends_with`) is useful for matching identifiers (e.g., function name) whose name ends with a given input. This is equivalent to a wildcard search (e.g., `*name`) in an aspect program.

```
(define name_ends_with
      (lambda Registry:MatchingCondition
           (let (;; (= [search_string (reference string)]
                     (Graph:HGHandling:GetString arguments:1))
                      [sub_string string]
                      [search_string_size natural]
                      [search_id_size natural]
                      [start_index natural]
                );;
                (value
                   (;;
                         (= search_string_size
                            (size (@(AST:GetString arguments:1))))
                         (= search_id_size (size
                            (@(AST:GetString arguments:2))))
                         (= start_index
                            (- search_string_size search_id_size))
                         (= sub_string (Strings:Substring
                            (AST:GetString arguments:1)
                         (+ start_index 1) search_id_size))

                         (ifthen(== sub_string
                         (@(AST:GetString arguments:2)))
                           (return ~t)
                         )ifthen

                      (return ~f)
                      );;
                      ~f
                )value
           )let
      )lambda
 )define
```

Figure 4-13 – Reusable external function in the GenAWeave framework

In Figure 4-13, `arguments:2` indicates the location where the search is to be conducted, while `arguments:1` is the given input to be matched. In addition to the example shown in Figure 4-13, GenAWeave provides other similar functions used for wildcard matching for identifiers (e.g., `name_begins_with`, `name_contains`). Appendix E provides more examples of reusable helper routines that are available in the GenAWeave framework. However, there are other external functions that are not completely reusable and depend on the grammar of the base language. Nevertheless, these routines, which implement a particular join point, generally conform to identical pattern matching algorithms (e.g., a `function call join point`, `a loop execution join point` in languages `A` and `B`). Examples of external functions for `loop execution join point` for Object Pascal and FORTRAN are given in Appendix E.

4.5 Experimental Evaluation – Object Pascal and FORTRAN Weavers

In order to experiment with the approach presented in the previous sections, two aspect weavers were constructed – one for Object Pascal and another for FORTRAN using the GenAWeave framework. A subset (e.g., primitive pointcuts like `call`, `execution`, `loop`, `withincode`, and `args`) of standard AOP features was built into both weavers in an AspectJ-like style. The FORTRAN weaver was constructed after the completion of the Aspect Pascal weaver and reused several functionalities, code and knowledge from the previous construction without much alteration to the core artifacts. For example, both weavers shared the generic front-end, which constituted around 50% of the overall front-end LOC (written in KM3 and TCS). Moreover, the FORTRAN

weaver reused 60% of the Object Pascal rule generator code without any alteration, and an additional 25% with minor customization. A detailed discussion of experimental results is given in Section 4.5.4. Most of the time and effort on building the FORTRAN weaver was spent on understanding the concrete syntax of the language and on the conceptual design of the weaver. The rest of the section is devoted to evaluating the basic functionalities of these weavers through sample case study applications.

4.5.1 Object Pascal Weaver

The initial experimentation towards evaluating our Aspect Pascal weaver was realized within the scope of a commercial distributed application written in Object Pascal. The case study application and all the examples discussed here were first introduced in [Gray and Roychoudhury, 2004] and also presented in Chapter 3. One specific application used for evaluation was a utility that assisted in upgrading a database after a schema change. The first example presented in this section is concerned with updating a processing dialog meter within the schema evolution tool. The second example relates to synchronization between various database error handlers. These Object Pascal examples were earlier introduced in Chapter 3. However, the technique presented there directly used program transformation rules instead of a high-level aspect language based on a model-driven approach.

Processing Dialog Meter: Figure 3-1 in Chapter 3 described the crosscutting concern that was present in a progress dialog meter in the database schema evolution utility. Figure 4-14 shows the `UpdateProgressMeter` aspect that encapsulates the crosscutting concern in a separate module. The pointcut `IncrCall_` captures all calls to

procedure `Inc`. The advice code shown between Lines 5-11 is triggered once this "procedure call join point" is reached. It may be noted that the entire "if statement" (Lines 5-11 defined internally as an `OpaqueStatement`) is not parsed by the front-end but delegated to the back-end parser. The aspect conforms to the Aspect Pascal metamodel and any syntactic errors are reported back to the user.

```
1. aspect UpdateProgressMeter {
2.   pointcut IncrCall_() : call(procedure  *.Inc(Integer));
3.   after() : IncrCall_()
4.   {
5.      if not ProcDlg1.Process(TotalInsertionsPerformed /
6.            TotalInsertionsCalculated) then
7.       begin
8.         ProcDlg1.Canceled := True;
9.         Result := True;
10.        exit;
11.      end;
12.  }
13. }
```

Figure 4-14 - Aspect to capture progress meter updating

Following TCS injection on the aspect from Figure 4-14, the corresponding Aspect Pascal model is generated (shown in Figure 4-15). The model (represented in XML format) conforms to the APascal and GAspect metamodels introduced in Section 4.4. After applying the ATL transformation (a `method call join pint`) on this Aspect Pascal model, the resulting RSL model is generated that conforms to the target RSL metamodel. Finally, the lower-order RSL transformation rule is extracted from the RSL model using TCS extraction. The resultant RSL model and the RSL transformation rules are available in [GenAWeave, 2008]. Note that the complete transformation scenario was initially introduced in Section 4.3 (also refer to Figure 4-6) and is fully automated using Another Neat Tool (ANT) scripts (http://ant.apache.org/). More details

about Ant scripts and integration of the framework within Eclipse are given in Section 4.6.

```
<APascal xmlns="APascal" xmlns:_1="GAspect" name="UpdateProgressMeter">
  <domain name="ObjectPascal"/>
  <pointcut name=" IncrCall_">
    <pctexpr xsi:type="CallExpr">
      <funcOrProcSig xsi:type="ProcedureDef" name="Inc" classifier="*">
        <paramdefs name="*" type="Integer"/>
      </funcOrProcSig>
    </pctexpr>
  </pointcut>
  <advice xsi:type="_1:BeforeAdvice" pctname="//@pointcut.0">
    <advStmt xsi:type="_1:OpaqueStatement" stmt="..."/>
  </advice>
</APascal>
```

Figure 4-15 **-** Aspect Pascal model generated from Aspect Pascal source program

The next example in our case study shows how a synchronization aspect is captured using the Aspect Pascal weaver constructed from our model-driven framework.

Database Error Handler Synchronization: Synchronization and thread safety issues are often considered as a major concern in software development because they are difficult to modularize with traditional object-oriented techniques. Such concerns end up scattered across many modules and tangled with other concerns of the system. An example of a concurrency concern was presented in Figure 3-5.

Figure 4-16 shows the aspect to support a synchronization concern. The pointcut `funcHandler_` captures execution of all database handler functions. Synchronization is realized by an around advice that wraps calls to the `LockHandle` and `UnlockHandle` methods inside a `try/finally` block. The `proceed` statement allows the database error handling code to execute normally within the synchronization

aspect. We applied the same steps as in the previous example to separate this concern from the main code base.

```
1. aspect SyncDBErrHandler {
2.    pointcut funcHandler_() :
                execution(function *.Handle(..));
3.    void around() : funcHandler_()
4.    {
5.       TExHandleColl(Collection).LockHandle;
6.       try
7.          proceed ();
8.          finally
9.            TExHandleColl(Collection).UnLockHandle;
10.      end;
11.   }
12. }
```

Figure 4-16 **-** Aspect to capture synchronization in a database error handler

The Aspect Pascal model shown in Figure 4-17 is obtained by applying TCS injection on the aspect from Figure 4-16. The complete ATL transformation used to translate this aspect (`method execution join point`) along with all other associated artifacts is available at the GenAWeave project website [GenAWeave, 2008]. It should be noted that it is this lower-order RSL code that does the actual weaving on the base program, but the general user of this framework is oblivious to its presence. Instead, the front-end aspect language acts as a façade to the back-end PTE and hides all the accidental complexities associated with it (*Challenge C3*).

The XML representation shown in Figure 4-17 is only an internal representation of the Aspect Pascal model and is generally used for analyzing and transforming the aspect specification. A software developer does not see this internal representation. However, the information is useful for more advanced users who want to construct aspect weavers for different GPLs using the technique described in this chapter.

```
<APascal xmlns="APascal" xmlns:_1="GAspect" name="SyncDBErrHandler">
  <domain name="ObjectPascal"/>
  <pointcut name="funcHandler_">
    <pctexpr xsi:type="ExecExpr">
      <funcOrProcSig xsi:type="FunctionDef" name="Handle"
        classifier="*">
        <paramdefs name="*" type="*"/>
      </funcOrProcSig>
    </pctexpr>
  </pointcut>
  <advice xsi:type="_1:AroundAdvice" pctname="//@pointcut.0">
    <advStmt xsi:type="_1:OpaqueStatement" stmt="..."/>
    <advStmt xsi:type="_1:TryCatchFinallyStatement">
      <stmts xsi:type="_1:ProccedStatement"/>
      <finallyStmts xsi:type="_1:OpaqueStatement" stmt="..."/>
    </advStmt>
  </advice>
</APascal>
```

Figure 4-17 – Generated Aspect Pascal model from Aspect Pascal source program

4.5.2 FORTRAN Weaver

Although most of the AOP research is centered around Java, we believe several numerical and scientific computing applications that are written in legacy languages like FORTRAN can benefit from AOP. We constructed a FORTRAN weaver using the generic model-driven framework and were able to reuse a majority of the code generator libraries that were previously written for Object Pascal. The front-end of the FORTRAN weaver is based on the same generic aspect metamodel that was used by the Object Pascal weaver. We evaluated our weaver within the scope of several FORTRAN programs that internally used the Message Passing Interface (MPI) [Gropp *et al*., 1996]. The first example shows how a security concern can be weaved into such programs and the second example illustrates how to monitor and weave an aspect around loops.

Security Aspect: MPI is a library specification for message-passing and is largely used in high-performance scientific computing applications [Gropp *et al*., 1996]. MPI provides more than 125 core functions that include all the basic functionalities to assist in

writing parallel programs. There are several implementations of MPI written in various languages (e.g., C, FORTRAN, C++ and Java). In order to provide security to FORTRAN-based MPI programs, it is often required to encrypt/decrypt messages while they are sent or received across the network. Figure 4-18 shows a snippet of a FORTRAN MPI program, in which lines 9 and 12 illustrate how a security concern (i.e., a call to the `encrypt` function) is added before each call to `MPI_SEND`. The implementation of the security concern is scattered over the entire code base for all messages that require encryption during `MPI_SEND`.

```
1.    program send_recv_with_MPI
2.    ...! original code
3.    real :: a_msg
4.    real :: b_msg
5.    ...! original code
6.    allocate (a_msg(msg_len))
7.    allocate (b_msg(msg_len))
8.    ...
9.    call encrypt(a_msg)
10.   call MPI_SEND(a_msg,...)
11.   ...
12.   call encrypt(b_msg)
13.   call MPI_SEND(b_msg,...)
14.   ...
15.   deallocate (a_msg)
16.   deallocate (b_msg)
17.   ...! original code
18.   end
```

Figure 4-18 - Encryption of messages during MPI_SEND

Figure 4-19 shows the aspect program required to enable security for all messages during MPI message send and receive. The pointcut captures all calls to `MPI_SEND` and passes the message to be encrypted as an argument. In a similar way, security to messages may be enabled during calls to `MPI_RECV`. The internal representation of the generated artifacts (e.g., Aspect FORTRAN model, RSL model and RSL transformation

rule) is not shown here but the transformation process is similar to previous descriptions and available on the GenAWeave project website [GenAWeave, 2008].

```
1. aspect enable_encryption {
2.    pointcut mpi_send_(real :: orig_msg) :
3.         call(MPI_SEND(real,*)) && args(orig_msg);
4.    before(real :: orig_msg): mpi_send_(orig_msg)
5.    {
6.         call encrypt(orig_msg )
7.    }
8. }
```

Figure 4-19 - Aspect to enable encryption during MPI calls

4.5.3 Join Point for Loops

It is often desired to monitor the performance of loops for some high-performance scientific applications. Harbulot *et al.* first introduced this concept in an extension to AspectJ [Harbulot and Gurd, 2005]. We borrowed from their definition and added this feature into our FORTRAN and Object Pascal weavers. According to our definition, the join point for a loop has the following signature:

```
<loop_name>(init::<val>, exit::<val>, stride::<val>)
```

`Init` specifies the loop initialization value, `exit` specifies the loop termination value and `stride` specifies the loop increment counter.

Figure 4-20 shows an implementation of `MPI_GATHER` written in FORTRAN. In MPI, messages can be forwarded by intermediate nodes where they are split (for scatter) or concatenated (for gather). Often it is required to measure timing statistics around critical parts of program execution. One such case is shown in Figure 4-20. Lines 9-14 shows the execution of the outer `do` loop which has initial value as 1, exit value as 10 and a default stride as 1. In a manual approach, it is required to invasively add the

timer (Lines 8 and 16) and change the source program at every place whenever the program runs into the execution of a loop join point that matches the given loop condition.

```
1.    program gather_vector
2.    ... ! original code
3.    parameter(niters=10)
4.    parameter(xmax=100,ymax=100)
5.    parameter (totelem=xmax*ymax)
6.    ... ! original code omitted
7.    ! start timer
8.    time_begin = MPI_Wtime()
9.    do iter = 1,niters
10.      ...
11.      do i=1,totelem
12.        ...
13.      enddo
14.   enddo
15.   ! stop timer
16.   time_end = MPI_Wtime()
17.   ... ! original code omitted
18.   end
```

Figure 4-20 - Adding timer around `do` loops

Figure 4-21 shows the aspect program that can automatically add the timing functions during the execution of the loop join point. The join point for loops matches any loop expression in the base program that satisfies the loop initialization value, the loop finalization value (exit) and the loop stride value. Note that the variables defined in the advice code (Lines 6-8) are generally not validated by the front-end and are delegated to the PTE for semantic validation. The wildcard '*' may be interpreted as 'any'. Currently, both integer and string value types are supported, but future extensions can support other value types. However, as a side effect, the behavior of a base program may be altered if there are logical errors (e.g., `init=1`, `exit=1`, `stride=2`) in the loop expression and there is a corresponding match. Such a situation may be avoided in the

future by adding semantic validations to the existing pattern matching functionality. In addition, future requirements may alter the semantics of the loop join point by including the variable names in the pointcut specification.

```
1.   aspect AddTimerAroundLoops
2.   {
3.      pointcut loop_timer_() :
           execution(do(init::1,exit::10,stride:*));
4.      void around(): loop_timer_()
5.      {
6.         time_begin = MPI_Wtime()
7.         proceed()
8.         time_end = MPI_Wtime()
9.      }
10.  }
```

Figure 4-21 - Aspect to add timer around `do` loops

```
<AFortran xmlns="AFortran" xmlns:_1="GAspect"
name="AddTimerAroundLoops">
 <domain name="FORTRAN"/>
 <pointcut name="loop_timer_">
  <pctexpr xsi:type="_1:LoopExpr">
   <loopStmt xsi:type="DoLoop">
      <loopInitCond xsi:type="1:IntLoopInitCond" condition="1"/>
      <loopExitCond xsi:type="1:IntLoopExitCond" condition="10"/>
      <loopStrideCond xsi:type="1:StringLoopStrideCond"
       condition="*"/>
   </loopStmt>
  </pctexpr>
 </pointcut>
 <advice xsi:type="_1:AroundAdvice" pctname="//@pointcut.0">
  <advStmt xsi:type="_1:OpaqueStatement" stmt="time_begin =
    MPI_Wtime()"/>
  <advStmt xsi:type="_1:ProceedStatement">
  <advStmt xsi:type="_1:OpaqueStatement" stmt="time_end =
    MPI_Wtime()"/>
  </advice>
</AFortran>
```

Figure 4-22 - Aspect FORTRAN model generated from source aspect program

The Aspect FORTRAN model (XML format) corresponding to the aspect program of Figure 4-21 is shown in Figure 4-22. The model conforms to the Aspect

FORTRAN metamodel shown in Figure 4-9. The corresponding ATL transformation for `loop execution join point` is available in Appendix D. The generated RSL code can be found at the project website [GenAWeave, 2008].

4.5.4 Discussion of Experimental Results

In terms of reusability, all the examples listed in Section 4.5 reuse the same generic aspect metamodel (GAspect). Moreover, the ATL transformation for translating a particular join point reveals non-trivial reuse among weavers constructed for different GPLs. This was illustrated in Section 4.4.2 through Figures 4-11 and 4-12 (i.e., an ATL transformation for translating a `method call join point` in FORTRAN and Object Pascal). In that particular example, 230 lines of model transformation code (out of 280 LOC) were reused without any modification. The remaining 50 LOC were reused with minor customization.

Similarly, for translating a `loop execution join point` in FORTRAN and Object Pascal, 265 LOC out of 305 were reused without any modification, while the remaining 40 LOC were reused with minor customization. Examples of an ATL rule for translating a `loop execution join point` for Object Pascal is shown in Appendix D.

A visual comparison between ATL rules (`loop execution join point`) for Object Pascal and FORTRAN weavers is shown in Figure 4-23, which suggests the level of reuse among the two ATL rules. This level of reuse is a direct benefit of using the GenAWeave framework, which enforces the model transformation rules to conform to a common abstract structure. The difference between the rules is due to the

dissimilarity in the grammar of Object Pascal and FORTRAN (highlighted in Figure 4-23). A comparative analysis between other ATL rules for the Aspect Pascal and Aspect FORTRAN weaver is available at the GenAWeave website [GenAWeave, 2008].

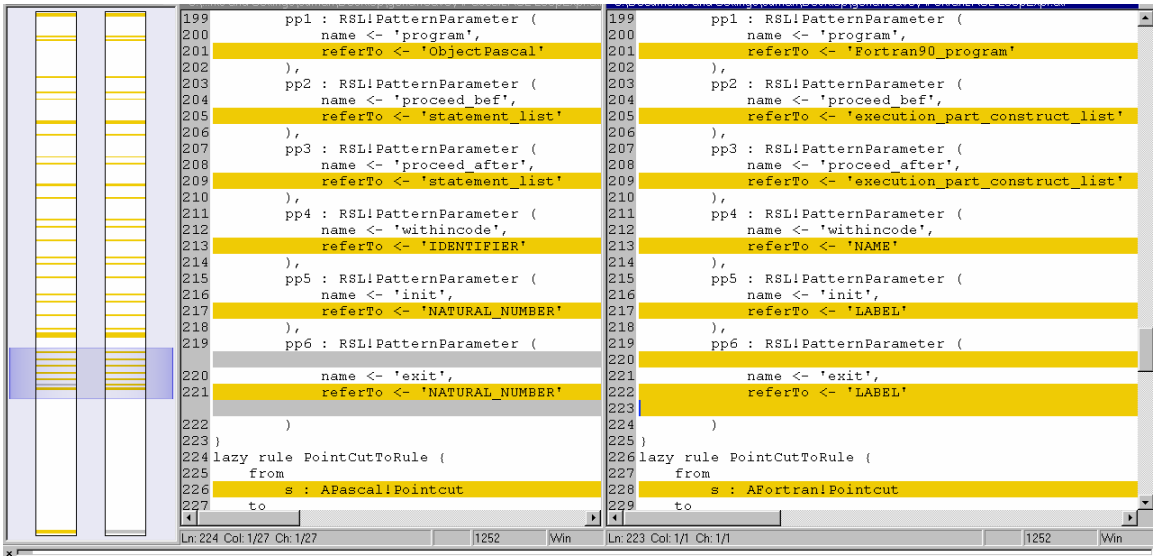


Figure 4-23 – A comparative analysis of model transformation rules

Likewise, the front-end of all weavers share a generic metamodel (i.e., GAspect). Out of 550 LOC used for defining the front-end metamodel (KM3 and TCS specifications), nearly 280 LOC were shared among the two weavers. However, it should be noted that the current weavers have limited functionalities and the reuse may decrease with mutually exclusive functionalities (e.g., *with* join point is present only in Object Pascal and not in FORTRAN). Nevertheless, the purpose of the Aspect Pascal and Aspect FORTRAN weavers were to experimentally evaluate the generality of our model-driven framework for building aspect weavers. The main objective was to evaluate the reusability of features that can be shared among multiple weavers without writing them from scratch. In the current stage of our investigation, we have adopted a simple join

point model (a subset of AspectJ) with primitive pointcuts like `call`, `execution`, `loop`, `withincode`, `with`, `within` and `args` and advice declarations like `before`, `after` and `around`. It was observed that the Aspect FORTRAN weaver that was constructed after the completion of the Aspect Pascal weaver reused a majority of the available front-end artifacts (e.g., generic metamodel and ATL specifications).

In addition to front-end reuse, GenAWeave provides a reusable library of back-end external functions (please see Section 4.4.3) that can be used to provide low-level transformation support for every aspect weaver. Currently, there are 11 such functions that are shared by the Object Pascal and FORTRAN weavers. A few of these shared functions are shown in Appendix E.1. However, not all external functions are reusable or shared, especially, the ones that are dependent on the syntax of the base language. In such cases, the functions adopted by multiple weavers generally use identical algorithms and conform to a common abstract structure (please see Appendix E.2 for such an example).

Figure 4-24 shows the reusability summary for the FORTRAN and Object Pascal weavers. It can be observed that the front-end reusability is considerably larger than the back-end reusability, overall nearly 55-65% of the artifacts are reused. Moreover, it should be noted that the two languages (i.e., Object Pascal and FORTRAN) are distinctively dissimilar in syntax and belong to two different paradigms (i.e., object-oriented and procedural). It is expected that the reusability will increase among languages that belong to the same paradigm (e.g., object-oriented).

FRONT-END REUSABILITY				
METAMODEL	KM3+TCS (LOC)	Shared LOC		Percentage
Aspect Pascal	565	280		**49.5**
Aspect FORTRAN	550			**50.1**
MODEL TRANSFORMATION	ATL (LOC)	Shared LOC		Percentage
Aspect Pascal	1890	1290		**68.2**
Aspect FORTRAN	1585			**81.3**
BACK-END REUSABILITY				
PARLANSE FUNCTIONS	Total LOC	Shared LOC	No. of Shared Functions	Percentage
Aspect Pascal	873	310	11	**35.5**
Aspect FORTRAN	775			**40**
OVERALL REUSABILITY				
OVERALL	LOC	Shared LOC		Percentage
Aspect Pascal	3328	1880		**56.4**
Aspect FORTRAN	2910			**64.6**

Figure 4-24 – Reusability summary for FORTRAN and Object Pascal weavers

Although more advanced pointcuts like control flow (`cflow`) and reflection (`thisJoinPoint`) were omitted from the current investigation due to limited static/control flow analysis in DMS for Object Pascal and FORTRAN, future research aims to introduce them at a later stage. Currently, DMS provides more mature analysis engines for languages like C++ and Java. As part of possible future extensions, it is planned to experiment with such advanced pointcut mechanisms (`cflow`, `reflection`, `loops`) for these two languages.

4.6 Integrating the GenAWeave Framework within Eclipse

The GenAWeave framework is integrated within the Eclipse IDE as shown in Figure 4-25. The individual aspect weavers could be run using Ant scripts available in the project website [GenAWeave, 2008]. The scripts take the input source file and the aspect

program as input and through a chain of transformation processes (model transformation followed by program transformation) produces the transformed target program. Both the ATL model transformation engine and DMS PTE are invoked within the Eclipse IDE.

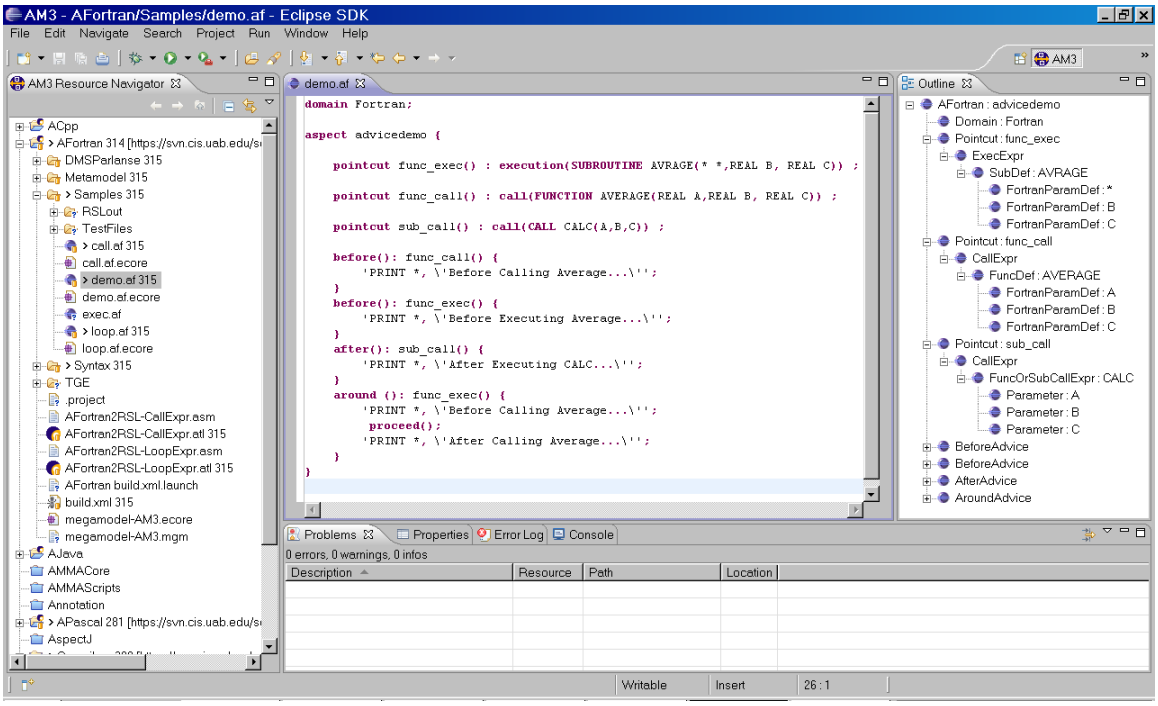


Figure 4-25 – GenAWeave framework within Eclipse

Moreover, there is a separate editor available for specifying aspects for source GPLs. The editor provides an outline view for the individual aspect programs, and any syntax errors present within the program can be displayed to the user (e.g., in Figure 4-26, the editor indicates the misspelling of the pointcut name `timer_around_loops`). In addition, the aspect code can be presented in different colors and fonts using the standard syntax highlighting feature present within the editor. Figure 4-26 shows how the outline view, syntax highlighting and syntax errors are displayed within the editor for a given aspect program.

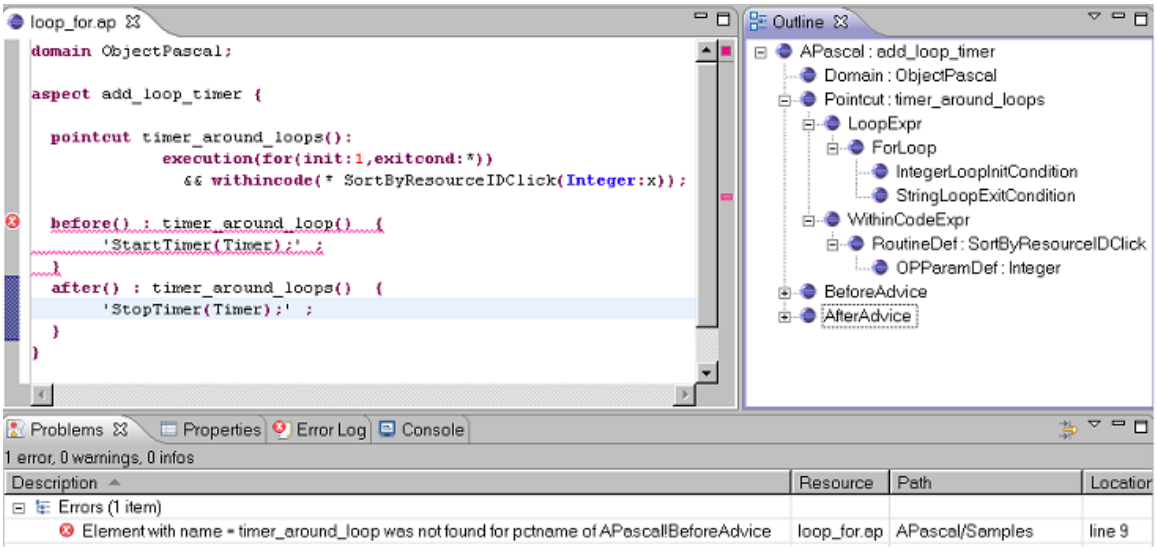


Figure 4-26 – Syntax errors displayed within the editor

4.7 Related Work

Section 2.2.2 presented a comparative discussion of language-independent modernization tools to support AOP. This section provides other related work towards generic AOP adoption and corresponding tool support. In a recent paper, Heidenreich *et al*. showed a generic approach for implementing aspect orientation for arbitrary languages using invasive software composition [Heidenreich *et al*., 2007]. However, their technique is more useful for declarative DSLs than for GPLs .

Morin *et al*. presented a generic aspect-oriented modeling framework to represent aspects that can be adapted to any modeling domain [Morin *et al*., 2007]. Although our work tends to capture the generality of aspect languages and not individual aspects, nevertheless, it can gain interesting insights from such an approach.

A recent addition to the class of language extension tools is MetaBorg [Bravenboer and Visser, 2004], which provides an ability to embed domain-specific languages into general purpose languages. However, the embedding permitted by

MetaBorg is focused on localized adaptations, and cannot accommodate the global effects of aspects. MetaBorg also requires specific transformation rules to be written for each GPL.

An initiative to develop an Abstract Syntax Tree Metamodel (ASTM) for GPLs has been proposed in OMG's Architecture-Driven Modernization program [OMG ADM, 2008]. The work described in this research may be benefited from this initiative although the primary focus of this dissertation is based on General-Purpose Aspect Languages (GPALs).

In [Roover *et al*., 2007], advanced pattern detection techniques are suggested by applying a logic-based query language that uses concrete source code templates to match against a combination of structural and behavioral program representations, including call-graphs, points-to analysis results and abstract syntax trees. This is similar to the rule specification language available in DMS that is used for pattern matching. RSL also provides external patterns and conditions that make calls to external functions written in PARLANSE for more advanced program analysis and transformation.

Ramos et al. proposed a framework for expressing patterns as *model-snippets* and showed how pattern matching can be performed with model-snippets for any given metamodel [Ramos *et al*., 2007]. In our current framework, all pattern matching and analysis is done through the back-end where the metamodel is used to express the front-end aspect language and its generic extensions. Eventually, all of the higher-order aspect specifications are translated to lower-order back-end program transformation code that does the actual weaving.

As an alternative approach to model-to-model (M2M) transformation followed by TCS extraction, an interesting technique that can be used is model-to-text (M2T) transformation [Eclipse M2T, 2007]. In the M2T approach, models of particular software solutions are refined and transformed into software source code (e.g. Java, C++). Such transformations generally make use of "templates." A template may be generally described as a text sequence interspersed with commands that extract information from a model. The Jet or Acceleo template language can be used for such a purpose [Eclipse M2T, 2007]. We recognize that this is an interesting solution and could serve as an alternative approach towards constructing the RSL rule generator. However, using M2T, we may loose a precise concept mapping between the source and the target model, and rely on mapping concepts to strings. Nevertheless, any alternative approach can still benefit from the technique described in this dissertation.

The goal of this research has been to reuse most of the software tools and artifacts (e.g., existing parsers and analysis engines) that are already available for a variety of legacy and modern programming languages. The science and theory to construct such tools are already well-established and it would require considerable engineering effort to build them from scratch without gaining any additional scientific knowledge. On the other hand, new language-independent techniques like .Net CLI / CodeDOM are not always feasible to support various legacy languages like FORTRAN, COBOL and Object Pascal due to their non-conformance to .Net specification. Unless those languages are forced to comply with a language-independent CLI specification, new experimentation to impart AOP features to them is virtually impossible.

The research presented in this dissertation provides an initial solution to such challenges by reusing most of the existing software artifacts (e.g., lexers, parsers, analyzers, evaluators) that are already available for a variety of GPLs. Thus, it enables new experimentation with advanced software engineering principles like AOP for existing legacy languages. The research also addresses new challenges that arise from the usage of complex PTEs like DMS by providing a suitable front-end that hides most of the accidental complexities that are generally associated with them.

CHAPTER 5

FUTURE WORK

This chapter outlines research directions that will be investigated as part of future work. In particular, future research goals are based on three specific directions:

1. Improve the generality of the framework.
2. Apply and evaluate the framework towards construction of other aspect weavers for legacy and modern programming languages.
3. Investigate other software engineering techniques like generalized refactoring and generic aspect mining based on the knowledge gained in developing a generic framework for aspect weaving. Both of these techniques may help to improve the quality of a software system and can benefit from a model-driven program transformation based approach presented in this dissertation.

Future enhancements towards improving the generality of the framework are listed below:

- To improve the reusability of the generic aspect metamodel, research into the idea of metamodel inheritance will be explored. This will assist tool developers to group commonalities among various GPLs into individual metamodels that could be inherited or extended by language-specific metamodels.

- To improve the reusability of the ATL rule generator, the idea of rule inheritance will be explored. Using rule inheritance some of the common features among model transformation rules can be extracted. In addition, a weaving model using AMW [Jossic *et. al.*, 2007] is proposed to automate the construction of the ATL rule generator. This could partially remove the manual construction effort of the ATL rules from one GPL to another.

To evaluate the framework further, the following steps are proposed:

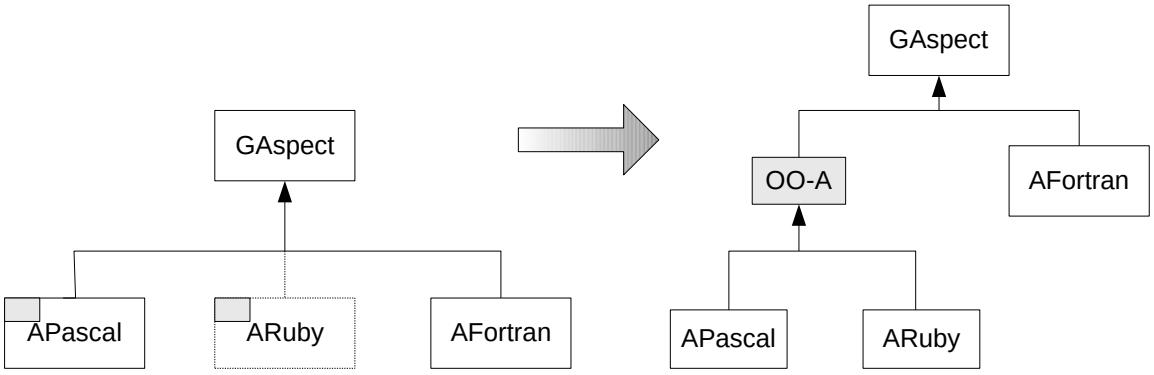
- Constructing aspect weavers for other legacy and modern GPLs.
- Applying the approach to domain-specific aspect languages (DSALs).
- Applying the approach towards high-performance scientific computing applications, especially towards specialization of scientific libraries.

5.1 Improving Reusability of the Generic Aspect Metamodel

Future research goals aim to improve the reusability of features among aspect weavers by further enhancement to the existing design of the generic aspect weaving framework. For example, it is desired to create a generic metamodel for Object-Oriented constructs, from which the weavers constructed for Object-Oriented languages can inherit.

The current generic metamodel (i.e., GAspect) generalizes what is common between APascal and AFortran (i.e., abbreviation of Aspect Pascal and Aspect FORTRAN, the aspect languages for Object Pascal and FORTRAN, respectively). Figure 5-1a shows this current design. In the future, if the construction of ARuby (i.e., an aspect language for Ruby) is considered using the GenAWeave framework, this new language

could directly extend from GAspect as shown in Figure 5-1a. However, it is expected that both ARuby and APascal will have some commonality (e.g., related to the object paradigm) not shared with AFortran. Figure 5-1b shows the improved metamodel design. The commonalities between APascal and ARuby are extracted into OO-A (the common object-oriented constructs of ARuby and APascal), which extends GAspect. In [Steel and Jézéquel, 2007], a proposal for typing models as a collection of interconnected objects is discussed. The formalism described there is an extension to object-oriented typing, however suitable to a model-oriented context. The proposed approach of defining an abstract metamodel and its conformance with another metamodel via metamodel extensions is similar to the concept described in [Steel and Jézéquel, 2007]. Similarly, it is desired to create a generic metamodel for JPMs. All weavers can inherit from the generic JPM, and (if required) add new join point extensions to their specific JPM.



a – Current metamodel design ***b*** – Improved metamodel design

Figure 5-1 – Improving the front-end metamodel design

5.2 Improving Reusability of ATL Rule Generator

The current design of the ATL rule generator uses generic interfaces to enforce a common structure among ATL rules from one GPL to another. However, using rule

inheritance the design could be further enhanced, as demonstrated in [Mernik *et al.*, 2002; Jouault and Kurtev, 2005]. An example of rule inheritance is shown in Figure 5-2. In the example, ATL `rule A` is defined as abstract while `rule B` is extended from `A`. Similarly, `rule C` is inherited from `rule B`. Thus, rule inheritance will allow certain rules that have similar functionality (e.g., an ATL rule for translating a loop join point for Object Pascal and FORTRAN) to be defined as generic (i.e., abstract) and concrete rules belonging to concrete weavers could extend from it. This will improve the reusability of model transformation rules used in this framework.

```
1.    abstract rule A {
2.       from [fromA]
3.       using [usingA]
4.       to [toA]
5.       do [doA]
6.    }
7.
8.    rule B extends A {
9.        from [fromB]
10.       using [usingB]
11.       to [toB]
12.       do [doB]
13.    }
14.
15.    rule C extends B {
16.       from [fromC]
17.       using [usingC]
18.       to [toC]
19.       do [doC]
20.    }
```

Figure 5-2 – An example showing ATL rule inheritance

In addition to rule inheritance, a proposal to automate the ATL rule construction effort will also be explored. Currently, although a large percentage of the model transformation rules conform to a specific structure and can be significantly reusable for a particular join point, one has to still customize them partly for those parts that depend

on the concrete syntax of the source language. For example, among the ATL rules presented in Figures 4-11 and 4-12 in Chapter 4, part of the difference is because of the dissimilarity in certain grammar symbols for Object Pascal and FORTRAN (e.g., `execution_part_construct_list`, `statement_list`, `IDENTIFIER`, `NAME`). In the future, it is proposed to capture these variable grammar parts in a separate weaving model that conforms to a weaving metamodel in AMW [Jossic *et. al.*, 2007]. This will remove any manual customization of ATL rules and help to further automate the rule construction process. It may be noted that AMW is a MDE-based tool for establishing relationships (i.e., links) between models. Specific examples of use cases using AMW are available at [AMW Use Case, 2008].

5.3 Constructing Weavers for other GPLs

Another possible extension of our work is to construct aspect weavers for other GPLs including object-oriented scripting languages like Ruby and Python. In addition, future directions in this area intend to experiment with adding new types of join points (e.g., loops) to existing GPLs like Java and C++. Advanced pointcuts like `cflow` and certain reflective techniques like `thisJoinPoint` would also be taken up as part of future work. This could serve as a future evaluation platform for the framework. Some steps have already been taken in this direction. Besides Object Pascal and FORTRAN, aspect-oriented metamodels for Java and C++ have already been constructed and are available at the GenAWeave project website [GenAWeave, 2008].

5.4 Applying the approach to DSALs

Although the majority of research in the AOSD community focuses on general-purpose aspect languages (e.g., AspectJ), there have been a number of influential research efforts on domain-specific aspect languages (DSALs) (e.g., COOL for concurrency management and RIDL for serialization [Lopes, 1997]). So far we have only considered the construction of GPALs using our framework, but it would be interesting to investigate how the framework can also accommodate the development of weavers targeting DSALs. In particular, future research will apply the GenAWeave framework towards the construction of an aspect weaver for the ANTLR grammar (ANother Tool for Language Recognition) [ANTLR, 2008], which is a DSL used for LR based parser generators. The proposed work would be similar to AspectG [Wu *et al.*, 2005], however the underlying technique would be different (i.e., the proposed research would use the GenAWeave framework instead). This would help us to compare the two techniques and their relative ease of use in constructing aspect weavers.

5.5 Applying the Approach to Scientific Computing Applications

Scientific computing applications like Blitz++ [Veldhuizen, 1998] and HPL [Petitet *et al*., 2004] play a critical role in solving several challenging problems within the high-performance scientific computing domain. The research presented in this dissertation has demonstrated techniques to improve the modularization of such applications. The GenAWeave framework could be applied suitably to weave in aspects that are identified in Blitz++; however, there is an equal need to extend the framework for non-AOP based transformations. For example, the transformations to specialize HPL are different in

intent than what is generally supported in an aspect-oriented language. Therefore, another possible extension of the GenAWeave framework will be to support DSLs to capture language constructs suitable for applying non-AOP based transformation techniques (e.g., specialization of HPL described in Chapter 3). However, such techniques are not just limited to libraries like HPL but can be applied to other scientific computing libraries like Matrix Template Library (MTL) [Siek and Lumsdaine, 1998] and POOMA [Reynders *et al*., 1996]. Nevertheless, the translation scheme from the high-level DSL to low-level RSL would be very similar to the one used in the current GenAWeave framework. Another interesting concept that is not explored in the current GenAWeave framework is multi-language weaving. Some programs, especially in the field of scientific computing, may have mixed-mode syntax, i.e., one language embed in another (e.g., FORTRAN calls inside a C program). In such cases, it may be required to weave in the language specific parts and write aspects having mixed-mode syntax. Future experimentation using GenAWeave may add such functionality to support mixed-mode weaving.

5.6 Generic Refactoring and Generic Aspect-Mining Engines based on Model-Driven Program Transformation

Similar to aspect weaving, refactoring [Fowler *et al*., 1999] can also improve modularization and readability of legacy software. Although the core idea behind refactoring (e.g., rename method, extract method, pull up method to a superclass and push down method to a subclass) is language-independent, current refactoring tools are generally tied to the source language. Therefore, the time and effort spent in constructing a refactoring tool for one language is almost wasted when applied to a different language context. A generalized framework that can capture the semantics of refactoring rules in

an abstract metamodel can improve the reusability and construction effort of refactoring engines. We believe that the knowledge gained during the current research in developing a generic framework for aspect weavers can be suitably applied to realize a similar goal to construct generic refactoring engines. Such an approach could also explore a model-driven metamodel based front-end with low-level program transformation support at the back-end. However, such techniques would necessitate more sophisticated control flow and data flow analysis that can be interchangeably used across GPLs.

The current GenAWeave framework does not support aspect mining, which is a technique to identify crosscutting concerns in an existing software system [Roy *et al.*, 2007]. Aspect mining enforces software systems to comply with an aspect-oriented design. Most of the aspects that were identified in the various case study applications during the course of this research were done manually. A future extension to automatically identify aspects in existing legacy applications can act as a complement to the current aspect weaving framework. Moreover, the analysis techniques (e.g., identifier analysis, fan-in analysis and dynamic analysis) [Roy *et al.*, 2007] are fairly language-independent and their intensions may be captured with suitable high-level language specifications. Therefore, as part of future research directions, both generic refactoring and generic aspect mining techniques will be investigated, which can serve as valuable extensions to the existing GenAWeave aspect weaver framework.

CHAPTER 6

CONCLUSION

Given the historical tendency of languages to evolve by adopting new paradigms, it is reasonable to assume that aspect-oriented concepts will be integrated into many more programming languages. To expedite this adoption, tools and frameworks that provide assistance for program restructuring are needed [Griswold and Notkin, 1993]. This will help early adopters assess the feasibility of AOSD within their own organization. However, the general focus of AOP has been based on a few popular programming languages like C++ and Java, neglecting the multiple billion lines of legacy code that exist in other languages. Given the large number of languages in use, a solution that mitigates the effort needed to create each new aspect weaver is more desirable than an approach that manually recreates a weaver from scratch for each legacy language. However, such a proposition raises several new challenges that represent the key focus of the research presented in this dissertation. Specifically, the research demonstrates how modern software engineering techniques like modeling combined with program transformation can assist in promoting aspect orientation in a generalized way for legacy languages.

6.1 Challenges addressed by the GenAWeave Framework

The research presented in this dissertation raised several key challenges (identified in Chapter 1) in designing a generic framework to construct aspect weavers. In particular, the research demonstrated an approach that combines PTEs with MDE to construct aspect weavers for modern and legacy programming languages. The unification of PTEs with MDE offered more possibilities to address the challenges raised in Chapter 1 than each of the two techniques considered separately.

It is our contention that initial efforts to bring aspect orientation to legacy systems should be robust and mature to the degree that they can be applied readily to large pre-existing applications. The scalability of such a requirement demands the availability of parsers that have been proven capable of handling large collections of source code. Toy parsers will only frustrate users to the point of potential abandonment of adoption. A mature PTE like DMS offers a repository of complete parsers and a program transformation language (i.e., RSL) for manipulating syntax trees. These two features help to reduce significantly the effort required to construct new weavers. Chapters 3 and 4 of the dissertation presented several case studies to demonstrate how RSL transformation rules could be used to construct aspect weavers for various GPLs (e.g., Object Pascal, C++ templates and FORTRAN). Thus, *Challenges C1 (parser construction problem)* and *C2 (weaver construction problem)* were addressed through adoption of a mature program transformation engine (i.e., DMS) as the back-end of the framework. However, a PTE-based weaver construction process raises new challenges and faces inherent accidental complexities; i.e., the transformation rules used to modify base programs are difficult to compose, which makes it accessible to only language

researchers and is generally hard to comprehend by average software developers. The research described in this dissertation illustrated how these accidental complexities (*Challenge C3*) that are generally associated with a PTE can be eliminated using a model-driven front-end (Chapter 4).

We believe that the development of each new weaver should minimize the duplication of effort from an earlier weaver construction. However, the dependence of weaver transformation rules on the grammar of the base GPL (e.g., Object Pascal) makes a previously constructed weaver almost impossible to be applied in a different language context. This concern was raised as *Challenge C4* that deals with generality, reusability, and transfer of knowledge from one weaver to another. In our opinion this is the most difficult challenge of the four. The research demonstrated that by making the front-end generic, along with a systematic program transformation rule generator, significant inroads could be made to address this challenge. To evaluate the usefulness of the generic framework, two aspect weavers were constructed for Object Pascal, and FORTRAN. The FORTRAN weaver was built after the successful construction of the Object Pascal weaver. When constructing the second weaver, it was observed that more than 50% of the artifacts (generic front-end and rule generator) that were created during the construction of the first could be reused. All these results and experiments are available in the GenAWeave project website [GenAWeave, 2008].

The current approach has been evaluated against a simple join point model. More advanced pointcuts like control flow and reflective techniques like `thisJoinPoint` are currently not available in GenAWeave. However, with the availability of a mature static/control flow analysis engine for Object Pascal and FORTRAN in DMS,

GenAWeave can be extended to support advanced aspect language features. This limitation will be addressed later as a part of future work. It may be noted that most of the analysis and pattern matching is realized through the back-end RSL/PARLANSE code and the front-end only acts as a wrapper to the back-end. If the back-end PTE can support advanced program analysis, it would not be difficult to wrap those features through the front-end, avoiding all the accidental complexities (*Challenge C3*) that are generally associated with complex PTEs. The complete source code for the GenAWeave framework, several case study examples and video demonstrations are available at the project web site [GenAWeave, 2008]. The following section presents a summary of the important lessons that we learned throughout the course of this research.

6.2 Lessons Learned

In this section, we summarize the seven main lessons that we have learned while working on the research presented in this dissertation. These lessons are enumerated below:

- ***Lesson 1 - Generalizing the weaver front-end:*** During the course of the research, it has been realized that parts of the aspect language front-end can be reused by making it generic. By generalizing the front-end metamodel, several aspect languages can extend a single core (e.g., GAspect) while the differences can be captured within their specific part. The solution can be achieved using MDE techniques like metamodel extension.
- ***Lesson 2 - Improving the generic metamodel:*** The current generic metamodel (i.e., GAspect) generalizes what is common between APascal and AFortran (i.e., the

aspect languages for Object Pascal, and FORTRAN). However, an extension of GAspect may categorize commonalities within a paradigm that can be reused (e.g., a metamodel named Object-Oriented that extends GAspect with common OO concepts, which is then extended by concrete OO languages). The idea was introduced in Chapter 5 (Section 5.1).

- ***Lesson 3 – Use of generic interfaces in the rule generator:*** The concept of a generic interface was introduced in Chapter 4 to generalize the design of the rule generator. As a result, the rule generator library (i.e., model transformation rules for translating specific join points like `call`, `loop` and `execution`) can be reused across languages with minimum customization.
- ***Lesson 4 - Modeling can be suitably applied to PTEs:*** From our research, it has been realized that higher-order model transformation rules could be used to generate lower-order program transformation rules. Thus, it is possible to model and automate the creation of low-level program transformation rules using MDE. The combination of both PTE and MDE (i.e., two distinct technical spaces) offers more possibilities than each considered separately.
- ***Lesson 5 - Changing the target PTE:*** The source aspect metamodel need not be altered even if one chooses to opt for a different target PTE (e.g., ASF+SDF [van den Brand et al., 2002]). In such a case, a new PTE metamodel needs to be developed, as well as a new rule generator for this new target. It is expected that it may be possible to generalize part of the transformation code by introducing a PTE pivot metamodel that abstracts common properties of many PTEs. It may be noted

that MDE is about platform independence and the ability to support multiple PTEs is another benefit of the approach described in this dissertation.

- ***Lesson 6 - Changing the source language:*** Conversely, for every new aspect language, one needs to add the appropriate metamodel extensions to the GAspect metamodel, but no change to the target metamodel is needed.
- ***Lesson 7 - Automation of rule generator:*** During the construction of the ATL rule generator, it was realized that most of the time and effort on building a new weaver for a particular GPL was spent on understanding the concrete syntax or grammar of the base language. We believe that it is possible to extract the join point model from transformation rules, and model it in terms of the concrete syntax. A future proposal towards this direction is discussed in Chapter 5 (Section 5.2).

LIST OF REFERENCES

[Aßmann, 2003] Uwe Aßmann, *Invasive Software Composition*, Springer, 2003.

[Aßmann and Ludwig, 1999] Uwe Aßmann and Andreas Ludwig, "Aspect Weaving as Graph Rewriting," *Generative Component-based Software Engineering*, Springer-Verlag LNCS 1799, Erfurt, Germany, October 1999, pp. 24-36.

[Adams, 2005] Bram Adams, "Language-Independent Aspect Weaving," *Summer School on Generative and Transformational Techniques in Software Engineering*, Braga, Portugal, July 2005.

[Agrawal, 2003] Aditya Agrawal, "Metamodel Based Model Transformation Language to Facilitate Domain Specific Model-Driven Architecture," *Object-Oriented Programming, Systems, Languages, and Applications Companion*, Anaheim, CA, October 2003, pp. 118-119.

[Alblas, 1991] Henk Alblas, "Attribute Evaluation Methods," *Proceedings on Attribute Grammars, Applications and Systems*, Springer-Verlag LNCS 545, 1991, pp.48-113.

[AMW Use Case, 2008] AMW Use Case - Metamodel Comparison and Model Migration, 2008, http://www.eclipsecon.com/gmt/amw/usecases/compare/.

[ANTLR, 2008] ANother Tool for Language Recognition, 2008, http://www.antlr.org.

[AOM, 2008] *Aspect-Oriented Modeling Workshop*, http://www.aspect-modeling.org/.

[Apostle, 2008] Apostle: Aspect Programming in Smalltalk, 2008, http://www.cs.ubc.ca/labs/spl/projects/apostle/.

[Arranga, 2000] Ed Arranga, "In Cobol's Defense," *IEEE Software*, March/April 2000, pp. 70-75.

[AspectR, 2008] AspectR - Simple Aspect-Oriented Programming in Ruby, 2008 http://aspectr.sourceforge.net/.

[Barbero *et al*., 2007] Mikhail Barbero, Frédéric Jouault, Jeff Gray, and Jean Bézivin, "A Practical Approach to Model Extension," *European Conference on Model Driven Architecture Foundations and Applications*, Haifa, Israel, June 2007, pp. 32-42.

[Batory, 2003] Don Batory, "A Tutorial on Feature Oriented Programming and Product-lines," *International Conference on Software Engineering*, Portland, OR, May 2003, pp. 753-754.

[Batory *et al*., 1998] Don Batory, Bernie Lofaso, and Yannis Smaragdakis, "JTS: Tools for Implementing Domain-Specific Languages," *International Conference on Software Reuse*, Victoria, Canada, June 1998, pp. 143-153.

[Baxter, 1992] Ira Baxter, "Design Maintenance Systems," *Communications of the ACM*, vol. 35, no. 4, April 1992, pp. 73-89.

[Baxter *et al.*, 2004] Ira Baxter, Christopher Pidgeon, and Michael Mehlich, "DMS: Program Transformation for Practical Scalable Software Evolution," *International Conference on Software Engineering*, Edinburgh, Scotland, May 2004, pp. 350-354.

[Bergmans and Aksit, 2001] Lodewijk Bergmans and Mehmet Aksit, "Composing Crosscutting Concerns using Composition Filters," *Communications of the ACM*, vol. 44, no. 10, October 2001, pp. 51-57.

[Bounif and Pottinger, 2006] Hassina Bounif and Rachel Pottinger, "Schema Repository for Database Schema Evolution," *International Conference on Database and Expert Systems Applications*, Krakow, Poland, September 2006, pp. 647-651.

[Brichau, 2002] Johan Brichau, "Composable Aspect-Specific Languages," *Generative Programming and Component Engineering, Young Researchers Workshop*, Pittsburgh, PA, October 2002.

[Bravenboer and Visser, 2004] Martin Bravenboer and Eelco Visser, "Concrete Syntax for Objects: Domain-specific Language Embedding and Assimilation without Restrictions," *Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, Canada, October 2004, pp. 365-383.

[Budinsky *et al*., 2003] Frank Budinsky, Dave Steinberg, Ed Merks, Ray Ellersick, and Timothy J. Grose, *Eclipse Modeling Framework*, Addison-Wesley, 2003.

[Capra *et al*., 2007] Eugenio Capra, Chiara Francalanci, and Francesco Merlo, "The Economics of Open Source Software: An Empirical Analysis of Maintenance Costs," *IEEE International Conference on Software Maintenance*, Paris, France, October 2007, pp. 1-10.

[Cazzola *et al*., 2005] Walter Cazzola, Sonia Pini, and Massimo Ancona, "AOP for Software Evolution: A Design Oriented Approach," *ACM Symposium on Applied Computing*, Santa Fe, NM, March 2005, pp. 1346-1350.

[Chalabine and Kessler, 2006] Mikhail Chalabine and Christoph Kessler, "Crosscutting Concerns in Parallelization by Invasive Software Composition and Aspect Weaving," *Hawaii International Conference on System Sciences*, Kauai, HI, January 2006.

[CHAOS, 2006] *Annual CHAOS Report*, The Standish Group International, Inc., 2006.

[Chaplin *et al*., 2001] Ned Chapin, Joanne Hale, Khaled Kham, Juan Ramil, and Wui-Gee Tan, "Types of Software Evolution and Software Maintenance," *Journal of Software Maintenance: Research and Practice*, vol. 13, no. 1, January 2001, pp. 3-30.

[Clarke and Baniassad, 2005] Siobhàn Clarke and Elisa Baniassad, *Aspect-Oriented Analysis and Design: The Theme Approach*, Addison-Wesley, 2005.

[Clarke *et al*., 1999] Siobhàn Clarke, William Harrison, Harold Ossher, and Peri Tarr, "Subject-Oriented Design: Towards Improved Alignment of Requirements, Design, and Code," *Object-Oriented Programming, Systems, Languages, and Applications*, Denver, CO, October 1999, pp. 325-339.

[Coady and Kiczales, 2003] Yvonne Coady and Gregor Kiczales, "Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code," *International Conference on Aspect-Oriented Software Development*, Boston, MA, March 2003, pp. 50-59.

[Colyer *et al*., 2004] Adrian Colyer, Andy Clement, George Harley, and Matthew Webster, *AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*, Addison-Wesley, 2004.

[Constantinides *et al*., 2002] Constantinos Constantinides, Tzilla Elrad, and Mohamed Fayad, "Extending the Object Model to Provide Explicit Support for Crosscutting Concerns," *Software - Practice and Experience*, vol. 32, no. 7, June 2002, pp. 703-734.

[Cordy *et al*., 2002] James Cordy, Thomas Dean, Andrew Malton, and Kevin Schneider, "Source Transformation in Software Engineering using the TXL Transformation System," *Journal of Information and Software Technology*, vol. 44, no. 13, October 2002, pp. 827-837.

[Cottenier *et al*., 2007] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad "Motorola WEAVR: Aspect and Model-Driven Engineering," *Journal of Object Technology* (Special Issue: Aspect-Oriented Modeling), vol. 6, no. 7, August 2007, pp. 51-88.

[Czarnecki and Eisenecker, 2000] Krzysztof Czarnecki and Ulrich Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[Czarnecki and Helsen, 2006] Krzysztof Czarnecki and Simon Helsen, "Feature-Based Survey of Model Transformation Approaches," *IBM Systems Journal*, vol. 45, no. 3, July 2006, pp. 621-645.

[Devanbu, 1999] Prem Devanbu, "GENOA—A Customizable, Front-end-retargetable Source Code Analysis Framework," *ACM Transactions on Software Engineering and Methodology*, vol. 8, no. 2, April 1999, pp. 177-212.

[Dig and Johnson, 2005] Danny Dig and Ralph Johnson, "The Role of Refactorings in API Evolution," *IEEE International Conference on Software Maintenance*, Budapest, Hungary, September 2005, pp. 389-398.

[Dijkstra, 1982] Edsger Dijkstra, *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982.

[Dongarra, 2002] Jack Dongarra, "Basic Linear Algebra Subprograms Technical Forum Standard," *International Journal of High Performance Applications and Supercomputing*, vol. 16, no. 1, 2002, pp. 1-111.

[Eaddy *et al*., 2007] Marc Eaddy, Alfred Aho, and Gail C. Murphy, "Identifying, Assigning, and Quantifying Crosscutting Concerns," *International Workshop on Assessment of Contemporary Modularization Techniques*, Minneapolis, MN, May 2007, pp. 2.

[Eclipse M2T, 2007] *Eclipse Model to Text (M2T) project*, http://www.eclipse.org/modeling/m2t/.

[Filman *et al*., 2004] Robert Filman, Tzilla Elrad, Siobhan Clarke, and Mehmet Aksit, *Aspect-Oriented Software Development*, Addison-Wesley, 2004.

[Fayad and Schmidt, 1997] Mohamed Fayad and Douglas Schmidt, "Object-Oriented Application Frameworks-Introduction," *Communications of the ACM*, vol. 40, no. 10, October 1997, pp. 32-38.

[Filman and Friedman, 2004] Robert Filman and Daniel Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness," in *Aspect-Oriented Software Development*, Addison-Wesley, 2004.

[Fowler *et al*., 1999] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[Fradet and Südholt, 1998] Pascal Fradet and Mario Südholt, "Towards a Generic Framework for Aspect-Oriented Programming," *Third AOP Workshop, ECOOP '98 Workshop Reader*, Springer-Verlag LNCS 1543, Brussels, Belgium, July 1998, pp. 394-397.

[France *et al*., 2004] Robert France, Indrakshi Ray, Geri Georg, and Sudipto Ghosh, "Aspect-Oriented Approach to Design Modeling," *IEE Proceedings - Software* (Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design), vol. 151, no. 4, August 2004, pp. 173-185.

[Gamma *et al*., 1994] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[GenAWeave, 2008] A Generic Aspect Weaver Framework based on Model-Driven Program Transformation, 2008, http://www.cis.uab.edu/softcom/GenAWeave/.

[Germon, 2001] Roy Germon, "Using XML as an Intermediate Form for Compiler Development," *XML Conference and Exposition*, Orlando, FL, December 2001.

[Getov *et al*., 1998] Vladimir Getov, Susan Flynn Hummel, and Sava Mintchev, "High-performance Parallel Programming in Java: Exploiting Native Libraries," *Concurrency: Practice and Experience*, vol. 10, no. 11-13, September-November 1998, pp. 863-872.

[Giese and Vilbig, 2006] Holger Giese and Alexander Vilbig, "Separation of non-orthogonal concerns in software architecture and design," *Journal of Software and Systems Modeling*, vol. 5, no. 2, June 2006, pp. 136-169.

[Gray *et al*., 2001] Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck, "Handling Crosscutting Constraints in Domain-Specific Modeling," *Communications of the ACM*, vol. 44, no. 10, October 2001, pp. 87-93.

[Gray and Roychoudhury, 2004] Jeff Gray and Suman Roychoudhury, "A Technique for Constructing Aspect Weavers using a Program Transformation Engine," *International Conference on Aspect-Oriented Software Development*, Lancaster, UK, March 2004, pp. 36-45.

[Gray *et al.*, 2003] Jeff Gray, Yuehua Lin, and Jing Zhang, "Aspect Model Weavers: Levels of Supported Independence," *Middleware 2003: Workshop on Model-driven Approaches to Middleware Applications Development*, Rio de Janeiro, Brazil, June 2003.

[Gray *et al*., 2007] Jeff Gray, Juha-Pekka Tolvanen, Steven Kelly, Aniruddha Gokhale, Sandeep Neema, and Jonathan Sprinkle, "Domain-Specific Modeling," *Handbook of Dynamic System Modeling*, CRC Press, 2007.

[Griswold and Notkin, 1993] William Griswold and David Notkin, "Automated Assistance for Program Restructuring," *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 3, July 1993, pp. 228-269.

[Gropp *et al*., 1996] William Gropp, Ewing Lusk, Nathan Doss and Anthony Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard," *Parallel Computing*, vol. 22, no. 6, 1996, pp. 789-828.

[Harbulot and Gurd, 2004] Bruno Harbulot and John Gurd, "Using AspectJ to Separate Concerns in a Parallel Scientific Java Code," *International Conference on Aspect-Oriented Software Development*, Lancaster, UK, March 2004, pp. 122-131.

[Harbulot and Gurd, 2005] Bruno Harbulot and John Gurd, "A Join Point for Loops in AspectJ," *Workshop on Foundations of Aspect-Oriented Languages*, Chicago, IL, March 2005.

[Hedin and Magnusson, 2003] Görel Hedin and Eva Magnusson, "JastAdd-an Aspect-Oriented Compiler Construction System," *Science of Computer Programming*, vol. 47, no. 1, April 2003, pp. 37-58.

[Heidenreich *et al*., 2007] Florian Heidenreich, Jendrik Johannes, and Steffen Zschaler, "Aspect Orientation for Your Language of Choice," *International Workshop on Aspect-Oriented Modeling*, Nashville, TN, September 2007.

[Ho *et al*., 2002] Wai-Ming Ho, Jean-Marc Jézéquel, François Pennaneac'h, and Noël Plouzeau, "A Toolkit for Weaving Aspect-Oriented UML Designs," *International Conference on Aspect-Oriented Software Development*, Enschede, Netherlands, March 2002, pp. 99-105.

[IJSEKE, 2006] *International Journal of Software Engineering and Knowledge Engineering*, (Special Issue on Aspect-Oriented Software Design Models), vol. 16, no. 3, June 2006.

[Irwin *et al*., 1997] John Irwin, Jean-Marc Loingtier, John Gilbert, Gregor Kiczales, John Lamping, Anurag Mendhekar, and Tatiana Shpeisman, "Aspect-Oriented Programming of Sparse Matrix Code," *International Scientific Computing in Object-Oriented Parallel Environments*, Springer-Verlag LNCS 1343, Marina del Ray, CA, December 1997, pp. 249-256.

[JBoss, 2008] JBoss Project, 2008, http://www.jboss.org/.

[Jackson and Clarke, 2004] Andrew Jackson and Siobhán Clarke, "SourceWeave.NET: Cross-Language Aspect-Oriented Programming," *Generative Programming and Component Engineering*, Springer-Verlag LNCS 3286, Vancouver, Canada, October 2004, pp. 115-135.

[Jacobson and Ng, 2005] Ivar Jacobson and Pan-Wei Ng, *Aspect-Oriented Software Development with Use Cases*, Addison-Wesley, 2005.

[Johnson and Gannon, 1997] Elizabeth Johnson and Dennis Gannon, "HPC++: Experiments with the Parallel Standard Template Library," *International Conference on Supercomputing*, Vienna, Austria, July 1997, pp. 124-131.

[Jossic *et al*., 2007] Albin Jossic, Marcos Didonet Del Fabro, Jean-Philippe Lerat, Jean Bézivin, and Frédéric Jouault, "Model Integration with Model Weaving: a Case Study in System Architecture," *International Conference on Systems Engineering and Modeling*, Haifa, Israel, March 2007, pp. 79-84.

[Josuttis, 1999] Nicolai M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*, Addison-Wesley, 1999.

[Jouault *et al*., 2006] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev, "TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering," *Generative Programming and Component Engineering*, Portland, OR, October 2006, pp. 249-254.

[Jouault and Bézivin, 2006] Frédéric Jouault and Jean Bézivin, "KM3: a DSL for Metamodel Specification," *Formal Methods for Open Object-Based Distributed Systems*, Springer-Verlag LNCS 4037, Bologna, Italy, June 2006, pp. 171-185.

[Jouault and Kurtev, 2005] Frédéric Jouault and Ivan Kurtev, "Transforming Models with ATL," *Model Transformations in Practice Workshop at MoDELS*, Montego Bay, Jamaica, September 2005.

[Kiczales *et al*., 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin, "Aspect-Oriented Programming," *European Conference on Object-Oriented Programming*, Springer-Verlag LNCS 1241, Jyväskylä, Finland, June 1997, pp. 220-242.

[Kiczales *et al*., 2001] Gregor Kiczales, Eric Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold, "Getting Started with AspectJ," *Communications of the ACM*, vol. 44, no. 10, October 2001, pp. 59-65.

[Klint *et al*., 2004] Paul Klint, Tijs van der Storm, and Jurgen Vinju, "Term Rewriting Meets Aspect-Oriented Programming," *REPORT SEN-E0421*, http://www.cwi.nl/ftp/CWIreports/SEN/SEN-E0421.pdf, December 2004.

[Klint *et al*., 2005] Paul Klint, Ralf Lämmel, and Chris Verhoef, "Toward an Engineering Discipline for Grammarware," *ACM Transactions on Software Engineering and Methodology*, vol. 14, no. 3, 2005, pp. 331-380.

[Kurtev *et al*., 2006] Ivan Kurtev, Jean Bézivin, Frédéric Jouault, and P. Valduriez, "Model-based DSL Frameworks," *Object-Oriented Programming, Systems, Languages and Applications Companion*, Portland, OR, October 2006, pp. 602-616.

[Laddad, 2003] Ramnivas Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*, Manning, 2003.

[Lafferty and Cahill, 2003] Donal Lafferty and Vinny Cahill, "Language-Independent Aspect-Oriented Programming," *Object-Oriented Programming, Systems, Languages, and Applications*, Anaheim, CA, October 2003, pp. 1-12.

[Lahire *et al*., 2007] Philippe Lahire, Brice Morin, Gilles Vanwormhoudt, Alban Gaignard, Olivier Barais, and Jean-Marc Jézéquel, "Introducing Variability into Aspect-Oriented Modeling Approaches," *Model Driven Engineering Languages and Systems*, Nashville, TN, October 2007, pp. 498-513.

[Lämmel, 1999] Ralf Lämmel, "Declarative Aspect-Oriented Programming," *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, San Antonio, TX, January 1999, pp. 131-146.

[Lämmel and Verhoef, 2001] Ralf Lämmel and Chris Verhoef, "Cracking the 500 Language Problem," *IEEE Software*, vol. 18, no. 6, November/December 2001, pp. 78-88.

[Lämmel and Schutter, 2005] Ralf Lämmel and Kris De Schutter, "What Does Aspect-Oriented Programming Mean to Cobol?" *International Conference on Aspect-Oriented Software Development*, Chicago, IL, March 2005, pp. 99-110.

[Lämmel *et al*., 2003] Ralf Lämmel, Eelco Visser, and Joost Visser, "Strategic Programming Meets Adaptive Programming," *International Conference on Aspect-Oriented Software Development*, Boston, MA, March 2003, pp. 168-177.

[Lédeczi *et al*., 2001] Ákos Lédeczi, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, vol. 34 no. 11, November 2001, pp. 44-51.

[Lehman *et al*., 1997] Meir Lehman, Juan Ramil, Paul Wernick, Dewayne Perry, and Wladyslaw Turski, "Metrics and Laws of Software Evolution - The Nineties View," *International Software Metrics Symposium*, Albuquerque, NM, November 1997, pp. 20-33.

[Lieberherr *et al*., 2001] Karl Lieberherr, Doug Orleans, and Johan Ovlinger, "Aspect-Oriented Programming with Adaptive Methods," *Communications of the ACM*, vol. 44, no. 10, October 2001, pp. 39-41.

[Lippert and Lopes, 2000] Martin Lippert and Cristina Lopes, "A Study on Exception Detection and Handling Using Aspect-Oriented Programming," *International Conference of Software Engineering*, Limmerick, Ireland, June 2000, pp. 418-427.

[Lohmann *et al*., 2004] Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk, "Generic Advice: On the Combination of AOP with Generative Programming in AspectC++," *Generative Programming and Component Engineering*, Springer-Verlag LNCS 3286, Vancouver, BC, October 2004, pp. 55-74.

[Lopes, 1997] Cristina Lopes, "D: A Language Framework for Distributed Programming," Ph.D. Dissertation, College of Computer Science, Northeastern University, December 1997 (1998).

[López, 1992] Jesús López, "Generalized LR parsing," *Journal of Machine Translation*, vol. 7, no.3, September 1992, pp. 214-217.

[Masuhara and Kiczales, 2003] Hidehiko Masuhara and Gregor Kiczales, "Modeling Crosscutting in Aspect-Oriented Mechanisms," *European Conference on Object-Oriented Programming*, Springer-Verlag LNCS 2743, Darmstadt, Germany, July 2003, pp. 2-28.

[McKinsey, 2004] *How IT spending is changing*, McKinsey Quarterly, 2004.

[Mens and Tourwe, 2003] Tom Mens and Tom Tourwe, "A Survey of Software Refactoring," vol. 30, no. 2, *IEEE Transactions on Software Engineering*, February 2003, pp. 126-139.

[Mernik *et al.*, 2002] Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer, "LISA: An Interactive Environment for Programming Language Development," *International Conference on Compiler Construction*, Grenoble, France, April 2002, pp. 1-4.

[Mernik *et al*., 2005] Marjan Mernik, Jan Heering, and Anthony Sloane, "When and How to Develop Domain Specific Languages," *ACM Computing Surveys*, vol. 37, no. 4, 2005, pp. 316-344.

[MoDisco, 2008] *Model Discovery*, http://www.eclipse.org/gmt/modisco/.

[Morin *et al*., 2007] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, and Rodrigo Ramos, "Towards a Generic Aspect-Oriented Modeling Framework," *Workshop on Models and Aspects*, Berlin, Germany, July 2007.

[Models and Aspects, 2008] *Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSD*, http://www.kircher-schwanninger.de/workshops/MDD&AOSD/.

[Neema *et al*., 2002] Sandeep Neema, Ted Bapty, Jeff Gray, Aniruddha S. Gokhale, "Generators for Synthesis of QoS Adaptation in Distributed Real-Time Embedded Systems," *Generative Programming and Component Engineering*, Springer-Verlag LNCS 2487, Pittsburgh, PA, September 2002, pp. 236-251.

[Object Technology, 2007] *Journal of Object Technology* (Special Issue on Aspect Modeling), vol. 6, no. 7, August 2007.

[OMG ADM, 2008] *OMG: Architecture-Driven Modernization*, 2008, http://adm.omg.org/.

[OMG MOF, 2003] *OMG: Meta Object Facility (MOF) 2.0 Core Specification*, OMG Document ptc/03-10-04, http://www.omg.org/docs/ptc/03-10-04.pdf, 2003.

[OMG OCL, 2001] *OMG: Object Constraint Language Specification, version 2.0*, formal/2006-05-01, http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf, 2001.

[OMG QVT, 2001] *OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*, OMG Document http://www.omg.org/docs/ptc/05-11-01.pdf, 2001.

[Opdyke, 1992] William Opdyke, *Refactoring Object-Oriented Frameworks*, Ph.D. Dissertation, 1992.

[Ossher *et al.*, 1994] Harold Ossher, William Harrison, Frank Budinsky, and Ian Simmonds, "Subject-oriented programming: Supporting decentralized development of objects," *IBM Conference on Object-Oriented Technology*, Santa Clara, CA, July 1994.

[Paakki, 1995] Jukka Paakki, "Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation," *ACM Computing Surveys*, vol. 27, no. 4, 1995, pp. 196-255.

[Parnas, 1972] David Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, December 1972, pp. 1053-1058.

[Petitet *et al*., 2004] Antoine Petitet, Clinton Whaley, Jack Dongarra, and Andrew Cleary, "HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers," *Version 1.0a*, http://www.netlib.org/benchmark/hpl, 2004.

[PostSharp, 2008] PostSharp for Microsoft .NET, 2008, http://www.postsharp.org/.

[Quinlan *et al*., 2004] Daniel Quinlan, Markus Schordan, Brian Miller, and Markus Kowarschik, "Parallel Object-Oriented Framework Optimization," *Concurrency: Practice and Experience*, vol. 16, no. 2-3, February-March 2004, pp. 293-302.

[Ramos *et al*., 2007] Rodrigo Ramos, Olivier Barais, and Jean-Marc Jezequel, "Matching Model-Snippets," *International Conference on Model Driven Engineering Languages and Systems*, Nashville, TN, September 2007, pp. 121-135.

[Reddy *et al*., 2006] Raghu Reddy, Sudipto Ghosh, Robert France, Greg Straw, James Bieman, Nathan McEachen, Eunjee Song, and Geri Georg, "Directives for Composing Aspect-Oriented Design Class Models," *Transactions on Aspect-Oriented Software Development*, Springer-Verlag LNCS 3880, 2006, pp. 75-105.

[Reynders *et al*., 1996] John Reynders, Paul Hinker, Julian Cummings, Susan Atlas, Subhankar Banerjee, William Humphrey, Steve Karmesin, Katarzyna Keahey, Marikani Srikant, and Mary Dell Tholburn, "POOMA: A Framework for Scientific Simulations of Paralllel Architectures," *Parallel Programming Using C++*, MIT Press, 1996.

[Roover *et al*., 2007] Coen De Roover, Theo D'Hondt, Johan Brichau, Carlos Noguera, and Laurence Duchien, "Behavioral Similarity Matching Using Concrete Source Code Templates in Logic Queries," *ACM Sigplan Workshop on Partial Evaluation and Program Manipulation*, Nice, France, January 2007, pp. 92-101.

[Roy *et al.*, 2007] Chanchal Kumar Roy, Mohammad Gias Uddin, Banani Roy, and Thomas Dean, "Evaluating Aspect Mining Techniques: A Case Study," *IEEE International Conference on Program Comprehension*, Alberta, Canada, June 2007, pp. 167-176.

[Roychoudhury, 2004] Suman Roychoudhury, "A Language-Independent Approach to Software Maintenance using Grammar Adapters," *Object-Oriented Programming Systems, Languages, and Applications Companion*, Vancouver, BC, October 2004, pp. 52-53.

[Roychoudhury *et al*., 2007] Suman Roychoudhury, Frédéric Jouault and Jeff Gray, "Model-Based Aspect Weaver Construction," *International Workshop on Language Engineering*, Nashville, TN, October 2007, pp. 117-126.

[Roychoudhury *et al*., 2008] Suman Roychoudhury, Jeff Gray, Jing Zhang, Purushotham Bangalore, and Anthony Skjellum, "Modularizing Scientific Libraries with Aspect-oriented and Generative Programming Techniques," *Acta Electrotechnica et Informatica*, 2008.

[Schach, 1996] Stephen Schach, "The Cohesion and Coupling of Objects," *Journal of Object-Oriented Programming*, vol. 8, no. 8, January 1996, pp. 48-50.

[Schach and Tomer, 2000] Stephen Schach and Amir Tomer, "A Maintenance-Oriented Approach to Software Construction," *Journal of Software Maintenance-Research and Experience*, vol. 12, no. 1, February 2000, pp. 25-45.

[Schmidt, 2006] Douglas Schmidt, "Guest Editor's Introduction: Model-Driven Engineering," *IEEE Computer*, vol. 39, no. 2, February 2006, pp. 25-31.

[Schordan and Quinlan, 2003] Markus Schordan and Daniel Quinlan, "A Source-To-Source Architecture for User-Defined Optimizations," *Joint Modular Languages Conference*, Springer-Verlag LNCS 2789, Klagenfurt, Austria, August 2003, pp. 214-223.

[Schutter and Adams, 2007] Kris De Schutter and Bram Adams, "Aspect-orientation for Revitalising Legacy Business Software," vol. 166, *Electronic Notes in Theoretical Computer Science*, January 2007, pp. 63-80.

[Sharp, 2000] David Sharp, "Component-Based Product Line Development of Avionics Software," *First Software Product Lines Conference*, Denver, CO, August 2000, pp. 353-369.

[Shonle *et al*., 2003] Macneil Shonle, Karl Lieberherr, and Ankit Shah, "XAspects: An Extensible System for Domain Specific Aspect Languages," *Object-Oriented Programming, Systems, Languages, and Applications Companion*, Anaheim, CA, October 2003, pp. 28-37.

[Siek and Lumsdaine, 1998] Jeremy Siek and Andrew Lumsdaine, "The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra," *International Scientific Computing in Object-Oriented Parallel Environments*, Springer-Verlag LNCS 1505, Santa Fe, NM, December 1998, pp. 59-70.

[Skjellum *et al*., 2004] Anthony Skjellum, Purushotham Bangalore, Jeff Gray, and Barrett Bryant, "Reinventing Explicit Parallel Programming for Improved Engineering of High Performance Computing Software," *ICSE 2004 Workshop: International Workshop on Software Engineering for High Performance Computing System Applications*, Edinburgh, Scotland, May 2004.

[Spinczyk *et al*., 2002] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat, "AspectC++: An Aspect-Oriented Extension to C++," *International Conference on Technology of Object-Oriented Languages and Systems,* Sydney, Australia, February 2002, pp. 53-60.

[Steel and Jézéquel, 2007] Jim Steel and Jean-Marc Jézéquel, "On Model Typing," *Journal of Software and Systems Modeling*, vol. 6, no. 4, December 2007, pp. 452-468.

[Stein *et al*., 2002] Dominik Stein, Stefan Hanenberg, and Rainer Unland "A UML-based Aspect-Oriented Design Notation for AspectJ," *International Conference on Aspect-Oriented Software Development*, Enschede, Netherlands, March 2002, pp. 106-112.

[Sullivan *et al*., 2005] Kevin Sullivan, William Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan, "Information Hiding Interfaces for Aspect-Oriented Design," *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Lisbon, Portugal, September 2005, pp.166-175.

[Tarr *et al.*, 1999] Peri Tarr, Harold Ossher, William Harrison, and Stanley Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns," *International Conference on Software Engineering*, Los Angeles, CA, May 1999, pp. 107-119.

[Thai and Lam, 2003] Thuan Thai and Hoang Lam, *.NET Framework Essentials*, O'Reilly & Associates, 2003.

[Ubayashi *et al*., 2006] Naoyasu Ubayashi, Tetsuo Tamai, Shinji Sano, Yusaku Maeno, and Satoshi Murakami, "Metamodel Access Protocols for Extensible Aspect-Oriented Modeling," *International Transactions on Systems Science and Applications*, vol. 1, no.1, 2006, pp. 93-10.

[Ulrich, 2002] William Ulrich, *Legacy Systems: Transformation Strategies*, Prentice-Hall, 2002.

[van den Brand *et al*., 2002] Mark van den Brand, Jan Heering, Paul Klint, and Pieter Olivier, "Compiling Rewrite Systems: The ASF+SDF Compiler," *ACM Transactions on Programming Languages and Systems*, vol. 24, no. 4, July 2002, pp. 334-368.

[Veldhuizen, 1998] Todd Veldhuizen, "Arrays in Blitz++," *International Scientific Computing in Object-Oriented Parallel Environments*, Springer-Verlag LNCS 1505, Santa Fe, NM, December 1998, pp. 223-230.

[Veldhuizen and Gannon, 1998] Todd Veldhuizen and Dennis Gannon, "Active Libraries: Rethinking the Roles of Compilers and Libraries," *SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, Yorktown Heights, NY, October 1998.

[Visser, 2001] Eelco Visser, "Stratego: A Language for Program Transformation Based on Rewriting Strategies. System Description of Stratego 0.5," *International Conference on Rewriting Techniques and Applications,* Springer-Verlag LNCS 2051, Utrecht, Netherlands, May 2001, pp. 357-361.

[Wu *et al*., 2005] Hui Wu, Jeff Gray, Suman Roychoudhury, and Marjan Mernik, "Weaving a Debugging Aspect into Domain-Specific Language Grammars," *ACM Symposium on Applied Computing*, Santa Fe, NM, March 2005, pp. 1370-1374.

APPENDIX A

ASPECT PASCAL METAMODEL SPECIFICATIONS

The individual specifications within this Appendix show the KM3 and TCS specifications for the Aspect Pascal weaver that is extended from the Generic Aspect (GAspect) metamodel.

A.1. Generic Aspect Metamodel KM3 Specification

The following represents the complete KM3 specification for the GAspect metamodel. Note the GAspect metamodel is common to both Aspect Pascal and Aspect FORTRAN weaver specifications.

```
package GAspect {

    abstract class LocatedElement {
         attribute location[0-1] : String;
         attribute commentsBefore[*] ordered : String;
         attribute commentsAfter[*] ordered : String;
    }

    class GAspect extends LocatedElement {
         attribute name : String;
         reference domain container : Domain;
         reference pointcut[*] container : Pointcut oppositeOf aspect;
         reference advice[*] container : Advice oppositeOf aspect;
    }

    class Domain extends LocatedElement {
         attribute name : String;
    }

    abstract class Element extends LocatedElement {
         attribute name : String;
    }

    class Parameter extends LocatedElement {
         attribute name : String;
    }

    abstract class ParameterDef extends LocatedElement {
         attribute name : String;
         attribute type : String;
    }
```

```
class Pointcut extends Element {
     reference aspect : GAspect oppositeOf pointcut;
     reference paramdefs[*] container : ParameterDef;
     reference pctexpr[*] container : Expression oppositeOf
     pointcut;
}

abstract class Advice extends LocatedElement {
     reference aspect : GAspect oppositeOf advice;
     reference pctname : Element;
     reference paramdefs[*] container : ParameterDef;
     reference advStmt[*] container : Statement;
}

class BeforeAdvice extends Advice { }

class AfterAdvice extends Advice { }

class AroundAdvice extends Advice { }
     abstract class Expression extends LocatedElement {
     reference pointcut : Pointcut oppositeOf pctexpr;
}

class ArgsExpr extends Expression {
     attribute name : String;
}

abstract class Statement extends LocatedElement { }

class ProccedStatement extends Statement { }

abstract class LoopStatement extends Statement {
     reference loopInitCondition container : LoopInitCondition;
     reference loopExitCondition container : LoopExitCondition;
     reference loopStrideCondition container : LoopStrideCondition;
}

abstract class LoopInitCondition extends LocatedElement { }

abstract class LoopExitCondition extends LocatedElement { }

abstract class LoopStrideCondition extends LocatedElement {  }

class IntegerLoopInitCondition extends LoopInitCondition {
     attribute condition : Integer;
}

class IntegerLoopExitCondition extends LoopExitCondition {
     attribute condition : Integer;
}

class IntegerLoopStrideCondition extends LoopStrideCondition {
     attribute condition : Integer;
}

class StringLoopInitCondition extends LoopInitCondition {
     attribute condition : String;
```

```
    }

    class StringLoopExitCondition extends LoopExitCondition {
        attribute condition :String;
    }


    class StringLoopStrideCondition extends LoopStrideCondition {
        attribute condition :String;
    }

    class OpaqueStatement extends Statement {
        attribute  stmt : String;
    }

    class TryCatchFinallyStatement extends Statement {
        reference stmts[1-*] ordered container : Statement;
        reference finallyStmts[*] ordered container : Statement;
        reference catchStmts[*] ordered container : CatchStatement;
    }

    class CatchStatement extends LocatedElement {
        reference  stmts[*] container : Statement;
        reference exceptions[*]  container : ParameterDef;
    }

    class LoopExpr extends Expression {
        reference loopStmt container : LoopStatement;
    }

}

package PrimitiveTypes {
      datatype Boolean;
      datatype Integer;
      datatype String;
}
```

A.2. Aspect Pascal KM3 Specification

The following represents the complete KM3 specification for the Aspect Pascal metamodel, which is extended from the GAspect metamodel.

```
package APascal {

    class APascal extends GAspect { }

    abstract class FuncOrProcDefExpr extends Expression {
        reference funcOrProcSig container : FunctionOrProcSignature;
    }
```

```
    class ExecExpr extends FuncOrProcDefExpr { }

    class WithinCodeExpr extends FuncOrProcDefExpr { }

    abstract class FunctionOrProcSignature extends LocatedElement {
         attribute name : String;
    }

    abstract class FunctionOrProcDef extends FunctionOrProcSignature {
         attribute classifier[0-1] : String;
         reference paramdefs[*] container : OPParamDef;
    }

    class FunctionDef extends FunctionOrProcDef { }

    class ProcedureDef extends FunctionOrProcDef { }

    class RoutineDef extends FunctionOrProcDef {
         attribute wildcard[0-1] : String;
    }

    class OPParamDef extends ParameterDef { }

    class CallExpr extends Expression {
         reference funcOrProcSig container : FunctionOrProcSignature;
    }

    class FunctionOrProcCallExpr extends FunctionOrProcSignature {
         reference params[*]  container : Parameter;
    }

    class WithExpr extends Expression {
         attribute parent : String;
         attribute child : String;
    }

    class ForLoop extends LoopStatement { }

    class WhileLoop extends LoopStatement { }

}
```

A.3. Aspect Pascal TCS Specification

The following shows the TCS specification for the Aspect Pascal metamodel. The lexical part is not included here but available at [GenAWeave, 2008].

```
syntax APascal {

      primitiveTemplate identifier for String default using NAME:
            value = "%token%";
```

```
primitiveTemplate stringSymbol for String using STRING:
      value = "%token%",
      serializer="'\'' + %value%.toCString() + '\''";

primitiveTemplate integerSymbol for Integer default using INT:
      value = "Integer.valueOf(%token%)";

primitiveTemplate floatSymbol for Double default using FLOAT:
      value = "Double.valueOf(%token%)";

template Expression abstract;

template ParameterDef abstract;

template Element abstract;

template Advice abstract;

template Statement abstract;

template Domain
      :  "domain" name ";"
      ;

template Pointcut context addToContext
      :  "pointcut" name "(" paramdefs{separator = ","} ")"
          ":" pctexpr{separator = "&&"} ";"
      ;

template BeforeAdvice
      :  "before" "(" paramdefs{separator = ","} ")" ":"
            pctname{refersTo = name}
            "(" paramdefs{separator = ","} ")"
              "{"
                 advStmt
               "}"
      ;

template AfterAdvice
      :  "after" "(" paramdefs{separator = ","} ")" ":"
            pctname{refersTo = name}
            "(" paramdefs{separator = ","} ")"
              "{"
                 advStmt
               "}"
      ;

template AroundAdvice
      :  "around" "(" paramdefs{separator = ","} ")" ":"
            pctname{refersTo = name}
            "(" paramdefs{separator = ","} ")"
              "{"
                 advStmt
              "}"
      ;
```

```
template OpaqueStatement
      :  stmt {as = stringSymbol} ";"
      ;

template ProccedStatement
      :  "proceed" "("  ")" ";"
      ;

template LoopStatement abstract;

template LoopExpr
      :  "execution" "(" loopStmt ")"
      ;

template LoopInitCondition abstract;

template LoopExitCondition abstract;

template LoopStrideCondition abstract;

template IntegerLoopInitCondition
      :  "init" ":" condition
      ;

template IntegerLoopExitCondition
      :  "exitcond" ":" condition
      ;

template IntegerLoopStrideCondition
      :  "stride" ":" condition
      ;

template StringLoopInitCondition
      :  "init" ":" condition
      ;

template StringLoopExitCondition
      :  "exitcond" ":" condition
      ;

template StringLoopStrideCondition
      :  "stride" ":" condition
      ;

template TryCatchFinallyStatement
      :  "try" "{"
                  stmts
            "}" catchStmts
            (isDefined(finallyStmts) ?
                  "finally" "{"
                        finallyStmts
                  "}"
            )
      ;
```

```
template Parameter
      :  name
      ;

template CatchStatement
      :  "catch" "(" exceptions ")" "{" stmts "}"
      ;

template ArgsExpr
      :  "args" "(" name ")"
      ;

template APascal main
      :  domain "aspect" name "{" pointcut advice "}"
      ;

template WithExpr
      :  "with" "(" parent "." child ")"
      ;

template ExecExpr
      :  "execution" "(" funcOrProcSig ")"
      ;

template WithinCodeExpr
      :  "withincode" "(" funcOrProcSig ")"
      ;

template CallExpr
      :  "call" "(" funcOrProcSig ")"
      ;

template FunctionOrProcSignature abstract;

template FunctionOrProcCallExpr
      :  name "(" params{separator = ","} ")"
      ;

template FuncOrProcDefExpr abstract;

template FunctionOrProcDef abstract;

template FunctionDef
      :  "function"
         (isDefined(classifier) ? classifier ".")
         name "(" paramdefs{separator = ";"} ")"
      ;

template ProcedureDef
      :  "procedure"
         (isDefined(classifier) ? classifier ".")
         name "(" paramdefs{separator = ";"} ")"
      ;
```

```
      template RoutineDef
            :  wildcard
               (isDefined(classifier) ? classifier ".")
               name "(" paramdefs{separator = ";"} ")"
            ;

      template OPParamDef
            :  name ":" type
            ;


      template ForLoop
            :  "for" "(" loopInitCondition "," loopExitCondition ")"
            ;

      template WhileLoop
            : "while" "(" loopExitCondition ")"
            ;

--- For Lexical Specification, refer to [GenAWeave, 2008] ----

}
```

APPENDIX B

ASPECT FORTRAN METAMODEL SPECIFICATIONS

The individual specifications within this Appendix show the KM3 and TCS specifications for the Aspect FORTRAN weaver that is extended from the Generic Aspect metamodel (GAspect).

B.1. Aspect FORTRAN KM3 Specification

The following represents the complete KM3 specification for the Aspect FORTRAN metamodel, which is extended from the GAspect metamodel. The GAspect metamodel was earlier presented in Appendix A.1. and is not shown here.

```
package AFortran {

    class AFortran extends GAspect { }

    abstract class FuncOrSubDefExpr extends Expression {
         reference func_sub_Sig container : FuncOrSubSignature;
    }

    class ExecExpr extends FuncOrSubDefExpr { }

    class WithinCodeExpr extends FuncOrSubDefExpr { }

    class NotWithinCodeExpr extends FuncOrSubDefExpr { }

    abstract class FuncOrSubSignature extends LocatedElement {
         attribute name : String;
    }

    abstract class FuncOrSubDef extends FuncOrSubSignature {
         reference paramdefs[*]  container : FortranParamDef;
    }

    class FuncDef extends FuncOrSubDef { }

    class SubDef extends FuncOrSubDef { }

    class FortranParamDef extends ParameterDef { }

    class CallExpr extends Expression {
         reference func_sub_Sig container : FuncOrSubSignature;
    }
```

```
    class FuncOrSubCallExpr extends FuncOrSubSignature {
         reference params[*]  container : Parameter;
    }

    class DoLoop extends LoopStatement { }
}
```

B.2. Aspect FORTRAN TCS Specification

The following shows the TCS specification for the Aspect FORTRAN metamodel. The lexical part is not included here but available at [GenAWeave, 2008].

```
syntax AFortran {

      primitiveTemplate identifier for String default using NAME:
            value = "%token%";

      primitiveTemplate stringSymbol for String using STRING:
            value = "%token%",
            serializer="'\'' + %value%.toCString() + '\''";

      primitiveTemplate integerSymbol for Integer default using INT:
            value = "Integer.valueOf(%token%)";

      primitiveTemplate floatSymbol for Double default using FLOAT:
            value = "Double.valueOf(%token%)";

      template Expression abstract;

      template ParameterDef abstract;

      template Element abstract;

      template Advice abstract;

      template Statement abstract;

      template Domain
            :  "domain" name ";"
            ;

      template Pointcut context addToContext
            :  "pointcut" name "(" paramdefs{separator = ","} ")"
                 ":" pctexpr{separator = "&&"} ";"
            ;
```

```
template BeforeAdvice
      :  "before" "(" paramdefs{separator = ","} ")" ":"
            pctname{refersTo = name}
            "(" paramdefs{separator = ","} ")"
                "{"
                  advStmt
                "}"
      ;

template AfterAdvice
      :  "after" "(" paramdefs{separator = ","} ")" ":"
            pctname{refersTo = name}
            "(" paramdefs{separator = ","} ")"
                "{"
                  advStmt
                "}"
      ;

template AroundAdvice
      :  "around" "(" paramdefs{separator = ","} ")" ":"
            pctname{refersTo = name}
            "(" paramdefs{separator = ","} ")"
                "{"
                  advStmt
                "}"
      ;

template Parameter
      :  name
      ;

template OpaqueStatement
      :  stmt {as = stringSymbol} ";"
      ;

template ProccedStatement
      :  "proceed" "("  ")" ";"
      ;

template LoopStatement abstract;

template LoopExpr
      :  "execution" "(" loopStmt ")"
      ;

template LoopInitCondition abstract;

template LoopExitCondition abstract;

template LoopStrideCondition abstract;

template IntegerLoopInitCondition
      : "init" ":" condition
      ;
```

```
template IntegerLoopExitCondition
      : "exitcond" ":" condition
      ;

template IntegerLoopStrideCondition
      : "stride" ":" condition
      ;

template StringLoopInitCondition
      : "init" ":" condition
      ;

template StringLoopExitCondition
      : "exitcond" ":" condition
      ;

template StringLoopStrideCondition
      : "stride" ":" condition
      ;

template ArgsExpr
      : "args" "(" name ")"
      ;

template AFortran main
      :  domain "aspect" name "{" pointcut advice "}"
      ;

template FuncOrSubSignature abstract;

template ExecExpr
      :  "execution" "(" func_sub_Sig ")"
      ;

template WithinCodeExpr
      :  "withincode" "(" func_sub_Sig ")"
      ;

template NotWithinCodeExpr
      :  "!" "withincode" "(" func_sub_Sig ")"
      ;

template CallExpr
      :  "call" "(" func_sub_Sig ")"
      ;


template FuncOrSubCallExpr
      :  "CALL" name "(" params{separator = ","} ")"
      ;

template FuncOrSubDefExpr abstract;

template FuncOrSubDef abstract;
```

```
      template FuncDef
            :  "FUNCTION" name "(" paramdefs{separator = ","} ")"
            ;

      template SubDef
            :  "SUBROUTINE" name "(" paramdefs{separator = ","} ")"
            ;

      template FortranParamDef
            : type name
            ;

      template DoLoop
            : "do" "(" loopInitCondition "," loopExitCondition ","
                       loopStrideCondition")"
            ;

--- For Lexical Specification, refer to [GenAWeave, 2008] ----

}
```

APPENDIX C

RSL METAMODEL SPECIFICATION FOR BACK-END PTE

The individual specifications within this Appendix show the KM3 and TCS specifications for the target RSL used by the back-end PTE.

C.1. KM3 Specification for RSL

The following represents the complete KM3 specification for RSL.

```
package RSL {

      abstract class LocatedElement {
            attribute location : String;
            attribute commentsBefore[*] ordered : String;
            attribute commentsAfter[*] ordered : String;
      }

      abstract class RSLelements extends LocatedElement {
            reference rsl : RSL oppositeOf rslelems;
      }

      class RSL extends LocatedElement {
            reference domain container : Domain;
            reference rslelems[*] container : RSLelements oppositeOf
                                              rsl;
            reference ruleset container : RuleSet;
      }

      class Domain extends LocatedElement {
            attribute dname : String;
      }

      class Pattern extends RSLelements {
            reference phead container : PatternHead;
            attribute ptoken : String;
            reference ptext container : PatternText;
      }

      abstract class PatternText extends LocatedElement {
            attribute ptext : String;
      }

      class SimplePatternText extends PatternText { }

      class ConditionalPatternText extends PatternText {
            reference pref container : PatternRef;
      }
```

```
class ExternalPattern extends Pattern {
      attribute dname : String;
      attribute eptext : String;
}

class PatternHead extends LocatedElement {
      attribute name: String;
      reference params[*] container : PatternParameter;
}

class PatternParameter extends LocatedElement {
      attribute name : String;
      attribute referTo : String;
}

class Condition extends RSLelements {
      reference chead container : ConditionHead;
      attribute ctext  : String;
}

class ConditionHead extends LocatedElement {
      attribute name: String;
      reference params[*] container : ConditionParameter;
}

class ConditionParameter extends LocatedElement {
      attribute name : String;
      attribute referTo : String;
}

class Rule extends RSLelements {
      attribute rname : String;
      reference params[*] container : RuleParameter;
      attribute type : String;
      reference r_lhs_pattern container : RuleLHS;
      reference r_rhs_pattern container : RuleRHS;
}

class RuleParameter extends LocatedElement {
      attribute name : String;
      attribute referTo : String;
}

class RuleLHS extends LocatedElement {
      reference ruletext container : RuleText;
}

class RuleRHS extends LocatedElement {
      reference ruletext container : RuleText;
      reference condition[*] container: RuleCondition;

}

abstract class RuleText extends LocatedElement {
      attribute text : String;
}
```

```
        class SimpleRuleText extends RuleText { }

        class IDRuleText extends RuleText { }

        class ComplexRuleText extends RuleText {
              reference pref container : PatternRef;
        }

        abstract class RuleCondition extends LocatedElement {
              attribute lhs:String;
              reference pref container : PatternRef;
        }

        class RuleEqCondition extends RuleCondition { }

        class RuleNotEqCondition extends RuleCondition { }

        abstract class Parameter extends LocatedElement {
              attribute name : String;
        }

        class PatternRef extends Parameter {
              reference params[*] container : Parameter;
        }

        class RealParameter extends Parameter { }

        class RuleSet extends LocatedElement {
              attribute rsname : String;
              attribute rname[*] : String;
        }

}

package PrimitiveTypes {
        datatype Boolean;
        datatype Integer;
        datatype String;
}
```

C.2. TCS Specification for RSL

The following shows the TCS specification for RSL. Note that the source metamodels (concrete syntax) for Aspect Pascal and Aspect FORTRAN do not require the capture of any pretty printing information. However, the target RSL metamodel requires the capture of formatting information (e.g., `indentIncr = 1`) to decorate the generated output (RSL code).

```
syntax RSL {

      primitiveTemplate identifier for String default using NAME:
            value = "%token%";

      primitiveTemplate stringSymbol for String using STRING:
            value = "%token%",
            serializer="'\'' + %value%.toCString() + '\''";

      primitiveTemplate treeFragmentSymbol for String using
            TREEFRAGMENT:
            value = "%token%";

      primitiveTemplate integerSymbol for Integer default using INT:
            value = "Integer.valueOf(%token%)";

      primitiveTemplate floatSymbol for Double default using FLOAT:
            value = "Double.valueOf(%token%)";

      template RSL main
            :  [domain rslelems ] {nbNL = 2,indentIncr = 0,
                                   startNL = false}
               [ruleset]{nbNL = 1,indentIncr = 0}
            ;
      template RSLelements abstract;

      template Domain
            :  "default" "base" "domain" dname "."
            ;

      template ExternalPattern
            :  "external" "pattern" phead ":"
               ptoken ["="] {nbNL = 1,indentIncr = 1} eptext {as =
                             stringSymbol} "in" "domain" dname "."
            ;

      template Pattern abstract
            :  "pattern" phead ":"
               ptoken  ["=" ptext  "."]
            ;

      template PatternText abstract;

      template SimplePatternText
            :  ptext{as = treeFragmentSymbol}
            ;

      template ConditionalPatternText
            :  ptext "if" pref
            ;

      template PatternHead
            :  name "(" params{separator = ","} ")"
            ;
```

```
template PatternParameter
      : name ":" referTo
      ;

template Condition
      :  "external" "condition" chead
         "=" ctext {as = stringSymbol} "."
      ;

template ConditionHead
      :  name "(" [params{separator = ","} ")"] {indentIncr = 1}
      ;

template ConditionParameter
      :  name ":" referTo
      ;

template Rule
      : "rule" rname "("
            [params{separator = ","} ")"]{indentIncr = 1} ":"
            [type "->" type]  "="
            [r_lhs_pattern]  "->"
            [r_rhs_pattern "."]
      ;

template RuleParameter
      :  name ":" referTo
      ;

template RuleLHS
      :  ruletext
      ;

template RuleRHS
      :  ruletext [condition] {indentIncr = 0}
      ;

template RuleText abstract;

template SimpleRuleText
      :  text{as = treeFragmentSymbol}
      ;

template IDRuleText
      :  text
      ;

template ComplexRuleText
      :  pref
      ;

template RuleCondition abstract;

template RuleEqCondition
      :  "if" lhs "==" pref
      ;
```

```
      template RuleNotEqCondition
            :  "if" lhs "~=" pref
            ;

      template Parameter abstract;

      template PatternRef
            :  name "(" params{separator = ","} ")"
            ;

      template RealParameter
            :  name
            ;

      template RuleSet
            :  "public" "ruleset" rsname "=" "{" rname
               {separator = ","} "}" "."

            ;

--- For Lexical Specification, refer to [GenAWeave, 2008] ----

}
```

APPENDIX D

MODEL TRANSFORMATION RULES FOR ASPECT PASCAL AND ASPECT FORTRAN WEAVER

The individual specifications within this Appendix show the model transformation rules for the Aspect Pascal and Aspect FORTRAN weavers. The visual comparisons between the model transformation rules for the two weavers are available at the GenAWeave project website [GenAWeave, 2008].

D.1. ATL Rule for Translating Call Expression Join Point in Aspect Pascal

The following ATL rule shows the complete specification for translating a primitive call expression join point in Aspect Pascal to low-level RSL code.

```
module APascal2RSL;

create OUT : RSL from IN : APascal;

helper context String
    def: startsWith(s : String) : Boolean =
        s.size() <= self.size() and self.substring(1, s.size()) = s;

helper context String
    def: endsWith(s : String) : Boolean =
        let start : Integer = self.size() - s.size() + 1 in
            start > 0 and self.substring(start, self.size()) = s;

rule ApDomain2RSLDomain {
      from
            s : APascal!Domain
      to
            t : RSL!Domain (
                  dname <- s.name
            )
}

rule APascal2RSL {
      from
            s: APascal!APascal
      to
            t: RSL!RSL (
                  domain <-  s.domain,
                  rslelems <- Sequence {
                              s.advice,
                              thisModule.PointCutExprToMethodName(
                              s.pointcut->first().pctexpr->first()),
                              thisModule.PointCutExprToWithinCode(
                              s.pointcut->first().pctexpr->last()),
```

```
                                s.pointcut->collect(e |
                                thisModule.PointCutToExternalPattern(e)),
                                s.pointcut->collect(e |
                                thisModule.PointCutToRule(e))
                                },
                        ruleset <- rs
                        ),
                        rs : RSL!RuleSet (
                        rsname <- s.name,
                        rname <- s.pointcut->collect(e|e.name)
                )
}

rule BeforeAdvice2Pattern {
        from
                s : APascal!BeforeAdvice
        to
                t : RSL!Pattern (
                        phead <- ph,
                        ptoken <- 'statement_list',
                        ptext <- spt
                ),
                ph : RSL!PatternHead (
                        name <- 'before_advice_stmt'
                ),
                spt : RSL!SimplePatternText (
                        ptext <- s.advStmt->iterate(
                                        e; acc : String = '' | acc + if acc = ''
                                                then '' else '\r\n\t' endif + e.stmt
                                                        )
                )
}

rule GenerateAfterAdviceDummy extends BeforeAdvice2Pattern {
        from
                s : APascal!BeforeAdvice (
                        not s.aspect.advice->exists(e |
                                e.oclIsKindOf(APascal!AfterAdvice))
                )
        to
                t : RSL!Pattern,
                at : RSL!Pattern (
                        rsl <- s.aspect,
                        phead <- aph,
                        ptoken <- 'statement_list',
                        ptext <- aspt
                ),
                aph : RSL!PatternHead (
                        name <- 'after_advice_stmt'
                ),
                aspt : RSL!SimplePatternText (
                        ptext <- ''
                )
}
```

```
rule AfterAdvice2Pattern {
	from
		s : APascal!AfterAdvice
	to
		t : RSL!Pattern (
			phead <- ph,
			ptoken <- 'statement_list',
			ptext <- spt
		),
		ph : RSL!PatternHead (
			name <- 'after_advice_stmt'
		),
		spt : RSL!SimplePatternText (
			ptext <- s.advStmt->iterate(
				e; acc : String = '' | acc + if acc
				  = '' then '' else '\r\n\t' endif + e.stmt
					)
		)
}

rule GenerateBeforeAdviceDummy extends AfterAdvice2Pattern{
	from
		s : APascal!AfterAdvice (
			not s.aspect.advice->exists(e |
				e.oclIsKindOf(APascal!BeforeAdvice))
		)
	to
		t : RSL!Pattern,
		bt: RSL!Pattern (
			rsl <- s.aspect,
			phead <- bph,
			ptoken <- 'statement_list',
			ptext <- bspt
		),
		bph : RSL!PatternHead (
			name <- 'before_advice_stmt'
		),
		bspt : RSL!SimplePatternText (
			ptext <- ''
		)
}

lazy rule PointCutToExternalPattern {
	from
		s : APascal!Pointcut
	to
		t : RSL!ExternalPattern (
			dname <-  'ObjectPascal',
			eptext <- 'around_advice_call',
			ptoken <- 'ObjectPascal',
			phead <- ph
		),
		ph : RSL!PatternHead (
			name <-   'around_advice_call',
			params <- Sequence {pp1,pp2,pp3,pp4,pp5,pp6}
		),
		pp1 : RSL!PatternParameter (
```

```
                name <- 'program',
                referTo <- 'ObjectPascal'
        ),
        pp2 : RSL!PatternParameter (
                name <- 'method_name',
                referTo <- 'IDENTIFIER'
        ),
        pp3 : RSL!PatternParameter (
                name <- 'proceed_call',
                referTo <- 'IDENTIFIER'
        ),
        pp4 : RSL!PatternParameter (
                name <- 'proceed_bef',
                referTo <- 'statement_list'
        ),
        pp5 : RSL!PatternParameter (
                name <- 'proceed_aft',
                referTo <- 'statement_list'
        ),
        pp6 : RSL!PatternParameter (
                name <- 'withincode',
                referTo <- 'IDENTIFIER'
        )
}

lazy rule PointCutExprToWithinCode {
        from
                s : APascal!Expression
        to
                t : RSL!Pattern (
                        ptext <- spt ,
                        ptoken <- 'IDENTIFIER',
                        phead <- ph
                ),
                ph : RSL!PatternHead (
                        name <-   'within_code'
                ),
                spt : RSL!SimplePatternText (
                ptext <- if s.pointcut.pctexpr.size() > 1 then
                                ' '+ s.funcOrProcSig.name
                                else ' ' + 'mc_' endif

                )
}


lazy rule PointCutExprToMethodName {
        from
                s : APascal!Expression
        to
                t : RSL!Pattern (
                        ptext <- spt,
                        ptoken <- 'IDENTIFIER',
                        phead <- ph
                ),
                ph : RSL!PatternHead (
                        name <-   'method_name'
```

```
            ),
            spt : RSL!SimplePatternText (
            ptext <- ' '+ s.funcOrProcSig.name
            )
}

lazy rule PointCutToRule {
      from
            s : APascal!Pointcut
      to
            t : RSL!Rule (

                  rname <- s.name,
                  type <-   'ObjectPascal',
                  params <- Sequence {rp1},
                  r_lhs_pattern <- rlhsp,
                  r_rhs_pattern <- rrhsp
            ),
            rp1: RSL!RuleParameter (
                  name <- 'program',
                  referTo <- 'ObjectPascal'
            ),
            rlhsp : RSL!RuleLHS (
                        ruletext <-   irt -- IDRuleText
            ),
            irt : RSL!IDRuleText (
                  text <- 'program'
            ),
            rrhsp : RSL!RuleRHS (
                        ruletext <-   crt, -- ComplexRuleText
                        condition <- Sequence {rcon}
            ),
            crt : RSL!ComplexRuleText (
                  pref <- rule_rhs_pattern
            ),
            rule_rhs_pattern : RSL!PatternRef (
                        name <-   'around_advice_call',
                        params <- Sequence {
                                          param1,
                                          param2,
                                          param3,
                                          param4,
                                          param5,
                                          param6
                                    }
            ),
            param1 : RSL!RealParameter (
                           name <- 'program'
            ),
            param2 : RSL!PatternRef (
                           name <- 'method_name'
            ),

            param3 : RSL!PatternRef (
                           name <- 'method_name'
            ),
            param4 : RSL!PatternRef (
```

```
                    name <-  'before_advice_stmt'
                ),
                param5 : RSL!PatternRef (
                    name <-  'after_advice_stmt'
                ),
                param6 : RSL!PatternRef (
                    name <-  'within_code'
                ),
                rcon :  RSL!RuleNotEqCondition (
                    lhs <-  ' program',
                    pref <-   rule_rhs_cond
                ),
                rule_rhs_cond : RSL!PatternRef (
                    name <-   around_advice_call',
                    params <- Sequence {p1,p2,p3,p4,p5,p6}
                ),
                p1 : RSL!RealParameter (
                    name <- 'program'
                ),
                p2 : RSL!PatternRef (
                    name <- 'method_name'
                ),

                p3 : RSL!PatternRef (
                    name <- 'method_name'
                ),
                p4 : RSL!PatternRef (
                    name <- 'before_advice_stmt'
                ),
                p5 : RSL!PatternRef (
                    name <- 'after_advice_stmt'
                ),
                p6 : RSL!PatternRef (
                    name <- 'within_code'
                )

}
```

D.2. ATL Rule for Translating Loop Expression Join Point in Aspect Pascal

The following ATL rule shows the complete specification for translating a loop expression join point in Aspect Pascal to low-level RSL code.

```
module APascal2RSL;

create OUT : RSL from IN : APascal;

helper context String
    def: startsWith(s : String) : Boolean =
        s.size() <= self.size() and self.substring(1, s.size()) = s;
```

```
helper context String
    def: endsWith(s : String) : Boolean =
        let start : Integer = self.size() - s.size() + 1 in
            start > 0 and self.substring(start, self.size()) = s;

rule APascal2RSL {
      from
            s:    APascal!APascal
      to
            t:    RSL!RSL (
                  domain <-  s.domain,
                  rslelems <- Sequence {
                              s.advice,
                              thisModule.PointCutExprToInitExpr(
                              s.pointcut->first().pctexpr->first()),
                              thisModule.PointCutExprToExitExpr(
                              s.pointcut->first().pctexpr->first()),
                              thisModule.PointCutExprToWithinCode(
                              s.pointcut->first().pctexpr->last()),
                              s.pointcut->collect(e |
                              thisModule.PointCutToExternalPattern(e)),
                              s.pointcut->collect(e |
                              thisModule.PointCutToRule(e))
                              },
                  ruleset <- rs
            ),
            rs : RSL!RuleSet (
                  rsname <- s.name,
                  rname <- s.pointcut->collect(e|e.name)
            )
}


rule ApDomain2RSLDomain {
      from
            s : APascal!Domain
      to
            t : RSL!Domain (
                  dname <- s.name
            )
}

rule BeforeAdvice2Pattern {
      from
            s : APascal!BeforeAdvice
      to
            t : RSL!Pattern (
                  phead <- ph,
                  ptoken <- 'statement_list',
                  ptext <- spt
            ),
            ph : RSL!PatternHead (
                  name <- 'before_advice_stmt'
            ),
            spt : RSL!SimplePatternText (
```

```
                ptext <- s.advStmt->iterate( e; acc : String = '' |
                                        acc + if acc = '' then '' else
                                          '\r\n\t' endif + e.stmt )
            )
}

rule GenerateAfterAdviceDummy extends BeforeAdvice2Pattern {
      from
            s : APascal!BeforeAdvice (
                  not s.aspect.advice->exists(e |
                                    e.oclIsKindOf(APascal!AfterAdvice))
            )
      to
            t : RSL!Pattern,
            at : RSL!Pattern (
                  rsl <- s.aspect,
                  phead <- aph,
                  ptoken <- 'statement_list',
                  ptext <- aspt
            ),
            aph : RSL!PatternHead (
                  name <- 'after_advice_stmt'
            ),
            aspt : RSL!SimplePatternText (
                  ptext <- ''
            )
}

rule AfterAdvice2Pattern {
      from
            s : APascal!AfterAdvice
      to
            t : RSL!Pattern (
                  phead <- ph,
                  ptoken <- 'statement_list',
                ptext <- spt
            ),
            ph : RSL!PatternHead (
                  name <- 'after_advice_stmt'
            ),
            spt : RSL!SimplePatternText (
                  ptext <- s.advStmt->iterate(e; acc : String = '' |
                                        acc + if acc = '' then '' else
                                         '\r\n\t' endif + e.stmt)
            )
}

rule GenerateBeforeAdviceDummy extends AfterAdvice2Pattern{
      from
            s : APascal!AfterAdvice (
                  not s.aspect.advice->exists(e |
                            e.oclIsKindOf(APascal!BeforeAdvice))
            )
      to
            t : RSL!Pattern,
            bt: RSL!Pattern (
                  rsl <- s.aspect,
```

```
                phead <- bph,
                ptoken <- 'statement_list',
                ptext <- bspt
        ),
        bph : RSL!PatternHead (
                name <- 'before_advice_stmt'
        ),
        bspt : RSL!SimplePatternText (
                ptext <- ''
        )
}

lazy rule PointCutExprToInitExpr {
        from
                s : APascal!Expression
        to
                t : RSL!Pattern (
                        ptext <- spt,
                        ptoken <- 'NATURAL_NUMBER',
                        phead <- ph
                ),
                ph : RSL!PatternHead (
                        name <-   'init'
                ),
                spt : RSL!SimplePatternText (
                        ptext <-  if
                                  s.loopStmt.loopInitCondition.condition.
                                        toString() = '*'  then
                                     '123456789'
                                else
                                  s.loopStmt.loopInitCondition.condition.
                                        toString()
                                endif
                )

}

lazy rule PointCutExprToWithinCode {
        from
                s : APascal!Expression
        to
                t : RSL!Pattern (
                        ptext <- spt ,
                        ptoken <- 'IDENTIFIER',
                        phead <- ph
                ),
                ph : RSL!PatternHead (
                        name <-   'within_code'
                ),
                spt : RSL!SimplePatternText (
                        ptext <- if s.pointcut.pctexpr.size() > 1 then
                               ' '+ s.funcOrProcSig.name
                               else ' ' + 'wc_' endif

                )
}
```

```
lazy rule PointCutExprToExitExpr {
      from
            s : APascal!Expression
      to
            t : RSL!Pattern (
                  ptext <- spt,
                  ptoken <- 'NATURAL_NUMBER',
                  phead <- ph
            ),
            ph : RSL!PatternHead (
                  name <-   'exit'
            ),
            spt : RSL!SimplePatternText (
                  ptext <-  if
                              s.loopStmt.loopInitCondition.condition.
                                    toString() = '*'  then
                                       '123456789'
                           else
                              s.loopStmt.loopInitCondition.condition.
                                    toString()
                           endif
            )
}

lazy rule PointCutToExternalPattern {
      from
            s : APascal!Pointcut
      to
            t : RSL!ExternalPattern (
                  dname <-  'ObjectPascal',
                  eptext <- 'around_advice_for',
                  ptoken <- 'ObjectPascal',
                  phead <- ph
            ),
            ph : RSL!PatternHead (
                  name <-   'around_advice_for',
                  params <- Sequence {pp1,pp2,pp3,pp4,pp5,pp6}
            ),
            pp1 : RSL!PatternParameter (
                  name <- 'program',
                  referTo <- 'ObjectPascal'
            ),
            pp2 : RSL!PatternParameter (
                  name <- 'proceed_bef',
                  referTo <- 'statement_list'
            ),
            pp3 : RSL!PatternParameter (
                  name <- 'proceed_after',
                  referTo <- 'statement_list'
            ),
            pp4 : RSL!PatternParameter (
                  name <- 'withincode',
                  referTo <- 'IDENTIFIER'
            ),
            pp5 : RSL!PatternParameter (
                  name <- 'init',
                  referTo <- 'NATURAL_NUMBER'
```

```
		),
		pp6 : RSL!PatternParameter (
			name <- 'exit',
			referTo <- 'NATURAL_NUMBER'
		)
}

lazy rule PointCutToRule {
	from
		s : APascal!Pointcut
	to
		t : RSL!Rule (

				rname <- s.name,
				type <-   'ObjectPascal',
				params <- Sequence {rp1},
				r_lhs_pattern <- rlhsp,
				r_rhs_pattern <- rrhsp
			),
			rp1: RSL!RuleParameter (
				name <- 'program',
				referTo <- 'ObjectPascal'
			),
			rlhsp : RSL!RuleLHS (
				ruletext <-   irt
			),
			irt : RSL!IDRuleText (
				text <- 'program'
			),
			rrhsp : RSL!RuleRHS (
				ruletext <-   crt,
				condition <- Sequence {rcon}
			),
			crt : RSL!ComplexRuleText (
				pref <- rule_rhs_pattern
			),
			rule_rhs_pattern : RSL!PatternRef (
				name <-   'around_advice_for',
				params <-  Sequence {
							param1,
							param2,
							param3,
							param4,
							param5,
							param6
						}
			),
			param1 : RSL!RealParameter (
					name <- 'program'
			),
			param2 : RSL!PatternRef (
					name <- 'before_advice_stmt'
			),

			param3 : RSL!PatternRef (
					name <- 'after_advice_stmt'
			),
```

```
                    param4 : RSL!PatternRef (
                                name <- 'within_code'
                    ),
                    param5 : RSL!PatternRef (
                                name <- 'init'
                    ),
                    param6 : RSL!PatternRef (
                                name <- 'exit'
                    ),
                    rcon :  RSL!RuleNotEqCondition (
                                lhs <- 'program',
                                pref <- rule_rhs_cond
                    ),
                    rule_rhs_cond : RSL!PatternRef (
                                name <-  'around_advice_for',
                                params <- Sequence {p1,p2,p3,p4,p5,p6}
                    ),
                    p1 : RSL!RealParameter (
                                name <- 'program'
                    ),
                    p2 : RSL!PatternRef (
                                name <- 'before_advice_stmt'
                    ),

                    p3 : RSL!PatternRef (
                                name <- 'after_advice_stmt'
                    ),
                    p4 : RSL!PatternRef (
                                name <- 'within_code'
                    ),
                    p5 : RSL!PatternRef (
                                name <- 'init'
                    ),
                    p6 : RSL!PatternRef (
                                name <- 'exit'
                    )
}
```

D.3. ATL Rule for Translating Loop Expression Join Point in Aspect FORTRAN

The following ATL rule shows the complete specification for translating a loop expression join point in Aspect FORTRAN to RSL code.

```
module AFortran2RSL;

create OUT : RSL from IN : AFortran;

helper context String
    def: startsWith(s : String) : Boolean =
        s.size() <= self.size() and self.substring(1, s.size()) = s;
```

```
helper context String
    def: endsWith(s : String) : Boolean =
        let start : Integer = self.size() - s.size() + 1 in
            start > 0 and self.substring(start, self.size()) = s;


rule AFortran2RSL {
      from
            s:    AFortran!AFortran
      to
            t:    RSL!RSL (
                  domain <-  s.domain,
                  rslelems <- Sequence {
                              s.advice,
                              thisModule.PointCutExprToInitExpr(
                              s.pointcut->first().pctexpr->first()),
                              thisModule.PointCutExprToExitExpr(
                              s.pointcut->first().pctexpr->first()),
                              thisModule.PointCutExprToWithinCode(
                              s.pointcut->first().pctexpr->last()),
                              s.pointcut->collect(e |
                              thisModule.PointCutToExternalPattern(e)),
                              s.pointcut->collect(e |
                              thisModule.PointCutToRule(e))
                              },
                  ruleset <- rs
            ),
            rs : RSL!RuleSet (
                  rsname <- s.name,
                  rname <- s.pointcut->collect(e|e.name)
            )
}

rule AFDomain2RSLDomain {
      from
            s : AFortran!Domain
      to
            t : RSL!Domain (
                  dname <- s.name
            )
}

rule BeforeAdvice2Pattern {
      from
            s : AFortran!BeforeAdvice
      to
            t : RSL!Pattern (
                  phead <- ph,
                  ptoken <- 'execution_part_construct_list',
                  ptext <- spt
            ),
            ph : RSL!PatternHead (
                  name <- 'before_advice_stmt'
            ),
            spt : RSL!SimplePatternText (
                  ptext <- s.advStmt->iterate(
                              e; acc : String = '' | acc + if acc
```

```
                                        = '' then '' else '\r\n\t' endif
                                            + e.stmt
                                          )
                )
}

rule GenerateAfterAdviceDummy extends BeforeAdvice2Pattern {
      from
            s : AFortran!BeforeAdvice (
                  not s.aspect.advice->exists(e |
                              e.oclIsKindOf(AFortran!AfterAdvice))
            )
      to
            t : RSL!Pattern,
            at : RSL!Pattern (
                  rsl <- s.aspect,
                  phead <- aph,
                  ptoken <- 'execution_part_construct_list',
                  ptext <- aspt
            ),
            aph : RSL!PatternHead (
                  name <- 'after_advice_stmt'
            ),
            aspt : RSL!SimplePatternText (
                  ptext <- ''
            )
}

rule AfterAdvice2Pattern {
      from
            s : AFortran!AfterAdvice
      to
            t : RSL!Pattern (
                  phead <- ph,
                  ptoken <- 'execution_part_construct_list',
                  ptext <- spt
            ),
            ph : RSL!PatternHead (
                  name <- 'after_advice_stmt'
            ),
            spt : RSL!SimplePatternText (
                  ptext <- s.advStmt->iterate(
                                    e; acc : String = '' | acc + if acc
                                      = '' then '' else '\r\n\t' endif
                                          + e.stmt
                                      )
            )
}

rule GenerateBeforeAdviceDummy extends AfterAdvice2Pattern{
      from
            s : AFortran!AfterAdvice (
                  not s.aspect.advice->exists(e |
                              e.oclIsKindOf(AFortran!BeforeAdvice))
            )
      to
            t : RSL!Pattern,
```

```
            bt: RSL!Pattern (
                  rsl <- s.aspect,
                  phead <- bph,
                  ptoken <- 'execution_part_construct_list',
                  ptext <- bspt
            ),
            bph : RSL!PatternHead (
                  name <- 'before_advice_stmt'
            ),
            bspt : RSL!SimplePatternText (
                  ptext <- ''
            )
}

lazy rule PointCutExprToInitExpr {
      from
            s : AFortran!Expression
      to
            t : RSL!Pattern (
                  ptext <- spt,
                  ptoken <- 'LABEL',
                  phead <- ph
            ),
            ph : RSL!PatternHead (
                  name <-   'init'
            ),
            spt : RSL!SimplePatternText (
            ptext <-    if
                           s.loopStmt.loopInitCondition.condition.
                              toString() = '*'  then
                                 '12345'
                              else
                                s.loopStmt.loopInitCondition.condition.
                                   toString()
                              endif
            )

}

lazy rule PointCutExprToExitExpr {
      from
            s : AFortran!Expression
      to
            t : RSL!Pattern (
                  ptext <- spt,
                  ptoken <- 'LABEL',
                  phead <- ph
            ),
            ph : RSL!PatternHead (
                  name <-   'exit'
            ),
            spt : RSL!SimplePatternText (
            ptext <-    if
                           s.loopStmt.loopInitCondition.condition.
                              toString() = '*'  then
                                 '12345'
                              else
```

```
                                    s.loopStmt.loopInitCondition.condition.
                                       toString()
                                endif
            )

}

lazy rule PointCutExprToWithinCode {
      from
            s : AFortran!Expression
      to
            t : RSL!Pattern (
                  ptext <- spt ,
                  ptoken <- 'NAME',
                  phead <- ph
            ),
            ph : RSL!PatternHead (
                  name <-   'within_code'
            ),
            spt : RSL!SimplePatternText (
            ptext <- if s.pointcut.pctexpr.size() > 1 then
                        ' '+ s.func_sub_Sig.name
                        else ' ' + 'wc_' endif

            )
}

lazy rule PointCutToExternalPattern {
      from
            s : AFortran!Pointcut
      to
            t : RSL!ExternalPattern (
                  dname <-  'FORTRAN',
                  eptext <- 'around_advice_do',
                  ptoken <- 'Fortran90_program',
                  phead <- ph
            ),
            ph : RSL!PatternHead (
                  name <-   'around_advice_do',
                  params <- Sequence {pp1,pp2,pp3,pp4,pp5,pp6}
            ),
            pp1 : RSL!PatternParameter (
                  name <- 'program',
                  referTo <- 'Fortran90_program'
            ),
            pp2 : RSL!PatternParameter (
                  name <- 'proceed_bef',
                  referTo <- 'execution_part_construct_list'
            ),
            pp3 : RSL!PatternParameter (
                  name <- 'proceed_after',
                  referTo <- 'execution_part_construct_list'
            ),
            pp4 : RSL!PatternParameter (
                  name <- 'withincode',
                  referTo <- 'NAME'
            ),
```

```
            pp5 : RSL!PatternParameter (
                  name <- 'init',
                  referTo <- 'LABEL'
            ),
            pp6 : RSL!PatternParameter (
                  name <- 'exit',
                  referTo <- 'LABEL'
            )
}

lazy rule PointCutToRule {
      from
            s : AFortran!Pointcut
      to
            t : RSL!Rule (

                        rname <- s.name,
                        type <-   'Fortran90_program',
                        params <- Sequence {rp1},
                        r_lhs_pattern <- rlhsp,
                        r_rhs_pattern <- rrhsp
                  ),
                  rp1: RSL!RuleParameter (
                        name <- 'program',
                        referTo <- 'Fortran90_program'
                  ),
                  rlhsp : RSL!RuleLHS (
                        ruletext <-   irt -- IDRuleText
                  ),
                  irt : RSL!IDRuleText (
                        text <- 'program'
                  ),
                  rrhsp : RSL!RuleRHS (
                        ruletext <-   crt, -- ComplexRuleText
                        condition <- Sequence {rcon}
                  ),
                  crt : RSL!ComplexRuleText (
                        pref <- rule_rhs_pattern
                  ),
                  rule_rhs_pattern : RSL!PatternRef (
                        name <-   'around_advice_do',
                        params <- Sequence {
                                          param1,
                                          param2,
                                          param3,
                                          param4,
                                          param5,
                                          param6
                                        }
                  ),
                  param1 : RSL!RealParameter (
                              name <- 'program'
                  ),
                  param2 : RSL!PatternRef (
                              name <- 'before_advice_stmt'
                  ),
```

```
                    param3 : RSL!PatternRef (
                                name <- 'after_advice_stmt'
                    ),
                    param4 : RSL!PatternRef (
                                name <- 'within_code'
                    ),
                    param5 : RSL!PatternRef (
                                name <- 'init'
                    ),
                    param6 : RSL!PatternRef (
                                name <- 'exit'
                    ),
                    rcon :  RSL!RuleNotEqCondition (
                                lhs <-  'program',
                                pref <-  rule_rhs_cond
                    ),
                    rule_rhs_cond : RSL!PatternRef (
                                name <-   'around_advice_do,
                                params <-  Sequence {p1,p2,p3,p4,p5,p6}
                    ),
                    p1 : RSL!RealParameter (
                                name <- 'program'
                    ),
                    p2 : RSL!PatternRef (
                                name <- 'before_advice_stmt'
                    ),

                    p3 : RSL!PatternRef (
                                name <- 'after_advice_stmt'
                    ),
                    p4 : RSL!PatternRef (
                                name <- 'within_code'
                    ),
                    p5 : RSL!PatternRef (
                                name <- 'init'
                    ),
                    p6 : RSL!PatternRef (
                                name <- 'exit'
                    )
}
```

APPENDIX E

BACK-END WEAVER TRANSFORMATION FUNCTIONS

The PARLANSE external functions that are reusable or shared among multiple weavers are shown in Appendix E.1.

E.1. PARLANSE Reusable External Functions

The following PARLANSE external functions are useful for traversing the AST and match conditions specified by the RSL. The function `name_begins_with` is useful for matching names (e.g., function name, identifiers), which begin with the given input. This is equivalent to a *wildcard* search `name*` in an aspect program.

```
(define name_begins_with
    (lambda Registry:MatchingCondition
         (let (;; (= [search_string (reference string)]
                   (Graph:HGHandling:GetString arguments:1))
                   [start_index natural]
                  );;
                  (value
                      (;;
                        (= start_index
                          (Strings:Find
                            (AST:GetString arguments:1)
                            (AST:GetString arguments:2)))
                        (ifthen(== 1 start_index)
                          (return ~t)
                        )ifthen
                      );;
                     ~f
                 )value
         )let
    )lambda
)define
```

The function `name_contains` is useful for matching names (e.g., method name, identifiers), which include the given input. This is equivalent to a *wildcard* search `*name*` in an aspect program.

```
(define name_contains
      (lambda Registry:MatchingCondition
            (let (;;
                 (= [search_string (reference string)]
                 (Graph:HGHandling:GetString arguments:1))

                   [start_index natural]
               );;
               (value
                  (;;
                     (= start_index
                       (Strings:Find
                         (AST:GetString arguments:1)
                         (AST:GetString arguments:2)))
                     (ifthen(> start_index 0)
                       (return ~t)
                      )ifthen
                  );;
                  ~f
            )value
            )let
      )lambda
  )define
```

The following external function `GetChildFromParent` is a helper routine useful for finding a child node with a given property from the parent node.

```
(define GetChildFromParent
    (lambda (function AST:Node
              (structure
                [parent_tree AST:Node]
                [node_type natural]
              )structure
              critical
             )function
    (let [child_node AST:Node]
       (value
          (;;
             (= child_node (AST:FindChildWithProperty
               parent_tree
                (lambda (function boolean AST:Node
                )function
                  (value (local (;; );;
                     (;;
                       (ifthen
                        (== (AST:GetNodeType ?) node_type)
                        (return ~t)
                       )ifthen
                       (return ~f)
                     );;
```

```
                    )local
                    ~f
                    )value
                  )lambda
                )
              )
            );;
           child_node
         )value
       )let
  )lambda
)define
```

The external function `GetParentFromChild` is a helper routine useful for finding a parent node with a given property from the child node.

```
(define GetParentFromChild
   (lambda (function AST:Node
              (structure
                 [child_tree AST:Node]
                 [node_type natural]
              )structure
              critical
            )function
   (let [parent_node AST:Node]
      (value
          (;;
           (= parent_node
             (AST:FindParentWithProperty  child_tree
               (lambda (function boolean AST:Node
               )function
               (value (local (;; );;
                   (;;
                     (ifthen
                      (== (AST:GetNodeType ?) node_type)
                      (return ~t)
                     )ifthen
                     (return ~f)
                   );;
                  )local
                  ~f
                )value
               )lambda
              )
             )
          );;
          parent_node
        )value
      )let
  )lambda
)define
```

The PARLANSE external functions that are algorithm-specific or grammar-dependent are shown in Appendix E.2. Reuse among these functions is more at the conceptual level.

E.2. PARLANSE External Function for Loop Execution Join Point

The two external functions presented in Appendix E.2 show the internal details for implementing a `loop execution join point` for Object Pascal (`for_loop`) and FORTRAN (`do_loop`), respectively. Both these functions follow similar search algorithms. Thus, in order to implement a `loop execution join point` for a new language `L`, the external function for `L` should also follow the same conceptual algorithm and abstract program structure as used by the previous weavers. A careful observation reveals the dependency on the underlying grammar symbols (e.g., `GrammarConstants:NodeTypes:_for_statement_1`, `GrammarConstants:NodeTypes:_block_do_construct_2`) for the two external functions. However, the algorithm remains conceptually the same.

```
(define around_advice_for
  (lambda Registry:CreatingPattern
      (value (local (;;
       (= [proc_def_node AST:Node] AST:VoidNode)
       (= [func_def_node AST:Node] AST:VoidNode)
       (= [for_stmt_node AST:Node] AST:VoidNode )
       (= [init_node AST:Node] AST:VoidNode)
       (= [exit_node AST:Node] AST:VoidNode)
       (= [slist_node AST:Node] AST:VoidNode)
       [parent AST:Node]
       [representation_instance AST:RepresentationInstance]
       [new_node AST:Node]
       [new_node_1 AST:Node]
       [empty_node AST:Node]
       [semicolon AST:Node]
       (= [ctr natural] 0)
       (= [withincode_node AST:Node] AST:VoidNode)
     );;
```

```
(;;
(= representation_instance
(AST:GetForestRepresentationInstance
(AST:GetForest arguments:1 )
(AST:GetRepresentation arguments:1)))
(= empty_node (AST:CreateNode representation_instance
GrammarConstants:NodeTypes:_empty_statement_1))
(= semicolon (AST:CreateNode representation_instance  453))

(AST:ScanTreeNodes  arguments:1
  (lambda (function boolean AST:Node
  )function
   (value (local (;; );;
        (;;
        (ifthen (== (AST:GetNodeType ?)
         GrammarConstants:NodeTypes:_IDENTIFIER)
          (ifthen (== (@(AST:GetString ?))
           (@(AST:GetString arguments:4)))
            (;;
            (= proc_def_node
            (GetParentFromChild
            ? GrammarConstants:NodeTypes:
            _implementation_declaration_6))
            (= func_def_node
            (GetParentFromChild
            ? GrammarConstants:NodeTypes:
            _implementation_declaration_7))
            );;
          )ifthen
        )ifthen
        (return ~t)
        );;
       )local
       ~t
      )value
    )lambda
  )

 (ifthenelse(~= proc_def_node AST:VoidNode)
 (= withincode_node proc_def_node)
 (ifthen(~= func_def_node AST:VoidNode)
  (= withincode_node func_def_node)
 )ifthen
 )ifthenelse

 (ifthen (== withincode_node AST:VoidNode)
 (= withincode_node arguments:1)
 )ifthen

 (AST:ScanTreeNodes  withincode_node
 (lambda (function boolean AST:Node
 )function
 (value (local (;; );;
   (;;
     (ifthen (== (AST:GetNodeType ?)
     GrammarConstants:NodeTypes:_for_statement_1)
     (;;
```

```
(= init_node
(GetChildFromParent
(AST:GetNthChild ? 4)
GrammarConstants:NodeTypes:_NATURAL_NUMBER))
(= exit_node
(GetChildFromParent
(AST:GetNthChild ? 6)
GrammarConstants:NodeTypes:_NATURAL_NUMBER))

(ifthen (!!
            (!! (&&
                (== (AST:GetNatural init_node)
                (AST:GetNatural arguments:5))
                  (== (AST:GetNatural exit_node)
                  (AST:GetNatural arguments:6))
                )&&
                (&& (== 123456789
                (AST:GetNatural arguments:5))
                  (== (AST:GetNatural exit_node)
                  (AST:GetNatural arguments:6))
                )&&
            )!!
                (&& (== (AST:GetNatural init_node)
                (AST:GetNatural arguments:5))
                (== 123456789
                (AST:GetNatural arguments:6))
                )&&
       )!!
  (;;
  (= ctr (+ ctr 1))
  (= for_stmt_node AST:VoidNode)
  (= for_stmt_node  (GetParentFromChild
       init_node GrammarConstants:
       NodeTypes:_for_statement_1))
  (= slist_node (GetParentFromChild  ?
   GrammarConstants:NodeTypes:_statement_list_2))

  (ifthen (~= for_stmt_node AST:VoidNode)
  (;;
    (= new_node
    (AST:CreateNode representation_instance
    GrammarConstants:NodeTypes:_statement_list_2))
    (= new_node_1
    (AST:CreateNode representation_instance
    GrammarConstants:NodeTypes:_statement_list_2))
    (= parent (AST:GetParent for_stmt_node))
    (AST:ConnectNthChild new_node 1 arguments:2)
    (AST:ConnectNthChild new_node 2 empty_node )
    (AST:ConnectNthChild new_node 3 new_node_1)
    (AST:ConnectNthChild new_node_1 1 for_stmt_node )
    (AST:ConnectNthChild new_node_1 2 semicolon )
    (AST:ConnectNthChild new_node_1 3 arguments:3 )
    (AST:ReplaceNthChild parent 1 new_node)
    (AST:ReplaceNthChild slist_node 2 empty_node)
   );;
   )
  );;
```

```
            )ifthen
            );;
          )ifthen
        (return ~t)
        );;
       )local
      ~t
      )value
     )lambda
    )
   (return arguments:1)
    );;
    )local
    (void AST:Node)
  )value
 )lambda
)define
```

The following external function is used to implement a `do loop execution` join point in FORTRAN. It is similar to the previous function, but instead of `for loop` (as in case of Object Pascal), it searches for `do loop` in FORTRAN programs.

```
(define around_advice_do
  (lambda Registry:CreatingPattern
   (value (local (;;
   (= [sub_def_node AST:Node] AST:VoidNode)
   (= [func_def_node AST:Node] AST:VoidNode)
   (= [do_stmt_node AST:Node] AST:VoidNode)
   (= [init_node AST:Node] AST:VoidNode)
   (= [exit_node AST:Node] AST:VoidNode)
   (= [withincode_node AST:Node] AST:VoidNode)
   [parent AST:Node]
   [representation_instance AST:RepresentationInstance]
   [new_node_1 AST:Node]
   [new_node_2 AST:Node]

   );;
   (;;
    (= representation_instance
    (AST:GetForestRepresentationInstance
    (AST:GetForest arguments:1 )
    (AST:GetRepresentation arguments:1)))

    (AST:ScanTreeNodes  arguments:1

    (lambda (function boolean AST:Node
    )function
```

```
 (value (local (;; );;
    (;;
      (ifthen (==
      (AST:GetNodeType ?)
      GrammarConstants:NodeTypes:_NAME)
      (ifthen (==
      (@(AST:GetString ?))
      (@(AST:GetString arguments:4)))
      (;;
         (= sub_def_node
         (GetParentFromChild  ?
         GrammarConstants:NodeTypes:
         _subroutine_subprogram_1))
         (= func_def_node
         (GetParentFromChild  ?
         GrammarConstants:NodeTypes:
         _function_subprogram_1))
       );;
      )ifthen
     )ifthen
     (return ~t)
     );;
   local
   ~t
 )value
)lambda

(ifthenelse(~= sub_def_node AST:VoidNode)
  (= withincode_node sub_def_node)
  (ifthen(~= func_def_node AST:VoidNode)
     (= withincode_node func_def_node)
  )ifthen
)ifthenelse
(ifthen (== withincode_node AST:VoidNode)
(= withincode_node arguments:1)
)ifthen

(AST:ScanTreeNodes  withincode_node
    (lambda (function boolean AST:Node
    )function
     (value (local (;; );;
      (;;
          (ifthen (== (AST:GetNodeType ?)
           GrammarConstants:NodeTypes:_loop_control_1)
            (;;
               (= init_node (GetChildFromParent
               (AST:GetNthChild ? 4)
               GrammarConstants:NodeTypes:_LABEL))
               (= exit_node (GetChildFromParent
               (AST:GetNthChild ? 6)
               GrammarConstants:NodeTypes:_LABEL))
               (ifthen
               (!!
                 (!!
                    (&& (== (AST:GetNatural init_node)
                        (AST:GetNatural arguments:5))
                        (== (AST:GetNatural exit_node)
```

```
                        (AST:GetNatural arguments:6))
                    )&&
                    (&& (== 12345 (AST:GetNatural arguments:5))
                        (== (AST:GetNatural exit_node)
                        (AST:GetNatural arguments:6))
                    )&&
                   )!!
                    (&&  (== (AST:GetNatural init_node)
                        (AST:GetNatural arguments:5))
                        (== 12345 (AST:GetNatural arguments:6))
                    )&&
                  )!!
                  (;;
                    (ifthen (~= init_node AST:VoidNode)
                      (= do_stmt_node  (GetParentFromChild
                      init_node GrammarConstants:NodeTypes:
                      _block_do_construct_2))
                    )ifthen

                    (ifthen (~= do_stmt_node AST:VoidNode)
                      (;;
                       (= new_node_1 (AST:CreateNode
                       representation_instance
                       GrammarConstants:NodeTypes:
                       _execution_part_construct_list_2))
                       (= new_node_2 (AST:CreateNode
                       representation_instance
                       GrammarConstants:NodeTypes:
                       _execution_part_construct_list_2))
                       (= parent (AST:GetParent do_stmt_node))

                       (AST:ConnectNthChild new_node_1 1 arguments:2)
                       (AST:ConnectNthChild new_node_1 2 new_node_2)
                       (AST:ConnectNthChild new_node_2 1 do_stmt_node)
                       (AST:ConnectNthChild new_node_2 2 arguments:3)
                       (AST:ReplaceNthChild parent 1 new_node_1)
                      );;
                      )
                    );;
                   )ifthen
                );;
              )ifthen
          (return ~t)
          );;
          )local
         ~t
          )value
         )lambda
        )
        (return arguments:1)
        );;
        )local
        (void AST:Node)
      )value
  )lambda
)define
```

APPENDIX F

DMS PARLANSE FUNCTIONS TO SPECIALIZE HPL

The DMS PARLANSE functions required to specialize HPL are shown in this Appendix.

F.1. PARLANSE External Function to Remove Macro Definitions from HPL

The following PARLANSE external function is used to remove macro definitions from HPL. Note that the function is actually called from the RSL shown in Figure 3-32.

```
(define remove_macro
  (lambda Registry:CreatingPattern
      (value (local (;;
       [empty_node AST:Node]
       [representation_instance AST:RepresentationInstance]
       [search_string (reference string)]
       [comment_string string]
       [out_file_name string]
       [dir_name string]
       [if_dir_node AST:Node]
       [scanner StringScan:Scan]
       [last_index natural]
       [first_index natural]
       [search_node AST:Node]
       [comments CommentHashTree:SequenceOfComments]
       [check_string string]
       [rem_comments string]
       [flag boolean]
       (= [output_stream OutputStream:OutputStream]
       OutputStream:VoidOutputStream)
      );;
      (;;
       (= out_file_name (@ (AST:GetAbstractFileName arguments:1)))
       (= first_index (Strings:LastIndex (. out_file_name) "\"))
       (= dir_name (Strings:Segment (. out_file_name) 1 first_index))
       (= dir_name (concatenate dir_name (@
       (AST:GetString arguments:2))))
       (= last_index (Strings:Find (. out_file_name) (.`.c')))
       (= out_file_name (Strings:Segment
       (. out_file_name) first_index last_index))
       (= out_file_name (concatenate out_file_name
       (@ (AST:GetString arguments:2))))
       (= out_file_name (concatenate dir_name out_file_name))
       (= output_stream (OutputStream:OpenFile (. out_file_name)))
```

```
(AST:ScanTreeNodes  arguments:1

(lambda (function boolean AST:Node
)function
 (value (local (;; );;
       (;;
       (= flag ~f)
       (= rem_comments `')
       (= comment_string `')
       (= representation_instance
       (AST:GetForestRepresentationInstance
       (AST:GetForest arguments:1 )
       (AST:GetRepresentation arguments:1)))
       (= empty_node (AST:CreateNode representation_instance
       GrammarConstants:NodeTypes:_identifier))
       (ifthen(== ~t (AST:ContainsString ?))
       (;;
       (= search_string (AST:GetString ?))
       (ifthen (== (@ search_string) (@
       (AST:GetString arguments:2)))
       (;;

       (= search_node (AST:GetParent (AST:GetParent ?)))
       (= if_dir_node (AST:GetFirstChild
       (AST:GetFirstChild search_node)))

       (ifthen(== ~t (AST:HasPreComments if_dir_node))
          (;;
            (= flag ~t)
            (= comments (AST:GetPreComments if_dir_node))

            (do [c natural] 1 (coerce natural
            (upperbound (@ comments) 1)) 1
            (;;
            (= rem_comments (concatenate rem_comments
            comments:c:CommentString))
            (= rem_comments (append rem_comments "~l"))
            (= check_string (Strings:Segment
            (. comments:c:CommentString) 1 12))
            (ifthen(== check_string `//>>>>>>HPL_')
              (;;
               (= rem_comments
               (append rem_comments "~s"))
               (= rem_comments (append rem_comments "~s"))
               (= rem_comments (append rem_comments "~s"))
               (= rem_comments (append rem_comments "~s"))
               (= rem_comments (append rem_comments "~s"))
               (= rem_comments (append rem_comments "~s"))
              );;
            )ifthen':
            );;
           )do
         );;
        )ifthen

        (= comment_string (concatenate `//>>>>>>'
        (@ search_string)))
```

```
                (= comment_string
                (concatenate comment_string
                `<<<<<<<<<MARKER>>>>>>>>>'))
                (ifthen(== ~t flag)
                  (;;
                    (= comment_string (concatenate rem_comments
                    comment_string))
                    (AST:SetUnitPreComment if_dir_node (. `'))
                  );;
                )ifthen
                (AST:SetString empty_node (. comment_string ))
                (Registry:Print
                  (. `Cpp~~ISO14882c1998')
                     Registry:DefaultSyntaxTreeDomainRepresentation
                     search_node
                     output_stream)
                (OutputStream:Put output_stream (.`$******$'))
                (OutputStream:PutNewline output_stream)
                (AST:ReplaceTree search_node empty_node)
                (return ~t)
                );;
              )ifthen
            );;
         )ifthen
         );;
        )local
        ~t
      )value
    )lambda
)define
```

F.2. PARLANSE External Function to Specialize HPL

The following PARLANSE external function is used to specialize or add specific macros to the core HPL library based on the requirement (e.g., CBLAS, FBLAS or VSIPL).

```
(define add_macro
  (lambda Registry:CreatingPattern
      (value (local (;;
       [empty_node AST:Node]
       [representation_instance AST:RepresentationInstance]
       [search_comments CommentHashTree:SequenceOfComments]
       [comment_string string]
       [input_file_name string]
       [dir_name string]
       [last_index natural]
       [first_index natural]
```

```
[Line string]
[temp_line string]
[comments CommentHashTree:SequenceOfComments]
[input_stream InputStream:InputStream]
[scanner StringScan:Scan ]
[string_len natural]
[rem_comments string]
[flag boolean]

);;
(;;
(= Line `')
(= flag ~f)
(= comment_string (concatenate `//>>>>>>'
(@ (AST:GetString arguments:2))))
(= comment_string (concatenate comment_string
`<<<<<<<<<MARKER>>>>>>>>>'))
(= input_file_name (@ (AST:GetAbstractFileName arguments:1)))

(= first_index (Strings:LastIndex (. input_file_name) "\"))
(= dir_name (Strings:Segment (. input_file_name) 1 first_index))
(= dir_name (concatenate dir_name (@
(AST:GetString arguments:2))))
(= last_index (Strings:Find (. input_file_name) (.`.c')))

(= input_file_name (Strings:Segment
(. input_file_name) first_index last_index))

(= input_file_name (concatenate input_file_name
(@ (AST:GetString arguments:2))))
(= input_file_name (concatenate dir_name input_file_name))
(= input_stream (InputStream:OpenFileUsingDefaultEncoding
(. input_file_name)))
(try (loop (;;
            (= Line (InputStream:GetLine input_stream))
            (= temp_line (concatenate temp_line Line))
            (= temp_line (append temp_line "~l"))
           );;
     )loop
     (ifthenelse (== (exception) InputStream:EndOfInputStream)
              (acknowledge
                (;;
                  (= temp_line (concatenate temp_line `'))
                  ; make a dummy line to indicate EOF seen
                );;
              ) ; no more input
            (propagate)
     )ifthenelse
)try

(AST:ScanTreeNodes  arguments:1

(lambda (function boolean AST:Node
)function
 (value (local (;; );;
      (;;
      (= Line `')
```

```
            (= flag ~f)
            (ifthen(== ~t (AST:HasPreComments ?))
                (;;
                (= comments (AST:GetPreComments ?))
                (= rem_comments `')
                (do [c natural] 1 (coerce natural
                (upperbound (@ comments) 1)) 1
                  (;;
                    (ifthenelse (== comments:c:CommentString
                        comment_string)
                        (;;
                          (= flag ~t)
                        (= last_index (Strings:Find
                        (. temp_line) (.`$******$')))
                        (= Line (Strings:Segment (. temp_line) 1
                        (-- last_index)))
                        (= Line (concatenate rem_comments Line))
                        (= rem_comments `')
                        (= scanner (StringScan:MakeScan (.
                        temp_line)))
                        (= string_len 0)
                        (while (== (StringScan:End? (. scanner)) ~f)
                          (;;
                          (= string_len (++ string_len))
                          (StringScan:Advance (. scanner))
                          );;
                        )while
                        (= temp_line (Strings:Segment (. temp_line)
                          (+ last_index 8) string_len))
                        );;
                        (;;
                          (= rem_comments (concatenate rem_comments
                           comments:c:CommentString))
                          (= rem_comments (append rem_comments "~l"))
                        );;
                     )ifthenelse
                    );;
                   )do
                  );;
              )ifthen
              (ifthen(== ~t flag)
              (;;
              (= Line (concatenate Line rem_comments))
              (AST:SetUnitPreComment ? (. Line))
              );;
             )ifthen
            );;
           )local
           ~t
          )value
       )lambda
)define
```