ASPECT-ORIENTED DOMAIN-SPECIFIC MODELING: A GENERATIVE APPROACH USING A METAWEAVER FRAMEWORK

By

Jeffrey G. Gray

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

May, 2002

Nashville, Tennessee

Approved:	Date:
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

To Marla,

Many women do noble things, but you surpass them all.

Proverbs 31:29 (NIV)

To Mom and Dad,

The glory of children is their parents.

Proverbs 17:6 (NRSV)

ACKNOWLEDGEMENTS

I have been blessed to have many people in my life that have truly believed in me. My parents, Gene and Joan Gray, have never ceased to provide unbounded opportunities and resources, along with tireless patience and encouragement. I am here today, "All Because Two People Fell in Love." Thanks Mom and Dad – I know that you will always be in the stands cheering me on. To my wife, Marla, as the seasons of this life often change, it's good to know where I belong – my heart has a compass and it always points to you. To all my family – your sustaining support has been the rock to which I have anchored my stability during difficult times. To Scott, Dave, George, Ben, and Wendell – thanks for standing beside me on July 15, 2000.

In my younger years, I was fortunate to have several influential mentors. During my time at the Linsly School, Dr. Garth Innocenti and Mr. Mike Chokel opened my eyes to the excitement and joy of research. Their willingness to give me independence in my various science fair projects and paper competitions allowed me to cultivate the thrill of discovery that motivates me still to this day. At WVU, Dr. Frances Van Scoy and Dr. Murali Sitaraman continued to shape and give guidance to my youthful energy.

To my thesis advisor, Dr. Stephen Schach, you deserve my deepest gratitude for the unwavering devotion that you have exhibited toward my best interests. You have demonstrated the patience of Job when my constant meanderings seemed to take me off-course, yet you always had the wisdom to guide me back to safety. You have been one of my most zealous motivators and have always been there to provide me with a jolt of invigorating enthusiasm.

To my committee members, I am indebted to your willingness to assist me in improving this work. Dr. Fritz Barnes – as my youngest committee member, your fresh outlook concerning this whole process has been very beneficial to me. Your detailed suggestions on earlier drafts have greatly improved the content and style of this dissertation. Dr. Larry Dowdy – as one of my earliest influences on campus, you gave me the perfect role model of what it means to care about students in an extraordinary way. Dr. Mike Fitzpatrick – I have learned to place a high premium of value on the suggestions that you have made to me while at Vanderbilt, especially regarding this research. Dr. Gábor Karsai – your uncommon ability to quickly isolate the essence of a problem, and to then discern a clever solution, has always amazed me. Thank you for allowing me to sit at your feet and learn from you over the past three years. Dr. Janos Sztipanovits – thanks for building a first-class research institute and for letting me play a very small part.

On October 8th, 1997, Gregor Kiczales dazzled me with his keynote presentation at the OOPSLA conference in Atlanta, Georgia. Two years later, at the OOPSLA in Denver, Colorado, Gregor took time out of his day to help me sketch out some of the ideas that would eventually become a part of this dissertation. Gregor, thanks for your enduring support of students, and for the time that you have spent in helping to build my confidence and understanding of how our world might be carved up in new and exciting ways.

I am very grateful for the opportunities that have been provided to me while at ISIS. In particular, I am thankful for the financial support of DARPA and the encouragement of Dr. Doug Schmidt. This work was supported by the DARPA Information Technology Office (DARPA/ITO), under the *Program Composition for Embedded Systems* (PCES) program, Contract Number: F33615-00-C-1695.

ISIS is unlike any other place where I have worked. The quality of research is evident by the success in funding, yet the environment is very collegial. Dr. Ted Bapty and Dr. Sandeep Neema have been both my mentors and colleagues on the DARPA PCES project. Ted and Sandeep, thanks for your patience and for making this such a rewarding experience. You also have been great traveling buddies at the PI meetings.

There are many others who have made my time at ISIS pleasant, while contributing to my growth as a person and young researcher: Dr. James Davis, Dr. Greg Nordstrom, Dr. Ákos Lédeczi, Beatrice Richardson, Lorene Morgan, Michele Codd, and Larry Howard. To my fellow students: Jonathan Sprinkle (my cube neighbor/secretary – ha!), Jason G., Brandon Eames, Jim Nichols, Jason Scott, Mark Briski, and Tal Pasternak – you have helped to make this a place where scholarly research and laughter coexist.

My earlier days at Vanderbilt were surrounded by a lot of friendly faces. Trey Tinnell, Doug Morse, Rob Bland, Sriram Narasimhan, Dr. Tamara and Natasha Balac, Roger Farmer, Dr. Liz Varki, Cindy Childers, Julie Johnson, Dr. Kirsten Whitley, Dr. Ravi Kapadia, and Dr. Manish Madhukar – you all brightened my experience here.

It has been my privilege to be in association with the congregation at the Brentwood church of Christ. Your fellowship and support has continually helped me to keep my focus on the more important things in this journey. To many of the inmates at the Riverbend prison, I have learned the definition of grace by watching the transformations in your lives. Ron, thanks for letting me tag along for the ride. To the ladies at the Brighton Gardens Nursing Home, I know that I have a dozen adopted grandmothers who will always be there to offer a kind smile and a hug.

Thank you, God, for bringing all of these people into my life.

TABLE OF CONTENTS

Appendices

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACS – Adaptive Computing Systems

AO – Aspect Oriented

AOD – Aspect-Oriented Design

AODSM – Aspect-Oriented Domain-Specific Modeling

AOP – Aspect-Oriented Programming

AOSD – Aspect-Oriented Software Development

AP – Adaptive Programming

API – Application Program Interface

ASDL – Abstract Syntax Description Language

ASOC – Advanced Separation of Concerns

AST – Abstract Syntax Tree

ATR – Automatic Target Recognition

C3I – Command, Control, Communication, and Information

CCM – CORBA Component Model

CDL – Contract Description Language

CF – Composition Filters

CLOS – Common Lisp Object System

CORBA – Common Object Request Broker Architecture

CSS – Cascading Style Sheet

DLL – Dynamic Link Library

DOC – Distributed Object Computing

DOM – Document Object Model

DSL – Domain-Specific Language

DSM – Domain-Specific Modeling

DSVL – Domain-Specific Visual Language

DTD – Document Type Definition

ECBS – Engineering of Computer-Based Systems

ECL – Embedded Constraint Language

ECOOP – European Conference on Object-Oriented Programming

GME – Generic Model Editor

GP – Generative Programming

GUI – Graphical User Interface

ICSE – International Conference on Software Engineering

IDE – Integrated Development Environment

IP – Intentional programming

ISIS – Institute for Software Integrated Systems

JSP – JavaServer Pages

JTS – Jakarta Tool Suite

KWIC – Key Word in Context

MCL – Multigraph Constraint Language

MDA – Model-Driven Architecture

MDSOC – Multi-Dimensional Separation of Concerns

MIC – Model-Integrated Computing

MOBIES – Model-Based Integration of Embedded Software

MOP – Metaobject Protocol

NCI – National Compiler Infrastructure

OCL – Object Constraint Language

OMG – Object Management Group

OO – Object Oriented

OOP – Object-Oriented Programming

OOPSLA – Object-Oriented Programming, Systems, Languages, and Applications

PCCTS – Purdue Compiler Construction Tool

PCES – Program Composition for Embedded Systems

QoS – Quality of Service

SOP – Subject-Oriented Programming

StratGen – Strategy Code Generator

SUIF – Stanford University Intermediate Format

UAV – Unmanned Aerial Vehicle

UML – Unified Modeling Language

WCET – Worst Case Execution Time

XML – Extensible Markup Language

XSLT – Extensible Stylesheet Transformations

YACC – Yet Another Compiler-Compiler

CHAPTER I

INTRODUCTION

> *Even for this let us divided live…That by this separation I may give that due to thee which thou deservest alone.*
>
> William Shakespeare, *Sonnet XXXIX* [Bevington, 1997]

In any engineering endeavor, a key requirement is the ability to compose large structures from a set of primitive elements. This is true for children who are constructing toy models of bridges and buildings using Lego™ or Erector™ sets. This is true, on a larger scale, for civil engineers who design and supervise the construction of skyscrapers. This is especially true for software engineers who compose increasingly complex systems from components, classes, and methods.

An important difference between the engineering of software, and the other undertakings enumerated above, is the recognition that the set of available core elements for software construction is often significantly larger. The composition of these elements can be specified at a much finer level of granularity. As a contrast, the "bricks" used to build Lego™ houses, or the steel beams used in the construction of a bridge, come in but a few different shapes and sizes, and are composed using a simple standard interface (e.g., the prong and receptacle parts of a Lego™ block have been unchanged since 1932 [Lego, 2002]; likewise, since around 1850, the standard dimensions for an "air cell" masonry brick in the United States has been 2.5 x 3.75 x 8 inches [Chrysler and Escobar, 2000]).

Furthermore, the compositional permutations and dynamic interactions that are possible with software elements are several orders of magnitude richer than those found in other engineering activities. For example, a generic function can be parameterized with a seemingly unlimited number of other elements (e.g., a template function that can sort any data type using numerous functors). Parametric polymorphism is but one factor that contributes to the exponential state explosion problem that makes the composition of software so difficult. A reason for this complexity is that the essence of software elements is expressed as logical abstractions, as opposed to physical materials, which results in the generation of an enormous state-space that must be tested. In fact, the core of Brooks' "No Silver Bullet" essay is a commentary that the molding of complex conceptual entities is the essence of software construction [Brooks, 1995].

It has been a longstanding understanding among software engineering researchers that the proverbial Gordian knot has appeared as a consequence of the exponential complexities involved in composing a set of software building blocks, or modules. Separation of concerns has emerged at the center of many helpful techniques for loosening the grip of this knot.

Separation of Concerns

Separation of concerns is not a new idea. In fact, over the past quarter-century, issues related to concern separation have been at the heart of the intersection of software engineering and programming language design research. A *concern* is generally defined as some piece of a problem whose isolation as a unique conceptual unit results in a desirable property. Concerns arise as intentional artifacts of a system. They are the primary stimulus for structuring software into localized modules. The *IEEE*

Recommended Practice for Architectural Description of Software-Intensive Systems defines a concern as, "…those interests that pertain to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders. Concerns include system considerations such as performance, reliability, security, distribution, and evolvability" [IEEE 1471, 2000]. Other researchers have defined a concern to be, "any matter of interest in a software system" [Sutton and Rouvellou, 2001], and, "a slice through the problem domain that addresses a single issue" [Nelson et al., 2001]. Concerns are a central point of interest at any stage of the development cycle.

Criteria for Decomposition

> *Abstraction is doing just what our small minds need: making it possible for us to think about important properties of our program – its behavior – without having to think about the entirety of the machinations.*
>
> [Kiczales, 1992]

Modularity, abstraction, information hiding, and variability are important topics in software engineering that are associated with separation of concerns [Schach, 2002]. A clean separation of concerns provides a system developer with more coherent and manageable modules. From the structured paradigm of the 1960s and 1970s, to the Object-Oriented (OO) paradigm of the past few decades, there has always been an interest in creating new abstraction mechanisms that provide improved separation of concerns. There are several new paradigms on the horizon, as will be discussed in the next subsection ("Advanced Separation of Concerns"), to assist in further separation.

The most influential paper related to the study of modularization, and perhaps even in all of software engineering, is David Parnas' "On the Criteria to Be Used in

Decomposing Systems into Modules" [Parnas, 1972]. Parnas' criteria aid a designer in achieving module independence. Parnas recognized that the decomposition of a system into its constituent parts must be performed with several specific goals in mind. To illustrate the consequences and tradeoffs from different design decisions, Parnas introduced a simple indexing program called KWIC ("Key Word in Context"). From a comparison of two separate modularizations for KWIC, Parnas suggested that modules be composed with the following objectives: changeability, independent development, and comprehensibility. The criterion of information hiding was shown by Parnas to be important in all three of these objectives.

Changeability

> *The way to evaluate a modular decomposition, particularly one that claims to rest on information hiding, is to ask what changes it accommodates.*
>
> [Hoffman and Weiss, 2001]

A change to a module should not necessitate numerous invasive changes to many other modules. Parnas' work has revealed that the structure of a system has a direct effect on the cost of change and maintenance. The potential that a module will undergo change should always be kept in mind when considering several different possibilities for modularization. Those implementation decisions that have the possibility of being changed, or those decisions that offer the most degree of flexibility in adaptation, should be hidden from the client of that module. This observation was key toward the discovery of the properties of encapsulation and information hiding, where abstraction is the principal idea for delimiting the "what" from the "how." Designs that are created with the principle of information hiding permit the substitution of different implementations for

the same abstraction. This improves the capacity to make changes based upon different desiderata (e.g., the typical "time versus space" arguments in data structure implementation).

Independent Development

> *Modularity is about separation: When we worry about a small set of related things, we locate them in the same place. This is how thousands of programmers can work on the same source code and make progress.*
>
> [Gabriel and Goldman, 2000]

As the complexity and size of a software system soars, the ability of developers to independently work on separate modules becomes increasingly important. This is a vital attribute of the open-source community, where multiple developers work independently on a common collection of source code. The task of modularization, then, turns out to be a type of work assignment for each developer. The details of the design decisions and responsibilities of each developer should be hidden behind an exposed abstract interface. The interface supplies the only means of access to the services offered by the module.

Comprehensibility

> *In many pieces of code the problem of disorientation is acute. People have no idea what each component of the code is for and they experience considerable mental stress as a result.*
>
> [Gabriel, 1995]

> *When Microsoft first began conducting usability studies in the late 1980s to figure out how to make their products easier to use, their researchers found that 6 to 8 out of 10 users couldn't understand the user interface and get to most of the features.*
>
> [Maguire, 1994]

Comprehensibility can be negatively affected, within any context, by a poorly designed interface. Comprehensibility is a major goal of modular reasoning; that is, it should be possible for a developer to study one module at a time without being overwhelmed with the details of extraneous implementation information defined outside of the module context. Several popular ideas in software engineering (e.g., Dijkstra's "Go To Statement Considered Harmful" [Dijkstra, 1968], and Wulf and Shaw's "Global Variables Considered Harmful" [Wulf and Shaw, 1973]), were in fact arguments made from the perspective of comprehensibility. An early result of object-oriented research demonstrated a strong link between comprehensibility and low coupling [Lieberherr and Holland, 1989].

Cohesion and Coupling

An obvious connection exists between highly cohesive and lowly coupled modules, and the objectives identified by Parnas. The seminal definitions of cohesion and coupling were provided within the context of structured design [Stevens et al., 1974]. A measure of cohesion and coupling can often provide an assessment of the quality of a design.

Cohesion represents the degree of functional correlation between the individual pieces of a module (i.e., the extent to which a module is concentrated on a specific, well-defined concept). A method that exhibits low cohesion often contains code to perform several tasks that are conceptually different (e.g., a stack class where the push method also computes a square root). In a highly cohesive module, the various relationships within the module can be easily discerned because of the distinct focus of the module. This is a great attribute for supporting independent development.

Coupling can be described as the extent to which modules are connected with each other. Highly coupled modules are very brittle because a change to one module often requires the modification of a number of other modules. This also negatively affects independent development because highly coupled modules will often reveal their underlying internal implementation details to other modules. The comprehensibility of such modules is reduced, too, because several different modules must be examined to understand the intent of a module. Coupling is, to a large extent, the opposite of good modularity.

Advanced Separation of Concerns

Even though the general notion of separation of concerns is an old idea, one can witness the nascence of a research area devoted to the investigation of new techniques to support advanced separation of concerns. Recall that the opening paragraphs of this chapter highlighted the importance of modular composition within several engineering activities. It has been recognized by numerous researchers that the software modularization constructs developed over the past quarter-century are sometimes inadequate for capturing certain types of concerns. This has serious consequences with respect to modular composition.

Previously defined modularization constructs are most beneficial at separating concerns that are orthogonal [Tarr et al., 1999]. However, these constructs often fail to capture the isolation of concerns that are non-orthogonal. Such concerns are said to be crosscutting, and their representation is scattered across the description of numerous other concerns. Crosscutting concerns are denigrated to second-class citizens in most languages (i.e., there is no explicit representation for modularization of crosscutting concerns). As a

result, crosscutting concerns are difficult to compose and change without invasively modifying the description of other concerns (i.e., crosscuts are highly coupled with other concerns). The three objectives of changeability, independent development, and comprehensibility are sacrificed in the presence of crosscutting concerns because of the lack of support for modularization (see [Gudmundson and Kiczales, 2001] for an evaluation of these objectives in the context of newly proposed modularization constructs).

The latest research efforts, under the general name of Aspect-Oriented Software Development (AOSD) [AOSD, 2002], explore fundamentally new ways to carve a system into a set of elemental parts in order to support crosscutting concerns. The goal is to capture crosscuts in a modular way with new language constructs called *aspect*s. A large portion of the second chapter thoroughly explains the problem of crosscutting concerns and surveys solution techniques.

The next section is not about AOSD, *per se*, but rather shows how crosscutting enters into other areas of human life, as well.

Organization Theory

> *Thus my central theme is that complexity frequently takes the form of hierarchy and that hierarchic systems have some common properties independent of their specific content.*
>
> [Simon, 1996]

Various types of organizations encompass elaborate hierarchies. The subject of organizational hierarchy has been studied for nearly a century. Within the disciplines of management and administration sciences, there is a popular corpus known as organization theory. Organization theory has a basis for comparison with software

development whenever a hierarchic approach to software decomposition is adopted. It is worth noting that some of the influential work in organization theory was conducted by a Turing Award winner – Herbert Simon – who also received the Nobel Prize for his work on decision-making in organizations. This section will offer a short assessment of organization theory as it relates to software construction.

Division of Labor

Since Adam Smith's *The Wealth of Nations* [Smith, 1776], the concept of division of labor has been an important topic within the discourse of economics, and the study of supporting institutions. A keen contribution by Smith was a quantifiable justification for the benefits that division of labor and specialization garner vis-à-vis efficiency and productivity. Division of labor is to a large extent correlated to the general objectives of separation of concerns as it relates to information hiding and the independent development of modules. Parnas actually gave a definition for the term "module" that would support such an assertion, as he stated, "In this context 'module' is considered to be a responsibility assignment rather than a subprogram" [Parnas, 1972]. The responsibility assignment of a module to a programmer relates to the specialization of effort that exists in division of labor.

Interdependence

> *Organizations display degrees of internal interdependence. Changes in one component or subpart of an organization frequently have repercussions for other parts – the pieces are interconnected.*
>
> [Daft et al., 1987]

After an organization is hierarchically constructed (as a result of the specialization of labor division), it is almost assured that the boundaries of the hierarchy will be broken as a result of interdependence among the different divisions. Large organizations naturally have certain kinds of concerns that are non-orthogonal to the hierarchic structure. Such facets of the organization increase the coupling of each division of the organization and expose particular characteristics of the division's specialization (an example of this is provided in the next section, within the context of a student requesting a transcript). These are the crosscutting concerns of the organization. Studies have been conducted on the mechanisms by which organizations have the ability to adapt to feedback [Daft et al., 1987]. These self-correcting behaviors are analogous to the reflective methods that are surveyed in Chapter 2.

Communication Channels and "Red Tape"

> *A multitude of rules and regulations appears to be the very essence of a bureaucracy. The term 'red tape[1]' adequately conveys the problem.*
>
> [Perrow, 1986]

Hierarchic decomposition is a tool for accomplishing goals and objectives within an organization. It is normal for organizations to have multiple goals, some of which may be conflicting [Hall, 1998]. The multiple rules that are spread throughout the hierarchy of an organization are the result, in many cases, of the implementation of some policy, or protocol. A policy is a mechanism that coordinates specific objectives across a set of dislocated organizational units. A policy, and the rules that implement it, could be

[1] The term seems to have first appeared in Sir Walter Scott's *Waverley* novels, written in 1814. At that time, bureaucratic documents were traditionally wrapped in pieces of red tape.

considered a type of crosscutting concern within the organization. The pejorative meaning of "red-tape" is tied to the frustrations that result from bureaucratic rules of policy implementation. In order for the policy to be realized, the specialization of many different organizational departments is needed. Intriguingly, the initial concept of bureaucracy, as proposed by [Weber, 1946], was promoted as the best structure for dealing with a changing environment – today, it is mostly associated with a negative connotation.

Most graduating college students have witnessed this bureaucratic process first-hand. A single request for a transcript may lead to the involvement of the registrar, student accounts, financial aid, and even the parking/traffic department. The policy for ensuring that a transcript is not issued to a student who has many outstanding debts necessitates the participation of all of these organization divisions. An interesting case study is presented in [Perrow, 1986], where a formal process at the University of Wisconsin was scrutinized. The policy that was examined corresponded to the process for a university faculty member to make a formal suggestion, or complaint. It was discovered that a complete review of the formal request would require that it pass through over fifteen levels of the university hierarchy. This example is comparable to crosscutting concerns in software implementations that execute a protocol across a large code base. As will be shown in a later chapter (see Figure 9 through Figure 11), the communication path in a hierarchy can introduce unnecessary overhead in both organizations and software.

The concept of an "Independent Integrator" has been advocated as a coordinator of the policies involving myriad interdependent departments [Dessler, 1986]. An integrator is the closest entity within organization theory that has a relation to techniques

for advanced separation of concerns. The role of an integrator is to step outside the hierarchical bounds and assist in the weaving of a crosscutting policy throughout the organization.

Research Objectives

This dissertation is about advanced separation of concerns at the system modeling level, and the construction of support tools that facilitate the elevation of crosscutting modeling concerns to first-class citizens (i.e., explicit constructs for the representation of such concerns). The contributions described in this dissertation can be summarized by two research objectives:

- **Raise Aspect-Oriented (AO) concepts to a higher level of abstraction**

  An AO approach can be beneficial at different stages of the software lifecycle and at various levels of abstraction; that is, it also can be advantageous to apply AO at levels closer to the problem space (e.g., analysis, design, and modeling), as opposed to the solution space (e.g, implementation and coding). Whenever the description of a software artifact exhibits crosscutting structure, the principles of modularity espoused by AO offer a powerful technology for supporting separation of concerns. This has been found to be true also in the area of domain-specific modeling [Gray et al., 2000]. Although there have been other efforts that explore AO at the design and analysis levels (see Chapter 2 for more details), the work described in [Gray et al., 2001a] represents the first occurrence in the literature of an actual aspect-oriented weaver (see Figure 8 in Chapter 2) that is focused on system modeling issues, rather than topics that are applicable to traditional programming languages.

- **Assist in the creation of new weavers using a generative framework**

  Because the syntax and semantics of each modeling domain are unique, a different weaver is needed for each domain. A metaweaver framework (illustrated in Figure 25 of Chapter 4) has been created as an aid toward the rapid construction of new domain-specific weavers. This framework uses several code generators that take metalevel specifications, described in a Domain-Specific Language (DSL), as input. The generators produce code that serves as a hook into the framework. The initial dissemination of this objective appeared in [Gray et al., 2001a], [Gray, 2001a], and [Gray, 2001b].

These two objectives provide a contribution toward the synergy of AOSD and Model-Integrated Computing (MIC) (see [Sztipanovits and Karsai, 1997] for an overview of MIC). This union assists a modeler in capturing concerns that, heretofore, were very difficult, if not impossible, to modularize. A key benefit is the ability to explore numerous scenarios by considering crosscutting modeling concerns as aspects that can be rapidly inserted and removed from a model.

Outline

Begin with the end in mind.

[Covey, 1990]

A background survey of related literature can be found in Chapter 2. The chapter reviews several techniques that have been used over the past decade to provide the variability needed to support clean separation of concerns. That chapter's overview begins by examining topics such as reflection and metaprogramming. Chapter 2 also provides the incentive for, and summary of, the emerging research efforts in advanced

separation of concerns. Within the general context of generative programming, a cornucopia of topics is summarized at the end of the second chapter. This encompasses a brief synopsis of the literature on object-oriented frameworks, code generators, and domain-specific languages.

The third chapter of this dissertation provides an explanation of the research objective that is centered on domain-specific aspect modeling. That chapter will motivate the need for aspect modeling by first describing the reasons why current modeling techniques are ineffective at capturing crosscutting concerns. Chapter 3 also gives a definition of the Embedded Constraint Language (ECL) to support aspect modeling. The similarities between aspect modeling, and the constitutive elements of aspect-oriented programming, are also given.

In Chapter 4, a metaweaver framework is introduced. The framework described in that chapter can be instantiated to produce new domain-specific aspect-oriented modeling weavers. The realization that each modeling domain requires a separate weaver is presented as an impetus for the metaweaver framework. The issue of code generation from a domain-specific language (the ECL) is an additional topic of interest in that chapter.

A detailed outline of extensions to this work can be found in Chapter 5. Several of the prospective areas of enhancement are concentrated on exploiting further areas of adaptability within the metaweaver framework.

Concluding remarks appear in Chapter 6. A comprehensive bibliography is included at the end of the dissertation, but is preceded by several appendices. Each appendix provides further details regarding certain facets of the dissertation.

Credits

There are a few parts of the work described in this dissertation that have been created as either an extension of the previous work of others, or in collaboration with others. This section acknowledges these contributing works and collaborations.

The Embedded Constraint Language (ECL), described in Chapter 3, is an extension of the Multigraph Constraint Language (MCL). The MCL has been developed over the past five years at ISIS by Dr. Gábor Karsai, Dr. Ákos Lédeczi, and Dr. Greg Nordstrom, with others making minor contributions along the way. The definition of the ECL uses the MCL as a base from which to evolve a new language and parser.

The modeling paradigms that are used in the Case Studies in Appendix B were created by Dr. Sandeep Neema in collaboration with Dr. Ted Bapty. These paradigms were developed in support of several DARPA projects (e.g., ACS, MOBIES, and PCES). Although the modeling paradigms are not a product of this dissertation, the particular techniques for modeling aspects from those paradigms are a core contribution of the research described in Chapter 3. These techniques are further illustrated by the examples of Appendix B.

In Chapter 2, Figure 6, Figure 10, and Figure 11 were reprinted, with permission, from Gregor Kiczales. Figure 14, also in the second chapter, has been used numerous times in ISIS presentations given by others. That figure is adapted from an earlier version that seems to have first appeared in [Karsai et al., 1997]. One of the case studies from Appendix B contains a figure (Figure 56) that has been reprinted, with permission, from BBN.

CHAPTER II

BACKGROUND

This chapter contains a broad survey of many techniques that have been found useful for supporting modularization of software (e.g., reflection and metaobjects, advanced separation of concerns, generative programming, and frameworks). These techniques also are effective at providing the capability needed for software compositions to adapt and change to evolving requirements. The contributions of this dissertation, described in Chapters 3 and 4, are extensions of several of these ideas.

Reflection and Metaobjects

> *Problems cannot be solved at the same level of awareness that created them.*
>
> [Einstein, 1950]

> *"The question is," said Humpty Dumpty, "which is to be the master – that is all."*
>
> [Carroll, 1872]

Industry increasingly demands that systems be adaptable and extensible. This demand may be manifested in various forms, including:

- the malleability of an application with respect to a set of changing user requirements (i.e., the degree of difficulty to affect change in an application's source code implementation);
- the degree of adaptivity within a system in the presence of a changing environment (i.e., the capacity of an application to examine itself and modify its own internal state during run-time).

Reflection and metaprogramming provide powerful techniques for extensibility by separating the program's computation (the base level) from the specifics of how the program is interpreted (the metalevel). This separation permits the modification of the underlying implementation semantics (through changes to the metalevel) at run-time. These techniques have been shown to provide great flexibility in systems that must adapt to changing environments [Robertson and Brady, 1999].

Reflection

Oh wad some Power the giftie gie us, To see oursels as ithers see us!

[Burns, 1786]

A philosophical definition of reflection has been given as, "…the capacity to represent our ideas and to make them the object of our own thoughts" [Clavel, 2000]. As used in this sense, reflection was first introduced in logic as a way to extend theories [Hoftstadter, 1979]. Reflection also has been an active research area within the context of programming languages. Various forms of reflection are even appearing in popular programming languages like Java.

Procedural Reflection

The work of Brian Cantwell Smith provided the seminal ideas for formally applying reflection to programming languages [Smith, 1982]. Smith defined *procedural reflection* as the concept of a program knowing about its implementation and the context in which it is executed (later, Smith would prefer the term *introspection* in place of procedural reflection). A reflective system is capable of reasoning about itself in the same

way that it can reason about the state of some part of the external world. Introspection offers the capability of dynamically adjusting the way that programs are executed.

A reflective system has a *causally connected* self-representation [Smith, 1982]. Thus, a reflective system has access to the structures that are used to represent itself. Depending on the level of support for reflection, these internal representations can be inspected and even manipulated. Here, the term "causally connected" means that a manipulation of the internal representation structures directly affects the observable external behavior.

Smith identified three conditions that must be satisfied in order for a system to be considered introspective:

1. The system must be able to represent a description of its internal structure in such a way that it can be inspected and modified by facilities within the system.
2. The self-representation must be causally connected to the structure and behavior of the system. Each event and state in the system must be self-described and modifications to the description must result in a change in structure or behavior.
3. The self-representation must be at the proper level of abstraction. It must be low enough such that meaningful modifications can be made. Yet, it must not be so low-level that a programmer gets bogged down in a morass of detail.

Metacircular Interpreters

> *If I have seen farther than others, it is because I was standing on the shoulders of giants.*
>
> [Newton, 1676]

Smith also described a language, called 3-Lisp, that supported his model of reflection. In 3-Lisp, the notion of a reflective tower of metacircular interpreters [Steele and Sussman, 1978] supports the incremental changes to layers of interpreters. A *metacircular* interpreter is a program that is written in the same language that it interprets [Abelson and Sussman, 1996]. The reflective tower is an infinitely ascending stack of interpreters. All interpreters in this tower are implemented in 3-Lisp. Each new layer in the tower is interpreted by the layer above it. The interpreter at the very bottom of the layer is the traditional program that processes user input.

In 3-Lisp, as is typical of most Lisp or Scheme implementations, an expression, an environment, and a continuation argument capture the state of an interpreter. The layers in the tower are connected by reification and reflection. Reification is the inverse of reflection – it is about the ability to consider an abstract concept as concrete. Sobel and Friedman distinguish the two processes as, "…converting some component of the interpreter's state into a value that may be manipulated by the program is called *reification*; the process of converting a programmatically expressed value into a component of the interpreter's state is called *reflection*" [Sobel and Friedman, 1996].

Object Reflection

The first effort to incorporate "Smithsonian" reflection into an object-oriented language is described in [Maes, 1987]. Building on the foundation of procedural reflection, an object-oriented reflective architecture divides the object part from the reflective part. The object part describes and manipulates the application domain and the reflective part describes and manipulates the object computation semantics.

The reflective operations provided by some object-oriented programming languages are limited. For example, the model of reflection provided in Java is much weaker than that found in Smalltalk and the Common Lisp Object System (CLOS). The reflection mechanism in Java does not permit the modification of the internal representation [Anderson and Hickey, 1999], [Sullivan, 2001]. It only provides a type of "read-only" examination facility that allows run-time inspection of the internal representation of an object. A further limitation is that the reflective methods in Java are marked `final`, which prohibits their extension. Therefore, the reflective model provided in Java is not of the Smithsonian style because it does not provide the adaptation needed for being causally connected.

The definition of introspection is presented slightly differently in [Bobrow et al., 1993]. They define *introspection* as a program's ability to observe and reason about its own state. They define *intercession* as the more powerful capability of modifying the internal state to affect the underlying semantics. Using these definitions, Java can be said to provide support for introspection, but not intercession.

Metaobjects

> *"Meta" means that you step back from your own place. What you used to do is now what you see. What you were is now what you act on. Verbs turn to nouns. What you used to think of as a pattern is now treated as a thing to put in the slot of another pattern. A metafoo is a foo into whose slots you can put parts of a foo.*
>
> [Steele, 1998]

As Steele observes, the prefix *meta* is used to denote a description that is one level higher than the standard frame of perception. Meta is also used to mean "about,"

"between," "over," or "after." Hence, a metaprogram is usually defined as a program that modifies or generates other programs. A compiler is an example of a metaprogram because it takes a program in one notation as input and produces another program (usually object code) as output. Reflection is considered a form of metaprogramming where the target of the modification is the metaprogram itself. Metaprogramming can be a complex activity sometimes because there can be a blur between the base level and the metalevel.

Metaobject Protocols

Maes appears to be the first to introduce the notion of a metaobject [Maes, 1987]. In an object reflection system, a metaobject is just like any other object during run-time. Every object in the language has a corresponding metaobject and every metaobject has a pointer to its corresponding implementation object [Maes, 1988]. The metaobject contains information about its language object, such as details on its implementation and interpretation. During the execution of a system, the language objects may request information about their state, and even perform a modification on the internal representation.

Metaobject Protocols (MOPs) facilitate the modification of the semantics of the underlying implementation language [Kiczales et al., 1991]. Manipulating the interfaces that the MOP provides can incrementally modify the behavior and implementation of the underlying language. For example, CLOS has a MOP that specifies a set of generic functions [Steele, 1990].

There are five categories of functions that represent the core elements of CLOS (i.e., classes, slots, methods, generic functions, and method combination). A metaobject

represents each of these core elements. Each metaobject has a metaclass. The metaclasses behave like any other class such that the semantics of a metaobject can be adapted by modifying its metaclass. A programmer can alter the semantics of CLOS by using standard object-oriented techniques, like subclassing. The instance of each metaobject can be adapted at run-time. The behavior of the system at any particular time is dependent on the configuration of the set of metaobjects. The protocol, in this case, represents the interfaces of the metaclasses. Any modification to the behavior of the system must adhere to the interface definitions.

MOPs gain their adaptive power from a synergy of reflection and Object-Oriented Programming (OOP). As described in [Kiczales et al., 1991], there are three attributes of a metaobject protocol:

1. The core programming elements of a language are represented as objects. For example, the syntax and semantics for method calls, the rules for handling multiple-inheritance, and the rules of method lookup are all represented as objects.
2. The behavior of the language is encoded in a protocol based on these objects. The protocol is the interface of the metaclasses.
3. A default object is created for each kind of metaobject.

Concerning the first attribute from above, an example of the ability to modify multiple-inheritance rules is shown in [Kiczales et al., 1991]. A generic function called `compute-class-precedence-list` returns the rules that determine the resolution of conflicts due to multiple-inheritance. The programmer can modify this list so that new rules of conflict resolution are used. As another example, objects are created in CLOS by

calling `make-instance`. The implementation of this method can be redefined at run-time to perform specialized adaptations during object creation.

Although the majority of the literature on reflection and metaprogramming is described in some dialect of Lisp, there have been efforts to apply these techniques to other languages. For example, [Chiba and Masuda, 1993] describe a basic metaobject protocol for a language called Open C++. A more detailed description of a MOP for C++ is given in [Forman and Danforth, 1999]. While not analogous to MOPs, *per se*, there has also been research in C++ on an idea called static metaprogramming. A variant of this, which relies on C++ templates, provides a compile-time facility for generating code and component configuration [Czarnecki and Eisenecker, 2000].

Metaobjects also can be used in assisting in the separation of concerns in areas other than programming languages. Research at IBM recognized that, within middleware, there is an intermixing of application code and protocol code [Atsley et al., 2001]. The lack of modularity affects the ability to maintain and customize the middleware. A metaobject protocol cleanly separates the policy and protocol code from the underlying application. Some example metaobjects that were defined to represent communication events are `transmit` (what happens when a component sends a message), `deliver` (what happens when a message is received by a component), and `dispatch` (the received message a component decides to process). Nonfunctional system properties like security and persistence [Rashid, 2002] can be cleanly separated from the base level program to improve reuse. This has been termed *implementational reflection* in [Rao, 1991].

Within the scope of distributed object computing and middleware, the technique of CORBA interceptors is closely related to metaobject protocols. Interceptors are defined as, "non-application components that can alter application behavior" [Narasimhan et al., 1999]. An interceptor can transparently modify the behavior of an application by attaching itself to the invocation path of a client and server object. Interceptors have been shown to be useful in enhancing CORBA by providing adaptability with respect to profiling, protocol adaptation, scheduling, and fault tolerance [Narasimhan et al., 1999].

Evaluating MOPs

A detailed evaluation of the practical use of MOPs can be found in [Lee and Zachary, 1995]. In this study, a MOP was applied to a geometric CAD tool in order to add persistence to the CLOS implementation objects. The project was described as being very ambitious and a much more complicated application of MOPs than previously studied.

Much of the evaluation was positive. Because the majority of the effort to extend CLOS related to objects, the metaobject protocol provided a useful resource. However, the effort had several difficulties. Although the CLOS MOP is very useful when extension is based on a property of an object, the protocol is not helpful when there is a requirement to augment a feature that is not captured as an object property. For example, in CLOS, arrays and several other composite values are native to Common Lisp and are not available for extension in the MOP.

Another difficulty was found with respect to performance. In several experiments, it was found that object creation was sixteen times slower than the prior implementation

that did not use a MOP. Similarly, write access using the MOP was found to be about seven times slower. Performance has always been a problem for reflective approaches. Consider the following observation, with respect to Java-based reflection, "As of release 1.3, reflective method invocation was forty times slower on my machine than normal method invocation. Reflection was re-architected in release 1.4 for greatly improved performance, but is still twice as slow as normal access, and the gap is unlikely to narrow" [Bloch, 2001]. The performance penalty resulting from many dynamic calls in a reflective implementation will often rule-out reflection as an implementation alternative in some contexts.

Open Implementations

Traditionally, black-box abstraction states that a software module should expose its interface, but hide its implementation details. This is a corollary to [Parnas, 1972], and is similar to the *Open-Closed Principle*, described in [Meyer, 1997], which states that a module should be open for extension, yet closed for modification. However, the idea of an open implementation disagrees with this principle when applied fundamentally. Research in the area of open implementations has found that, in some cases, software can be more reusable when a client is allowed to control a module's implementation strategy [Kiczales, 1996]. Open implementation proponents agree that the base level should remain closed like a black-box. It is the metapart that they advocate opening to extension [Kiczales, 1992]. In fact, the initial motivation behind MOPs was a desire to open the language in such a way that better control could be exerted over the selection of the implementation with respect to certain performance concerns [Kiczales et al., 1993].

Advanced Separation of Concerns

The limits of my language are the limits of my world.

[Wittgenstein, 1961]

> *Language exists to communicate whatever it can communicate. Some things it communicates so badly that we never attempt to communicate them by words if any other medium is available.*
>
> [Lewis, 1967]

In Chapter 1, the importance of separation of concerns was motivated. During the latter part of the 1990s, research in this area increased with an invigorated interest. This was due, in part, to the recognition that the languages and tools used to develop software hampered the proper isolation of specific categories of concerns. The inadequacies of modern programming languages (with respect to separating certain concerns) prompted many researchers to take a fresh look at modularization constructs and extensions/complements to current languages. The focus of the problem can be discerned from the observation that programming languages are often used in a linear process. However, the things that we want to express in a language, and our conceptualization of key abstractions as a supporting mechanism, are certainly not linear. This section provides the initial motivation and problems that are being solved by a new area of research entitled Advanced Separation of Concerns (ASOC).

A Survey of Some Concerns and Their Separation

Before initiating the impetus behind advanced separation of concerns at the implementation level, it may be beneficial to first notice the various methods that have been suggested for managing concerns in other contexts. The examples in this section

represent concerns that are typically identified outside of the milieu of traditional programming language research.

Database Triggers

Assume that the following business rule is to be consistently enforced within a database: “Every time an employee’s salary is increased by 25%, log the employee’s social-security number, previous salary, and new salary into an audit table.” The implementation of this business rule requires that some action be taken every time that an update to the salary column occurs. This business rule is an archetype for a crosscutting concern.

Without triggers, the realization of this rule would require that the concern be placed in *all* of the stored procedures that update the employee’s salary. That is, the delta of a salary increase must be computed for each update and checked against the specified 25% rate increase. This could result in the insertion of redundant code throughout all stored procedures that are affected by this business rule. The problem is compounded when the salary update occurs within embedded SQL in a base programming language. In that case, the check must be made outside of the database in every location of the base program that implements this business rule.

Fortunately, a trigger mechanism facilitates a cleaner solution. A trigger-based solution, like that found in Figure 1, would provide a single location from which changes could be made to the semantics of the concern.

The trigger solution does not need access to metalevel control in order to capture the intent of the concern (i.e., it is not necessary to redefine the underlying semantics of the table update definition). As will be shown later, this is similar to the way that AspectJ

captures a concern without resorting to metaprogramming techniques (i.e., aspects and non-aspects are all at base-level code – there is no reference to the metalevel within AspectJ). This is an important point in differentiating triggers, and even aspect languages, from pure metaprogramming techniques. Later in this chapter, the constitutive parts of an aspect language will be described. A preview of these is now given in a comparison of aspect languages and triggers.

```
CREATE OR REPLACE TRIGGER salary_audit
AFTER UPDATE OF salary ON employee
FOR EACH ROW
WHEN (new.salary > 1.25 * old.salary)
CALL log_salary_audit(:new.ssn, :old.salary, :new.salary);
```

Figure 1: A Trigger for Logging Salary Increases

On the second line of Figure 1, the "`AFTER UPDATE`" statement indicates the point of execution when the trigger statement is applied. Using `BEFORE/AFTER`, an Oracle database trigger is able to influence the dynamic execution of a database server whenever certain operations (`DELETE`, `INSERT`, `UPDATE`) are executed on a database table. There are six different variations that can be given, resulting from the permutation of {`BEFORE`, `AFTER`} × {`DELETE`, `INSERT`, `UPDATE`}. Also, on the second line, the "`OF salary ON employee`" is similar to the pointcut idea in aspect languages. This construct identifies a particular point in the database table (e.g., a row and a table) that is affected by the trigger. The "`when` condition" syntactical construct on line 4 has some likeness to the "`if`" pointcut designator in AspectJ. The executable statement that is

associated with the trigger (this is the action that occurs when the trigger is fired), found on the last line of Figure 1, is akin to the concept of "advice" in AspectJ. The definition of these aspect-oriented terms will be clarified in a subsequent section.

Even though the database trigger mechanism permits the capture of crosscutting business rules within a database, it has several weaknesses when compared to pure aspect languages. The most evident limitation is the lack of the ability to create compositions of triggers. The trigger approach allows only the naming of a single table. It does not permit the logical composition of table property descriptions. That is, the type of pointcut model used within triggers is not composable in the same way as AspectJ. Triggers also do not support the concept of wildcards within the naming of a pointcut. For example, the second line from above could not be written as "`OF sal* ON emp*`" in order to designate multiple columns and tables that are affected by the trigger.

Mail Merge

Mail merge is an office automation tool that supports the separation of the form of a document from a data source of merge fields. By this separation, the insertion of each instance throughout the document can be better managed (see Figure 2). Consider the task of a lawyer who specializes in commercial foreclosures. He, or she, will typically need to process fifteen different documents in order to execute a foreclosure (according to information obtained from a personal conversation with a Nashville attorney). Furthermore, five or more different parties (with separate contact information) are typically involved. Their contact addresses, and other pertinent information, are diffused across the space of the various legal documents. By separating the instance from the form, the author of the document is spared from the tedious task of visiting multiple

locations in the document in order to make each change. Although the mail merge tool assists in a specific type of concern separation, it requires the document designer initially to visit every instantiation point in order to insert a field designator (because of this, the process is somewhat similar to the LaTeX macro command).

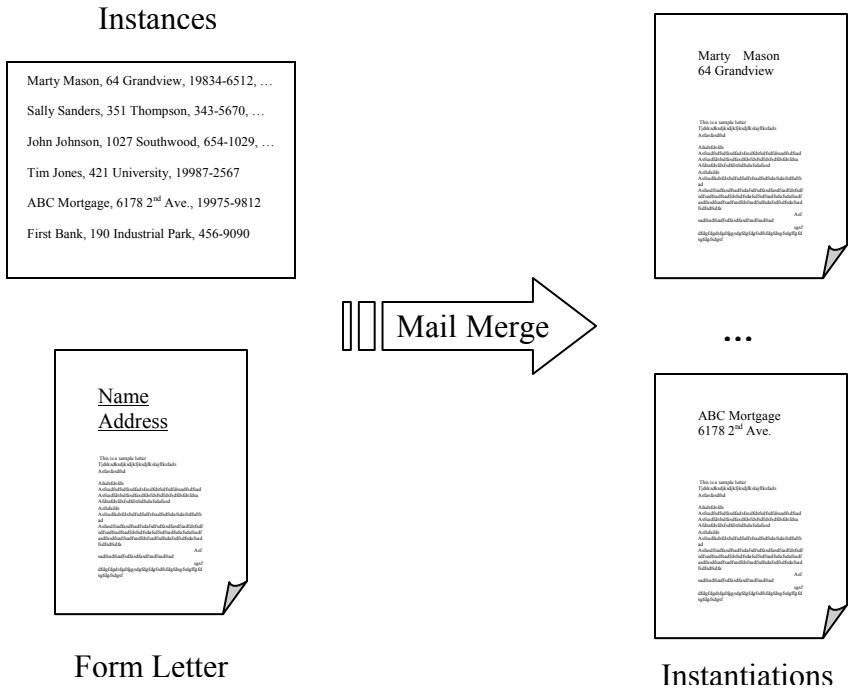


Figure 2: Mail Merge Example

Style Sheets

Within the context of web publishing, style sheets are a useful technique for separating the content of a document from its presentation style [Meyer, 2000]. Such a separation provides a method for making seamless global changes to the appearance of a document without the need for visiting numerous individual locations in the document. In a Cascading Style Sheet (CSS), a rendering engine visits each node of a document. As the

traversal proceeds over the document's hierarchy, the renderer attempts to match the current element with a pattern specified as a CSS rule. A CSS rule consists of two parts: a selector, which names the type of the element to which the style will be applied, and a declaration, which represents the type of style to be applied.

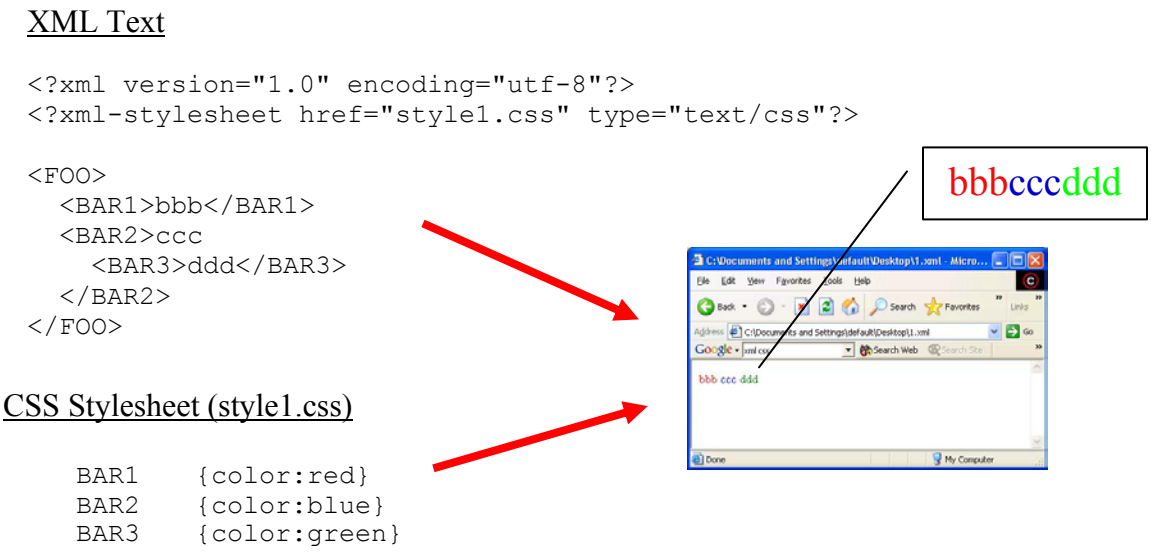


Figure 3: A Cascading Stylesheet Example

An illustration of the application of a CSS rule is shown in Figure 3. The top-left of the figure contains the content of a document as represented in the Extensible Markup Language (XML). The information regarding the name of the specific style that is to be applied (in this case, the style sheet named style1.css) is located within the preamble of this document. The specification of style1.css is listed in the bottom-left of the figure. As can be seen, this style sheet has a rule asserting that all elements of type BAR1 are to be rendered in the color red. In this example, it should be understood that the rendering engine resides within the browser.

Literate Programming and WEB

> *Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.*
>
> [Knuth, 1984]

The idea of literate programming was initially described by Donald Knuth and implemented with a tool called WEB [Knuth, 1984]. In WEB, a single program is a combination of source code, documentation text, and WEB commands. Literate programming assists a programmer in assembling programs that are more easily read by a human. This is done by treating the construction of documentation and source code as a simultaneous activity. The aim is to make the construction of programs more like the creation of a literary work.

The formal expression of a concern is so closely tied to the informal description that tools are needed to separate the two representations so that they are consumable by different parties (e.g., a compiler and a human). In WEB, source code is produced from the TANGLE tool, and documentation is formed by the WEAVE tool (see Figure 4). It is interesting to note that the structure of the process for creating WEB programs is almost opposite to that seen in Figure 2 and Figure 3. In those contexts, the concept of weaving a document entailed the notion of bringing separated entities together as one (where the separation provided some desirable property that assisted in change maintenance and comprehensibility). In literate programming, however, the concept of weaving represents the task of separating concerns of interest (e.g., the visual presentation of documentation) from an existing tightly coupled document.

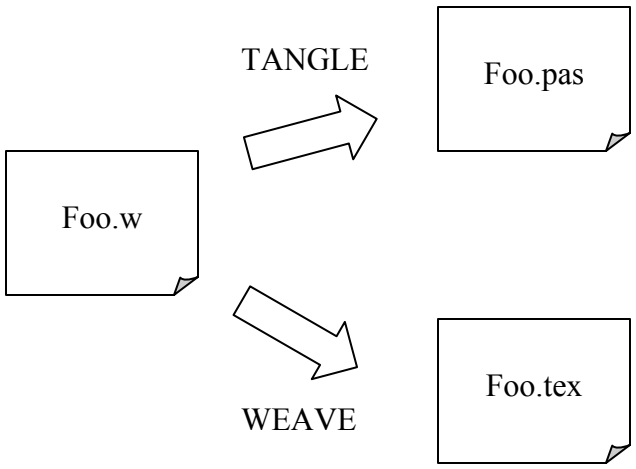


Figure 4: Separation of Concerns in WEB

The preceding subsections provided several examples of concern separation. Two of the four examples were in contexts not associated with software development (e.g., mail-merge and stylesheets). A common topic in each of these examples was the existence of an integration tool for assisting in the conceptual separation. In the following sections, the problems associated with crosscutting concerns are motivated, along with the need for a new type of software integration tool – a weaver.

Problems with Scattered Code

> *It is organization which gives birth to the dominion of the elected over the electors, of the mandataries over the mandators, of the delegates over the delegators. Who says organization, says oligarchy.*
>
> [Michels, 1915]

Non-orthogonal concerns can be described as crosscutting, because such concerns tend to be scattered across the traditional modularity boundaries provided by a development paradigm. In programming languages, two concerns crosscut when the

modularity constructs of a language allow one concern to be captured separately, but only to the detriment of another concern that must be captured in a way that is not cleanly localized. This has been referred to as the "tyranny of the dominant decomposition" [Tarr et al., 1999]. The "Iron Law of Oligarchy," quoted above from Michels, suggests that bureaucratic hierarchy tends to result in oligarchy; that is, those at the top of an organization are those that rule. In Chapter 1, an allusion was made to this tyranny under the Organization Theory section that described Interdependence. With respect to the dominant decomposition, this also seems to be true with traditional methods for software modularization.

Crosscutting has the potential to destroy modularity. The crosscutting phenomenon can occur in structured programming, where the procedure, function, and module delimit the modularity boundaries. It is also prevalent in object-oriented programming, where classes, methods, and inheritance define the boundaries of encapsulation.

Crosscutting concerns provide difficulties for a programmer because the implementation of the concern is scattered throughout the code; the concern is not localized in a single module. This can be a source of potential error when modifications are required. Comprehensibility is negatively affected in two ways [Tarr et al., 1999]:

- The scattering problem: The ability to reason about the effect of a concern is decreased because a programmer must visit numerous modular units in order to understand the intent of a single concern. The problem is that a concern often touches many different pieces of code.

- The tangling problem: Within a module, the tangling of numerous concerns decreases cohesion, and raises coupling. This reduces a programmer's ability to understand the core intent of a particular module. The problem is that many concerns may touch a single piece of code.

Programmers are often forced to keep track of crosscutting concerns in their heads. This is an error-prone activity, because even medium-sized programs can have hundreds of different crosscutting issues [Tristram, 2001]. Another problem of crosscutting concerns is maintenance. It is often the case that the global spreading of a concern, and the ramifications of its modifications, are not intuitive to those who inherit the code for maintenance. Maintenance becomes more of an archaeological metaphor, where a programmer must search through rubble in order to uncover a useful artifact [Hunt and Thomas, 2002]. The Parnasian objectives, found in Chapter 1, are usually sacrificed in the presence of non-orthogonal concerns.

Figure 5 provides an illustration of scattering and tangling. The three individual units (Unit A, B, and C) would be considered highly cohesive, if it were not for the tangling of the three concerns of logging, synchronization, and persistence. Furthermore, the scattering of these concerns would make it difficult to change their behavior, especially if the example were scaled to a much larger problem with thousands of units.

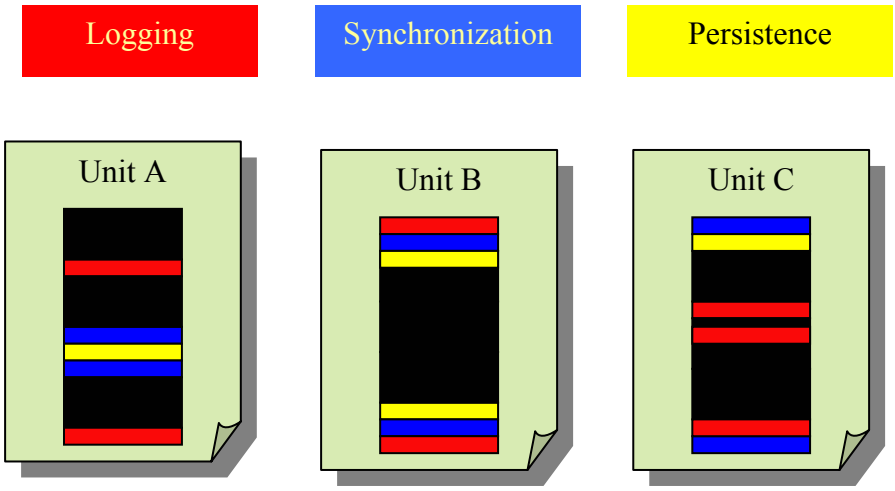


Figure 5: Crosscutting Concerns

For a pictorial representation of the problems of crosscutting in a real application, consider the following figure:

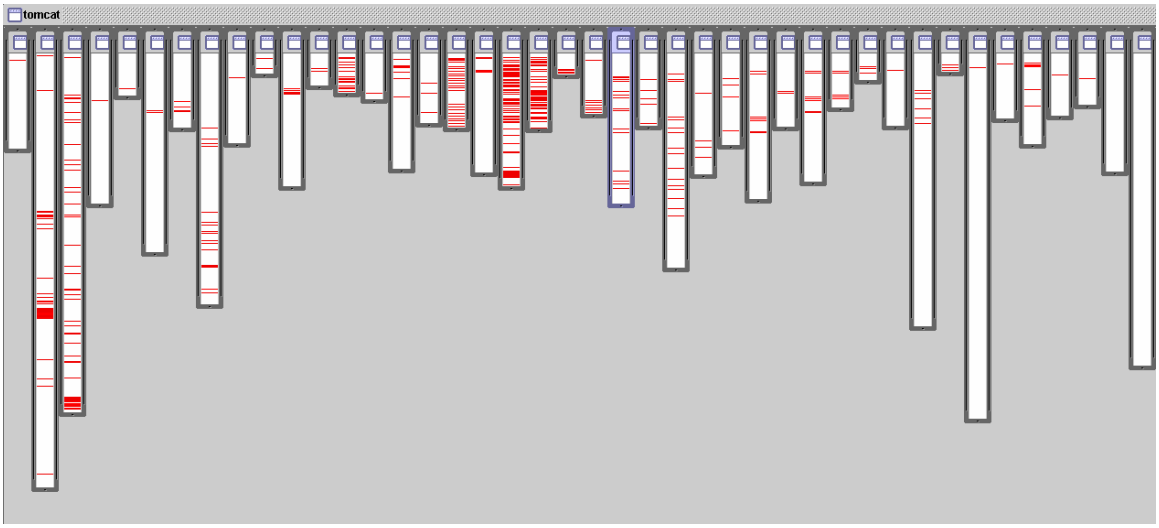


Figure 6: A Pictorial Representation of Crosscutting
(Reprinted from [Hilsdale et al., 2001], with permission from Gregor Kiczales.)

This figure represents a piece of the Apache Tomcat code. Tomcat is an implementation of the Java Servlet and JavaServer Pages (JSP) specifications. Tomcat can run as a standalone, or it can be integrated into the Apache Web Server. The white vertical boxes represent a few of the classes in a subset of the Tomcat implementation. The highlighted lines designate the lines of code related to the concern of logging.

Notice that the implementation of the logging concern is spread across the various classes. It is not located in a single spot. In fact, it is not even located in a small number of places. As reported in [Robillard and Murphy, 2002], a modification to the logging concern, "would require the developer to consider 47 of the 148 (32%) Java source files comprising the core of Tomcat." In this example, if the type of information to be logged is changed, then a developer may be required to make modifications to each of these 47 individual source files. From a software engineering viewpoint, this is not desirable. There is no cohesive module for representing the concept of logging – that concept is coupled among all of the other concerns.

To highlight the importance of this, forget for a moment that the highlighted code in Figure 6 represents logging. Assume, instead, that it represents all of the code for implementing the concerns of an employee in a payroll application (i.e., the implementation of employee features is scattered across multiple source files, in different modules). In that situation, it is easy to see that the basic principles of cohesion and coupling are being violated. The same can be said, then, when the highlighted concern is understood to be logging.

The problem just described is not the fault of a programmer who is guilty of poor design [Simonyi, 2001]. There is simply no traditional programming language construct

that would permit a better localization of the concern – it is, "a lack of expressibility in the technology available to the original designer to express interacting or overlapping concerns" [Robillard and Murphy, 2002]. Gregor Kiczales has commented that, "Many people, when they first see AOP, suggest that concerns…could be modularized in other ways, including the use of patterns, reflection, or 'careful coding.' But the proposed alternatives nearly always fail to localize the crosscutting concern. They tend to involve some code that remains in the base structure" [Kiczales, 2001]. These alternatives require that the code related to the concern be placed in numerous locations.

Appendix A includes a case study that documents several crosscutting concerns that emerged during the development of a commercial application using Delphi. The appendix also provides an aspect-oriented solution that exhibits better support for the separation of these concerns.

Granularity of Concerns

Suppose that the highlighted lines of code in Figure 6 represented all of the places that a square root function (sqrt) was called (as opposed to the real instance of logging). One of the differences between a scattered logging concern, and the use of sqrt in many locations in the source, is that the multiple appearances of sqrt in the code typically do not necessarily mean that there is a crosscutting concern present. That is, there is no general crosscutting concern called "square root." In fact, the sqrt function could be used in many different places that are orthogonal to each other. In such cases, the sqrt function is simply being used as a piece of a computational equation. It fits nicely within the functional decompositions where it appears. It is possible, however, for the sqrt function to be embedded within some type of crosscutting concern. In the instance of logging, the

crosscutting concern is not the code that performs the actual logging of the message. Rather, the reason that the concern is crosscutting comes from some higher-level requirement that logging should be done at a specific collection of points in the execution of the program, and that this should be done in a consistent manner across all logging locations. The essence of the logging concern is the set of points that perform the logging, not the actual code for performing the write to a log. It is this characteristic – the enumeration of the places where logging is to occur – that make the concern crosscutting. For instance, a requirement might state, "Within the packages of all Borland GUI classes, write to a log every time an update method is executed." This is obviously crosscutting. Also, such statements typically would not make sense when applied to a sqrt function. The sqrt function, most frequently, would be a part of a larger computation, not a crosscutting concern, *per se*. The example in Figure 6 is primarily about the consistent policy (or protocol) that is diffused across the code – it is not about replicated code.

In discussing this example with Gregor Kiczales, he imagined a scenario where there are three small functions that each call a square root method (sqrt) and a logging method (logit). Each of these functions call these methods in the same way, for the same reason. Kiczales points out that there are really four different concerns emerging from this example: 1) the implementation concern describing the functionality of sqrt, 2) the implementation concern describing the functionality of logit, 3) the implementation concern describing the functionality of each of the three simple functions that call sqrt and logit, and 4) the global consistency concern related to the logging policy. The first three concerns can all be implemented using traditional modularity constructs. However, notice that the policy relating to the consistent use of logit (i.e., the pertinent information

that is consistently passed, as the parameter to logit, by all occurrences of this concern) is distinct from the actual implementation of the logit method itself. It is this fourth concern that is crosscutting. On the other hand, the different ways that sqrt are called within these three functions is focused more on the implementation concerns within the three functions (i.e., the characteristics of the computation within each function) than on some globaly coordinated policy with a common structure. There really is nothing crosscutting about the invocation of the sqrt calls.

Twisted Plot Metaphor

As a metaphor for software development, and to help in understanding the problems of crosscutting in a completely different context, consider the task of writing a large novel. The interactions among characters, and the coincidences that occur as a result of overlapping events, can be said to add appeal to the novel (perhaps the level of "thickness" of the plot could be suggested as a complexity metric for novels). The author of a novel must be disciplined in preserving, throughout the entire story, the internal consistency within the plot – a change in the conclusion may necessitate global changes in all chapters. For example, a character that died in an early chapter must not appear resurrected several chapters later (unless, of course, the script is for a day-time soap opera, where such things seem to occur with unexpected frequency).

As the creative writing process unfolds, the author must make a mental note of all the twists of the plot (pictorially rendered in Figure 7 as the swirl that is linked to each chapter). This can be an arduous undertaking because the various concerns of the plot are distributed across multiple chapters (the chapters, sections, paragraphs, and sentences of

the novel represent the hierarchical boundaries of the dominant decomposition, with the plot being the crosscutting concern that is difficult to modularize).

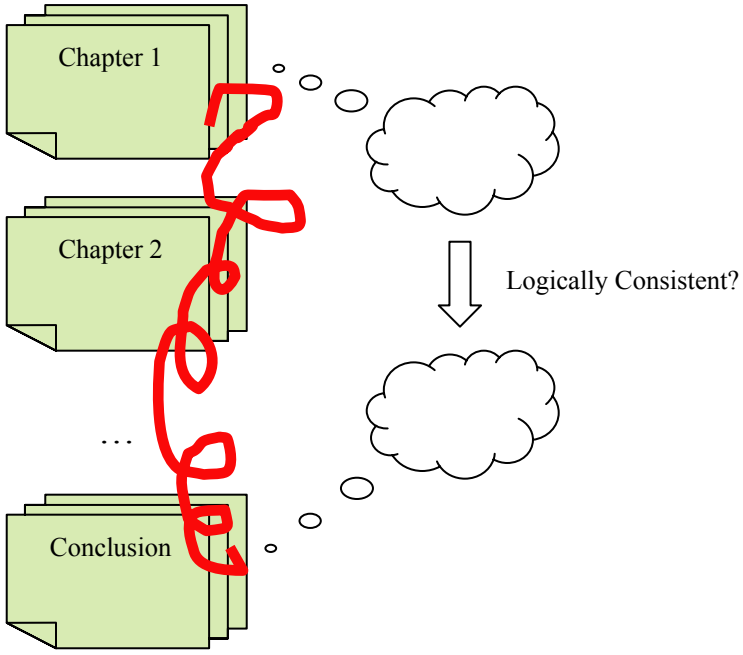


Figure 7: Twisted Plot Metaphor

It could be asserted, however, that software development is several orders of magnitude more complex than the process of writing a large novel. The discipline required to maintain and understand all of the locations where a crosscutting concern is affected beseeches that advanced modularization constructs be provided. Aspect-Oriented Programming (AOP) has been offered as a liberator from such burdens.

Aspect-Oriented Programming

> *I believe that the continued advance of programming as a craft requires development and dissemination of languages which support the major paradigms of their user's communities. The design of a language should be preceded by enumeration of those paradigms, including a study of the deficiencies in programming caused by discouragement of unsupported paradigms.*
>
> [Floyd, 1979]

Programming language support for separation of concerns has long been a core aid toward managing the complexity of large software projects. Support for the modularization and decomposition of certain dimensions of a system has improved comprehensibility and evolvability during software development. For example, objects support the decomposition of a system according to the dimensions of data abstraction and generalization (via inheritance), and structured programming techniques focus on a functional decomposition. Other dimensions of concern often concentrate on features that are crosscutting (e.g., persistence is a crosscutting feature) [Tarr et al., 1999]. Most modularization constructs, however, provide for the separation of concerns along only one dimension. The dominant form of decomposition forces other dimensions of the system to be scattered across other modules. When non-orthogonal concerns are spread out across multiple modules, the system becomes more difficult to develop, maintain, and understand. Moreover, reusability of such concerns is not possible due to the cross-pollination of one concern into many modules; there is no localized container to capture the concern.

As implied in the first section of this chapter, reflection and metaprogramming were an early attempt at resolving crosscutting. These techniques were somewhat low-level, but provided a lot of expressive power. With MOPs, for instance, there is a blurred

distinction between language user and language designer. Therefore, a more practical use of the techniques by less experienced programmers would require modularization constructs that offered more disciplined control over this power. As these techniques evolve, a new breed of programming languages is emerging to assist in the modularization of crosscutting concerns.

Aspect-Oriented Programming (AOP) provides a strategy for dealing with emergent entities that crosscut modularity [Kiczales et al., 1997]. AOP recognizes that crosscuts are inherent in most systems and are generally not random. The goal of AOP is to provide new language constructs that allow a better separation of concerns for these aspects. An *aspect*, therefore, is a piece of code that describes a recurring property of a program that crosscuts the software application (i.e., aspects capture crosscutting concerns). AOP supports the programmer in cleanly separating components and aspects from each other by providing mechanisms that make it possible to abstract and compose them to produce an overall system.

Gregor Kiczales and his colleagues at Xerox PARC developed the seminal ideas behind AOP in the mid-1990s. In *MIT Technology Review*, AOP was featured as one of the top 10 "Emerging Technologies That Will Change the World" [Tristram, 2001] and has been the subject of a special issue of *Communications of the ACM* [Elrad et al., 2001]. Notably, object-oriented guru Grady Booch labeled AOP as, "something deeper, something that's truly beyond objects…a disruptive technology on the horizon" [Booch, 2001].

Aspects – A Complement to Traditional Paradigms

In the structured paradigm, modular block structures were used to provide scope for separating the boundaries of concerns. The "go-to" statements that often resulted in tangled and scattered concerns were replaced with procedure calls [Dijkstra, 1968]. This improved the control flow of a program and enhanced its modularization. The Object-Oriented (OO) paradigm represents the generation that followed the structured paradigm. In OO, the key modularization technique focused on hierarchical structuring through classes and inheritance. Another key feature of OO, polymorphism, permits variation of behavior within a class hierarchy.

Each new generation of modularity technology builds upon the previous generation. AOP should be evaluated within the context of being another technology for supporting separation of concerns. The ideas of AOP should be viewed as a counterpart to procedures, packages, objects, and methods to the extent that they all support different ways of modularizing certain kinds of concerns. In this sense, AOP can be regarded as a complement to both the structured and OO paradigm, or any other paradigm for software construction (e.g., logic programming [De Volder and D'Hondt, 1999]). In AOP, the focus is on capturing, in a modular way, the crosscutting concerns of a system. The crosscuts will still exist, but the problems of scattered and tangled code are removed by encapsulating the crosscut in a single module. To quote a personal communication with Gregor Kiczales, "OO made inheritance explicit in language. AO makes crosscutting explicit in language. OO makes its bet on hierarchical structures, but AOP makes its bet on crosscutting structures."

AOP has been defined in terms of its ability to provide quantification and obliviousness. *Quantification* is the notion that a programmer can write single, separated

statements that introduce effects across numerous locations in the source code. Thus, quantification would provide the capability for saying the following: "In programs P, whenever condition C arises, perform action A" [Filman, 2001]. This can be stated more formally as: ∀C[A], where the crosscutting nature is captured in the universal quantifier, and the action to be performed within the concern is the parameterized action. The property of *obliviousness* holds when the quantified locations do not require modification in order to incorporate the effects of the quantification. As stated by the authors of this definition, "AOP can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantifications hold over programs written by oblivious programmers" [Filman and Friedman, 2000].

The idea of quantification does suggest a special property of aspect languages, but quantification also exists within pure metaprogramming techniques. Even though metaprogramming is one way to capture crosscutting concerns, and AOP has its roots in metaprogramming, it should be understood that there are some important differences. Perhaps a better characterization of aspect languages, in order to avoid confusion, would be those languages that provide constructs for quantification, yet do not refer to metalevel concepts.

In a first exposure to AOP, many compare it to macro expansion. However, this comparison is far from accurate. Although there are similarities with respect to code being inserted or expanded, the AOP model is much more powerful. A limitation to the strength of macros is the fact that the transformations that are performed are textually local [Kiczales et al., 1992]. For instance, to use a macro, a programmer must visit numerous locations in the source code and insert the name of the macro. If a change

needs to be made, or the macro needs to be removed from a specific context, then the programmer must visit all of these points in the code. Macros do not exhibit quantification. Aspects, on the other hand, operate under the property of reverse inheritance (also known as inversion of control[2]). The behaviour of an aspect is specified outside of the context where it is applied. Aspects, and their quantification, are described in one location – a programmer does not have to visit and insert code in any other place. This makes the addition and removal of aspects effortless.

It should be noted that the same distinction that has been made between AOP and macros could also be made in comparing AOP and *mixins* [Bracha and Cook, 1990]. A mixin is a class that is not intended to be instantiated. It provides some desired behavior (e.g., persistence) that is imported into other classes via inheritance. Mixin-based inheritance does not provide quantification and obliviousness. If a programmer wants to include mixin behavior in a class, the mixin must be explicitly imported within the purview of the class's predecessors. Mixin based inheritance is also missing the reverse inheritance property that can be provided through the kind of quantification available in aspect languages.

In comparing aspects to classes, there is almost an inverse relation between the way inheritance works in OO and the way aspects work in AOP. As stated in [Viega and Voas, 2000], "With inheritance, classes choose what functionality they wish to subsume from other objects. Aspects, on the other hand, get to choose what functionality other objects subsume."

---

[2] This also can be called the *Hollywood Principle* ("Don't Call Us, We'll Call You").

Examples of Commonly Recurring Crosscuts

There are several commonly recurring crosscutting concerns that have been identified from a wide variety of different systems. For example, the software described in Figure 6 highlighted the fact that the common concern of logging is often scattered across the code base. The case studies in Appendix A document several other common types of crosscuts. The dirty bit example, in Appendix A, for the LangMan application points to the fact that global coordination among many interacting objects often forces an implementation that has several concerns scattered among the various participant objects.

The study of operating systems code is ripe for the mining and understanding of crosscutting concerns. As pointed out in [Coady et al., 2001b], many of the key elements of operating systems crosscut. As an illustration, the prefetching activity that is performed in OS code is often highly scattered and tangled. As Coady and colleagues discovered, the FreeBSD v3.3 implementation of prefetching was spread across 260 lines of code in 10 clusters in 5 core functions from two subsystems. A refactoring of the prefetching implementation using an aspect language demonstrated an increased comprehensibility of the code with respect to independent development, as well as the ability to (un)plug different modes of prefetching [Coady et al., 2001a]. Their future research focus is in the investigation of other crosscutting concerns in FreeBSD; namely, scheduling, communication protocols, and the file system. It is also often the case that the implementation of specific protocols lead to tangled code, as does code that is introduced into the system to improve some performance optimization. This also can be true in implementations that provide resource sharing among a set of objects. The various policies, or protocols, contained within an operating system are typically implemented in

a crosscutting manner. This is similar to the observation made in Chapter 1 concerning policy implementations that have been studied in organization theory.

Perhaps the two most commonly observed crosscutting concerns are synchronization and exception handling. Both of these are also evident in the case studies of Appendix A. A detailed analysis has been performed on the ability of AOP to remove redundant code in exception handling [Lippert and Lopes, 2000]. This study looked at the code for JWAM, a framework for interactive business applications, which is implemented in over 614 Java classes in 44,000 lines of code. It was discovered that 11% of the overall code was focused on the concern of exception handling. The core of their work involved a refactoring of the exception handling code into AspectJ. The benefits of this refactorization are obvious. In many types of exceptions, they were able to reduce the amount of redundant code by a factor of 4. Of the top five types of exceptions in the JWAM application, over 90% of the number of `catch` statements were removed. For example, the number of catches of the generic `Exception` type went from 77 in the original code to only 7 catches in the refactored AspectJ code. Similarly, the number of catches of the `SQLException` type went from 46 catches in the original code to only 2 in the aspectized code.

Because the JWAM application was written using *Design by Contract* [Meyer, 1997], there are many assertions that test the pre- and post-conditions for a particular method. Lippert and Lopes found that over 375 post-conditions contained an assertion of "`result != null`" – this redundant assertion represented 56% of all post-conditions (here, redundancy referes to the replication of a single statement at the end of multiple methods). There were also 1,510 pre-conditions that contained the assertion of "`arg !=`

`null`"; using AspectJ, that number was cut down to 10. That is, the 1,510 pre-conditions were separated into 10 aspects, where each aspect contained a concise specification of the methods that were to contain the assertion.

The idea of superimposition, which is related to the "diffusing computation" concept initially proposed in [Dijkstra and Scholten, 1980], has recently been compared to aspect-orientation. A superimposition has been found helpful in distributed systems for maintaining and changing the global properties related to a distributed computation (e.g., deadlock detection, or the snapshot algorithm in [Chandy and Lamport, 1985]). Typically, the implementation that manages each globally distributed property is scattered in two ways: it is scattered across the processes that perform the distributed computation, and it is scattered across the source code implementation that is charged with the task of maintaining the global property. It has been noted that, "Algorithms which were intentionally designed to superimpose additional functionality on a basic program have a long history in distributed systems research, probably starting with algorithms to detect termination of basic algorithms" [Katz and Gil, 1999]. Like aspect-orientation, superimpositions impose additional functionality to a base program through quantification.

Enforcing Programmer Discipline

Aspects can be used to enforce certain properties of a system that would typically be left to programmer discipline. To understand this point, reconsider the trigger example from Figure 1. Rather than using a trigger, a database administrator could have written a stored procedure, called `UpdateSalary`, which provides a single point of control for

updating the salary field of the employee table. The `UpdateSalary` stored procedure could then contain, in one location, the semantics for implementing the business rule.

This solution, however, does not provide any guarantee that others will obey the rule for using only this stored procedure. There is nothing to prevent a user or developer from updating the table through means other than the stored procedure. The reliance on programmer discipline is unfeasible in large systems, and it is quite likely that certain system properties are violated when there is no direct way to enforce the concern. Aspects can be helpful in enforcing that a particular policy, or protocol, is observed in a way that does not rely on the programmer remembering to conform to a large set of unverifiable rules.

AspectJ

Early aspect languages, like COOL and RIDL [Lopes, 1997], dealt with specific types of concerns (e.g., synchronization and distribution). The most mature language, however, is a general aspect language (called AspectJ) that is an extension to Java. It is described as being general because it is not tied to capturing a particular kind of concern; instead, it provides general constructs that allow a programmer to capture a wide variety of different kinds of concerns. The language definition has undergone many changes since the first description in [Kiczales et al., 1997] to the most recent implementation, as documented in [Kiczales et al., 2001a] and [Kiczales et al., 2001a]. This section highlights some of the key characteristics of AspectJ.

AspectJ is being used in commercial development. CheckFree.com, which provides financial services for e-commerce, uses AspectJ [Miller, 2001]. An interesting anecdote is reported from this effort. A senior engineer at CheckFree stated that AspectJ

allowed his team to implement a crosscutting feature in four programmer-hours. The same feature, implemented in a previous version of the application in C++, is reported to have taken two programmer-weeks [Tristram, 2001].

It has been proposed that there are three critical parts to an aspect composition language: a join point model, a way of denoting join points, and the ability to specify behavior at those join points [Kiczales et al., 2001b]. Each of these items needs to be defined to allow for a proper understanding of the AspectJ examples in Appendix A.

Join Points and Pointcuts

In AOP languages like AspectJ, a *join point* denotes the location in the program that is affected by a particular crosscutting concern. This location can be either the static location of a specific line of source code, or it can represent a dynamic point during the execution of the program. There are many potential join points in a program. A *pointcut* specifies a collection of join points. The AOP literature does not provide the etymology of this term. Perhaps the intent of the terminology comes from graph theory, where the notion of a cutpoint represents a vertex in a graph whose removal would leave the graph in a disconnected state. It is a point of separation between nodes in a graph. Analogously, a pointcut is a place of potential separation for non-orthogonal concerns.

A pointcut designator is declarative and permits the composition of join points using logical operators. There are many different types of pointcut designators. Several designators that will be used in a later example are:

- `this(T)`: all join points where the currently executing object is an instance of class T
- `target(T)`: all join points where the target object of a call is an instance of class T
- `call(S)`: all join points (in a calling object) that are matched by a call specified by signature S
- `cflow(C)`: this powerful designator selects all join points within the control flow of pointcut C

Advice

Whereas a join point represents a location where an aspect adds behavior, *advice* represents the behavior to add (Note: The name "advice" was chosen because it is similar to the advice feature in early Lisp machines). Advice represents a type of method that can be attached to pointcuts. The definition of an advice relates a pointcut with specific code, contained in the advice body, which takes care of the crosscutting concern. The body of the advice is normal Java code.

There are three different designators for specifying the point of execution for advice: `before`, `after`, and `around`. The choice of these names appears to have been borrowed from CLOS [Steele, 1990]. In `before` advice, the advice body is executed prior to the execution of the join point's computation. The opposite is true with `after` advice; the advice runs after the join point computation. There are even three different kinds of `after` advice:

- after the successful execution of the join point (`after returning`);
- after an error was encountered during the execution of the join point (`after throwing`);
- either of the above two cases (`after`).

An example of both before and after advice is provided in Appendix A in Figure 42.

Weaving

Having divided to conquer, we must reunite to rule.

[Jackson, 1990]

Separation of concerns often necessitates subsequent integration. Whereas AOP provides the capability of separating numerous concerns during development, the effects of the crosscuts must be integrated back into the target code. The goal of the separation is to improve the conceptual ability of programmers during development – the end result at run-time, however, will certainly have crosscutting concerns that are transparent. As David Weiss states, in his introductory comments to one of Parnas' papers, "At run-time, one might not be able to distinguish what criteria were used to decompose the system into modules" [Hoffman and Weiss, 2001].

In AOP, a translator called a weaver is responsible for taking code specified in a traditional programming language, and additional code specified in an aspect language, and merging the two together. Because the aspect code describes numerous behaviors that crosscut a system, the concerns must eventually be integrated into the base code. This is the purpose of a weaver – it integrates aspects into the base code. In Figure 8, the weaving process is depicted using the previous example in Figure 5.

One way that a weaver can perform its translation is by creating two separate Abstract Syntax Trees (ASTs). One tree represents the base program and another tree represents the parsed aspects. Walking along the aspect tree drives the transformations that need to be performed to the tree that represents the base program.

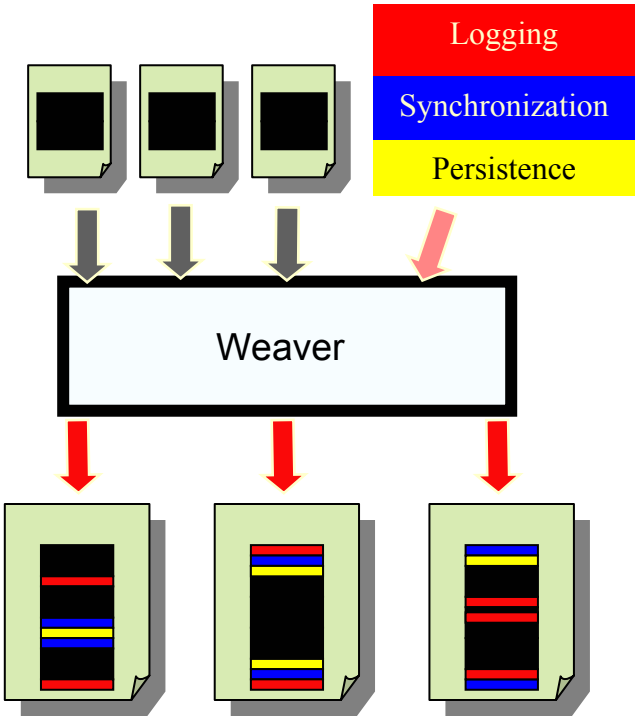


Figure 8: The Weaving Process

An Example: Bureaucracy and Wormholes

> *Where the hierarchy is used for communication, each step in the communication chain acts as a screening point to decide how much of the information needs to be communicated further down or up the line, as the case may be…Hierarchical channels are usually very slow, however, for the indispensable crosswise flow of information…Hence, in actual organizations information flows through many channels, formal and informal, other than the hierarchy.*
>
> [Simon, 1950]

In Chapter 1, a section was included that compared software modularization with some of the topics in organization theory. In that section, the hierarchical layers of communication were cited as a source for the problems usually associated with the "red-tape" of bureaucracies. This is characterized by the communication pattern depicted in Figure 9.

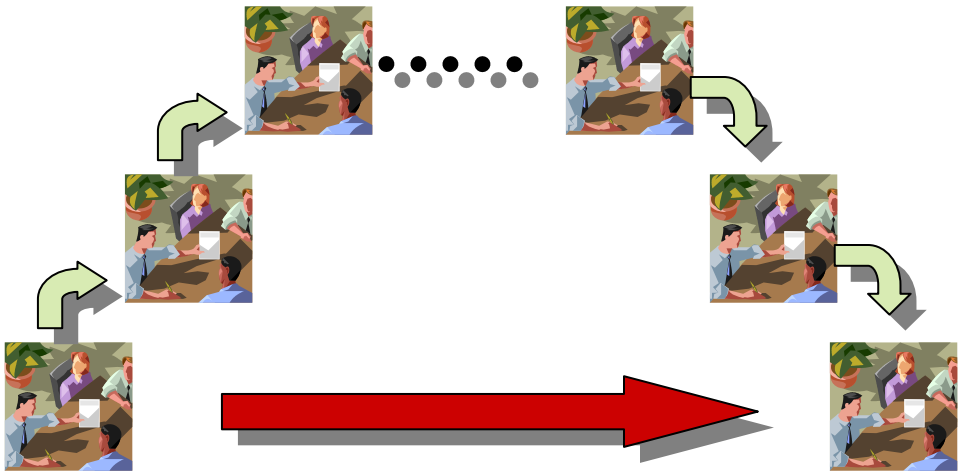


Figure 9: Organizational Bureaucracy

A portion of the communication pattern that is present in Figure 9 is very similar to a situation experienced occasionally by software developers. Within the life-time of every programmer, a situation will arise when contextual information that is located at the top of a layer of method calls is needed by methods that are at the bottom of the layer. In such circumstances, all of the signatures of the intermediate methods within the call flow need to be altered such that the contextual information passes through the chain of calls. This is an error-prone activity that can involve many invasive changes to the protocol of communication within the layer. Like the situation described by Simon in the

opening quote of this section (this is also pictorially represented in Figure 9 by the arrow along the bottom of the figure that represents the type of direct communication that circumvents the hierarchy), a better solution to the programming problem just described would involve some mechanism for direct communication between the top and bottom participants in the call flow.

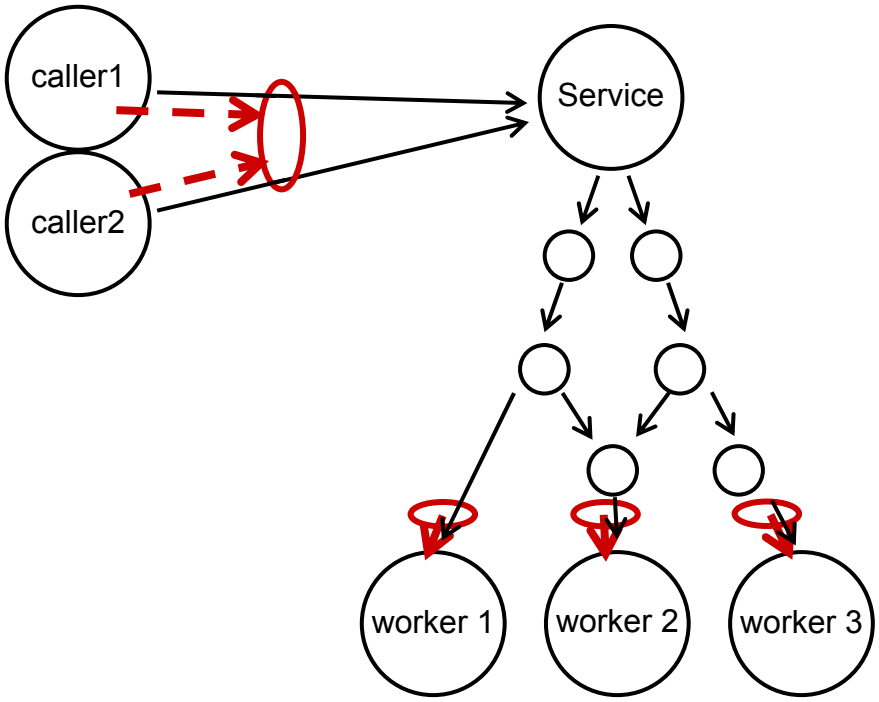


Figure 10: The AspectJ Wormhole Example
(Reprinted from [Hilsdale et al., 2001], with permission from Gregor Kiczales.)

Figure 10 represents an example that was borrowed from the AspectJ team (they term this their "wormhole" example). It provides an instance of the context passing problem just described. The contextual information that is passed from a caller is absorbed within a wormhole at the top of the layer, and reappears at the bottom of the

layer. The programmer does not need to explicitly modify the internal structure of the flow.

The solution to the above problem can be found in the aspect of Figure 11. In that solution, the `invocations` pointcut represents those callers that invoke an operation on the top-level of the layer. The `workPoints` pointcut denotes the objects at the bottom of the layer that implement some desired functionality. The real power of the technique is contained within the `perCallerWork` pointcut, which unites the `workPoints` with the control flow emanating from the caller `invocations`. The `before` advice on `perCallerWork` passes the context of the callers on to the worker objects.

```
abstract aspect ContextPassing {

  pointcut invocations(Caller c):
    this(c) && call(void Service.doService(String));

  pointcut workPoints(Worker w):
    target(w) && call(void Worker.doTask(Task));

  pointcut perCallerWork(Caller c, Worker w):
    cflow(invocations(c)) && workPoints(w);

  before (Caller c, Worker w): perCallerWork(c, w) {
    w.doSomethingWithContext(c);
  }

}
```

Figure 11: An AspectJ Wormhole Solution
(Adapted from [Hilsdale et al., 2001], with permission from Gregor Kiczales.)

Other Work in Aspect-Oriented Software Development (AOSD)

Several researchers are working in the area of AOSD to provide new language constructs to support crosscutting concerns [Tarr et al., 1999]. Aside from AOP, other examples of specific research in this area are Subject-Oriented Programming (SOP) [Osher et al., 1996], variants of Adaptive Programming (AP) [Lieberherr et al., 2001], and Composition Filters (CF) [Bergmans and Aksit, 2001]. A hybrid approach to applying these techniques has been suggested in [Rashid, 2001]. Several of these research areas can be considered a part of generative programming, the topic of the next section.

Multi-Dimensional Separation of Concerns (MDSOC)

Another successful approach for dealing with crosscutting concerns is Subject-Oriented Programming (SOP), a research effort at IBM Research. In this approach, it is recognized that objects have different roles that they represent. These different roles can be composed into system features [Ossher et al., 1996], [Ossher and Tarr, 2001]. For example, in an `Employee` class, an employee object plays different roles depending on whether the `Employee` is being sent to the payroll subsystem (where salary and tax information are pertinent) versus the same `Employee` being sent to the human resources, or personnel, subsystem (where years of service and address are appropriate). The separation of these roles into isolated views is referred to as a "hyperslice" [Tarr et al., 1999]. Hyperslices assist a team of programmers in independently developing different concerns that may apply to a single class. Note that this capability was one of the Parnas' criteria described in the first chapter [Parnas, 1972].

Earlier work on subdivided procedures provided a basis for the approach adopted in SOP [Harrison and Ossher, 1990]. Subdivided procedures promote extensible programming by separating the multiple cases of procedure bodies. A procedure that dispatches from a large case statement would be an example application of subdivided procedures. In such instances, the individual cases that comprise the procedure are somewhat related to the notion of a hyperslice.

An interesting comparison can be made between AOP and SOP. With AOP, the focus has always been on crosscutting concerns that are spread across multiple modules. A focus of SOP, however, has been the ability to capture several views of a single class. The separation of these views, it is argued, permits a better understanding of the implementation of each view in isolation so that the views do not become tangled. In the SOP literature, a translator called a *compositor* has numerous similarities to a weaver in AOP. A programmer creates composition rules that direct the output of the compositor [Ossher et al., 1996]. A tool called Hyper/J has been developed to support the idea of hyperslices in Java.

Adaptive Programming

The structure of objects within a class hierarchy has been found to be a type of crosscutting concern. In Adaptive Programming (AP), a key focus is the separation of behavior from structure. To aid in the modularization of this concern, visitor and traversal strategies are used [Lieberherr, 1996]. This modularization prevents the knowledge of the program's class structure from being tangled throughout the code, a desirable property that is called "structure shyness." Traversal strategies can be viewed as a specification of the class graph that does not require the hardwiring of the class structure throughout the

code [Lieberherr et al., 2001]. An example of a traversal/visitor language for supporting such modularization is described in [Ovlinger and Wand, 1999].

The AP community considers their research as a special case of AOP. The motivation for AP came from the earlier work on the *Law of Demeter*, which offered a set of heuristics for improving the cohesion and coupling of object-oriented programs (the motto of this work was the anti-social message of "Talk only to your immediate friends") [Lieberherr and Holland, 1989].

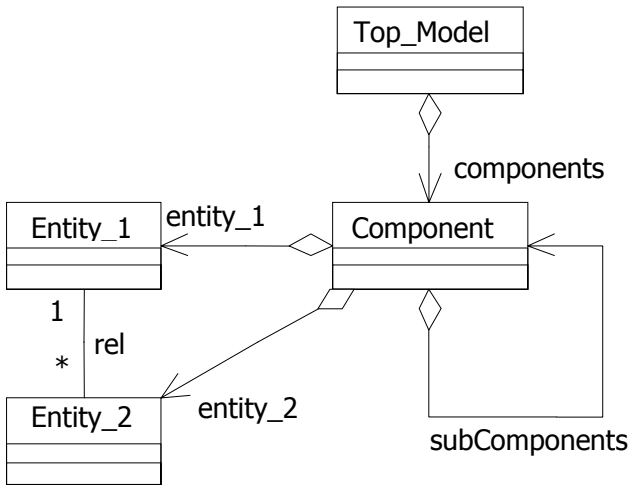


Figure 12: A Simple UML Tool Model Specification

In previous work at ISIS, an adaptive programming approach was used to solve a tool integration problem for a large aerospace firm [Karsai and Gray, 2000]. The domain for the integration focused on fault-analysis tools, where each tool persistently stored a model in either a database or a textual format (e.g., either comma-separated values, or a proprietary format). In that work, a model from one tool was translated into the representation of another tool. To accomplish this, semantic translators were used to

traverse the graph of an internal representation of a model. In a semantic translator, the specification of the traversal, and the actions to be performed at each traversed node, are separated.

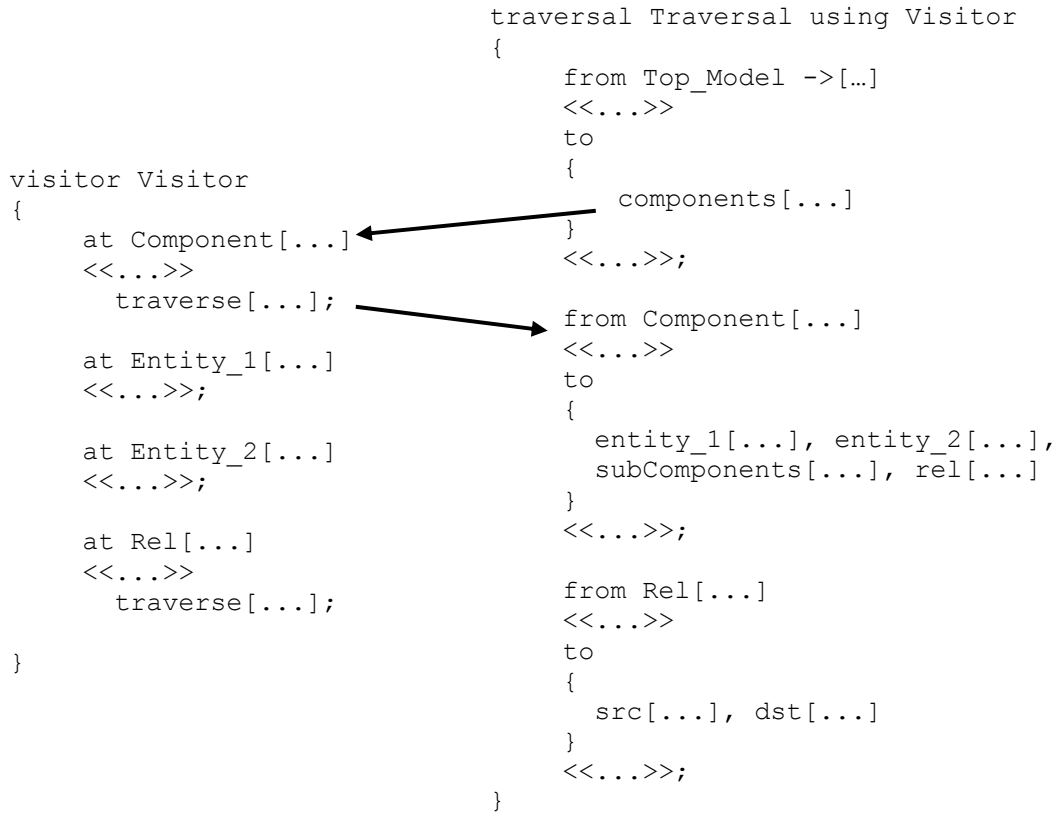


Figure 13: Traversal/Visitor Specifications

The illustration in Figure 12 represents a simple model that is specified in the Unified Modeling Language [Booch et al., 1998]. A domain-specific language (DSL) for textually representing this diagram is presented in [Karsai and Gray, 2000]. Another DSL is shown in Figure 13, which demonstrates the traversal/visitor specifications that appear within a translator. During a translation, the process begins with the top model and

follows along the traversal specifications. At visitor nodes, a specific action is performed that executes the required translation (these are elided inside of the inline code, which is denoted as <<…>>). In Figure 13, the first two steps in the model translation are shown by two arrows. The remaining traversal/visitor sequence would follow similarly.

Composition Filters

An earlier effort at isolating crosscutting concerns is the composition filters approach. With this technique, explicit message-level filters are added to objects and the messages that they receive [Aksit et al., 1992], [Bergmans and Aksit, 2001]. The motivation for composition filters came from the recognition that conventional object models lack the required support for separating functionality from message coordination code. As objects send messages to each other, the messages must pass through a layer of filters. Each filter has the possibility of transparently redirecting a message to other objects. Different types of filters have been found to be effective at isolating constraints and error checking [Aksit et al., 1994]. The CF approach can be very useful in executing actions before and after the interception of a method call. A related technique, proposed in [Filman et al., 2002], intercepts communication among functional components and injects behavior to support various additional capabilities (e.g., reliability, security). CORBA interceptors [Narasimhan et al., 1999] have some similarities with composition filters because they also can intercept messages.

Future Research Directions in AOSD

> *All technical evolution has a fundamental behavior pattern. First there is scientific discovery of a generalized principle, which occurs as a subjective realization by an experimentally probing individual. Next comes objective employment of that principle in a special case invention. Next the invention is reduced to practice. This gives humanity an increased technical advantage over the physical environment. If successful as a tool of society, the invention is used in bigger, swifter, and everyday ways. For instance, it goes progressively from a little steel steamship to ever-bigger fleets of constantly swifter, higher-powered ocean giants.*
>
> [Fuller, 1981]

There are many exciting things on the horizon for research in aspect-oriented software development. The remainder of this section surveys some of these other research areas.

Weaver Development and Tool Support

Some of the earliest aspect languages and weavers were focused on specific concerns like synchronization and distribution. Examples of these particular aspect languages include COOL and RIDL, as defined in the dissertation of Cristina Lopes [Lopes, 1997]. More recent work, like AspectJ, has focused on generic aspect languages. Aside from Java and AspectJ, other languages are being explored with respect to AOP. The use of AspectC was cited earlier in the discussion of prefetching [Coady et al., 2001]. Although there are many difficulties in writing a C++ parser, initial efforts at providing an AspectC++ weaver (in support of real-time systems) are reported in [Gal et al., 2002], [Mahrenholz, 2002]. AspectS is an approach to general-purpose AOP in the Squeak environment [Hirschfield, 2001]. Apostle is an aspect weaver for Smalltalk [de Alwis, 2001]. A simple weaver even exists for Ruby [Bryant and Feldt, 2001]. Additionally, there has been work on making the CORBA IDL aspect-oriented [Hunleth et al., 2001],

as well as efforts for bringing AOP into the realm of Microsoft .NET [Shukla et al., 2002], [Lam, 2002].

All of the weavers mentioned above are typically much more immature than the capabilities offered in AspectJ, yet they provide the major impetus for taking the ideas of AOP to other languages. In addition to weaver development, there are several other development tools that are being created to support AOP. A debugger for AspectJ, with GUI support, is available. There also has been effort to support AspectJ within several Integrated Development Environments (IDEs).

Another related interesting research area is the application of AOP to compilers. As observed in [Tsay et al., 2000], "The code to do one coherent operation is spread over all node classes, making the code difficult to maintain and debug." The advantages of using AOP techniques for a weaver can be found in [de Moor et al., 1999]. In their work, the descriptions of the effects on attribute grammars are separated from the grammar productions. The benefit of this was also recognized in [Van Wyk, 2000].

Debugging Aspect Code

> *I do not want to imply that support of paradigms is limited to our programming language proper. The entire environment in which we program, diagnostic systems, file systems, editors, and all, can be analyzed as supporting or failing to support the spectrum of methods for design of programs.*
>
> [Floyd, 1979]

Many aspect weavers are preprocessors that target their output code in another traditional programming language. Given the obfuscation created by the mangled names, and the numerous indirections present in the generated code, it seems that there is a

mismatch between the implementation space and the execution space. That is to say, how does a programmer write code using a particular conceptualization, and then debug the generated code that is void of that conceptualization? This question is not peculiar to AOP – the problem can be found in almost any implementation of a domain-specific language [Faith et al., 1997], [van Deursen and Knit, 1997].

To answer the question concerning the debugging of aspect code, it should be recognized that AOP is still in its early infancy. Although tool support is being developed, such as an aspect debugger, the technology is still immature. Yet, it is reasonable to expect future tools will be developed that will make the underlying execution transparent to the paradigm. In fact, the path that AspectJ is taking is not unlike the development of the earliest C++ compilers. The initial C++ compilers were merely preprocessors that generated C code. The resulting C code was void of any semblance of true object-oriented concepts – the C++ representation was merely simulated in a language that had more mature compilers. The same can be said of AspectJ and other languages concerning the incubation period needed for growth and stabilization. Perhaps a future solution to this problem will be found in an adaptation to the work in [Faith, 1997], which describes a tracking engine that interacts with a debugger and maps nodes from syntax trees.

Analysis and Design with Aspects

A study of the history of software development paradigms reveals that a new paradigm often has its genesis in programming languages and then moves out to design and analysis, or even other research areas (see [Rashid and Pulvermueller, 2000] for a description of aspects applied to databases). This same pattern also can be observed with

respect to aspect-orientation. Most of the existing work on advanced separation of concerns has been heavily concentrated on issues at the coding phase of the software lifecycle. There have been, however, efforts that have focused on applying advanced separation of concerns in earlier phases of the software lifecycle. One of the first examples of this type of work can be found in [Clarke et al., 1999], where the principles of SOP were applied at the design level. Similarly, [Herrero et al., 2000] have investigated the benefits of aspects at the design level. Extensions to the UML have been proposed in order to support composition patterns as a facility for handling crosscutting requirements [Clarke and Walker, 2001], [Clarke, 2002].

A set of generic design principles for aspect-oriented software development is the focus of [Chavez and de Lucena, 2001]. An analysis of design patterns, and the aspect-oriented techniques that can improve their specification and implementation, are the subject of [Nordberg, 2001]. There has been an increased interest in the need for formal verification of systems designed with support for crosscutting concerns. The most mature effort in this area can be found in [Nelson et al., 2001], where two formal languages are presented that assist in the verification of concerns focused on concurrent processes.

Aspect Mining

There is an overwhelming amount of legacy code that has been written in languages that do not support the clean separation of crosscutting concerns. To convert legacy code into languages that support AOSD, it is necessary to refactor the original program. A correct refactoring into a cleaner separation of concerns requires the examination of the original code with an eye toward aspect mining (i.e., the identification and isolation of aspects).

An aspect mining tool offers assistance in this process. The Aspect Browser tool, presented in [Griswold et al., 2001], is such an example. The tool has been applied to a case study that contained 500,000 lines of source code in FORTRAN and C. Another tool for aspect mining is described in [Hannemann and Kiczales, 2001]. This tool generated the graphic in Figure 6.

AOP Validation Research

Case studies that transform legacy applications into AspectJ, like [Lippert and Lopes, 2000] and [Kersten and Murphy, 1999], provide practitioners with heuristics for adopting AOP. Both a case study and an experimental method were used in [Walker et al., 1999] to assess AOP. In an experiment that studied the ease of debugging, three synchronization errors were introduced into a Java program. A separate program that duplicated the errors was also written in AspectJ. Several teams of programmers were given the task of tracking down the errors in each of the implementations. The results of this experiment show that AspectJ provided a clear benefit to increasing localized reasoning, but no benefit when the solution required non-localized reasoning. Here, localized reasoning refers to whether or not a programmer needs to leave the context of the module (in this study, the file) that contains the error. Overall, the program teams that used AspectJ isolated and fixed the errors quicker than those who used pure Java.

There are case studies that have compared the various different mechanisms for supporting advanced separation of concerns [Murphy et al., 2001]. Obviously, as AOP matures, additional studies will be needed to determine the benefits of these new approaches.

Aspect Reuse

As a large collection of different types of aspects is assembled, the idea of aspect reuse will become an interesting research topic. AOP presents new issues for reuse researchers [Grundy, 2000]. In order to be successful at aspect reuse, developers will need to begin writing their aspects in a more generic style than is currently prevalent. To see why this is so, consider the code fragments that are provided as Figure 41 and Figure 42, in Appendix A. The pointcuts of these aspects are concretized and bound specifically to the methods called `DisplayError` and `Handle`. This assumption is too strong. It may often be the case that others will want to reuse this aspect, but their code does not conform to these concrete names. To remedy this problem, a style of pointcut designation is needed such that the pointcuts of the reusable aspects are abstract. In this case, those who would wish to use and extend an abstract aspect must concretize it. In fact, AspectJ permits such designations, but its use is very infrequent in the current aspect code that is being developed. Some of the issues in support of aspect reuse and composition have been initially explored in the work on aspectual components [Lieberherr et al., 1999].

Another research issue occurs in the reuse of orthogonal aspects that apply to the same join point. This issue is important because the ordering of the generated code may be essential. For example, given the two previous aspects of locking and logging, it is often the case that, when applied to the same join point, the mutex code should appear before the logging instructions. AspectJ provides the `dominates` construct to allow the specification of priority between two different aspects. It is unclear, however, whether this construct alone is able to allay all of the possible problems in composing several aspects within the same join point.

Generative Programming

> *We must recognize the strong and undeniable influence that our language exerts on our ways of thinking and, in fact, delimits the abstract space in which we can formulate – give form to – our thoughts.*
>
> [Wirth, 1974]

The first FORTRAN compiler took 18 programmer-years to complete [Backus et al., 1957]. One could argue that the time that it would take today to write an equivalent compiler would be on the order of programmer-months, not programmer-years. Of course, much of the decreased development time would be related to the experience that has been collected on the topic of compiler construction. Most would agree, however, that the principal reason for the decreased development time would be that we have moved beyond the manual handcrafting of "one-of-a-kind" solutions to an approach that resembles an automated assembly line. To be specific, in the case of implementing a simplistic version of a FORTRAN compiler, a programmer today would use parser generators, specialized components, and perhaps even object-oriented frameworks.

In implementing a compiler using modern techniques, the reduction in development time is the result of a paradigm shift toward the engineering of families of systems, as proposed in [Parnas, 1976]. The idea of a family of systems is best categorized as a domain-specific product-line architecture, where a set of different products can be created from adaptations that are made from a set of varying features [Clements and Northrop, 2001]. An excellent example of this idea is found in [Delisle and Garlan, 1990], which describes development at Tektronix on a family of oscilloscopes.

An additional contributing factor to the relative ease in constructing a modern-day FORTRAN compiler is in the recognition that many of the arduous implementation

details of software construction can be handed off to a generator. This paradigm shift has led toward a research area that has been dubbed Generative Programming (GP). Generative programming is accomplished by transforming higher-level representations of programs into a lower-level equivalent representation.

This section surveys several of the promising research areas that are being associated with the GP movement. More detailed coverage of GP can be found in [Czarnecki and Eisenecker, 2000].

Domain-Specific Languages

> *The first order term in the success equation of reuse is the amount of domain-specific content and the second order term is the specific technology chosen in which to represent that content.*
>
> [Biggerstaff, 1998]

A Domain-Specific Language (DSL) is a, "programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain" [van Deursen et al., 2000]. DSLs assist in the creation of programs that are more concise than an equivalent program written in a traditional programming language. An upward shift in abstraction often leads to a boost in productivity. It has been observed that a few lines of code written in a DSL can generate a hundred lines of code in a traditional programming language [Herndon and Berzins, 1988]. A key advantage is that a DSL is perspicuous to the domain expert using the language. A DSL is typically more concise because much of the intentionality of the domain is built into the generator. To use a connotation borrowed from Polya, the intent of a DSL is "pregnant with meaning" [Polya, 1957].

A DSL can assist in isolating programmers from lower-level details, such as making the decisions about specific data structures to be used in an implementation. Instead, a programmer uses idioms that are closer to the abstractions found in the problem domain. This has several advantages:

- The tedious and mundane parts of writing a program are automated in the translation from the DSL to a traditional programming language.
- Repetitive code sequences are generated automatically instead of the error-prone manual cut-and-paste method. The generation of error-prone code also has advantages during the maintenance phase of a project's lifecycle. Programs written in a DSL are usually easier to understand and modify because the intention of the program is closer to the domain.
- Solutions can be constructed quickly because the programmer can more easily focus on the key abstractions.

The size and scope of a DSL is much smaller than that of a traditional programming language. In fact, DSLs are often called "little languages" [Bentley, 1986], [van Deursen and Knit, 1997], [Aycock, 1998]. Another common characteristic is the declarative nature of these languages. In some cases, a DSL can be viewed as a type of specification language in addition to a general purpose programming language. A DSL can be declarative because the domain provides a particular underlying interpretation. The notations and abstractions of the domain are built into the generator that synthesizes a program written in a DSL.

A DSL translator can be implemented using the standard approaches for constructing a compiler or interpreter [Aho et al., 1986]. However, the majority of the

literature implements DSLs with a preprocessor. Although this approach can be simpler than writing a complete compiler, it has several disadvantages. The main disadvantage is that the generated code is converted to a base programming language. This means that type checking and other compile-time tests are done outside of the domain. It also means that feedback from run-time errors are couched in terms of the base language, not the domain. A solution to this problem (previously cited in the section on "Debugging Aspect Code") is suggested in [Faith, 1997]. There are other disadvantages in using a DSL that often arise later in the development cycle. As observed in [van Deursen and Knit, 1997], the use of a DSL introduces new maintenance issues. For instance, the generators that process the programs in a DSL may often need maintenance.

Domain-Specific Modeling

> *...allows a computationally naïve user to describe problems using natural terms and concepts of a domain with informality, imprecision, and omission of details.*
>
> [Barstow, 1985]

> *An important step in solving a problem is to choose the notation. It should be done carefully. The time we spend now on choosing the notation may be well repaid by the time we save later avoiding hesitation and confusion. Moreover, choosing the notation carefully, we have to think sharply of the elements of the problem which must be denoted. Thus, choosing a suitable notation may contribute essentially to understanding the problem.*
>
> [Polya, 1957]

The aim of Domain-Specific Modeling (DSM) is similar to the objective of DSLs in that expressive power is gained from notations and abstractions aligned to a specific problem domain. Typically, a DSM relies on graphical representations of the domain abstractions, as opposed to the textual form of a traditional DSL. Also, a program in a

DSL is usually given a fixed interpretation, but a model in a DSM may have multiple interpretations (e.g., one interpretation may synthesize to C++, and a different interpretation may synthesize to a simulation engine).

Research on domain-modeling using UML has focused on the UML stereotype and profile facilities for introducing domain-specific knowledge (a few examples of this prevalent method can be found in [Bettin, 2001], [Clauß, 2001], [Gogolla, 2001]). A potential problem exists with this approach whenever the domain metamodel and domain model both are represented using this notation. The intent of these notational extensions is represented in a form that may not be familiar to the domain expert (i.e., models are simply represented as annotated UML class diagrams). The intention and semantics of the domain also tend to be scattered across the domain model diagrams. As observed in [Nordberg, 2001], "Object-oriented designs tend to become littered with mechanism classes – classes that serve a critical software function but have no correspondence to real world objects." In more mature domain-specific modeling synthesis environments, these "mechanism classes" often are absorbed within the model interpreter. That is, the behavior of these mechanism classes is understood by the modeler to be provided as part of the intention of the domain.

Like DSLs, domain-specific modeling raises the level of abstraction to highlight the key concerns of the domain. A Domain-Specific *Visual* Language (DSVL) is capable of removing the designer from being tied to specific notations like the UML. In domain-specific modeling using a DSVL, a design engineer describes a system by constructing a visual model using the terminology and concepts from a specific domain. Analysis can then be performed on the model, or the model can be synthesized into an implementation.

At ISIS, an approach called Model-Integrated Computing (MIC) has been refined over many years in order to assist in the creation and synthesis of computer-based systems [Karsai, 1995]. A key application area for MIC is in those systems that have a tight integration between the computation structure of a system and its physical configuration. In such systems, MIC has been shown to be a powerful tool for providing adaptability in frequently changing environments. An example of the flexibility provided by MIC is documented in [Long et al., 1998], where an installed system at Saturn was shown to offer significant improvements in throughput by being able to adapt to changes in business needs and the physical environment.

A specific instance of the type of domain-specific modeling supported by MIC is implemented using a core tool – the Generic Model Editor (GME) [Lédeczi et al., 2001]. The GME is a modeling environment that can be configured and adapted from metalevel specifications (this is called the modeling paradigm) that describe the domain [Nordstrom et al., 1999]. In using the GME, a modeler loads a modeling paradigm into the tool. This provides an environment containing all of the modeling elements and valid relationships that can be constructed in a specific domain.

A complex modeling task often requires the leveraging of knowledge and expertise in numerous scientific and engineering disciplines. The successful use of an environment like the GME necessitates the collaboration and the skillful execution of the roles of domain expert, environment developer, and experienced programmer. The participants in these roles must synergistically come together in the following way:

- Domain Expert: The role of the domain expert is to construct the domain-specific model. They do not possess intricate knowledge of the GME. They only need a

basic familiarity that would allow them to create and navigate around the model. They do, however, require detailed insight into the various minutiae of the underlying domain.

- Environment Designer: The creation of the domain-specific metamodel, which represents the description of a particular modeling environment, is an arduous task. The metamodel must contain all of the concepts that the domain expert needs to create a model. The individuals responsible for this role must have an understanding of the specific domain, as well as an appreciation of the GME API. This participant must wear two different hats – part programmer, part domain expert.
- GME Developers: The GME developers are unique in that they do not possess a priori knowledge of the specific domain in which the GME will be applied. They must, however, have a great understanding of general modeling concepts and how those concepts are implemented in a programming language.

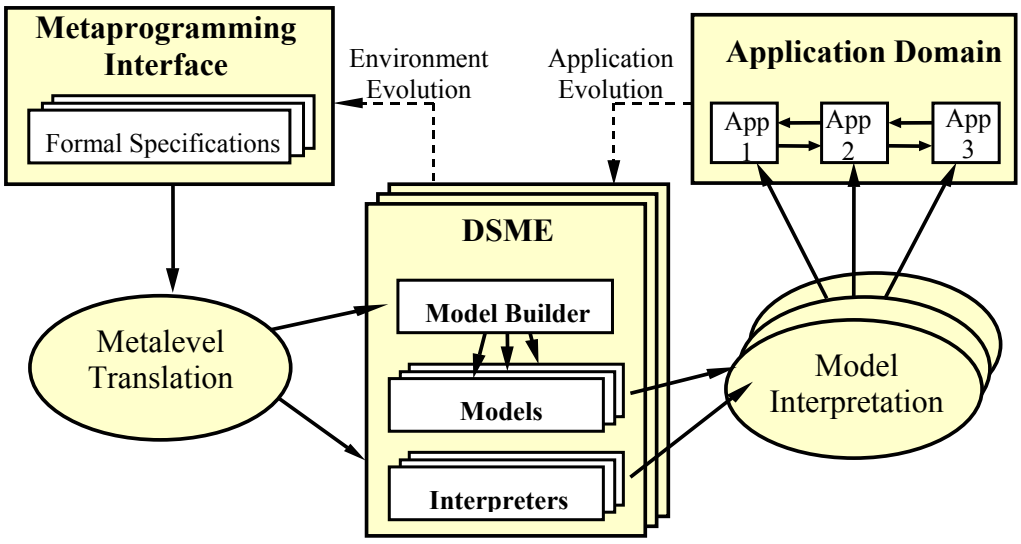


Figure 14: Model-Integrated Computing

The process for applying MIC is shown in Figure 14. The left-hand side of this figure describes the task of creating new modeling environments. From metalevel descriptions, new modeling environments are generated from metalevel translators (note that this process is self-descriptive – the metalevel descriptions are also created within a modeling environment [Nordstrom et al., 1999]). These metalevel specifications define the ontology of the domain. That is, the specifications identify the pertinent entities of the domain, as well as their various associations. Once a modeling environment is generated, a domain expert can then create models for the particular domain associated with the environment (see the middle of Figure 14). Once a model is created, it can then be processed by domain interpreters. An interpreter walks the internal data structure that represents the model and generates a new artifact. These interpreters can synthesize an application to specific execution platforms, as well as generate input to analysis tools. The synthesis task is shown on the right-hand side of Figure 14.

Within the context of the objectives of this dissertation, GME models will be presented in subsequent chapters to illustrate the aspect modeling approach.

Example Domains

There are numerous domains where DSLs have been applied. Some of the example domains are telecommunications [Bonachea et al., 1999], operating systems [Pu et al., 1997], typesetting and drawing [Bentley, 1986], web services [Fernández et al., 1999], caching policies [Barnes and Pandey, 1999], [Gulwani et al., 2001], and databases [Horowitz et al., 1985]. The concept of a domain-specific metalangauge has also been put forth as a technique for assisting in the domain of language translators [Van Wyk, 2000]. An extensive annotated bibliography of research in this area can be found in [van

Deursen et al., 2000]. Domain-specific modeling has been successfully applied in several different domains, including automotive manufacturing [Long et al., 1998], digital signal processing [Sztipanovits et al., 1998], and electrical utilities [Moore et al., 2000].

Generators

> *In the words written on the wall of a Stanford University graduate student office, "I would rather write programs to help me write programs than write programs."*
>
> [Floyd, 1979]

Compilers for DSLs have often been called *application generators* [Horowitz et al., 1985], [Cleaveland, 1988], [Smaragdakis and Batory, 2000]. A *generator* is a tool – a type of translator or compiler – that takes as input a domain-specific language and produces as output source code that can be compiled as a traditional programming language. The internal architecture of a generator is very similar to a compiler. A generator requires: a front-end to parse a source language into an intermediate representation, a translation engine to perform transformations and optimizations, and a back-end to produce the target code.

In [Hunt and Thomas, 2000], a distinction is made between passive code generators and active code generators. In a *passive code generator*, the generator is executed just once to produce a result. After the output of a passive generator is obtained, the result becomes freestanding. The origin of the file is forgotten. An example of this type of generator would be a design wizard, like that described in [Batory et al., 2000]. With a wizard, a user enters various configuration data as a response to interacting with a dialog window. Based upon this configuration information, the wizard can then generate code that would have been tedious to create by hand. The code produced from an *active*

code generator, though, frequently changes such that it is advantageous to invoke the generator on variations of the input.

There is some evidence that generators improve productivity and reliability. A comparative experiment for a Command, Control, Communication, and Information (C^3I) system is described in [Kieburtz et al., 1996]. This experiment compared the use of generators with a previously developed Ada template-based approach for implementing message translation and validation. The results of this experiment show that the teams that used the generator approach were three times more productive than those who performed the same task using templates. The generator approach also realized improvements in reliability, with under half as many test run failures.

GenVoca

GenVoca permits hierarchical construction of software through the assembly of interchangeable/reusable components [Batory and Geraci, 1997]. The GenVoca model is based upon stacked layers of abstraction that can be composed. A *realm* is a library of plug-compatible components. It can be thought of as a catalog of problem solutions that are represented as pluggable components that can be used to build applications in the catalog domain. Each realm exposes a common interface that all components in that realm must satisfy. This provides the ability to have many alternative implementations for the same interface. The layered decomposition of implementations offers component composition that is similar to the stacking of layers in a hierarchical system.

Each realm in the hierarchy is denoted by a GenVoca grammar. This grammar describes all of the legal compositions that may occur within the realm. The composition

of components in GenVoca is performed by writing parameterized type expressions. These expressions are checked against the grammar to preserve validity.

A comparison between GenVoca and AOP is made in [Cardone, 1999]. Both aspect languages and GenVoca type equations guide the transformation of programs. The AOP weaver and the GenVoca generator are the preprocessors that implement such transformations. GenVoca has the capability of validating the correctness of component compositions. This is an issue that has not received much focus within the AOP research community. As mentioned in an earlier section, control over the order in which a weaver applies multiple aspects on the same join point is very limited. GenVoca, though, provides control over the ordering of component composition.

Intentional Programming

> *To the designer of programming languages, I say: unless you can support the paradigms I use when I program, or at least support my extending your language into one that does support my programming methods, I don't need your shiny new languages.*
>
> [Floyd, 1979]

Intentional programming (IP) provides a software development environment that is not tied to a specific programming language. The power of IP is the ability to create new abstractions for languages. It allows the tailorability of a specific language to a new domain. As Charles Simonyi states, "Under IP, domain experts write models/specs/programs in domain terms" [Simonyi, 2001]. The IP system provides the functionality for defining the manner in which these new abstractions interact with the environment's text editor, as well as syntactic and semantic constructs for translating

these extensions to the abstractions already supported in the IP system [Simonyi, 1996]. Thus, IP allows a programmer to write ordinary programs and domain transformations.

The nodes of an Abstract Syntax Tree (AST) typically represent the semantic constructs of a language (e.g., a *while*-loop or *if*-statement). In IP, these nodes are called *intentions*. Many intentions are common across a wide variety of programming languages. The IP environment provides the capability to modify the semantics of an intention for a particular language, as well as introduce new intentions peculiar to that language. New intentions introduce their own syntax in addition to prescribing the effects of interactions with the programmer through an editor. The IP concept of an *enzyme* represents a transformation that is performed on an AST. An enzyme assists in the creation of new intentions that are built on top of existing intentions.

Parser Generators, Language Extenders, and Analysis Tools

Parser generators, like the Purdue Compiler Construction Tool (PCCTS) and YACC (Yet Another Compiler-Compiler), are programs that help in the creation of other programs that perform transformations on source code [Parr, 1993]. In the area of parser generators, an example of an extensible framework for building compilers in Python is described in [Aycock, 1998]. A framework that creates ASTs and associated tree-walker classes, based on the Visitor pattern [Gamma et al., 1995], is described in [Gagnon, 1998]. Other compiler frameworks, like Zephyr [Wang et al., 1997] and SUIF [SUIF2, 2000], provide an extensible framework to support collaborative experimental research. A primary goal of these efforts is to provide an infrastructure to benchmark different techniques that are used in compilers.

The Jakarta Tool Suite (JTS) contains the basic tools to support the addition of new programming features to the Java language [Batory et al., 1998]. It assists in the construction of new preprocessors for DSLs that are transformed into a host language. The supported host language in JTS is called Jak. Jak is described as a superset of Java that supports metaprogramming. It seems likely that JTS could be used to create a weaver for new aspect languages to support Java. The JTS environment builds upon the ideas of GenVoca. Each new extension to Java represents a new realm.

Within the context of the Ptolemy project, a code generator for transforming Java programs is available [Tsay et al., 2000]. This generator is situated within an infrastructure that can parse Java programs and perform transformations on the AST using the Visitor pattern [Gamma et al., 1995].

Program Transformation Systems

A program transformation system is an environment for specifying and performing semantic-preserving mappings from a source program to a new target program [Partsch and Steinbrüggen, 1983]. Typically, a program transformation can be thought of as a string-rewrite. Transformations are specified as rules that involve pattern matching on an AST. The application of numerous transformation rules evolves an AST to the target representation.

A transformation system is much broader in scope than a traditional generator for a DSL. In fact, a generator can be thought of as an instance of a program transformation system with specific hard-coded transformations. There are advantages and disadvantages to implementing a generator from within a program transformation system. A major advantage is evident in the pre-existence of parsers for numerous languages [Baxter,

2001]. The internal machinery of the transformation system may also provide better optimizations on the target code than could be done with a stand-alone generator. A lot of transformation systems incorporate pretty-printing facilities to output the transformed AST in a readable format.

There is a disadvantage to developing a generator using a transformation environment. As stated in [Smaragdakis and Batory, 2000], "Expressing a generator as a collection of transformations has the disadvantage of making the generator dependent on a complicated piece of infrastructure (the transformation system)." In this case, each user of a generator also must be a licensed user of the transformation system. This has the potential for becoming an expensive solution.

An example of a program transformation system can be found in [Baxter, 1992]. This initial research prototype has been enhanced and is now sold as a commercial tool [Baxter, 2001]. Another example of a program transformation system is described in [Faith et al., 1997]. The focus of this research used a transformation system as a test-bed for improving the debugging capabilities within a DSL.

Prem 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. The behavior of an analysis tool can be specified in the GENOA language. The Lisp-like specification is processed by a generator that assists in the construction of new software analysis tools [Devanbu, 1999]. GENOA provides the capability to traverse ASTs and to extract information needed by analysis tools.

Frameworks

> *We consider a set of programs to constitute a family, whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members.*
>
> [Parnas, 1976]

A *framework* can be defined as a skeleton of an application that can be extended to produce a customized program [Fayad et al., 1999]. This type of framework is usually defined as a collection of classes that together help support a domain-specific architecture. A framework architecture must define the objects that are to participate in the framework as well as the interaction patterns among all objects. In this architecture, there is a distinction between those who create the framework and core objects (the framework developer) and the programmer who extends the framework by plugging in their own application objects (the application programmer). Frameworks typically cost more to develop than a single application, although their cost can be amortized over each instantiation [Johnson, 1997].

Adaptability in frameworks is provided by factoring out component objects that implement the core functionality in the application domain from those objects that vary with each instantiation of the framework. A framework instantiation is defined as the insertion of instance-specific classes into the framework architecture. The locations of variability within a framework are referred to as the *hot spots* of the framework [Lewis, 1995]. The instance-specific classes must conform to a predefined interface in order to properly interact with the core objects.

The specification of the hot spots is needed for users of the framework because frameworks exhibit the property of inversion of control. In typical software development,

the components that are written contain the locus of control in the application and selectively pass control onto other library components or lower-level calls to an Application Program Interface (API). In a framework, however, the locus of control resides in the framework, rather than the application objects. The flow of control traverses through the objects of the framework until a hot spot is reached, at which time the application object is dispatched.

Event-based infrastructures also demonstrate the principle of inversion control [Gianpaolo et al., 1998]. In an event-based approach, there is a distinction in the architecture between suppliers, consumers, and the event dispatcher (see Figure 15). Suppliers submit events to a mediating dispatcher that forwards events to all consumer objects that have subscribed to the event (suppliers may also be consumers of other events). The asynchronous nature of the consumers suggests a type of control inversion that provides a high degree of dynamic reconfigurability within distributed object computing. A popular example of this architecture is present in the CORBA event service [Harrison et al., 1997].

Frameworks have been developed in practically every domain that supports variability among a family of products [Fayad et al., 1999], [Fayad, 2000]. One particular interesting research area combines the topic of a previous section (AOP) with a framework for a concurrent object system [Constantinides et al., 2000].

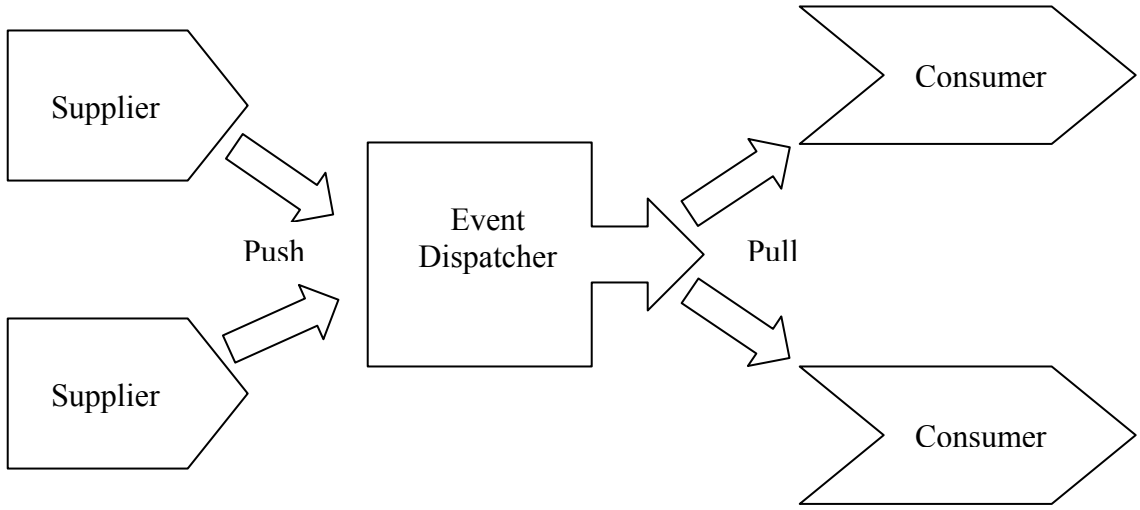


Figure 15: Architecture for Event-based Dispatching

Summary

This chapter provided a synopsis of the techniques that are useful in the development of software that must adapt to changing requirements. The first half of the chapter presented an overview of the literature on reflection, metaprogramming, and AOSD. The research in these areas has produced new ideas and methods for improving adaptability, and for separating crosscutting concerns. This separation provides an advantage for realizing the three objectives presented by Parnas (see "Criteria for Decomposition" in the Chapter 1). The second half of the chapter surveyed research that can be classified under the general area of Generative Programming. A generative approach captures the intent of the problem space at a higher level of abstraction. Generators map the higher abstractions to the lower-level details in the solution space.

In the next two chapters, these techniques (e.g., reflection and metamodeling, advanced separation of concerns, and generative programming) will be extended to support aspect-oriented domain-specific modeling.

CHAPTER III

ASPECT-ORIENTED DOMAIN-SPECIFIC MODELING

This chapter defines a contribution of this dissertation that is related to advancing the notion of Aspect-Oriented Domain-Specific Modeling (AODSM). The chapter begins by differentiating the goals of this research with the work that has been investigated initially within the realm of Aspect-Oriented Design (AOD). A motivation for the work is provided by assessing various techniques for concern separation that are supported by current modeling tools, and critiquing the way in which they fall short of capturing concerns that are crosscutting in nature. The core of the chapter is the presentation of a technique that supports AODSM.

Aspect-Oriented Modeling: Adjective or Verb?

A section in Chapter 2 ("Analysis and Design with Aspects") made reference to work that has been done with respect to bringing aspect-oriented techniques into the purview of analysis and design. The prominent work in that area has been published as [Clarke et al., 1999], [Herrero et al., 2000], [Clarke and Walker, 2001], and [Clarke, 2002]. A focal point of these efforts is the development of notational conventions that assist in the documentation of concerns that crosscut a design. These notational conventions advance the efficiency of expression of these concerns in the design. Moreoever, they also have the important trait of improving the traceability from design to implementation. Without the introduction of aspect-oriented notations into popular modeling languages (like UML), there could be a mismatch (in some cases) when an

object-oriented design is implemented using an aspect-oriented programming language. In the absence of these notations, the intent of a crosscut is captured in an object-oriented design in a way that is awkward. This progression of a paradigm, from implementation to design, is very similar to the evolution of the object-oriented and structured paradigms moving from the implementation level to the design level. The movement of the paradigm up the stages of the software lifecycle aid in reducing the semantic gap between each development phase.

Although these current efforts do well to improve the cognizance of AO at the design level, they treat the concept of Aspect-Oriented Design (AOD) as an "adjective." This is to say that their focus has been on the notational, semantical, and decorative attributes concerned with aspects and their representation within UML. A contribution of this dissertation is to consider AODSM as a "verb." That is, viewing AO as a mechanism to improve the modeling task, itself, by providing the ability to quantify properties across a model *during* the system modeling process. This action is performed by using a weaver that has been constructed with the concepts of modeling in mind. A research effort that also seems to have this goal in mind can be found in [Ho et al., 2002]. Although they claim that their approach is aspect-oriented, it is unclear from the examples they provide. Their work seems more aimed at providing a transformation tool that reifies design patterns at the level of object-oriented design.

Concern Separation in Domain-Specific Modeling

Research and development with the GME has produced several novel, and powerful, techniques for dealing with the problem of separation of concerns at the

modeling level. It is argued in this section, however, that these techniques generally fail to capture modeling concerns that are crosscutting.

Viewpoint Modeling

The concept of viewpoints has been researched frequently as a topic within requirements engineering [Nuseibeh et al., 1994]. Plainly stated, "A viewpoint is an encapsulation of partial information about a system" [Sommerville and Sawyer, 1997], and, "A view is a description of the system relative to a set of concerns from a certain viewpoint" [Hilliard, 1999]. The notion of views/viewpoints is even a key part of the *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems* [IEEE 1471, 2000]. Similarly, the term "view" is frequently used within databases to denote a subset of a table, or join, that highlights the pertinent parts of interest to a specific user of the database [Date, 1999].

The GME supports the concept of a viewpoint as a first-class modeling construct. This assists a modeler in separating the concerns of multi-perspective views [Lédeczi et al., 2001]. As models grow in size and complexity, it becomes unmanageable to view the contents of a model in its entirety; there are just too many participating entities. The viewpoint facility provided within the GME has been labeled as an "aspect." Each GME aspect describes a partitioning that selects a subset of entities to display. These partitions are determined at the metalevel (i.e., the modeling domain), though, and cannot be modified within individual model instances.

Although they offer a powerful conceptualization for concern separation, viewpoints do not fit completely within the definition of aspects (at least in the way that they are defined within the AOP community). Using only viewpoints, for example, a

modeler cannot quantify over a model's join points and apply advice. The key parts of AO, as enumerated in the last chapter, are not fully present in viewpoint-oriented techniques. Another example of research that is classified as being aspect-oriented, but in reality seems to be closer to the viewpoint model, can be found in [Carley and Stewart, 2001].

Type Hierarchies for Modeling

A study of the history of programming languages reveals the great benefit realized from the introduction of typing facilities [Wegner, 1976]. The programmer's ability to create their own user-defined types offers the advantage of being able to generalize and describe the key properties of a common set of entities from the problem domain. Of course, the ability to use types is not only an advantage for writing programs – there are also benefits that accrue when typing is provided in a modeling language.

Types and prototypes are two capabilities that can be very useful in modeling. Modeling tools that support these concepts provide mechanisms to share a common description among numerous objects. A prototype is a representative example of a group of objects that can be reused (or cloned) at other places in the application model. The idea, as it applies to modeling, borrows from the research that has been done in the area of prototype-based programming languages [Craig, 2000].

The GME supports the idea of types and prototypes in order to provide a facility to modelers for categorizing and managing general modeling concepts [Maroti et al., 2002]. Creating clones of prototypes is a simple operation in the GME – the prototype is selected and then dragged to the destination. Clones have the same set of attributes as their prototypes. By modifying the value of an attribute of some prototype, the change

propagates to all clones. As in all prototype-based programming systems, however, the clones are not identical mirror images of their prototypes. It is possible to overwrite any attribute value in the clone, and expect the new value not to be rewritten by the propagation mechanism. The prototype-clone relationship is preserved for the full lifetime of these objects, which distinguishes cloned objects from simple copies. There is no notion of instantiation, as in class-based OOP, because prototypes exist as independent entities [Lieberman, 1986].

Consider the fact that many programming languages support the notion of typing, yet, the modularization of crosscutting concerns cannot be captured using typing alone. The same can be said for the typing facility provided within GME. It does support a useful feature of generalization and reuse of properties, but it fails to provide the kind of quantifying separation found in AOP.

The combination of viewpoints and types within the GME, and the aspect weaver described herein, provides a modeler with the flexibility needed to examine the effect of numerous modeling scenarios. More importantly, these three techniques promote the ability of a modeler to make changes readily within the model – a desirable characteristic of any method that supports concern separation.

Handling Crosscutting Constraints in Domain-Specific Modeling

This section describes the difficulties caused by crosscutting constraints in domain-specific modeling. The modeling techniques enumerated in the previous section were shown to lack the ability to support crosscutting modeling concerns. The remainder of this chapter provides a description of how AO techniques can be used to ameliorate the problem of scattered and tangled modeling concerns. The goal is to encode important

issues about the system being modeled in a clean and localized manner. The purpose of the following section is to serve as a prologue to the driving need for constraints in domain-specific modeling.

Design Space Exploration

A beneficial approach toward domain-specific modeling considers the creation of a base model for representing a family of related systems (e.g., a product-line architecture). In such an approach, a design space corresponds to a set of implementation alternatives that are available within the product family. The selection of a fixed-point, among the set of alternatives possible from the base model, must be explored prior to synthesis [Neema, 2001]. The exploration of a design space requires the existence of constraints that are dispersed throughout a model. Some of these constraints designate a host of alternatives, and other constraints may specifiy a set of requirements for each alternative. Constraints codify properties of the model that must be satisfied during exploration. The next sub-section briefly introduces the language that is used to represent model constraints.

OCL and MCL

The standardization of the Unified Modeling Language (UML) [Booch et al., 1998] has provided software and system designers with a common notation for expressing the intent of an application, or system. The UML defines a graphical language that facilitates the description of a system in an object-oriented style. There are several semantic issues, however, that cannot be captured with the graphical formalisms offered in the UML (see [Nordstrom et al., 1999] and [Gray and Schach, 2000] for several

examples). In such cases, the Object Constraint Language (OCL) has been standardized as a formal language for specifying additional semantics on a UML model [Warmer and Kleppe, 1999]. The OCL is used to denote pre/post-conditions, class invariants, and even guard conditions for state machines. The OCL is a purely declarative modeling language; it is not a programming language. An important feature of the OCL is that it does not introduce side effects into the underlying model.

The Multigraph Constraint Language (MCL) is an extension of the OCL that is supported in the GME. During the creation of a domain's metamodeling paradigm, the MCL is used to stipulate specific semantics within the domain (e.g., a constraint that ensures the uniqueness of a name within a model). The GME has the ability to interpret an MCL constraint in order to verify the consistency of a model with respect to the behavioral intention of the constraint.

Constraint Driven Design Space Exploration

Model constraints are used to specify properties such as bit precision, timing, and power consumption. An example of a latency constraint is illustrated in Figure 16. A base model may have numerous constraints distributed across its various modeling elements. As mentioned earlier, design space exploration is guided by the evaluation of constraints during the exploration process. The constraints provide paths to several different implementation alternatives. Design space exploration is an iterative process that selectively evaluates a set of constraints that are chosen by a modeler using a tool [Neema and Lédeczi, 2001]. Each iteration of the exploration prunes the design space further. Focusing the exploration on different sets of constraints can lead the exploration and pruning algorithms along different elaborations of synthesis.

The utility of specifying constraints within a model, however, is often diminished due to their scattering throughout the model hierarchy. It is often the case that the metamodel forces the emergence of a dominant decomposition that imposes the subjugation of other concerns, such as those captured by constraints. Consequently, constraints represent a type of crosscutting concern. A technique will be presented in the remainder of this chapter for modularizing the concerns represented in these crosscutting constraints.

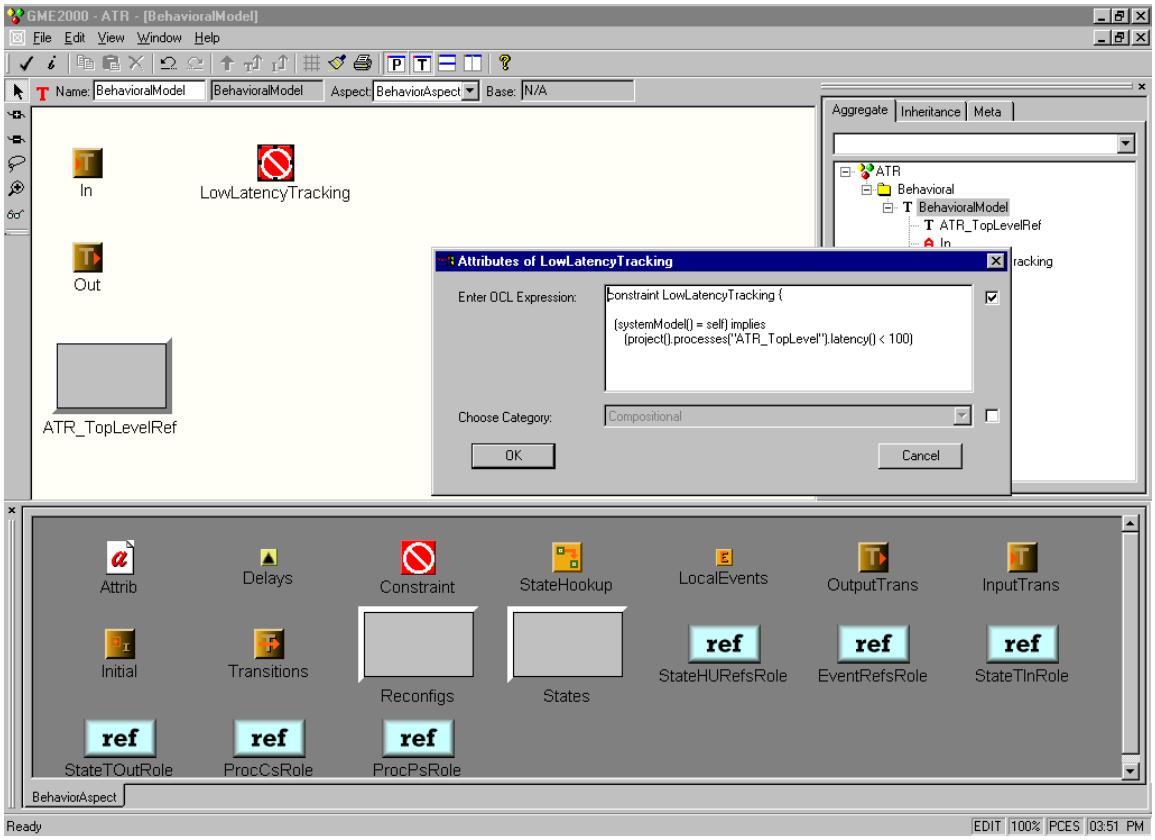


Figure 16: A Latency Modeling Constraint

Constraints as Aspects

> *The crucial choice is, of course, what aspects to study 'in isolation,' how to disentangle the original amorphous knot of obligations,* constraints *and goals into a set of 'concerns' that admit a reasonably effective separation.*
>
> [Dijkstra, 1976]

The same problems that result from crosscutting code in programming languages also occur in the scattered constraints of domain-specific models [Gray et al., 2000]. Often, the same constraint is repeatedly applied in many different places in a model, usually with slight node-specific variations. This can result in redundancy throughout the model. With respect to code, a large amount of redundancy can be removed using AO techniques [Lippert and Lopes, 2000] – the same applies to domain-specific models and constraints. It is also beneficial to be able to describe a common constraint in a modular manner and designate the places and conditions where it is to be applied.

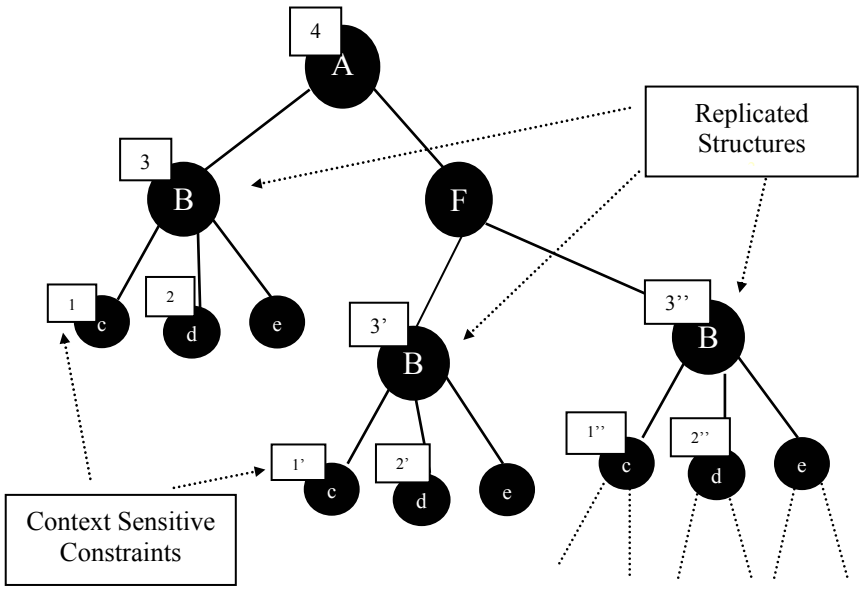


Figure 17: Illustration of the Difficulty in Managing Constraints

As illustrated in Figure 17, three replicated structures (i.e., the structurally similar sub-models with parent node B and children C, D, and E) are acted on by context sensitive constraints. An example of a context-sensitive constraint can be found in the discussion of processor assignment in Appendix B. The dominant form of decomposition shown in the above figure is concentrated on the functional hierarchy of the system being modeled. Notice that each constraint cuts across this hierarchy (for example, constraint "1" is scattered across several nodes in the model). The manner in which a constraint is applied also depends upon the context of the sub-model (for example, constraint "1" may be applied in different ways depending on the context of each node). In conventional system modeling tools, any change to the intention of a system property requires visiting and modifying each constraint, for every context, representing the property. This would require the modeler to "drill-down" (i.e., traverse the hierarchy by recursively opening, with the mouse, each sub-model), by hand, to many locations of the model. It is not uncommon for a model in the GME to contain hundreds of different modeling elements with hierarchies that are ten or more levels deep. The interdependent nature of each constraint makes change maintenance a daunting task for anything but a simple model. The benefits of a single model representation of a product family are nullified. The "Parnasian" objectives of changeability, comprehensibility, and independent development are all sacrificed in the presence of crosscutting constraints [Parnas, 1972].

Another consequence is that constraints become tangled and difficult to understand. A new approach, based on AO, provides a modular construct for separating such design decisions. Often, what is desired is the ability to express a global system-

wide constraint and have it propagated to all relevant nodes in a model. Constraints are the warp and weft of AODSM.

A concrete example of a crosscutting modeling concern will be provided later in this chapter. Likewise, there are several additional examples provided in Appendix B.

Embedded Constraint Language (ECL)

Syntactic sugar causes cancer of the semicolon.

[Perlis, 1982]

This new AO approach requires a different type of weaver from those that others have constructed in the past (e.g., the weaver for AspectJ [Kiczales et al., 2001a], [Kiczales et al., 2001b]) because the type of software artifact that is processed by the weaver differs. Other weavers process source code, but a domain-specific weaver works with the structured textual description of a model. In particular, this new weaver requires the capability of reading a model that has been stored in the Extensible Markup Language (XML). This weaver also requires the features of an enhanced constraint language.

This new approach uses a constraint language in three different ways:

- Model Constraints: This type of constraint appears as attributes of modeling elements. In this case, constraints are used in the same manner as the former approach (the constraint in Figure 16 is an example of a model constraint). It is these constraints that are traditionally scattered across the model. These model constraints assist in the latter stages of design space navigation.
- Specification Aspects: A *specification aspect* is a neologism for the new modular construct for defining modeling concerns across the hierarchy. Each specification

aspect describes the binding and parameterization of strategies to specific nodes in a model. A specification aspect may be described as similar in intent to a pointcut [Kiczales et al., 2001a]. Like a pointcut designator, a specification aspect is responsible for identifying the specific locations of a crosscutting concern.

- Strategies: A *strategy* is used to specify elements of computation, constraint propagation, and the application of specific properties to the model nodes. Strategies are generic in the sense that they are not bound to particular model nodes in their description. Each weaver that supports a specific metalevel GME paradigm will have disparate strategies (this is the topic of the next chapter). The intent of a strategy is to provide a hook that the weaver may call in order to process the node-specific constraint application and propagation. Thus, strategies offer numerous ways for instrumenting nodes in the model with crosscutting concerns.

The three types of entities enumerated above differ in purpose and in application, yet each is based on the same underlying constraint language. This constraint language is called the Embedded Constraint Language (ECL). ECL provides many of the common features of OCL [Warmer and Kleppe, 1999], such as arithmetic operators, logical operators, and numerous operators on collections (e.g., `size`, `forAll`, `exists`, `select`); see Table 1. ECL also provides special operators (to support model aggregates and connections) that provide access to modeling concepts that are within the GME. These aggregation and connection operators originally appeared in the MCL. The ECL is a contribution of this dissertation toward the support of AODSM.

Table 1: Included OCL Operators

Arithmetic Operators	`+, -, *, /, =, <, >, <=, >=, <>`
Logical Operators	`and, or, xor, not, implies, if/then/else`
Collection Operator/ Property Operator	`->` `.`

Standard OCL Collection Operators	`collection->size() : integer`
	`collection ->forAll( x | f(x) ) : Boolean`
	`collection ->select( x | f(x) ) : collection`
	`collection ->exists( x | f(x) ) : Boolean`

There are a few things that distinguish ECL from OCL:

- ECL provides a set of operators for navigating the hierarchical structure of a model (see Table 2). The aggregate and selection operators can be applied to first-class modeling objects (e.g., a container model or primitive model element) in order to obtain reflective information needed in either a strategy or specification aspect (e.g., `findModel`, `getID`, `findAttribute`). These operators can be considered as reflective and likened to introspective operators in Java (e.g., `getName`, `getType`, `getInt`); i.e., they are reflective to the internal representation used in the GME.
- Traditionally, OCL has been used as a declarative language to specify properties of UML diagrams [Warmer and Kleppe, 1999]. The use of ECL requires the capability to introduce side-effects into the underlying XML model. This is needed because the strategies often specify transformations that must be

performed on the model. This requires the ability to make modifications to the model as the strategy is applied. Therefore, ECL supports an imperative procedural style with numerous operations that can alter the state of the model (e.g., `addAtom`, `addAttribute`, `removeChild`). Because the underlying model hierarchy is stored as an XML file, these functions are often implemented as wrappers for the specific calls that are needed to the XML Document Object Model (DOM).

- The procedural nature of ECL permits the dependency between strategies. Strategies can be chained together as procedure calls. Recursion is also supported in ECL. Circular dependencies are possible (of course, the strategy must specify a termination condition in order for the strategy to complete its processing). An example of a set of strategies that have circular dependencies can be found in Figure 51.

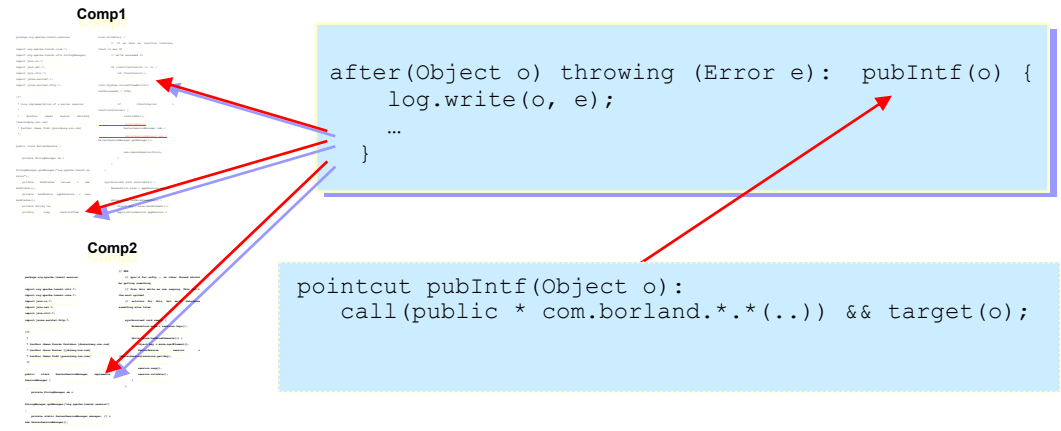
Table 2: ECL Model Operators

Aggregates	`folders, models, atoms, attributes, connections`
Connections	`connpoint, target, refs, resolveRefeeredID, resolveIDReferred`
Transformation	`addAttribute, addAtom, addModel, addConnection, removeNode`
Selection	`findFolder, findModel, findAtom, findAttributeNode`
General	`id, parent, getID, getInt, getStr`

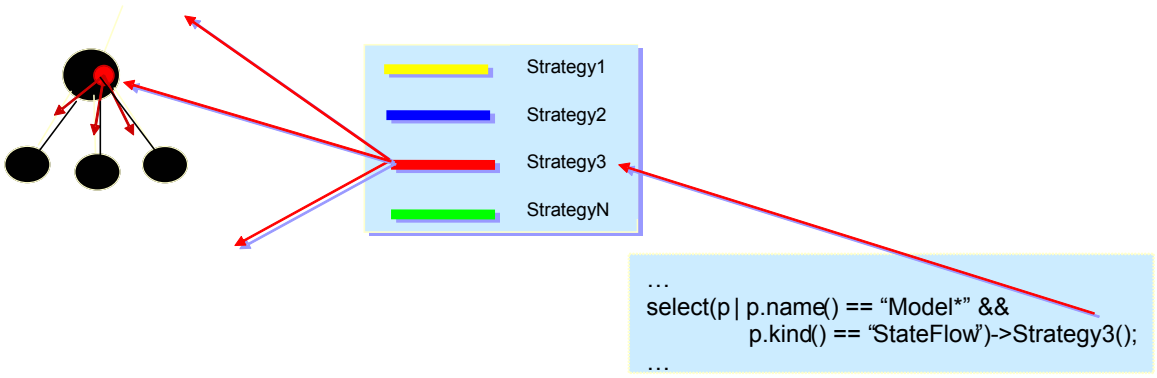
Relationship Between AOP and AODSM

Several comparisons can be made between the approach to AODSM, as described in this chapter, and traditional AOP. Please consider Figure 18. The illustration in Figure 18a depicts a pointcut that is associated with a specific piece of advice. The effect of this association is the quantification of a concern over multiple join points. The pointcut construct in AspectJ identifies several join points, and the advice construct describes the additional code to run at those join points. Comparatively, the box in the bottom-right of Figure 18b represents a subset of a specification aspect. In this specification aspect, a predicate within the `select` statement instructs the weaver to collect all nodes in the model that are of kind "StateFlow" and have a name that matches "Model*." Such a statement has a direct correspondence to a pointcut (as in AspectJ) that picks out specific points in the execution of a program satisfying some condition. The specification aspect also describes the strategy that is to be invoked on each node selected from the predicate. As the strategy is applied at each node, the graph is transformed according to the intent of the strategy. This has a direct correspondence to the association of pointcuts with advice in AspectJ, and how advice affects the execution of the program.

Table 3 provides a comparison of the critical elements that make a system aspect-oriented and enable quantification; i.e., the join point model, the pointcut designator construct, and the concept of advice. Note that the static nature of the AODSM join point model may be improved with some of the extensions suggested in the chapter on Future Work.



a) Aspect-Oriented Programming (AspectJ)



b) Aspect-Oriented Domain-Specific Modeling
Figure 18: Effects of AOP and AODSM

Table 3: Comparison of AspectJ and AODSM

	AspectJ	AODSM
Join Point Model	Well-defined points in the execution of a program	Currently, static points (nodes) in an XML document
Pointcut Designator	A declarative statement (formed from a set of primitives like `call`, `this`, and `target`) that describes a set of join points in a program	A declarative statement (formed from ECL collection operators) that identifies a set of locations within a model
Advice	A block of code that is executed at a join point	A strategy, or heuristic, for instrumenting a model node with information related to a concern

Sample Strategies and Specification Aspects

An introduction to applying the ECL can be found in Figure 19 and Figure 20 (Appendix B contains several additional sample strategies, along with a description of their intent). These two figures contain several sample strategies and a specification aspect. The first three strategies at the top of Figure 19 are generic strategies that can be used for constraint application, removal, and replacement. These simple strategies make use of standard functions that are provided within ECL. The strategy named `ReplaceConstraint` demonstrates that strategies may depend on the capability of other strategies.

```
defines ApplyConstraint, RemoveConstraint, ReplaceConstraint,
        PowerStrategy;

strategy ApplyConstraint(constraintName, expression : string)
{

  addAtom("OCLConstraint", "Constraint",
          constraintName).addAttribute("Expression", expression);

}


strategy RemoveConstraint(constraintName : string)
{

  findAtom(constraintName).removeNode();

}


strategy ReplaceConstraint(constraintName, expression : string)
{

  RemoveConstraint(constraintName);
  ApplyConstraint(constraintName, expression);

}


strategy PowerStrategy(level, power : integer)
{

  if (level < 3) then

    <<CComBSTR aConstraint = "power < " + XMLParser::itos(power); >>
    ApplyConstraint("PowerConstraint", aConstraint);

    power := power / 10;
    level := level + 1;

    models("")->forAll(PowerStrategy(level, power));

  endif;

}
```

Figure 19: Sample Strategies

```
aspect ATR_Power
{
  in Structural models("ProcessingCompound")->
           select(p | p.name() == "*_Top?")->PowerStrategy(1, 100);
}
```

Figure 20: ATR_Power Specification Aspect

The `PowerStrategy` inserts a new ECL model constraint that specifies power properties in an embedded system. There are a few features worth noting about this strategy.

- The header of the containing file must define all of the enclosed strategies.
- The strategy language uses ECL in such a way that conditional statements, assignment operations, and even recursion are available.
- It is possible to provide inlined C++ code inside of a strategy (this is indicated by the << .. >> syntax). As the language evolves, there will be less of a dependence on underlying C++ constructs. The implementation of ECL, at the time of the writing of this chapter, requires string manipulation to be performed inline.
- Constraint propagation can be passed along to sub-models by using the ECL functions. In this case, the `models` reflective function returns a collection of all immediate children that are sub-models. The `forAll` standard function then iterates over this collection and invokes `PowerStrategy` on each sub-model (with new values for `power` and `level`).
- Although not explicitly shown here, it is possible to create several different types of `PowerStrategy` by varying the strategy signature. Overloaded strategies

can offer various ways of applying the power constraint and propagating it to sub-models.

Notice that strategies are not bound to any particular node in the model. The binding and parameterization of strategies occurs within a specification aspect. An example specification aspect is shown in Figure 20. This simple specification aspect will find the nodes in the model that are of type `ProcessingCompound` and have the wildcard designator of "`*_Top?`." As may be surmised, this means that a search will be made for any string that matches zero or more occurrences of any letter, followed by `_Top`, and then a single character. The `PowerStrategy` will then be applied to the matching nodes using the parameters provided to the strategy.

The implementation of wildcard matching within ECL is worthy of a special note. Whenever a wild card appears within a string comparison, the wildcard pattern matcher is triggered within the equality operator (i.e., the detection of a "*" or "?" within a string invokes a special comparison function). In such cases, any number of "*" or "?" may appear within the match string. An alternative implementation to placing the pattern matcher in the equality operator would be to use special XPath functions. XPath is a standardized technology that offers a simple query language for searching XML documents. An XPath solution, for example, would translate the wildcard string represented by "Foo*" into the following sub-expression:

```
[starts-with(@kind, "Foo")]
```

For situations where the wildcard character appears as a prefix, as in "*Foo," the translation would be much more complicated:

```
[substring(@kind, string-length(@kind)-
string-length("Foo") + 1, string-length("Foo")) = "Foo"]
```

The equality operator implementation was chosen over the XPath solution due to the flexibility available in matching. It is very difficult to use XPath to detect anything other than a prefix or suffix. The comparison function associated with the equality operator permits any combination of wildcard characters to be detected within a given string.

Different sets of specification aspects can be weaved into a model. That is, a specification aspect with very different behavior could be applied as an alternative to `ATR_Power`. This gives the modeler the capability of constructing "what if" scenarios. This capability was impossible in the former approach because there was no modular construct for collecting the constraints in a single location. Specification aspects can be much more complicated than shown here. A single specification aspect can cause the weaver to visit many different nodes in the model hierarchy. It is even possible for one global aspect to be diffused across the entire model hierarchy. This is a testimony to the power of quantification.

A key feature of this approach is that it provides a framework that uses software code generators to create new domain-specific weavers. The process for creating new weavers is the topic of the following chapter. The manner in which a domain-specific weaver is used, however, can be understood by viewing Figure 21. The GME can export the contents of a model in the form of an XML document (in this case, the exported XML is related to the metalevel paradigm from which the model was constructed). In the former approach, the generated XML would be tangled with constraints throughout the

document. Under this new AO-based approach, however, it may be quite possible that the exported XML model is void of any constraints. In Figure 21, the solid arrows represent the output from tools that generate, or transform, a model. The white arrows indicate that the combination of a model and a specification aspect are sent to the weaver.

The input to the domain-specific weaver consists of the XML representation of the model, as well as a set of specification aspects provided by the modeler. These are positioned to the left of the weaver. The output of the weaving process is a new description of the model in XML. This enhanced model, though, contains new concerns that have been integrated throughout the model by the weaver.

One way to understand this process is to reconsider the diagram in Figure 17. The XML model that is fed into the weaver will often resemble the hierarchy depicted in this diagram but *without* the constraints (here, provided as the numbered rectangular blocks). The purpose of the specification aspects is to specify the manner in which the constraints are replicated and applied to the context-sensitive model elements. The resultant enhanced model, then, would resemble the diagram in Figure 17 *with* the added model constraints.

The benefits of this approach are numerous. Consider the case of embedded systems where constraints often have conflicting goals (e.g., latency and resource usage). In the former approach that did not use AO, latency and resource requirements would be scattered and tangled throughout the model. As a result, it was quite difficult to isolate the effects of latency or resource constraints on the design. By using aspects to represent these concerns, the designer may apply specification aspects separately to see how the system is affected in each case. In this way, areas of the system that will have more

difficulty meeting a requirement may be given more relaxed constraints, and other parts of the system may be given tighter constraints (e.g., using the example of Figure 60 in Appendix B, it could be the case that the size of the video frame is reduced in order to increase the video frame rate). In short, it enables the designer to quickly isolate and study the effects of concerns (here, as constraints) across the entire model. This is a very desirable property with respect to application-constraint tuning [Vahid and Givargis, 2001]. Therefore, the separation of concerns provided by the specification aspects improves the modular understanding of the effect of each constraint. The plugging/unplugging of various sets of specification aspects into the model can be described as creating "what if" scenarios. These scenarios help in the exploration of constraints that may have conflicting goals. The insertion and removal of scenarios is somewhat analogous to the ability that AspectJ offers in terms of being able to plug/unplug certain aspects (e.g., logging) into a core piece of Java code [Kiczales et al., 2001b].

In AspectJ, it is possible to be too cavalier in creating a pointcut designator. Without care, pointcut designators can be created that can cause advice to be applied in a contradictory manner, or even in a way that causes the weaver never to terminate. A partial solution to resolving contradictory advice in AspectJ is tool support (e.g., the plug-ins for JBuilder and other IDEs that are available from the AspectJ Team). In the approach described in this chapter, conflicting constraints may be resolved with the design space navigation. During navigation, the modeler can chose to apply from among a set of valid constraints.

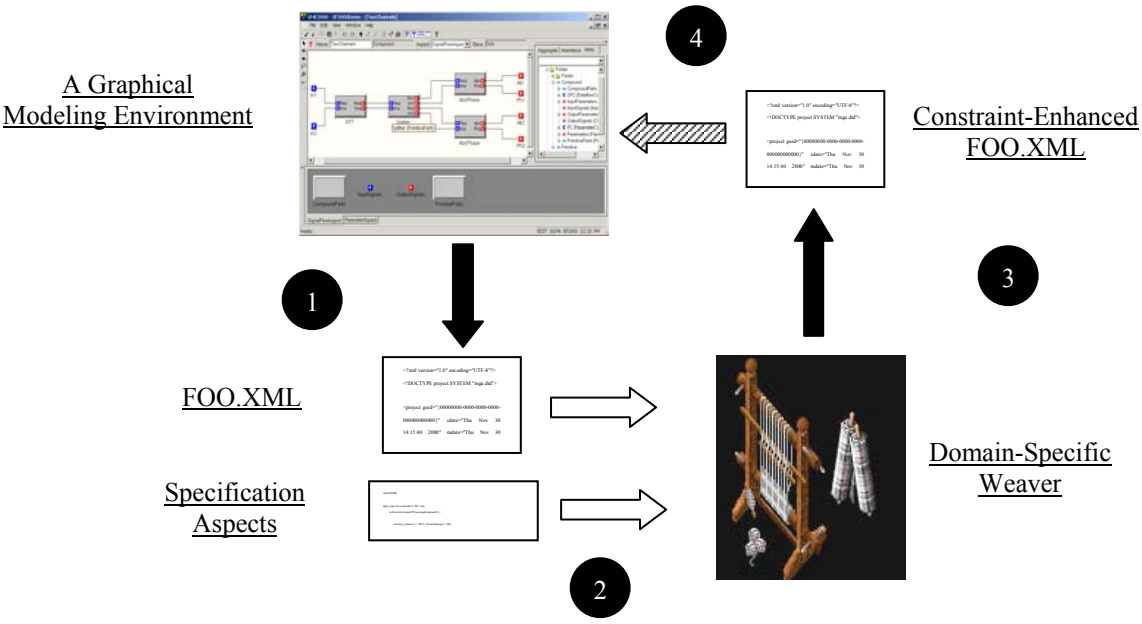


Figure 21: Process of Using the Constraint Weaver

Summary

This chapter introduced a contribution of this dissertation that is concentrated on Aspect-Oriented Domain-Specific Modeling. The approach expressed here serves as a complement to the previously developed modeling techniques supported by the GME. Incorporating AODSM into a modeling project can assist in the separation of concerns that were extremely difficult to isolate using only viewpoints and type hierarchies. The new concepts of specification aspects, and strategies, provide a unique contribution to the modeling science literature.

The following chapter describes the generative programming techniques that have been used to contribute toward the construction of a metaweaver framework to support AODSM in multiple domains.

CHAPTER IV

A METAWEAVER FRAMEWORK

Each specific GME modeling paradigm introduces different types of modeling elements, syntax, and semantics that are unique to a domain. For example, the modeling paradigm that is used to create models of the Saturn automobile factory is very different from the paradigm used to create avionics models for Boeing. Because of the existence of unique syntax and semantics, different weavers are needed for each new paradigm. This chapter describes the process in which new instances of domain-specific weavers are constructed using a metaweaver framework. Issues related to code generation are also described in this chapter.

The Motivating Need for Different Weavers

In the definition of a modeling paradigm, concepts from the domain are specified using a graphical modeling language. Different domains will have different dominant decompositions and different crosscutting concerns. Consequently, different weavers are required. As Figure 22 illustrates, the domain for Automatic Target Recognition (i.e, "ACS ATR") needs its own specialized weaver, as does the BBN Unmanned Aerial Vehicle (UAV) domain, and the Boeing BoldStroke domain (see the examples in Appendix B).

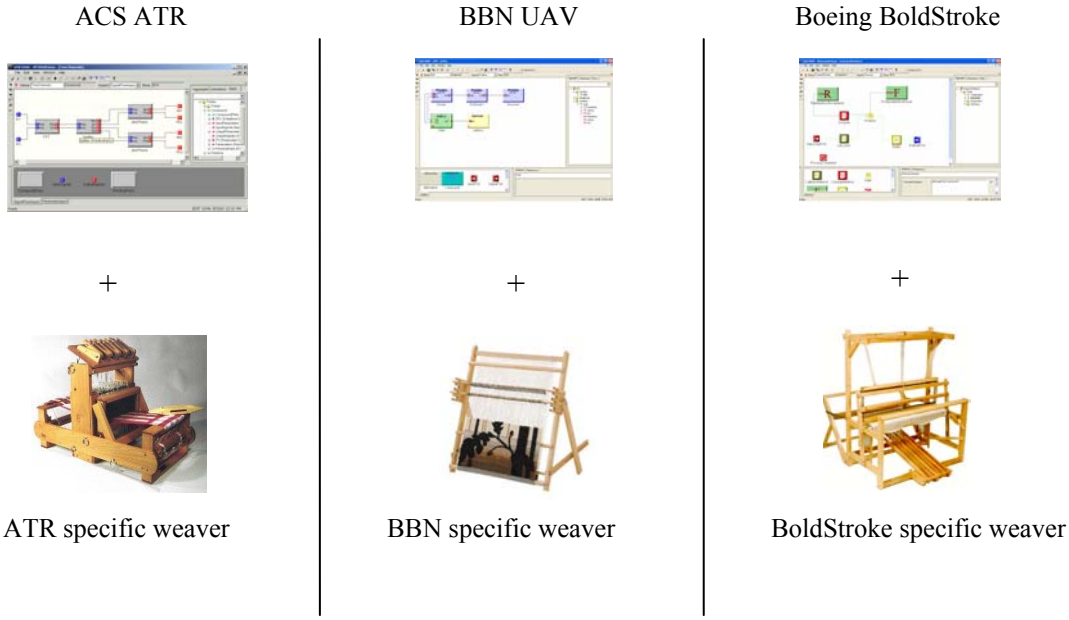


Figure 22: Separate Weavers for Different Paradigms

To understand the need for multiple weavers, consider the XML document in Figure 23. This XML file is a representation of a subset of the model illustrated in Figure 47. The document has distinctly named regions with respect to the kind of elements being presented (e.g., "`Component`"), as well as roles (e.g., "`ComputeMethod`"), name, and even attributes (e.g., "`WCET`").

```
<model id="id-05" kind="Component">
      <name>InertialSensor</name>
      <atom id="id-17" kind="ComputeMethod" role="ComputeMethod">
            <name>compute</name>
            <attribute kind="WCET">
                  <value>2</value>
            </attribute>
```

Figure 23: BoldStroke/CCM XML Model

Further consider the XML fragment in Figure 24. It also has its own unique modeling entities (e.g., "`State`," "`Transition`," "`Guard`"). Because each new GME metamodeling paradigm introduces different types of modeling elements, syntax, and semantics, different weavers are needed for each new paradigm. The situation is similar to the reason a different compiler is needed for a new programming language – the syntax and semantics varies too much between each language to permit a single instance of a generalized translator that compiles multiple languages.

```
<model id="id–544975-39" kind="State">
<name>frameRate</name>

<model id="id–544975-42" kind="State">
<name>Range1-7</name>

<connection id="id–544975-63" kind="Transition">
<name>Transition</name>
<connpoint role="dst" target="id–544975-42" />
<connpoint role="src" target="id–544975-46" />
      <attribute kind="Guard">
              <value>latency > 25</value>
      </attribute>
     <attribute kind="Action">
          <value>frameRate=4</value>
     </attribute>
```

Figure 24: BBN/UAV XML Model

Strategy Code Generator (StratGen)

Strategies are used to aid in the rapid construction of new domain-specific weavers. ECL constraints can succinctly capture portions of a strategy specification. A generative programming approach has been adopted with respect to constructing a weaver. A code generator has been created that is capable of translating the strategies into

C++ code that is then compiled within the metaweaver framework. Each domain-specific paradigm can then be considered as being a component within the weaver.

The C++ code that is generated by StratGen is much more complex than the strategy specification. All of the details of making the appropriate XML Document Object Model (DOM) calls and the iterations over collections are hidden from the strategy specifier. This allows the construction of a weaver at a higher-level of abstraction – a commonly recognized benefit of using domain-specific languages and code generators [van Deursen et al., 2000]. An example of generated code will be provided in a subsequent section.

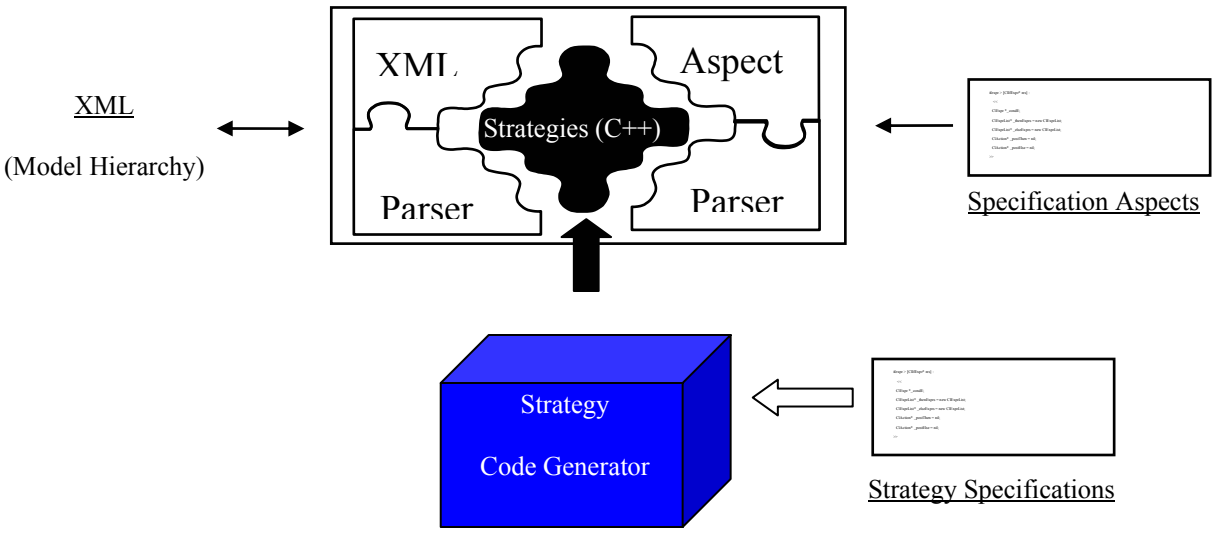


Figure 25: Metaweaver Framework

XML Parser

The C++ code that is generated by StratGen is dependent upon several key components. Strategies iterate and manipulate the model, as stored in the DOM. The XML Parser component is responsible for providing wrappers for the methods used to

interact with the DOM. The XML Parser is also given the task of encapsulating all of the functionality needed to load/save a model using XML. The generated code contains specific details for iterating and modifying the underlying XML representation of the model, according to the intent of the strategy (see Figure 27 for an example). Therefore, the generated C++ strategies are heavily dependent upon the XML Parser functionality.

Aspect Parser

The Aspect Parser is another piece of the metaweaver framework. Its purpose is to parse and apply the specification aspects. The application of a specification aspect will result in the invocation of some strategy. It is the task of the Aspect Parser to locate specific nodes in the model hierarchy and invoke specific strategies on those nodes.

An ECL grammar has been created that is used with the PCCTS parser generator [Parr, 1993]. The Aspect Parser uses this grammar, and the associated data structures that represent the parse tree, extensively. In fact, StratGen uses the same grammar during the translation of strategies into C++ code.

Metaweaver Instantiation vs. Weaver Invocation

A distinction should be made concerning the way these various components are used in the stages of metaweaver instantiation (i.e., the creation of a new domain-specific weaver) versus weaver invocation (i.e., executing a weaver on a specific model with a specific set of specification aspects).

Although strategies are particular to each instance of a domain-specific weaver, the aspect parser that processes specification aspects is the same for every weaver instance. Another difference between specification aspects and strategies is in the way

that they are realized. Specifically, ECL constraints that are applied within strategies are actually used to generate C++ code that is then compiled within the framework to create a new weaver. On the other hand, the ECL constraints used in specification aspects are interpreted, in memory, during weaver invocation.

Constraints used in strategies are synthesized during instantiation of the metaweaver. Constraints used in specification aspects are interpreted during the invocation of a specific weaver.

Sample Code Generation

Figure 26 contains a single statement from the strategy in Figure 51. This statement finds all of the models that match a specific id and then calls the `DetermineLaziness` strategy on those selected models. The amount of C++ code that is generated by StratGen, however, is far from being concise or simple (see Figure 27). Much of the code for implementing this strategy statement is focused on iterating over a collection and selecting elements of the collection that satisfy the predicate. In a different research effort, it was discovered that the details involved in generating selections from a DSL were also found to be much larger than expected [Karsai and Gray, 2000]. In that study, the code that was generated to represent traversal/visitor (like that in Figure 13) was compared to the corresponding generated C++.

```
components.models("")->select(c | c.id()==refID)->DetermineLaziness();
```

Figure 26: Fragment of the EagerLazy Strategy

The code in Figure 27 contains a generic value class named `ClData`. It is in this class where the equality operator performs a special match for string wildcards. The C++ code calls an XML Parser wrapper class that retrieves a set of all models. An iteration over the list of models checks to see if the name of the node referenced by the current iterator matches the wildcard.

```
CComPtr<IXMLDOMNodeList> models0 = XMLParser::models(components, "");
nodeTypeVector selectVec1 = XMLParser::ConvertDomList(models0);
nodeTypeVector selectVecTrue1 = new std::vector<nodeType>;
vector<nodeType>::iterator itrSelect1;
for(itrSelect1 = selectVec1->begin(); itrSelect1 != selectVec1->end();
    itrSelect1++) {
  nodeType selectNode1 = (*itrSelect1);
  nodeType c;
  c = selectNode1;
  CComBSTR id0 = XMLParser::id(c);

  ClData varforward1(id0);
  ClData varforward2(referredID);
  bool varforward3 = varforward1 == varforward2;
  if(varforward3)
     selectVecTrue1->push_back(*itrSelect1);
}

vector<nodeType>::iterator itrCollCall1;
for(itrCollCall1 = selectVecTrue1->begin();
    itrCollCall1 != selectVecTrue1->end(); itrCollCall1++)
  DetermineLaziness::apply(…);
```

Figure 27: Sample of Generated C++ Code

Comparing ECL to the Generated C++

Domain-Specific Languages gain their power by raising the intentionality of programmer expression. With a DSL, it is argued, a programmer can express their objective in a concise manner using a language that is much higher in expressiveness than

that typically offered in a traditional programming language. Because of this, it is often asserted that programs written in DSLs are much easier to maintain and modify.

It is reasonable to assume that any language that raises the level of expressiveness will be more concise than the underlying representation to which it is generated. A simple analogy of this would be a comparison of any high-level programming language to the equivalent assembly or object code that resides closer to the execution space. Typically, the representation of a single executable statement in a programming language translates to several assembly instructions, or more than a few bytes of object code.

There have been very few studies that have quantified the actual productivity improvements offered by DSLs. One of the earliest studies demonstrated an order of magnitude difference [Herndon and Berzins, 1988]. The most detailed study of this topic can be found in [Batory et al., 1994], where it was discovered that a DSL for specifying data structures led to a reduction of programming time by a factor of 3. It was also determined in that study that the number of lines of code needed to represent a specific intention was reduced by a factor of 4. These results are similar to observations that have been made in comparing the ECL to its underlying C++ translation.

The data presented in Table 4 is a comparison, along several different measures, of the conciseness offered by DSLs like ECL. The table lists several measurements taken between the ECL and the generated C++ along the criteria of lines of code, size of code (number of bytes), and word count (using the Unix `wc` utility). The subjects of this study were a subset of several of the strategies that were created to support this research. Most of these strategies are described elsewhere in this dissertation (in Appendix B). A ratio of

differences between the sizes of these two representations is also provided within each cell of the table.

Table 4: Size Comparison of DSL to Generated Code

	Lines of Code	Bytes of Code	Word Count
Power Distribution	ECL: 43 C++: 140 Ratio: 1::3.25	ECL: 859b C++: 3.08k Ratio: 1::3.50	ECL: 69 C++: 232 Ratio: 1::3.36
Processor Assignment	ECL: 39 C++: 137 Ratio: 1::3.50	ECL: 954b C++: 3.28k Ratio: 1::3.44	ECL: 76 C++: 251 Ratio: 1::3.30
Eager/Lazy	ECL: 85 C++: 230 Ratio: 1::2.71	ECL: 2.03k C++: 6.24k Ratio: 1::3.07	ECL: 169 C++: 499 Ratio: 1::2.95
Exhaustive State Transition	ECL: 70 C++: 184 Ratio: 1::2.62	ECL: 1.92k C++: 5.14k Ratio: 1::2.68	ECL: 160 C++: 399 Ratio: 1::2.49
State Generation	ECL: 128 C++: 242 Ratio: 1::1.89	ECL: 3.42k C++: 6.76k Ratio: 1::1.98	ECL: 312 C++: 570 Ratio: 1::1.82

With reference to bytes of code, Figure 28 visually represents the differences between ECL and the generated C++ code. An observation can be made regarding the State Generation strategy. Its translation yielded the lowest ratio of comparison. This strategy also contains the least amount of ECL collection statements, suggesting the somewhat obvious observation that all of the code needed to iterate over a collection increases the amount of generated C++ code.

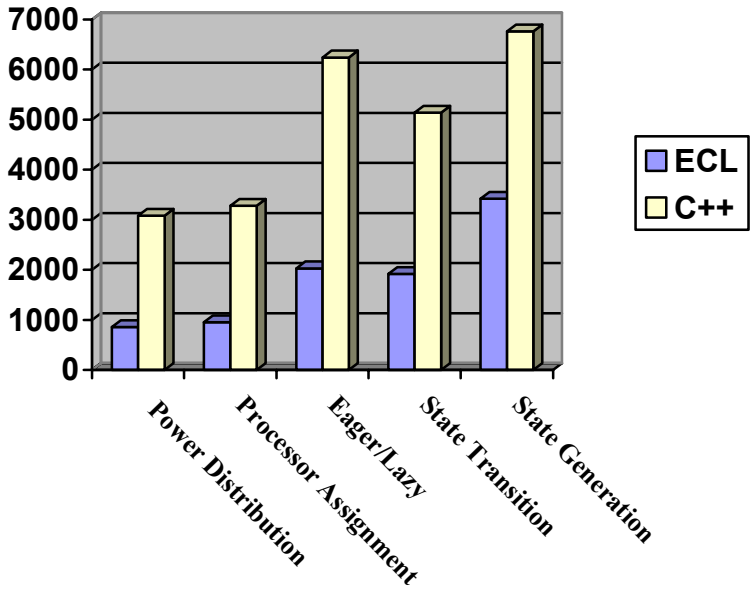


Figure 28: Bytes of Code Comparison of ECL and C++

XSLT as an Alternative to ECL

The Extensible Stylesheet Transformations (XSLT) is a language that can transform XML documents to other XML, HTML, or even plain-text documents [Tidwell, 2001]. XSLT is, itself, written in XML. The text of the XSLT template contains a specification of the desired transformation. The XSLT file that contains the transformation description is sent as input to an XSLT processor, along with the XML input file. The output from the XSLT processor will be some transformation, in some format, as specified by the input file.

It is possible that the strategies and specification aspects could be written as XSLT transformations. In fact, the generated C++ strategy code makes frequent use of calls to XPath in order to retrieve information from the XML model. XPath is a query language that is also used in the specification of XSLT transformations.

There are several problems with XSLT, though. Maintaining state information during complex computations is not an easy thing to specify in XSLT. Several authors have commented on the difficulties in using XSLT to process increasingly complex transformations:

- "XSL also has some procedural control structures, but these features are somehow limited when compared to the power offered by a real procedural language…This programmatic solution might be more scalable than a simple XSL stylesheet when the XSL code generator logic becomes too complicated." [Georgescu, 2002]
- "However, some transformations, particularly those that require some analysis of the information are difficult to express in XSLT." [Cleaveland, 2001]
- "The XSLT rewriting approach makes it very hard to attach arbitrary computations to the translation process." [Karsai, 2000]

An area for future study could compare equivalent transformations written in both the ECL and XSLT. It is my belief that the ECL will be shown to be more succinct than an equivalent XSLT solution.

Other OCL Generators

There have been a few contributing research efforts in the literature on OCL-based generators. The majority of these investigations are within the context of applications that use OCL to perform some type of analysis of UML class diagrams.

Within the area of query-based debugging, [Hobatr and Malloy, 2001] document a technique for translating OCL constraints (that appear in UML class diagrams) into code that is inserted into existing C++ classes. The underlying technique is based on a

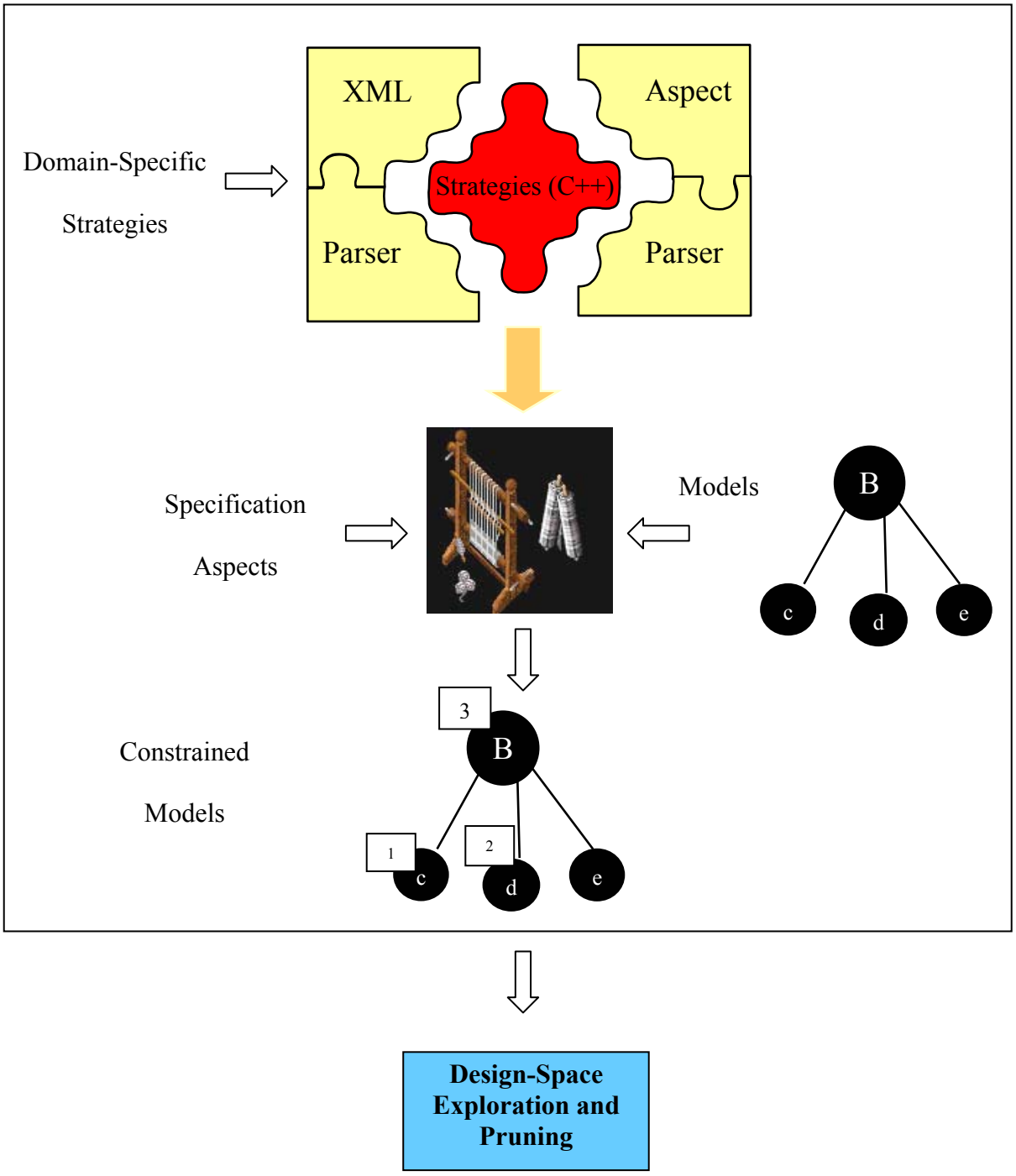
MOP; in this case, OpenC++ [Chiba and Masuda, 1993]. A key advantage of this approach is that the constraints formulated in the design phase can be used as a source for automating the application of Design by Contract [Meyer, 1997].

The notion of specification animation has been a topic in the literature on formal specification languages. The animation of a formal specification is a type of executable specification that is explored from within a support environment. The development of a specification animation environment for UML/OCL is documented in [Gray and Schach, 2000]. In this work, OCL constraints were converted to an executable form using an object-oriented version of Prolog. An OCL toolkit is described in [Hußman et al., 2000]. Much of their description is relative to an outline of different tools that need to be developed in order to advance the status and popularity of OCL. They also provide an example of generated code from OCL to Java.

Summary

An illustrative summary of the concepts presented in the past two chapters can be found in Figure 29. The first task, when introducing AODSM into a new domain, is to instantiate the metaweaver framework in order to generate a new domain-specific weaver. This instantiation is accomplished by creating strategies for the domain (using the ECL) and translating them with StratGen (not specifically shown on the following figure). The generated code can be compiled within the framework and a new weaver instantiation will be generated. Modelers can use the domain-specific weaver to separate the crosscutting modeling concerns from the structure of each model. The result is a new model that contains the scattered concerns. It is this constrained model that is explored

using design space navigation tools. The concept of Aspect-Oriented Domain-Specific Modeling is circumscribed by the box in Figure 29.



CHAPTER V

FUTURE WORK

> *My favorite story about the gold rush of 1849 is that the lasting fortune is Levi Strauss. Turns out that digging gold wasn't really where it was at; it was selling pants.*
>
> Nathan Myhrvold, Former Microsoft CTO
> *US News & World Report* (1/21/2002)

This chapter puts forth a few ideas for extending the work that was presented in the previous two chapters. The key ideas for future extension are: investigation into the modeling techniques needed to represent the textual format of strategies and specification aspects in the style of a visual programming language, and exploiting further areas of adaptability within the metaweaver framework.

Aspect Modeling in the Style of Visual Programming

A potentially rewarding subject for future investigation will be the subsumption of the textual descriptions formulated within the ECL into a graphical modeling language; that is, an investigation into the expression of specification aspects, and even strategies, using a graphical formalism similar to that of visual programming languages. This kind of visual aspect modeling would, of course, be perfectly suited for exploration from within the GME.

A new technique that will assist in the implementation of this idea has recently been added to the GME. The concept of composable metamodeling is available as a GME modeling construct. Multiple paradigm sheets are supported by the application of

this construct. This would assist in the construction of a general aspect modeling paradigm that could then be composed with other paradigms that were not designed with the concept of aspect modeling in mind.

A First Step: Moving the Weaver into the GME

In the current aspect weaving process, as typified by Figure 21, the weaver exists outside of the GME. To use the weaver, a modeler must save the model, export it to XML, invoke the weaver, and then import the new model back into the GME. This can be a disruptive progression. It would be beneficial to make the weaver invocation more transparent. The metaweaver framework will be altered so that it is capable of generating an interpreter that can be registered from within the GME. Thus, weaving would be no different than any other GME interpreter. The weaver, in this case, will still be domain-dependent, but it will be more integrated within the GME. In this scenario, it would be advantageous to explore the possibility of removing the dependency on XML. In such a situation, where the weaver is registered as an interpreter, it would make sense to perform the weaving on the internal representation of the model (instead of the XML representation). An opportunity also exists, in this new integration of the weaver within the GME, to explore the feasibility of supporting an undo/redo capability (i.e., the weaving process can be undone and a model returned to its state prior to weaving).

Generating Weavers from Visual Descriptions

The concept of generating weavers from visual formalisms (i.e., interpreting strategy specifications that are described visually) is appealing. It is unclear at this point, though, how some of the constitutive properties of aspect-orientation will be

implemented using a graphical formalism. The graphical specification of join points may benefit from collaboration with other efforts at ISIS that are focused on pattern matching within a graph. The inherent quantification that is present within aspect languages will need to be specified visually in a manner that allows a modeler to describe the essential characteristics of a pointcut. The task of composing various graphically specified pointcuts is another interesting question for future research. A graphical notation for representing the effect of advice will also need to be explored.

Extending the Metaweaver Framework

The previous chapter described the manner in which strategies are used to aid in the construction of domain-specific weavers. Although strategies allow for variability among different GME paradigms, there are other improvements that can be made to the framework in order to extend its variability. In this section, two other degrees of variability within this framework are proposed. The section also contains a generic description of a metaweaver framework that can support variability for weaving different programming languages and aspect languages.

Variability with Respect to Modeling Tools

The context of the previous two chapters assumed that the separation of modeling concerns was being performed on models created with the GME. In fact, this assumption is built into the XML Parser that was described in the last chapter. The limitation imposed by this assumption precludes other modeling tools (that also can export models using XML) from being able to employ the benefits of an aspect weaver. In addition to the GME, other examples of domain-specific visual modeling tools are Honeywell's

Domain Modeling Environment [DOME], and metaEdit+ (from metaCASE) [Tolvanen and Kelly, 2000]. It is possible that these, and other modeling tools (such as Ptolemy, from UC Berkeley [Lee, 2001]) and notations (such as an XML representation for EXPRESS [Barkmeyer and Lubell, 2001]), could benefit from an aspect-oriented modeling approach. Figure 30 illustrates the manner in which a new code generator could be inserted into the metaweaver framework in order to provide an added measure of variability. From the modeling tool's Document Type Definition (DTD), the functionality of the wrappers provided within the XML Parser can be generated. For example, in Chapter 3, within the context of the description of ECL, a parenthetical example of ECL reflective operators were given; namely, `findModel`, `findAtom`, and `findAttribute`. These reflective operators are actually implemented as wrappers within the XML Parser. The particular operators that were described in that previous section are tied to the DTD that GME uses during the import and export of models. A subset of the GME DTD is shown in Figure 31. That figure specifies the definition of GME models, atoms, and attributes. The definition of other modeling entities (e.g., connections and references, among others) would be specified similarly. Other tools, where the DTD may not contain modeling elements called "model," "atom," or "attribute" would require different adapters for accessing the XML DOM.

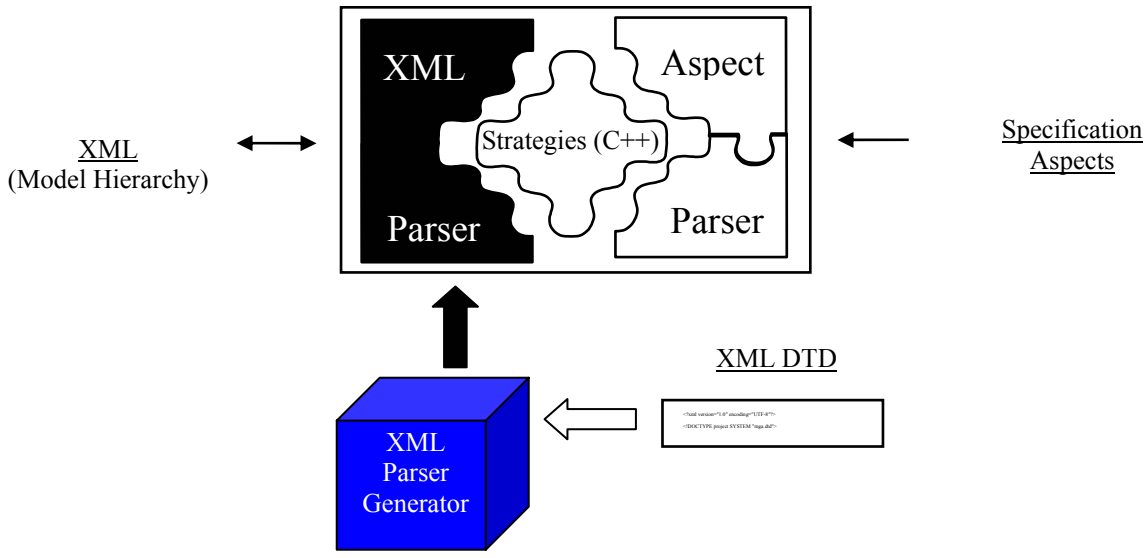


Figure 30: Variability with Respect to Modeling Tool

Given the XML element definitions from the figure below, there is a straightforward mapping to many of the XML Parser methods. A few such methods are listed in Figure 32. Throughout the code listing in that figure, it can be observed quite easily that the element definitions from the DTD have greatly influenced the methods defined in the XML Parser (to see this, just look through the source in Figure 32 for strings like “model,” “atom,” and “id”). Similar routines could be generated from the metalevel definition (found in the DTD) of other modeling tools that use XML for model persistence.

```
<!ELEMENT model (name, (constraint|attribute|model|atom|reference|set
                        connection)*)>
<!ATTLIST model
      id                ID                #IMPLIED
      kind              NMTOKEN           #REQUIRED
      role              NMTOKEN           #IMPLIED
>

<!ELEMENT atom (name, (regnode|constraint|attribute)*)>
<!ATTLIST atom
      id                ID                #IMPLIED
      kind              NMTOKEN           #REQUIRED
      role              NMTOKEN           #IMPLIED
>

<!ELEMENT attribute (value, regnode*)>
<!ATTLIST attribute
      kind              NMTOKEN           #REQUIRED
>
```

Figure 31: Subset of GME DTD

In Figure 32, the `addAtom` method simply calls another XML Parser support method named `addNode`. This method makes the necessary call to the DOM in order to attach a new node to the XML model (a structurally equivalent `addModel` method is coded in the same manner – atoms and models, as shown in the DTD, have the same attribute list). The `findModel` method basically executes an XPath query to the DOM in order to search for a model with a specific name (`submitXPath` is itself an adapter method whose details are not shown here). The `findFolder` and `findAtom` methods are written in the same style. The "id" attribute of any modeling element can be obtained by calling the XML Parser `id` method. Other attribute accessor methods are written in the same style.

```
nodeType XMLParser::addAtom(nodeType self, CComBSTR kind,
                            CComBSTR role, CComBSTR name)
{
      return addNode(self, "atom", kind, role, name);
}

nodeType XMLParser::findModel(nodeType aNode, CComBSTR name)
{

  CComBSTR bstrFind(L"./model[name=\"");
  nodeType res;

  bstrFind.Append(name);
  bstrFind.Append("\"]");

  res = submitXPath(aNode, bstrFind);

  return res;

}

CComBSTR XMLParser::id(nodeType aNode)
{
  CComBSTR res;
  CComPtr<IXMLDOMNode> attr = XMLParser::findAttribute(aNode, "id");
  XMLParser::getStr(attr, res);
  return res;
}
```

Figure 32: Sample Subset of XML Parser Methods

Generating a Code Generator

It may be interesting to observe the strategies specified in Appendix B in Figure 51. A perusal of the strategies in that figure should reveal that the following operators are referenced in the strategy definition: `connections`, `models`, `refs`, `connpoint`, `findFolder`, `findModel`, and `findAtom`. This suggests that tool-specific knowledge has crept into the intentions that can be expressed from within the ECL.

A survey of the methods within the StratGen code generator will reveal the presence of GME-specific concepts (it is recognized that many tools would use terms such as "atom" and "model" to denote specific modeling concepts, but the presence of methods like `findModel` is the result of a dependence on the GME, not a generalization of all modeling tools). This can be viewed in Figure 33, which contains the code to generate the C++ strategy for calling the `findModel` method that is in the XML Parser (see the second method in Figure 32). The generation methods for `findAtom`, `findConnection`, and a host of other tool-specific methods are constructed in an analogous manner by making reference to the methods provided in XML Parser.

```
void Generator::GenerateFindModel()
{

  static findModelCounter = 0;

  genOut << indentStr << "nodeType aModelFind" << findModelCounter <<
                    " = XMLParser::findModel(" << lastVariable << ", ";

  lastVariable.Format("%s%d", "aModelFind", findModelCounter++);

}
```

Figure 33: Code Generation for `findModel`

To reduce the tool dependency bias within the StratGen code generator, portions of StratGen itself could be generated from a tool's DTD, as suggested in Figure 34.

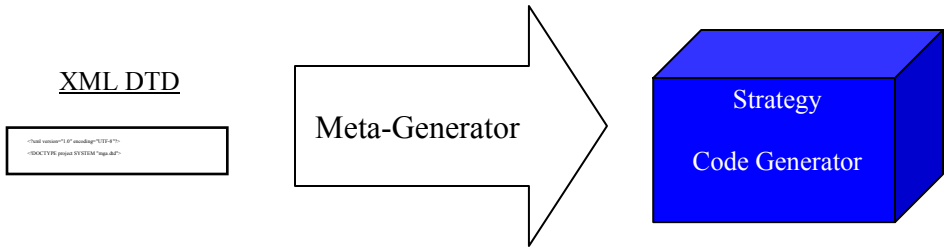


Figure 34: Generating StratGen from a Tool-Specific DTD

Variability with Respect to Aspect Languages

As noted previously, the metaweaver framework for domain-specific modeling uses the ECL for expressing both strategies and specification aspects. A point of variation within the framework is an extension that would allow the specification aspect parser to be replaced with some other language. Figure 35 highlights the modifications that are needed to permit this flexibility.

To provide variation with respect to the aspect parser, the output of a parser generator (e.g., YACC or PCCTS) needs to be integrated into the framework. Likewise, the input to the parser generator must be variable with respect to the aspect language grammar. Typically, the input to a parser generator is a grammar for a particular language, where each production in the grammar constructs a portion of an Abstract Syntax Tree (AST) from a set of data structures. These data structures that represent the AST for a language can be generated from a metalevel specification of the language, as represented by a DSL. An example tool that can provide this capability is the Abstract Syntax Description Language (ASDL), which is a part of the National Compiler Infrastructure (NCI) [Wang et al., 1997].

It is also uncertain at this point whether a framework that provides this level of variability needs the capabilities of strategies. In place of strategies, it may be the case that a traversal/visitor language is needed. This is something that will be investigated, but the open question concerning the need for a new "connector" language (e.g., strategies) leads to a discussion of a framework that provides the highest level of variability. This is described in the next section.

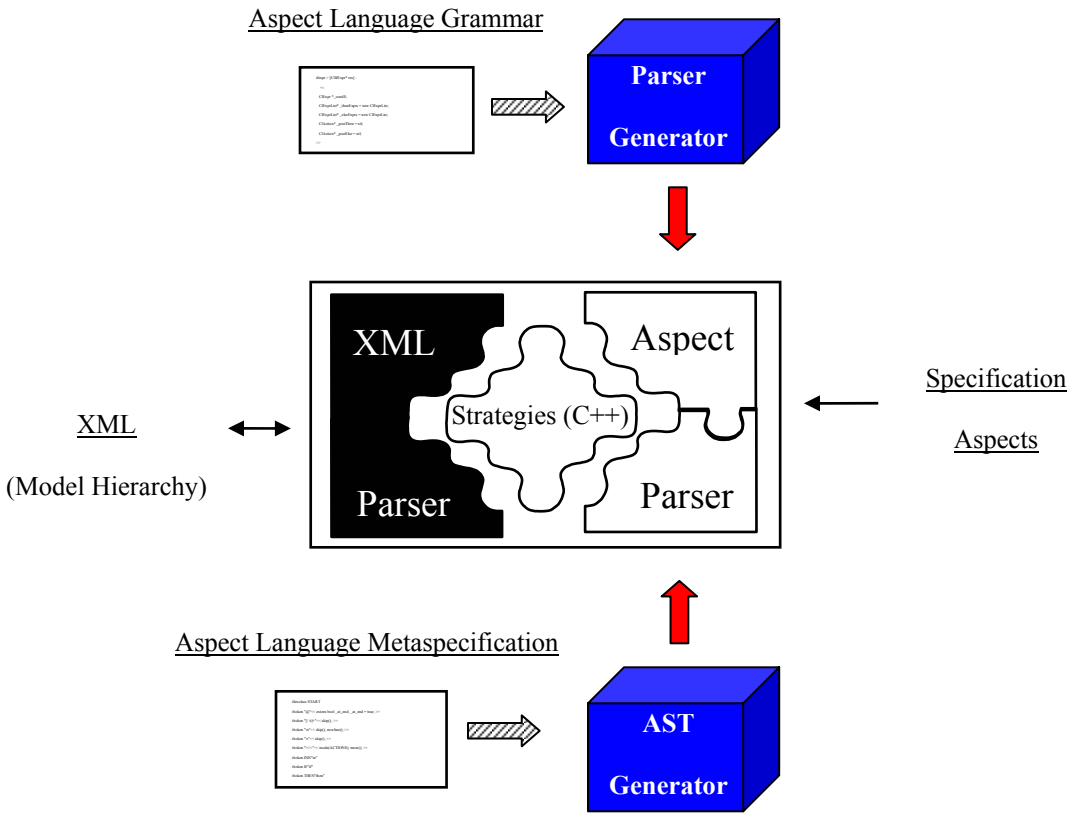


Figure 35: Variability with Respect to Aspect Language

A Metaweaver for Programming Languages

Software development occurs in a polyglot world. Recognizing this truth, it would be desirable to construct a new type of metaweaver that works with programming

languages rather than domain-specific models. This may be useful to those who want some of the benefits of AOP, but use languages other than Java and AspectJ. In a sense, each programming and aspect language becomes componentized within the weaver. This part of the proposal addresses the problem of creating new weavers for these other languages. A motivation for doing this can be found in Appendix A, where there is a recognized need for a weaver to support Delphi. Initial descriptions of this research objective were given in [Gray, 2001a] and [Gray, 2001b].

XML as an Intermediate Representation for Parsing

The potential for adaptability within the XML Parser, as suggested in Figure 30, could yield an advantage for the construction of a weaver for programming languages. There have been efforts reported in the literature that document the use of XML as an intermediate representation for ASTs. The most mature of these efforts is presented in [Badros, 2000], which describes a markup language called JavaML. An XML DTD for representing C programs is defined in [Zou and Kontogiannas, 2001]. Personal contact with a third-party Oracle tool vendor has also revealed that a tool exists that can export and import XML representations of Oracle PL/SQL. A prototype that combines an XML representation with AOP has been presented in [Schonger et al., 2002]. This prototype borrows from the work of [Badros, 2000] and extends it with an XML-based notation for specifying join points and advice.

A problem with this idea, of course, is that the original source code still must be parsed, and then converted into the intermediate XML representation. The work cited here does not offer a solution to the problem of constructing a parser for these languages. In several of these cases, parsers from other individuals were adapted to work with an

XML representation [Zou and Kontogiannas, 2001]. The benefit, though, is that these intermediate representations are in a format that can be manipulated by the weaver using specification aspects and strategies written in the ECL. There are other problems that may make this approach unfeasible for large programs; it may not scale well. Issues of performance, which were reported in a commercial tool, may make this option feasible only for small programs [Germon, 2001]. The amount of time to process 15,000 lines of Java code into JavaML, however, is reported to have taken only 12 seconds [Badros, 2000].

An experiment (reported in [Zou and Kontogiannas, 2001]) in this area examined the original file size of C source code and compared it to the size of the resulting XML AST representation. In one program, the size went from 164Kb of C source code to an equivalent XML representation of 1.6Mb. Another program, which was originally 628Kb of C source, ballooned to over 25Mb of XML. The worst case found an increase from 930Kb of C to over 47Mb of XML. Obviously, the large size of these files is due to the XML tags that are used within the document. The size of these representations make the approach (of using XML as an Intermediate Representation) impractical for anything but small programs.

Extending the Metaweaver Concept to Programming Languages

Building on the ideas of extension provided in the past few sections, the metaweaver framework can be extended to make it easier to mix and match different base programming languages (e.g., Ada, Delphi, Prolog) with various aspect languages. Thus, once the initial metalevel definitions are provided for all of the base and aspect languages, it would be possible, for example, to have a Delphi version of a weaver for

several aspect languages that have also been defined. In order to build a new weaver using this approach, the following must be provided to the framework.

1. A metalevel description of the key elements of the base programming language is needed, as well as a description of the relations between each element (e.g., classes contain methods and attributes). In a previous project, a language to accomplish a similar goal was defined [Karsai and Gray, 2000] (see Figure 12).
2. The same metalevel description is also needed for the aspect language.
3. The weaver must know how to parse the base programming language. Therefore, this must be described using a parser generator like YACC or PCCTS.
4. The weaver must also know how to parse the aspect language.

An interpreter must be able to walk the generated syntax trees of the base and aspect languages and be able to perform the weave of the joinpoints. In a past project, experience was gained in specifying higher-level traversal/visitor sequences [Karsai and Gray, 2000] (see Figure 13). An adaptation of this method could be useful here. Pieces of this idea borrow from the previous work of Adaptive Programming [Lieberherr, 1996] with respect to languages for traversal of object structures [Ovlinger and Wand, 1999].

Figure 36 describes the integration of a weaver using all of the above parts. This framework is more generic in the sense that the XML Parser has been removed and replaced with the metalevel description of a programming language. Note also that the framework depends upon three types of software generators (shown in ovals).

First, the *base.def* and *aspect.def* files must be supplied to a software generator that creates classes based upon the metalevel definitions contained in the files. These classes can be used to build a syntax tree to be used during parsing. The syntax tree

generator of Figure 36 is focused on the creation of data structures. Here, the data structures encode the structure of an AST for the defined language. Related work on using a DSL and a generator for creating data structures is introduced in [Smaragdakis and Batory, 1997]. Their work, however, is focused at a lower level of description and requires an Intentional Programming environment.

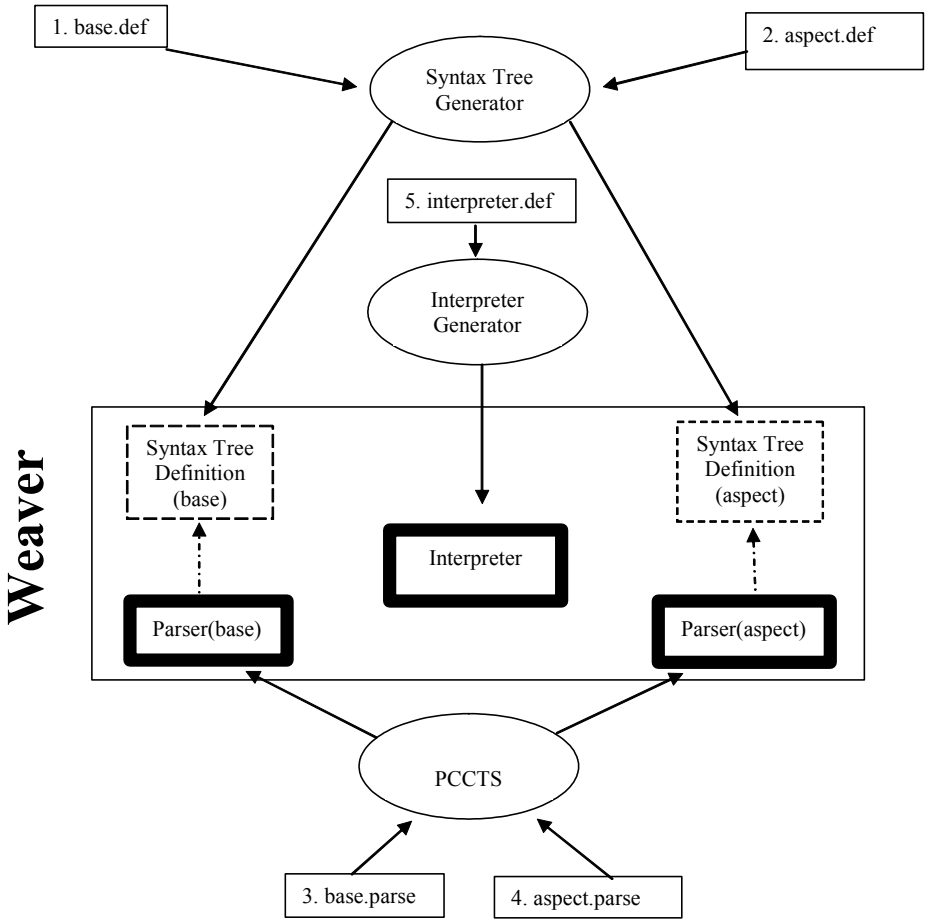


Figure 36: Metaweaver Framework for Programming Languages

Second, the *base.parse* and *aspect.parse* files must be processed by a parser generator. These files will generate the code that will parse the aspect and base programs.

Finally, the traversal/visitor strategies needed to perform the weave from the two syntax trees are specified in an interpreter definition file, *interpreter.def*. This is fed into a software generator that creates the code needed to perform the graph transformation.

To understand the operation of the weaver, consider the diagram in Figure 37. Given an instance of a program file and an aspect file, the parser uses the syntax tree definition classes to construct a syntax tree of both the base and aspect programs. The interpreter performs the weave by traversing/visiting these trees. This may require numerous stages and traversals. The interpreter component of the weaver is responsible for producing the woven program as output.

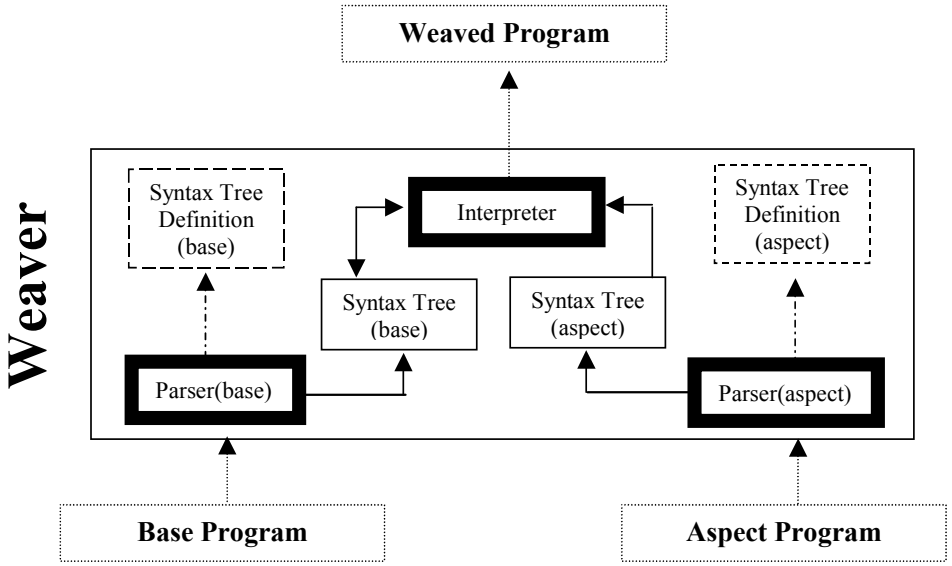


Figure 37: Inputs/Output of Weaving Process

From experience in other work [Karsai and Gray, 2000], it was found that a framework that uses software generators greatly reduces the amount of time needed to create new applications. When metalevel descriptions are provided to generators, much of the creation of the tedious and boring code is delegated to the generator. The metalevel specifications for a particular language permit the language itself to be treated as a kind of component that can be plugged into the framework. Inserting the metalevel language descriptions into the framework can create new weavers for different languages.

A metaweaver can offer the distinguishing advantage of being able to take the concepts of AOP to new languages. The power of AOP would be available to many new developers who use languages other than Java and AspectJ. This would have been beneficial in implementing the applications described in the case studies of Appendix A.

Among all of the extensions proposed in this chapter, the one described in this section is the most likely to fail. One of the difficulties of this proposed extension relates to the problem of obtaining a parser for a base language (see the base.parse file in Figure 36). Several approaches for adapting existing parsers from other tools are described in [Lämmel and Verhoef, 2001], but, as the authors of that article state, "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." In the absence of readily available parsers, this approach is very labor intensive. Given such restrictions, it is quite possible that program transformation systems (introduced in Chapter 2) can offer aid with respect to access to pre-existing parsers.

Open Development Environments

The metaweaver concept can be positioned as a means to assist in the creation of development tools outside of the purview of AOP. The whole concept actually applies to any situation where a base programming language is extended with a new type of DSL. There are some very interesting projects that are being made available as open-source that could be leveraged to provide an environment for exploring the extensions mentioned in this section.

For several years, compiler vendors (e.g., Borland) have "opened up" their development environments by providing an API for attaching plug-ins to an Integrated Development Environment (IDE). For instance, through Borland's OpenTools API, plug-ins can be created and attached to the development environment in such a way that the plug-in has access to the source code text in the editor, as well as control over all of the various windows and menus within the environment. These plug-ins can offer a powerful facility for customizing specialized tasks within an IDE.

The Eclipse project (www.eclipse.org), led by Erich Gamma, represents a completely open environment that can be extended by plug-ins. In Eclipse, all parts of the environment are open to extension – including the compiler and debugger. In addition to providing a host for the metaweaver, Eclipse plug-ins could be developed to assist in research that focuses on debuggers for DSLs.

Aspect Language Extensions

Perhaps less interesting are the possibilities for extending the ECL. The ECL has truly been an evolving language – each new strategy that was created brought some fresh insight into additional language constructs that would be beneficial. There are several

language constructs that were not completed at the time of the writing of this section. These will certainly be a focus for extension.

Because strategies can be chained together in the style of procedure calls, it makes sense to be able to return computed values from a strategy, as in a functional style. This is currently not possible, and future work will look at enriching the signatures of strategies to offer return types. It is also within reason to expect an extension to the parameter passing mechanism so that styles other than call-by-value are supported.

The ECL does not offer support for any type of container data structure. This is very limiting in situations where a fair amount of state needs to be stored in order to perform a computation. When those facilities are needed, currently, the strategy must resort to using inline code and native C++ containers. A primary goal in the development of the ECL is to grow the language so that the inline facility is not needed. This would include a provision for string manipulation routines.

A very powerful construct in AspectJ is the "cflow" pointcut designator. With cflow, all of the join points in the control flow of a pointcut are selected. There are certainly some advantages that could be realized if a similar feature were available in the ECL. In fact, the `Backflow` strategy in Figure 51 requires an inspection of "DFlow" in order to carry out its task.

CHAPTER VI

CONCLUSION

Question: *What are the most exciting/promising software engineering ideas or techniques on the horizon?*

David Parnas: *I don't think that the most promising ideas are on the horizon. They are already here and have been here for years but are not being used properly.*

[Parnas, 1999]

The objection that Parnas expresses in the above quote is specifically targeted to the lack of knowledge and training of those who would call themselves software engineers. The comment, however, also alludes to the fact that many of the same problems and key research issues that existed in the past are still here today. In many cases, the reason that these ideas are "not being used properly" is a direct result of the inability of our current tools and languages to support development using these longstanding ideas. In particular, the main objectives for modular decomposition, as suggested in [Parnas, 1972], are sometimes still unachievable because of the lack of support for separating certain kinds of concerns that tend to crosscut traditional boundaries of module demarcation. New research efforts into Aspect-Oriented Software Development are providing the language constructs that are needed to implement the support for separation of crosscutting concerns.

This dissertation has focused on a specific research area that has targeted separation of concerns as a central issue. A key goal of this work has been to raise crosscutting concerns to the level of first-class (i.e., to provide explicit representation

constructs). Specifically, a focal objective was the application of the concepts of aspect-oriented programming to domain-specific modeling. The implementation of this objective has resulted in a means for applying aspect modeling, *per se*, to the repertoire of a previously proven modeling tool (the GME). This dissertation's contribution is distinguished within the literature as being the earliest example of a weaver for aspect-oriented domain-specific modeling. As other work in the general area of aspect-oriented design has concentrated on notational and diagrammatic issues, the research described in this dissertation has brought the benefits of aspect-orientation to the modeling process itself.

There are several reasons that would support the adoption of these ideas into a general modeling approach. As presented in Chapter 3 ("Constraint Driven Design Space Navigation"), it was discovered on a previous DARPA project (i.e., Adaptive Computing Systems) that a lack of support for separation of concerns with respect to constraints can pose a difficulty when creating domain-specific models. Constraints may be specified throughout the nodes of a model in order to stipulate design criteria and limit design alternatives. For example, power constraints may be written for all of the nodes in a functional hierarchy. However, when the specification changes, each node expressing a power constraint must be visited and updated. Whether the constraints relate to the operation, composition, or resources of the system, their scattering throughout various levels of a model makes it difficult to maintain and reason about their effects and purpose.

The concept of a domain-specific weaver, which was introduced in this dissertation, can be used in many ways beyond the application of constraints. The weaver

can be used in order to distribute any system property (that is endemic to a specific domain) across the hierarchy of a model. The weaver can also be used to instrument structural changes within the model according to the dictates of some higher-level requirement that represents a crosscutting concern. Domain-specific weavers rely on specification aspects and strategies to carry out their duty. Specification aspects, similar in intent to pointcuts in AspectJ [Kiczales et al., 2001b], are used to describe *where* the concern will be applied in the model, and strategies describe *how* a concern is applied in the context of a particular node in the model.

To support the creation of weavers for numerous modeling domains, a metaweaver framework was created to aid in the construction of new weavers. In this dissertation, the framework, in conjunction with several code generators and DSLs, are used to provide the adaptability needed to construct new instances of the framework. A core component of this framework is a code generator that translates high-level descriptions of strategies into C++ source code.

The forecast from the preceding chapter outlined several possible extensions to this work. The extensions related to adding new constructs to the ECL, bringing the specification of modeling aspects into the visual programming context, and providing additional variability within the metaweaver framework. A long-term goal will be to apply the framework to the creation of weavers for programming languages, with a peripheral research goal of providing support for debugging of domain-specific languages.

The following were key elements to the successful realization of the contributions described in this dissertation.

1. The design of a framework, or product-line architecture, that permits the pluggability of application components.
2. Modeling of the configuration knowledge such that abstract specifications are translated into concretized components.
3. Implementation of the configuration knowledge using generators.

These three characteristics were pointed out in [Czarnecki and Eisenecker, 1999], where reference was made regarding the industrial revolution and automobile assembly:

> *The principle of interchangeable parts was the prerequisite for the introduction of the assembly line by Ransome Olds in 1901, which was further refined and popularized by Henry Ford, and finally automated using industrial robots in the early 1980s.*

APPENDIX A

CASE STUDIES IN ASPECT-ORIENTED PROGRAMMING

This appendix describes my own personal difficulties while encountering problems with respect to separation of crosscutting concerns using Delphi (a popular Windows development environment, based on Object Pascal, offered by Borland). Three different software products, which were all used in the deployment of a successful commercial application, each had their share of problems with respect to scattered code. The applications described in this appendix were developed from 1997 through 1999 for a telecommunications company, and are still being deployed to customers at the time of completion of this dissertation.

LangMan – Handling Dirty Bits

With today's commercial software, there is an economic incentive to internationalize an application so that the software can be sold in different countries for commercial advantage. There are many things involved in this process. One of the key challenges is the storage and retrieval of all the textual strings that appear in an application in a manner that permits the representation of those strings in different written languages. One technique for doing this 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 LangMan application is a tool that was created to support such a task.

The implementation of LangMan 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. There are 29 unique places in the source code where accesses to the Boolean variable `EditMadeDirtyBit` are made.

There were only four different types of contexts in which the `EditMadeDirtyBit` was accessed. Two of the contexts simply dealt with setting the value to true or false, based upon a particular situation. This was spread across several diverse classes and represented nine of the places where this concern occurred (i.e., these two kinds of modifications were found in nine different places in the program). The other two contexts in which access to the dirty bit appeared dealt with performing some action based upon the value of `EditMadeDirtyBit`. The code for deciding what to do, based on the value of the bit, was identical in each source code location. Thus, redundant code was found in many different places. Any modification or change to the way in which a text string is stored often required a change to the way in which this concern was implemented. This required the programmer to visit 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 string at run-time.

This type of concern represents an example of property-based crosscutting. A very simple property (here, the indication of whether a change has been made to GUI widgets) was spread across several different classes. The concern necessitated that it appear scattered within the textual context of several classes. Traditional techniques of

modularization would not permit a separation of this concern from the contexts in which it appears. Even if a simple class were created to contain this concern (e.g., a small class that contained a Boolean state variable, with accessor methods), it would still be necessary to scatter the concern among the numerous widgets.

Database Error Handler – Synchronization as a Concern

Often, a commercial application must work with databases from several different vendors. 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. The application, however, must make this transparent to the user while interpreting the exception and providing a meaningful message back to the end-user.

To accomplish this transparency, a database error handling DLL was created. This library contained 23 classes. The majority of these classes were responsible for handling specific types of exceptions. The "Chain of Responsibility" pattern was used where the exception was passed along to a list of potential handlers [Gamma et al., 1995].

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. This, of course, required the use of a mutex to ensure that only one error handler was invoked at a time. This required an invasive change to over 20 classes. Invasive changes like this are the source of maintenance nightmares.

An example of one of these error handlers is shown in Figure 38. In that figure, lines 5-7 and 21-23 are present because of this single synchronization concern.

Furthermore, this exact code is replicated in all of the entry and exit points of each database handler. It would be desirable to have a single location from which this single concern resides.

As an aside, note the statement that appears in line 17. The error that is displayed to the user is obtained from the name of a constant, `L_DBERR_NullFields`. This constant represents an index to the string to be used in the resource DLL that was created by the LangMan utility described earlier.

```
 0  // Null field exception object
 1  function TExNullField.Handle(ServerType : TServerType;
 2                               E : EDBEngineError) : Integer;
 3  begin
 4
 5    TExHandleCollection(Collection).LockHandle;
 6
 7    try
 8
 9    Result := -1;
10
11    if E.Errors[0].ErrorCode = BDEFieldRequired then
12      if ((GetNativeErr(E) = ORANotNull) and
13          (ServerType = svOracle)) or
14         ((GetNativeErr(E) = INTNotNull) and
15          (ServerType = svInterbase)) then
16          begin
17            Result := L_DBERR_NullFields;
18            DisplayError(L_DBERR_NullFields, E);
19          end;
20
21    finally
22      TExHandleCollection(Collection).UnLockHandle;
23    end;
24
25  end;
```

Figure 38: A Database Error Handler

Schema Manager – Processing Dialogs and Logging SQL Queries

The Schema Manager is a utility that assists customers in upgrading to a new database schema during an update to the application software. Utilities like the Schema 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. In fact, with respect to error handling, the following code fragment appears 33 times in different methods:

```
on E : Exception do
  begin
    dmSERVERS.HandleException(E);
    dmSERVERS.ProcessingDialog.Canceled := True;
  end;
```

Figure 39: Redundant Exception Handling Code

Code replication with respect to exception handling is a dangerous thing. As mentioned in Chapter 2, this problem was studied in depth by [Lippert and Lopes, 2000]. It would be desirable to have a way to create a single separate module that describes all of the functionality of updating the progress meter.

Another crosscutting concern that is scattered throughout the Schema Manager is the logging of SQL code. As the Schema Manager utility upgrades the customer's database to a new schema, all of the SQL that is generated to perform the upgrade is logged to a file so that it can be examined later in the event of a problem. 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 program. Again, the ability to collect the logging actions in a single module would aid in better separation of concerns. Unfortunately, for Delphi and most other programming languages, there are no language constructs to provide these desired capabilities.

This section offered a case study of specific problems resulting from scattered and tangled code. These difficulties were a result of the inability to represent crosscutting concerns in a particular programming language. The remainder of this chapter describes the way that AOP could be used to offer a solution to these problems. The solutions are considered from the point of view of AspectJ, rather than Delphi.

AspectJ Examples

Please reconsider the exception handling code fragment from Figure 38. In this section, that code fragment will be rewritten in AspectJ. The AspectJ code will contain the mutex lock, but will also write to a log immediately after displaying the error to the user. Only the details that are pertinent to highlighting the use of AspectJ are provided here, for clarity. It is also acknowledged that Java supplies keywords, such as `synchronize`, to handle some of these concerns. The specification of a synchronization concern here is provided merely for illustrative purposes of the use of AspectJ.

Figure 40 contains the core concern of the null field exception handler. Note that this class is a subclass of a generic error handling class (`TExErrorHandler`) and the

`Handle` method of this class has no provision for synchronization or logging. This first class is pure Java.

```
public class TExNullField extends TExErrorHandler
{
// other methods removed

  public int Handle(EDBEngineError e) {

      int Result = 0;

      if(e.ErrorCode == BDEFieldRequired)
      {
        Result = L_DBERR_NullFields;
        DisplayError(L_DBERR_NullFields, e);
      }

      return Result;

   }
}
```

Figure 40: Null Field Exception Class

An aspect that handles the concern of logging is coded in Figure 41. The `Logging` aspect contains its own copy of a logging object (`TLog`). A pointcut designator is defined in this aspect that applies to instances of the error handling superclass that call the `DisplayError` method. In this case, all signatures of `DisplayError` participate in this designator, as indicated by the wildcards. The `after` advice of this aspect simply makes the appropriate call that will add a message to the log. This message will contain the name of the error-handling object that was able to process the error.

```
aspect Logging
{
    TLog aLog = new TLog();

    pointcut display_error(TExErrorHandler c) : this(c) &&
                                        call(* *.DisplayError(..));

    after(TExErrorHandler c): display_error(c) {
      aLog.AddText("Display Error invoked by: " + c);
    }
}
```

Figure 41: Logging Aspect

The `Locking` aspect, given below in Figure 42, is very similar to the `Logging` aspect. The fundamental difference is that the pointcut designator in this case applies to the execution point of the `Handle` method. The `Logging` aspect has `before` advice that locks the mutex and `after` advice that unlocks the mutex.

The pointcut designators provided in these aspects are much more powerful than can be illustrated with this simple example. These designators actually have the potential of affecting numerous different classes and methods through quantification. The semantics of these pointcuts would push the advice into *all* instances of `TExErrorHandler` and instances of its subclasses. Variations on the pointcut designator could be given to provide variability with respect to the (un)plugging of logging and locking. This is a powerful capability whose importance is not captured very well with such a small example.

```
aspect Locking
{
    TLock aLock = new TLock();

    pointcut synchronize(TExErrorHandler c) : this(c) &&
                                    execution(* *.Handle(..));

    before(TExErrorHandler c): synchronize(c) {
      aLock.LockHandle();
    }


    after(TExErrorHandler c): synchronize(c) {
      aLock.UnLockHandle();
    }
}
```

Figure 42: Locking Aspect

Observe how the concerns of logging and locking have been removed from the concern related to the core functionality of handling a null exception. To weave the aspects into the error handler, the weaver can be called in the following way:

```
C:\AspectJ\ExceptionHandler> ajc nullfield.java log.java lock.java
```

Actually, the locking and logging concerns can apply to many other error handling objects, not just the null exception handler as shown here. To understand the effects of the weaving process, the next two figures show some of the generated code from the weaver.

There are numerous differences between the original `Handle` method and the generated code in the figure below. The most obvious addition to the generated code is the `try/finally` block that encompasses the method boundary. This block ensures

```
/*   Generated by AspectJ version 1.0.3 */
public class TExNullField extends TExErrorHandler {
  private int BDEFieldRequired;
  private int L_DBERR_NullFields;
  public int Handle(EDBEngineError e) {
    try {
      Locking.aspectInstance.before0$ajc(this);
      int Result = 0;
      if (e.ErrorCode == this.BDEFieldRequired) {
        Result = this.L_DBERR_NullFields;
        this.DisplayError$method_call(this,
                                      this.L_DBERR_NullFields, e);
      }
      return Result;
    } finally {
      Locking.aspectInstance.after0$ajc(this);
    }
  }

  public TExNullField() {
    super(); {
      this.BDEFieldRequired = 7;
      this.L_DBERR_NullFields = 100;
    }
  }

  private void DisplayError$method_call(TExNullField target,
                                        final int ErrNum,
                                        final EDBEngineError e) {
    try {
      target.DisplayError(ErrNum, e);
    } finally {
      Logging.aspectInstance.after0$ajc(this);
    }
  }
}
```

Figure 43: Generated Code for Null Field Exception Class

that the mutex lock does indeed get released, even in the event of an exception. The mutex is locked in the first statement of the block by passing control to a method that was created from the `Locking` aspect. Another obvious addition to this method is the introduction of code to implement the logging. This is accomplished by generating a new

method, with a mangled `DisplayError` name, that dispatches to the original `DisplayError` before calling a `Logging` method.

The code that the weaver generated for the `Logging` aspect is visible below. As can be seen, the code created by the weaver to implement the aspect is encapsulated within a Java class. The majority of the generated content is focused on methods that provide initialization and reflective information about the aspect. A mangled method name (e.g., `after0$ajc(TExErrorHandler c)`) represents the `after` advice that implements the logging. The generated code to handle locking would be similar in structure to the logging code below. The locking example, however, would also have a mangled method name for both `before` and `after` advice.

```
/*   Generated by AspectJ version 1.0.3 */
class Logging {
  TLog aLog;
  public final void after0$ajc(TExErrorHandler c) {
    this.aLog.AddText("Display Error invoked by: " + c);
  }

  Logging() {
    super();
    { this.aLog = new TLog(); }
  }
  public static Logging aspectInstance;
  public static Logging aspectOf() {
    return Logging.aspectInstance;
  }

  public static boolean hasAspect() {
    return Logging.aspectInstance != null;
  }

  static {
    Logging.aspectInstance = new Logging();
  }
}
```

Figure 44: Generated Code for Logging Aspect

APPENDIX B

CASE STUDIES IN ASPECT MODELING

Chapter 3 presented an example strategy for distributing a power constraint across a model. The purpose of this appendix is to provide three more examples, within two different domains, which show various strategies that have been used to separate crosscutting modeling concerns. The first set of examples further demonstrates the weaving of constraints that are used during design space navigation [Neema and Lédeczi, 2001], as described earlier. The third example illustrates the ability of a strategy to alter the structure of a model by providing adaptation within finite-state machines.

Boeing's BoldStroke/CORBA Component Model

Boeing's BoldStroke is a product-line framework for avionics navigation software [Sharp, 1998]. In this section, two applications of strategies will be presented in a domain for modeling a subset of BoldStroke applications and configurations. The two different strategies will be concerned with processor assignment and eager/lazy evaluation.

Consider the diagram in Figure 45. This represents a simple model, previously presented in [Gray et al., 2001b], which contains five components. The first component is an inertial sensor. This sensor outputs, at a 100Hz rate, the position and velocity deltas of an aircraft. A second component is a position integrator. It computes the absolute position of the aircraft given the deltas received from the sensor. It must at least match the sensor rate such that there is no data loss. The weapon release component uses the absolute position to determine the time at which a weapon is to be deployed. It has a fixed period

of 20Hz and a minimal latency requirement. A mapping component is responsible for obtaining visual location information based on the absolute position. A map must be constructed such that the current absolute position is at the center of the map. A fifth component is responsible for displaying the map on an output device. Notice the frequencies, latencies, and Worst Case Execution Times (WCET) of these components. The specific values of these properties will likely differ depending on the type of aircraft represented by the model (e.g., the latencies and WCETs for an F-18 would most likely be lower than a helicopter). The core modeling components describe a product family with the values for each property indicating the specific characteristics of a member of the family.

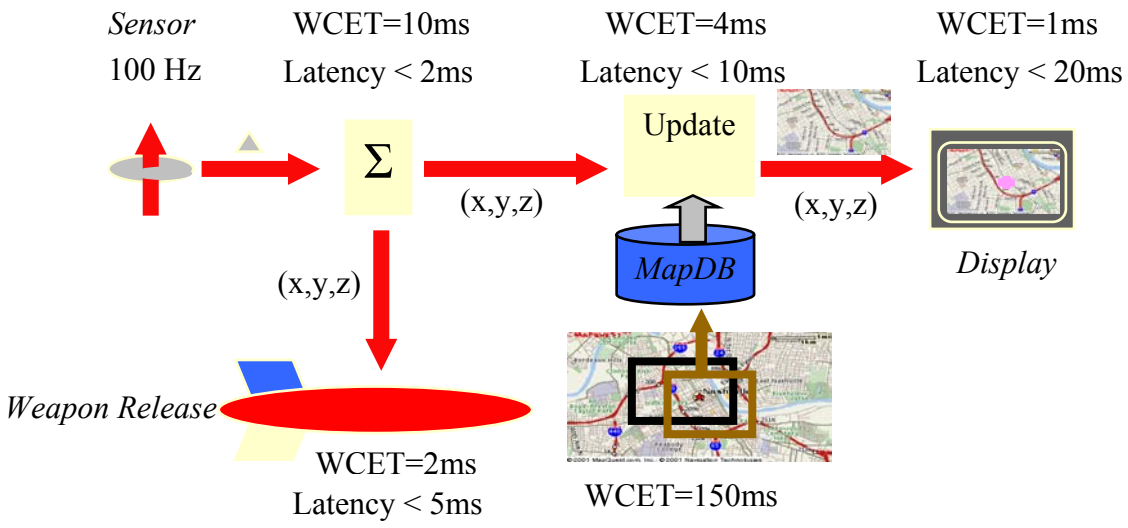


Figure 45: A Weapons Deployment Model

Figure 46 provides a depiction of the weapons deployment model, represented within the GME. The model is an instance of the paradigm that was initially developed for the DARPA MOBIES program, and later refined for the DARPA PCES project, to

assist in the modeling of BoldStroke applications (Note: the majority of the development of this paradigm was completed by Dr. Sandeep Neema – the paradigm itself is not claimed as a contribution of this dissertation). The extensions that were made for PCES permit the representation of CORBA Component Models (CCM) [Siegel, 2000]. The CCM provides capabilities that offer a greater level of reuse and flexibility for developers who need to deploy standardized components [Wang et al., 2001].

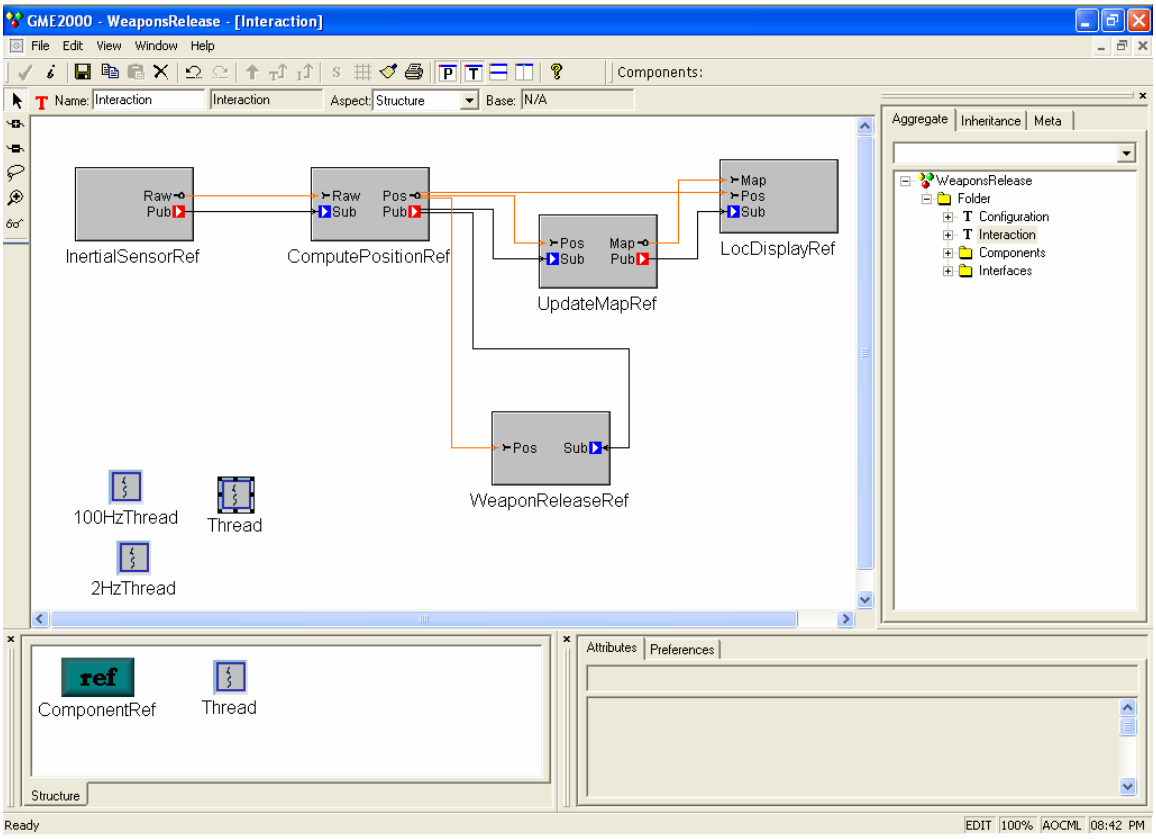


Figure 46: A GME Model of the Component Interactions

Each of the components in Figure 46 has internal details, in support of the CCM, that also are modeled. For instance, the contents of the Compute Position component are

rendered in Figure 47. As can be noticed from the internals of this component, the series of interactions actually take place using a publish/subscribe model. The figure specifically highlights the attributes of a method called “compute” (see the bottom-right of the figure). The attributes provide the name of the method, the C++ source file that contains the method, and the method’s estimated WCET.

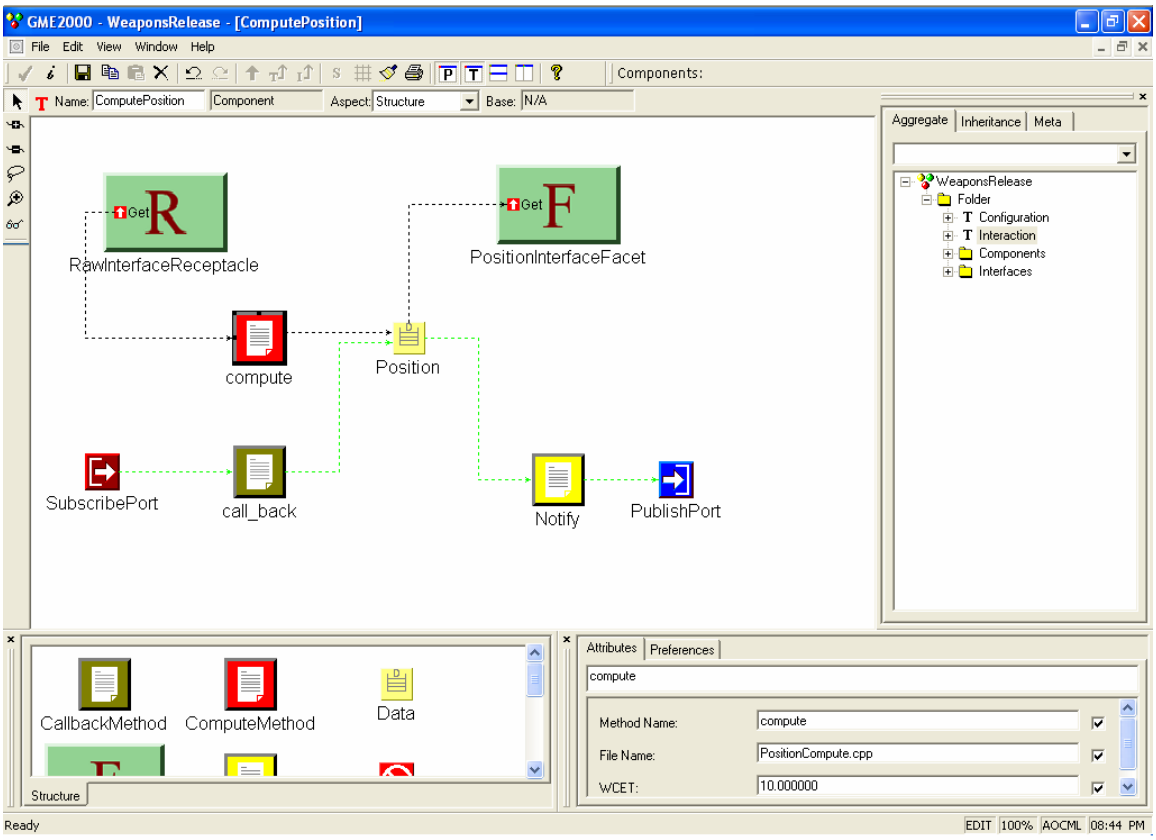


Figure 47: The Internals of Compute Position

Weaving Constraints: Processor Assignment

Suppose that we wanted to model the processor assignment of each component. That is, based upon the expected WCET, the component methods are executed as tasks on various processors. A notation is needed to specify the assignment of component methods/tasks to processors. One way to accomplish this representation issue is to specify the processor assignment as a constraint of the component model. The way that processor assignment is typically modeled involves the application of a set of heuristics that globally assign tasks to processors based on specific properties of each component. In modeling, this is often done by hand and requires the modeler to visit each component, or task, in order to manually apply the heuristic. For a model with a large number of components, this can be a daunting task. It becomes increasingly unmanageable in situations where the modeler would like to play "what-if" scenarios. These "what-if" scenarios are used to drive the iterative evolution of the model, such that intermediate scenarios may even be discarded. This is helpful because a modeler may want to change the values of different properties, or even modify the details of the heuristic, in order to observe the effect of different scenarios. A manual application of a heuristic would require that the modeler re-visit every component and re-apply the rules of the heuristic.

An example of a specification aspect and strategy to support processor assignment can be found in Figure 48. The interpretation of the aspect called `ProcessorAssignment` is that an iteration is specified over all of the modeling elements that are of type "Comp*" (note the use of the wildcard designator). The strategy, called `Assign`, is then invoked on each of these modeling components (here, a parameter bound to the value 10 represents a threshold of the execution time for each

processor load). The purpose of the `Assign` strategy is to look into the "compute" method of each component and find its WCET. The WCETs of each component are accumulated. Whenever this accumulated value reaches past the threshold, a new processor is created for component assignment. `Assign` will finally call another strategy, named `AddConstraint`, which will add a new constraint to the model. The new constraint, in this case, represents the processor assignment. Admittedly, this particular strategy for processor assignment is very simple and would not be a best choice. However, it has been chosen for its simplicity so that intricacies of the algorithm do not overshadow the intent of demonstrating the manner by which the processor assignment constraints are distributed. Also, observe that the entire strategy is written purely in ECL (except for one line of inlined code that is used for string creation).

Figure 49 shows the same component that was given in Figure 47. The only difference is that the component now contains a constraint that was added by the weaver as a result of applying the strategies described by the specification aspect. Notice that the strategy has assigned this component to processor 0. An examination of all the other components involved in this interaction would reveal that different components are assigned to processors based on their WCET and the parameterized threshold.

Note that the `ProcessorAssignment` aspect could be modified so that a different strategy is invoked (i.e., some strategy other than `Assign)`; or, a different parameter threshold could be provided that may result in a different set of constraints (i.e., the parameter to `Assign` may be changed from 10 to 20). The key advantage of this approach is realized in the observation that, from a change in *one* place, an entirely

different set of constraints can be weaved. This solves a serious scalability problem concerning maintenance issues, and the ability to change the constraints within a model.

```
defines AddConstraint, Assign, ProcessorAssignment;


strategy AddConstraint(constraintName, expression : string)
{

  addAtom("OCLConstraint", "Constraint",
           constraintName).addAttribute("Expression", expression);

}


strategy Assign(limit : integer)
{

  declare static accumulateWCET, processNum : integer;
  declare currentWCET : integer;

  self.compute.WCET.getInt(currentWCET);

  accumulateWCET := accumulateWCET + currentWCET;

  if (limit < accumulateWCET) then

    accumulateWCET := currentWCET;
    processNum := processNum + 1;

  endif;

  <<CComBSTR aConstraint = "self.assignTo() = processor" +
                           XMLParser::itos(processNum); >>

  AddConstraint("ProcessConstraint", aConstraint);

}
```

a) Processor Assignment Strategy

```
aspect ProcessorAssignment
{
  models("")->select(m | m.kind() = "Comp*")->Assign(10);
}
```

b) Specification Aspect for Assigning Components to Processors

Figure 48: Strategy and Specification Aspect for Processor Assignment

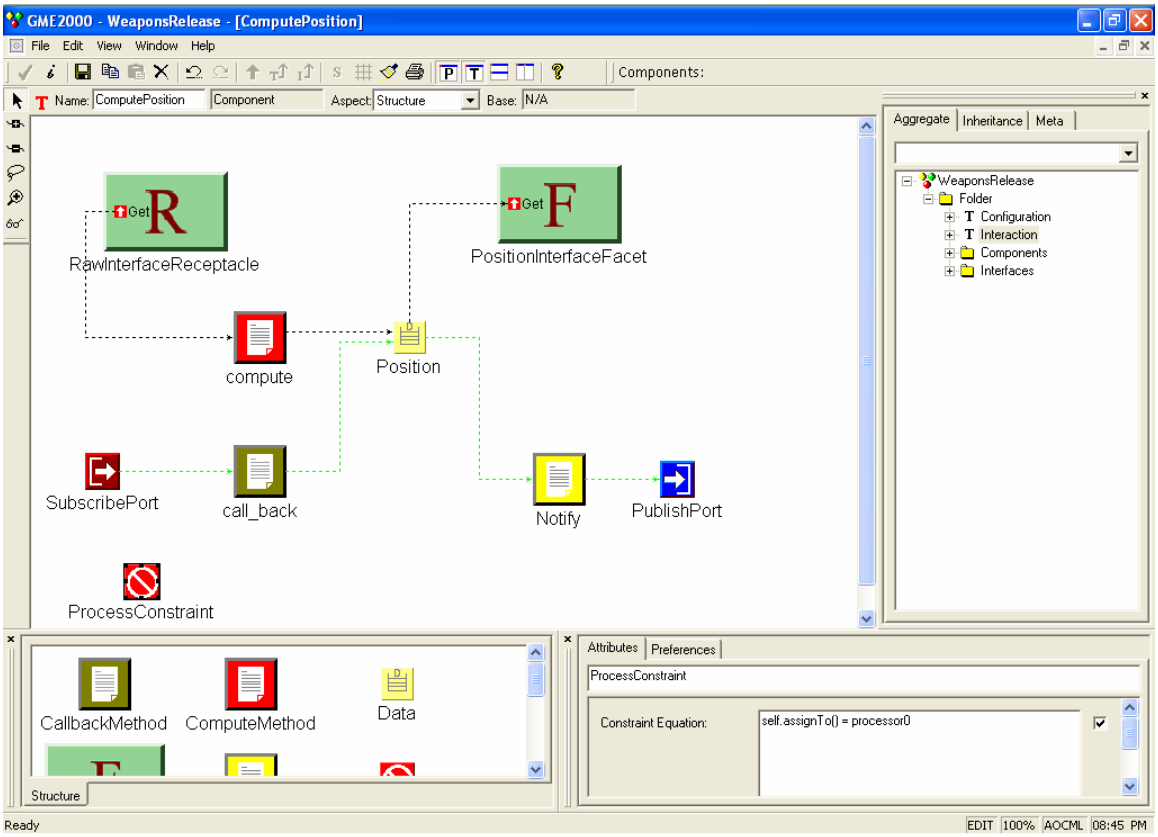


Figure 49: Component with Weaved Constraint

Determining an Eager/Lazy Evaluation Strategy

> *The point of time at which the resources are acquired can be configured using different strategies. The strategies should take into account different factors, such as when the resources will be actually used, the number of resources, their dependencies, and how long it takes to acquire the resources. Regardless of what strategy is used, the goal is to ensure that the resources are acquired and available before they are actually used.*
>
> [Kircher, 2002]

In the interactions among the various components in the weapons deployment example, there is a defined protocol for computing a value and notifying other components of a completed computation. These interactions are the result of a

publish/subscribe model that uses an event channel. The typical scenario for these interactions is:

1. One component (C) receives an event from another component (S), indicating that a new value is available from S.
2. At some point in time, C invokes the `get_data` function of S in order to retrieve the most up to data value from S. C then performs a computation based upon the newly retrieved value.
3. At some point in time, component C notifies all of the other components that have subscribed to the event published by C.

There are situations where early acquisition and computation of data can waste resources. The determination concerning how often a computation should be made is an optimization decision. In an eager evaluation, all of the steps to perform the computation for a component are done at once. An eager evaluation would follow the three steps above in a strict sequential order (see the top part of Figure 50 for a depiction of the protocol for eager evaluation) each time an event is received from a supplier component. A lazy evaluation is less aggressive in computing the most up to date value. The second step, from above, is performed late. That is, the value from the supplier, and the actual computation, are only performed when a client component requests a data value. The computation is performed only when needed, not during each reception of an event from a supplier. The concept of a lazy evaluation is shown in the bottom part of Figure 50.

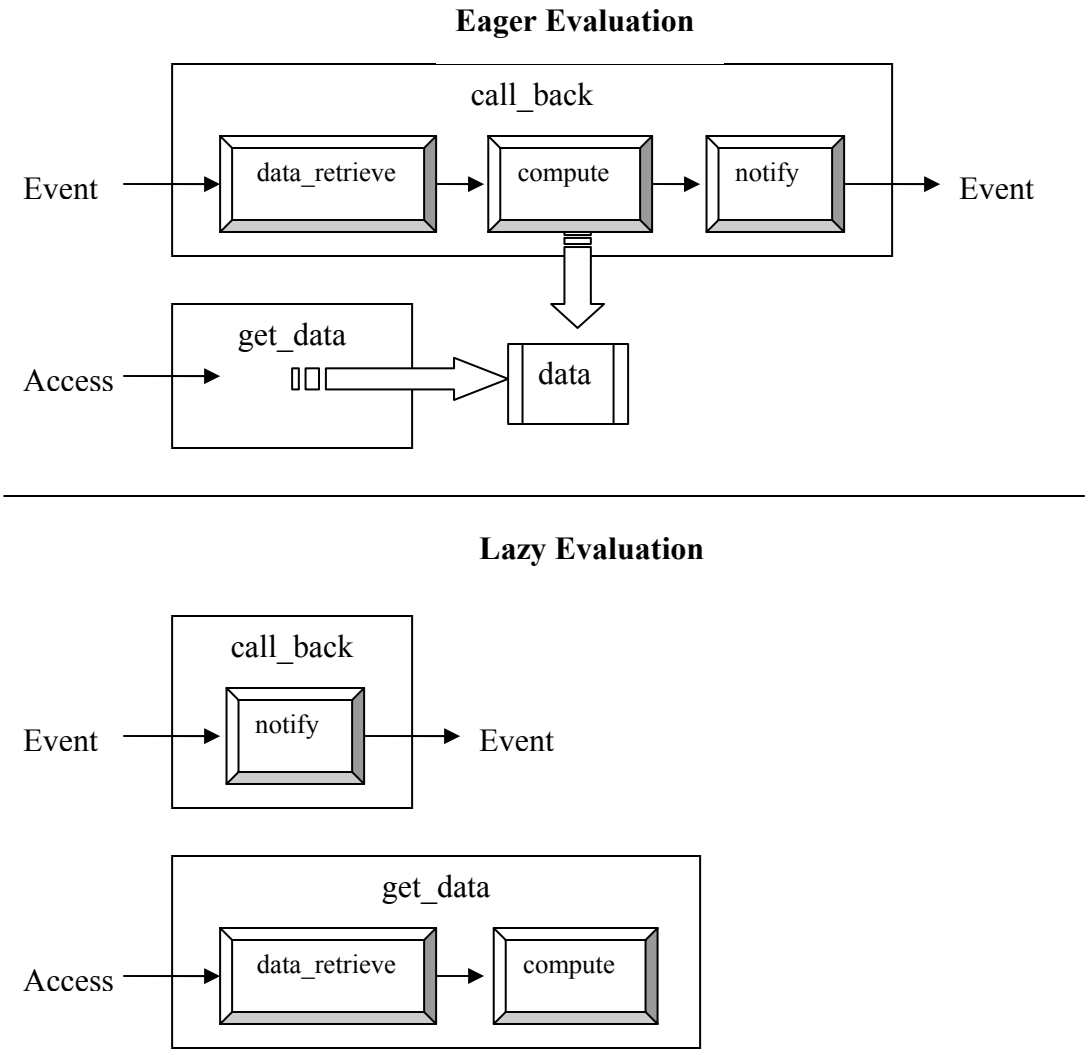


Figure 50: Eager/Lazy Evaluation Description

The manner by which a determination of eager/lazy evaluation is made can be considered a type of aspect. The determination is typically made according to some optimization protocol, which is spread across each component of the model. It would be useful to be able to separate the criteria used for deciding upon the assignment of an evaluation. Such separation would support changeability and exploration of different protocols. A specific strategy for determining eager/lazy evaluation is given in Figure 51.

From Figure 51, the `EagerLazy` strategy simply determines the location of the start and end nodes within the model. It also finds the context of folders and models that will be needed during the distribution of the concern. The parameterization of the start and end nodes, and also the latency threshold, enable this strategy to be called by a specification aspect in numerous ways.

The `DetermineLaziness` strategy is invoked on the start node (because the strategy works backwards, the start node is actually the node that is nearest to the end of the interaction). This strategy performs a simple computation to determine the evaluation assignment for the current node. If the current node is not the end node of the interaction, then the strategy named `BackFlow` is fired. This simple strategy collects all of the suppliers of the current node (this is done by finding the components that are on the current component's data flow, and serve as suppliers) and invokes a continuation on the collection. The `Continue` strategy fires the `DetermineLaziness` strategy on the collection of suppliers.

The effect of applying the `EagerLazy` strategy can be seen in Figure 52. That figure displays the modifications made to the internals of the `UpdateMap` component.

```
defines EagerLazy, DetermineLaziness, BackFlow, Continue;


strategy EagerLazy(StartName, EndName : string;
                   latencyThreshold : integer)
{

  declare components, interactions, startNode, endNode : node;

  components := findFolder("Components");
  interactions := findModel("Interaction");

  startNode := components.findModel(StartName);
  endNode := components.findModel(EndName);

  startNode.DetermineLaziness(components, interactions, endNode,
                              latencyThreshold);

}


strategy DetermineLaziness(components, interactions, endNode : node;
                           latencyThreshold : integer)
{

  declare static accumulateLatency : integer;
  declare latency : integer;
  declare currentID, endID : string;

  if (accumulateLatency < latencyThreshold) then
   AddConstraint("EagerLazy", "assignment = lazy");
  else
   AddConstraint("EagerLazy", "assignment = eager");
  endif;

  self.compute.latency.getInt(latency);
  accumulateLatency := accumulateLatency + latency;

  getID(currentID);
  endNode.getID(endID);

  if(currentID <> endID) then

     self.BackFlow(components, interactions, endNode,
                   latencyThreshold);

  endif;

}
```

Figure 51: Eager/Lazy Strategy

```
strategy BackFlow(components, interactions, endNode : node;
                  latencyThreshold : integer)
{

  declare currentID, referredID : string;

  self.getID(currentID);
  referredID := interactions.resolveReferredID(currentID);

  interactions.connections("DFlow")->select(c |
      c.connpoint("dst").refs() == referredID) ->Continue(components,
                          interactions, endNode, latencyThreshold);

}


strategy Continue(components, interactions, endNode : node;
                  latencyThreshold : integer)
{

  declare newID, referredID : string;
  declare newNode : node;

  newID := connpoint("src").refs();
  referredID := interactions.resolveIDReferred(newID);

  components.models("")->select(c | c.id() == referredID)
         ->DetermineLaziness(components, interactions, endNode,
                             latencyThreshold);

}
```

Figure 51 (cont): Eager/Lazy Strategy

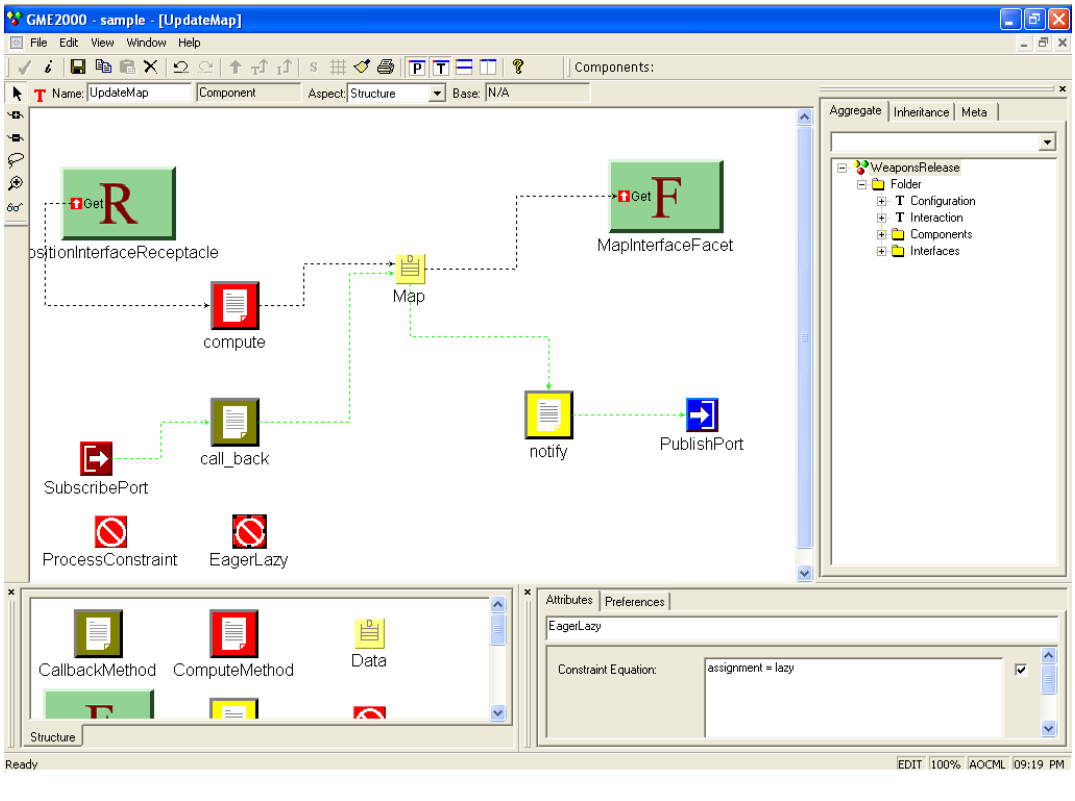


Figure 52: Effect of Eager/Lazy Strategy

Aspect Code Generation from Models

A future goal of our DARPA PCES project is the capability for generating the configuration of BoldStroke components from domain-specific models in such a way that specific parts of each component are weaved together as an aspect. This goal fits well with the OMG's Model Driven Architecture (MDA) [Bézivin, 2001], and also the idea of "fluid AOP" [Kiczales, 2001]. One possibility for realizing this objective would be the generation of AspectJ code from models. This is shown in Figure 53, where the model and specification aspects are sent through a weaver that constrains the model. The constrained model can then be sent to a GME interpreter that generates the aspect code.

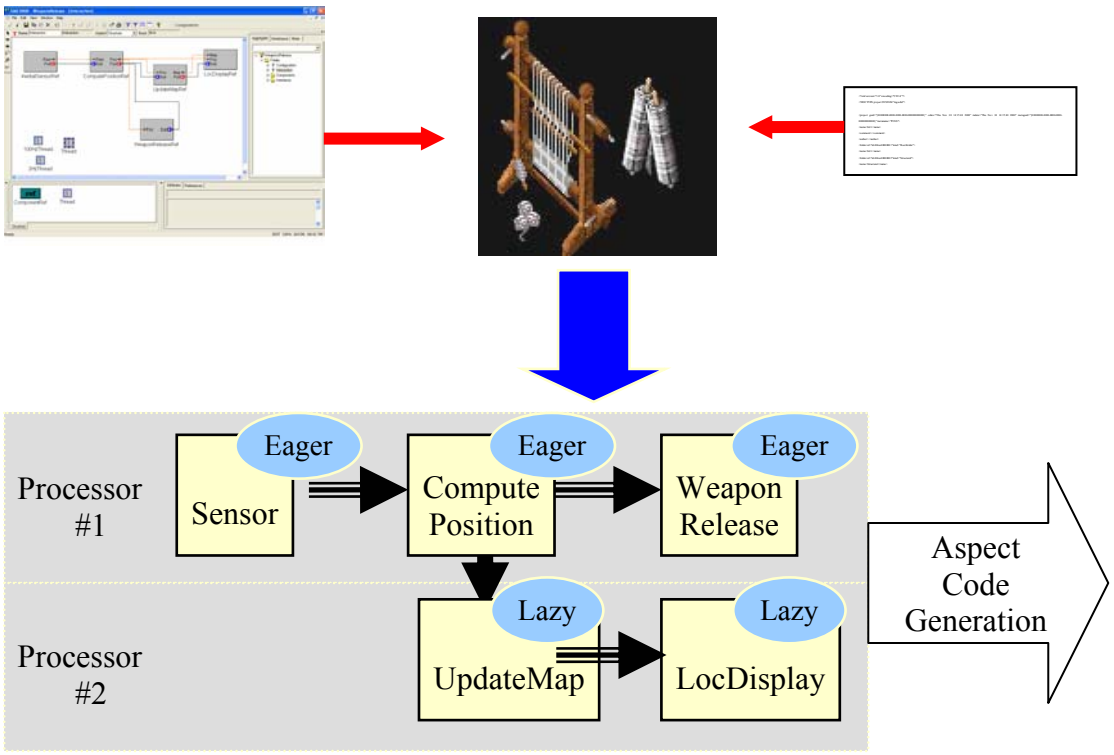


Figure 53: An MDA View of Aspect Code Generation

The amount of generated code produced from the aspect generator would actually be quite small. The assumption is that the core of the available components would already exist. Another assumption would be the existence of several different aspects of concern. These assumptions are in line with the work that other researchers are doing toward the goal of making a library of components and aspects available for a subset of the CORBA event-channel [Hunleth et al., 2001].

An example of a core library of components can be found in the Java code in Figure 54. This figure represents an abstract `Component` (a), and a `LocDisplay` component (b). The abstract component defines the required methods for the domain –

the same methods that can be found in models like Figure 47. The `LocDisplay` subclass, for clarity, simply provides stubs for each of the method implementations.

```
public abstract class Component
{

  public abstract void call_back();
  public abstract int get_data();
  public abstract void init();

  public abstract void data_retrieve();
  public abstract void compute();
  public abstract void notify_availability();

  protected int _data;

}
```

a) Component.java

```
public class LocDisplay extends Component
{

  public void call_back() {
    System.out.println("This was LocDisplay.call_back"); };

  public int get_data() { return _data; };

  public void init() { };

  public void data_retrieve() {
    System.out.println("This is LocDisplay.data_retrieve!");
    UpdateMap map = new UpdateMap();
    map.get_data();
  };

  public void compute() {
    System.out.println("This is LocDisplay.compute!"); };

  public void notify_availability() {
   System.out.println("This is LocDisplay.notify_availability!");

};
```

b) LocDisplay.java

Figure 54: Base Class Java Components

```
abstract aspect Lazy {

    abstract pointcut call_back(Component c);
    abstract pointcut get_data(Component c);

    after(Component c): call_back(c)
    {
      System.out.println("after:call_back (Lazy)!");
      c.notify_availability();
    }

    before(Component c): get_data(c)
    {
      System.out.println("before:get_data (Lazy)!");
      c.data_retrieve();
      c.compute();
    }

}
```

a) Lazy Aspect

```
aspect LocDisplayLazy extends Lazy {

    pointcut call_back(Component c) : this(c) &&
                        executions(void LocDisplay.call_back(..));

    pointcut get_data(Component c) : this(c) &&
                         executions(int LocDisplay.get_data(..));


}
```

b) Concretization of Lazy Aspect with LocDisplay

Figure 55: Sample Strategies and Specification Aspects

Example aspects are coded in Figure 55. The `Lazy` aspect contains abstract pointcuts. Other aspects will refine the definition of the pointcuts through extension. It is assumed that the `Lazy` aspect would exist in a library of reusable aspectual components. This abstract aspect captures the model of lazy evaluation, as described earlier. The `call_back` "after" advice simply forwards all notifications on to client components

without making any effort to retrieve data and compute the intention of the component. The `LocDisplayLazy` aspect, from the above figure, manifests the type of code that is expected to be actually generated by the aspect code generator. This code is very easy to generate. In fact, to synthesize the `LocDisplayLazy` aspect, all that is needed is the name of the class and the type of eager/lazy evaluation to weave. The code generator produces the concretized pointcuts that are needed to accomplish the weaving of the lazy evaluation concern with the `LocDisplay` component.

Adaptation in BBN's UAV Prototype

The ability to adapt is an essential trait for Distributed Object Computing (DOC) middleware solutions. In real-time embedded systems, the presence of Quality of Service (QoS) requirements demands that a system be able to adjust, in a timely manner, to changes imposed from the external environment. To provide adaptability within distributed real-time systems, there are three things that must be present: 1) the ability to express QoS requirements, in some form 2) a mechanism to monitor important conditions that are associated with the environment, and 3) a causal relation between the monitoring of the environment and the specification of the QoS requirements in such a way that there is a noticeable change in the behavior of the system as it adapts [Karr et al., 2001].

In addition to Boeing's BoldStroke framework, another Open Experimental Platform (OEP) in the DARPA PCES project has been managed by BBN. This project is a prototype application for an Unmanned Aerial Vehicle (UAV). A UAV is an aircraft that performs surveillance over dangerous terrain and hostile territories. The UAV streams video back to a central distributor that forwards the video on to several different

displays [Loyall et al., 2001]. The essence of this project is pictorially shown in Figure 56. The feedback cycle for utilizing the UAV as a surveillance device (with respect to the individual stages highlighted on the figure below) is: 1) video from the UAV is sent to the distributor that is located on a sea vessel, 2) the distributor broadcasts the video to numerous video display hosts on board the ship, 3) the video is received by each host and displayed to various operators, and 4) each operator at a display observes the video and sends commands, when deemed necessary, to control the UAV [Karr et al, 2001].

In the presence of changing conditions in the environment, the fidelity of the video stream must be maintained according to specified QoS parameters. The video must not be stale, or be affected by jittering, to the point that the operator cannot make an informed decision. Within the BBN implementation, a *contract* assists the system developer in specifying QoS requirements that are expected by a client and provided by a supplier. Each contract describes operating regions and actions that are to be taken when QoS measurements change. A domain-specific language was developed to assist in the specification of contracts; the name of this DSL is the Contract Description Language (CDL). A code generator translates the CDL into code that is integrated within the runtime kernel of the application. The textual intention of a CDL specification is very similar to the semantics of a hierarchical state machine.

Typically, there arises one dimension of the contract that is treated as a dependent variable, with numerous other independent variables that are adjusted to adapt the dependent variable according to some QoS requirement. For example, the end-to-end latency of the video stream distribution may be a dependent variable that drives the

adaptation of other independent variables (e.g., the size of a video frame, or even the video frame rate).

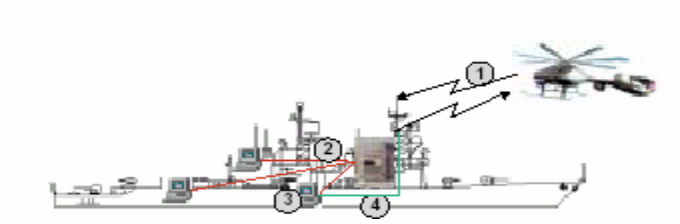


Figure 56: BBN UAV Example
(Reprinted from [Karr et al., 2001], with permission from BBN)

Weaving Across Finite State Machines

In consultation with the developers and users of CDL, it was believed that an approach toward contract synthesis (from models) would permit the creation of larger and more complex contracts. A GME paradigm has been created that synthesizes state-machine models into CDL contracts (Note: The paradigm and interpreter to generate CDL from a model was developed by Dr. Sandeep Neema, and is not a contribution of this dissertation). There also exists an interpreter to synthesize models into a Matlab simulation. Feedback from CDL developers and users has been very positive.

The weaver has also been applied to the BBN paradigm. Several strategies have been created to support the modeling of state machines that represent the behavior of a contract. The first strategy for this paradigm focused on issues related to the creation of state machines and their internal transitions.

The view of the model shown in Figure 57 pertains to the dataflow of the UAV prototype. The latency concern is the dependent variable in this case. It is represented here as a system condition object (the value of a system condition object is monitored from the environment). The latency is an input into a hierarchical state machine called

"Outer." Within Outer, there are state machines that describe the adaptation of identified independent control variables.

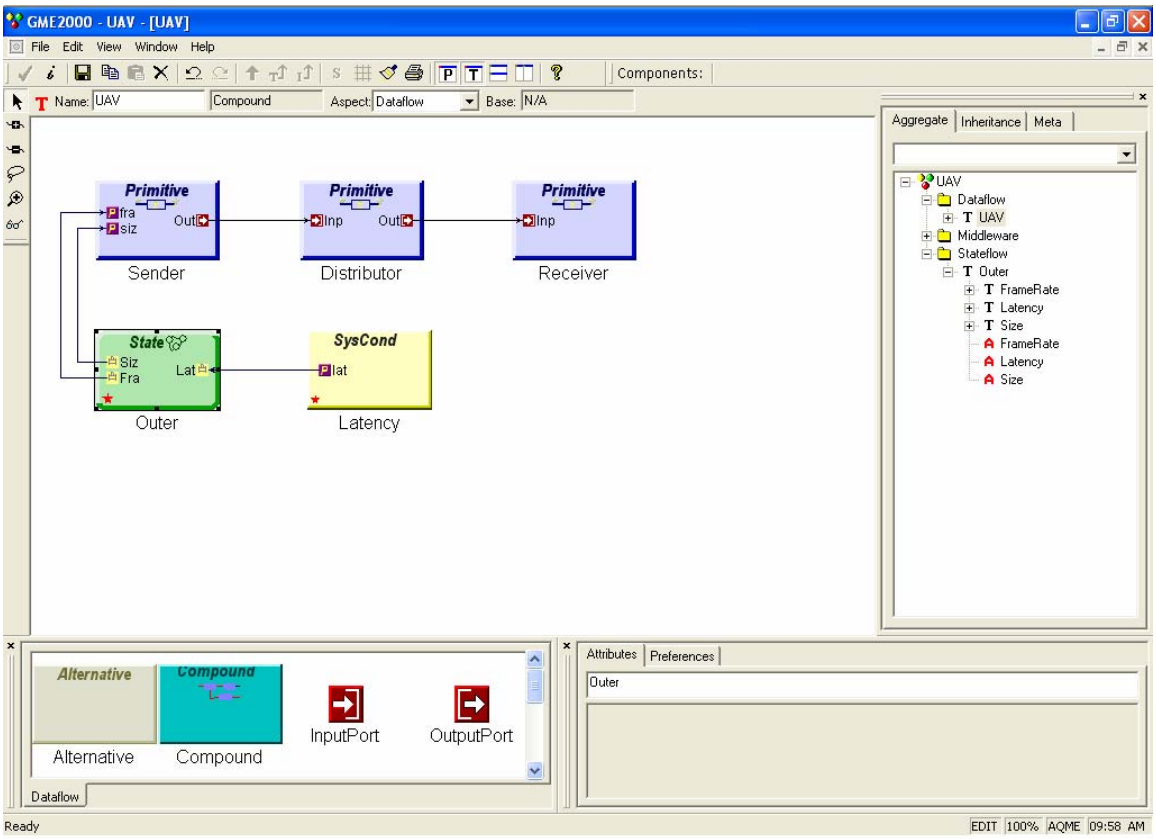
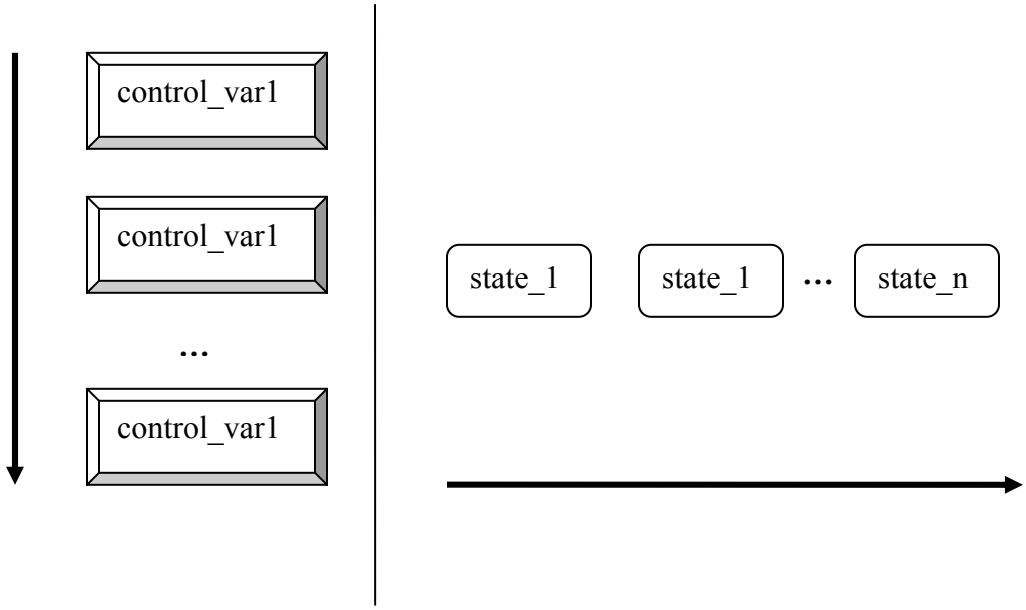


Figure 57: Dataflow for UAV Prototype

As depicted in Figure 58, there are two ways that a state machine model can be extended. Along one axis of extension, the addition of new dependent control variables often can offer more flexibility in adaptation toward the satisfaction of QoS parameters. It could be the case that other variables (e.g., color, video format, compression) would help in reducing the latency. Figure 58a captures the intent of this extension through the introduction of new control variables. It may also be the case that, within a particular

state, finer granularity of the intermediate transitions would permit better adaptation to QoS requirements. Figure 58b captures the intent of this extension.



a) Adding New Control Variables b) Adding More Intermediate Transitions in States
Figure 58: Axes of Variation within a State Machine

The internal details of Outer can be viewed in Figure 59. The three substates within Outer were actually created from a strategy. The strategy that generated these states is not shown in this appendix, but that strategy was implemented in order to support the two axes of extension shown in Figure 58.

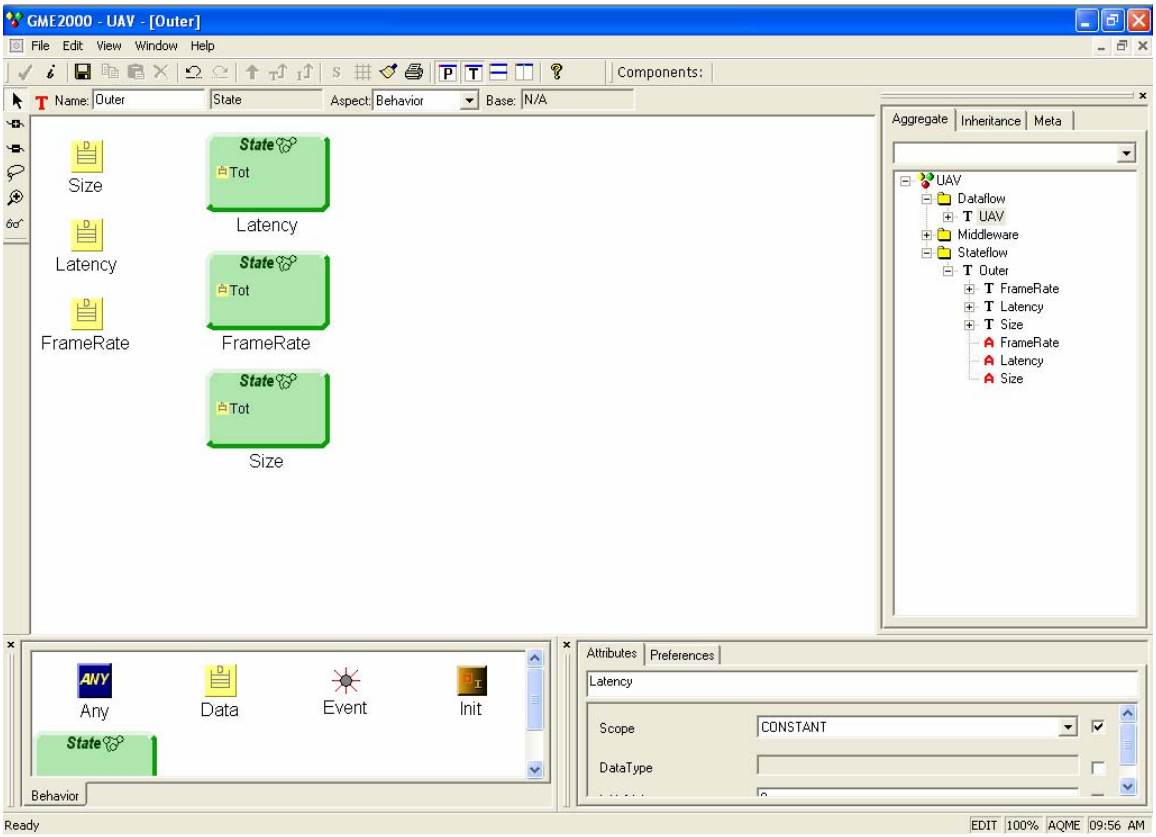
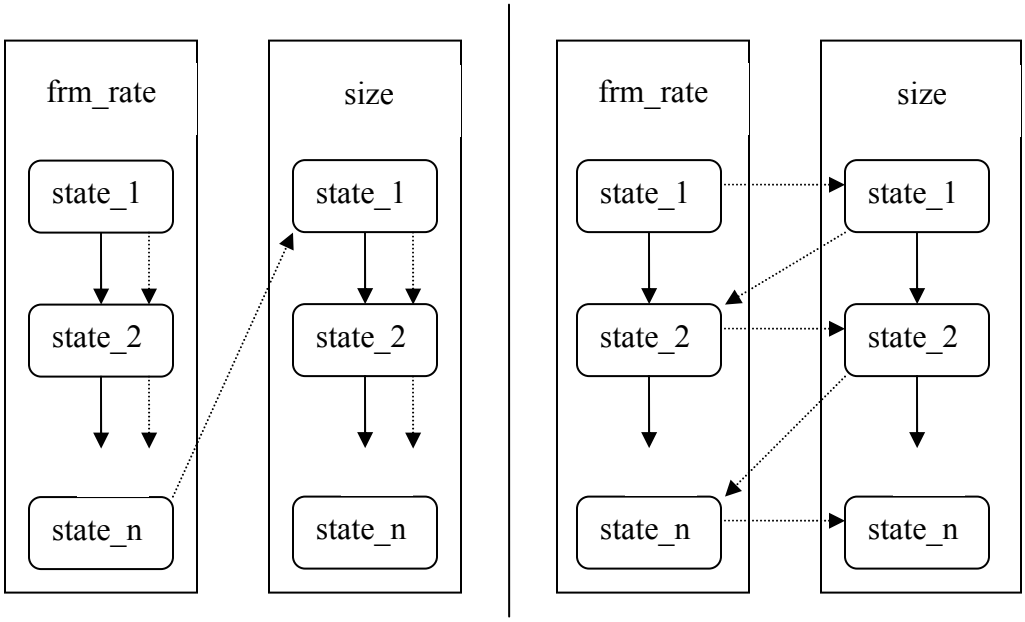


Figure 59: Top-Most View of Parallel State Machine

In addition to the strategy for creating control variables (and their intermediate states), an additional strategy was written to provide assistance in changing the adaptive protocol that spans across each state machine. There could be numerous protocols possible for adapting a system to meet QoS requirements. Two possibilities are given in Figure 60. The realization that each of these protocols is scattered across the boundaries of each participating state machine suggests that these protocols represent a type of crosscutting concern.

The left-hand side of the figure below specifies a protocol that exhausts the effect of one independent variable (frm_rate) before attempting to adjust another independent

variable (size). The semantics of this protocol pertain to the exhaustive reduction of one variable before attempting to reduce another one. Thus, the size variable is of a higher priority in this case because it is not reduced until there is no further reduction possible to the frame rate. The dotted-arrow in this figure indicates the order in which the transitions fire based upon the predicate guards.



a) Priority Exhaustive b) Zig-zag

Figure 60: State Protocols for Adapting to Environment

The right-hand side of Figure 60 represents a more equitable strategy for maintaining the latency QoS requirement. In this protocol, a zig-zag pattern suggests that the reduction of a variable is staggered with the reduction of a peer variable. Observe that the figure above involves only two control variables. The ability to change the protocol (by hand) becomes complicated when many variables are involved, or when there are numerous intermediate states. This crosscutting nature suggests that a strategy would be

beneficial. Figure 61 contains a strategy that supports the protocol highlighted above in Figure 60a.

```
defines AddTransition, FindConnectingState, ApplyTransitions;

strategy AddTransition(stateName, prevID, guard : string;
                       prevPri : integer)
{

  declare pri, minVal, maxVal, avgVal : integer;
  declare endID : string;
  declare aConnection : node;

  findAtom("Priority").findAttributeNode("InitialValue").getInt(pri);

  if(pri == prevPri + 1) then

    getID(endID);
    findAtom("Min").findAttributeNode("InitialValue").getInt(minVal);
    findAtom("Max").findAttributeNode("InitialValue").getInt(maxVal);

    avgVal := (minVal + maxVal) / 2;
    <<CComBSTR action(stateName);
      action.Append("="+XMLParser::itos(avgVal)); >>

    aConnection :=
      parent().addConnection("Transition", "Transition", "Transition",
                             endID, prevID);

    aConnection.addAttribute("Guard", guard);
    aConnection.addAttribute("Action", action);

  endif;

}
```

Figure 61: Latency Adaptation Transition Strategy

```
strategy FindConnectingState(stateName, guard : string)
{

  declare pri : integer;
  declare startID : string;

  findAtom("Priority").findAttributeNode("InitialValue").getInt(pri);
  getID(startID);

  if(pri < 4) then

    parent().models("State")->
                forAll(AddTransition(stateName, startID, guard, pri));

  endif;

}


strategy ApplyTransitions(stateName, guard : string)
{

  declare theModel : node;

  theModel := findModel(stateName);
  theModel.models("State")->forAll(FindConnectingState(stateName,
                                                       guard));

}
```

Figure 61 (cont): Latency Adaptation Transition Strategy

There may be several different variables that can be the focus of adaptation, depending on the contract and goals of an application. In this particular scenario, a smaller frame rate is tolerated in order to maintain a desired latency. The adaptation strategy just presented was used to produce the internal view of the "size" state, shown in Figure 62. Each of the states progressively reduces the size of the video frame. The guard condition for the selected transition appears in the lower-right hand side of the figure.

The guard condition states that the transition fires when the latency is not at the desired level, and also when the frame rate has been reduced to its smallest possible size.

In order to keep the specification of the strategy to a minimum, transitions that adapt to improved latency were not provided. These would permit the size and frm_rate states to improve the values of their respective variables whenever the latency improved.

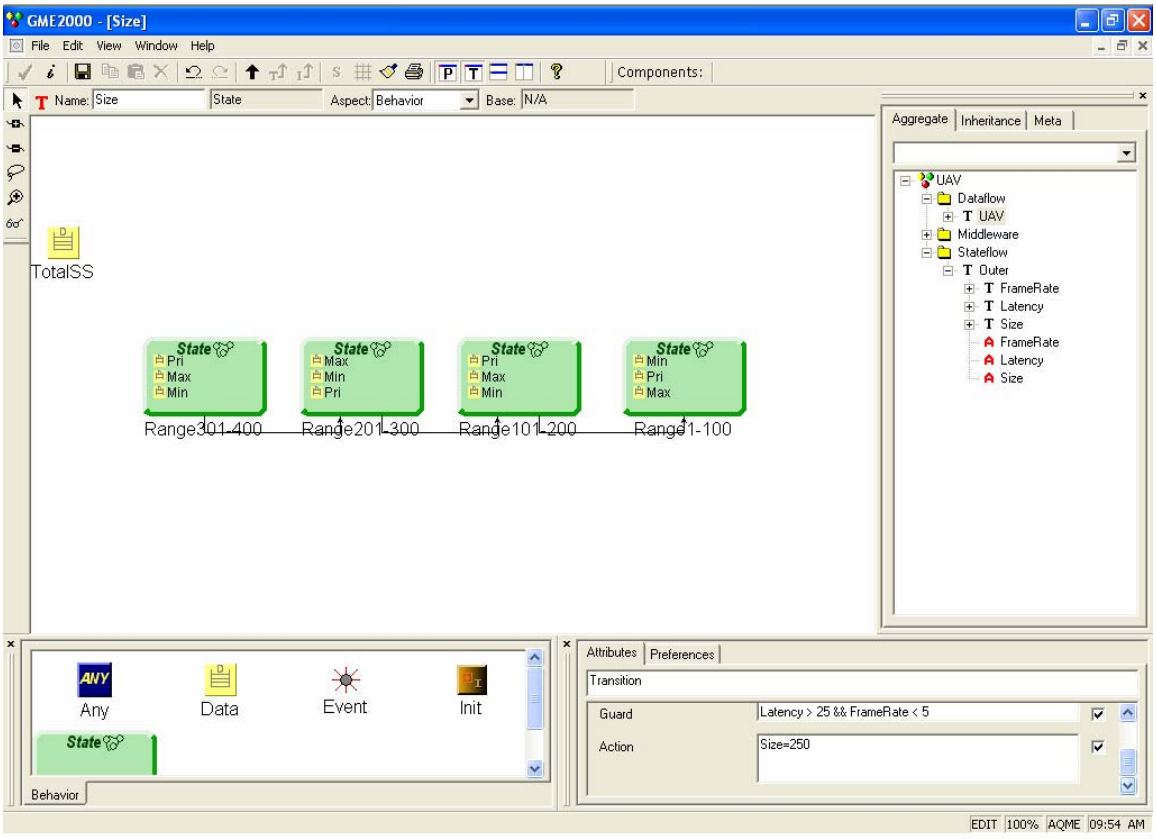


Figure 62: Internal Transitions within the Size State

REFERENCES

[Abelson and Sussman, 1996] Harold Abelson and Gerald Jay Sussman (with Julie Sussman), *Structure and Interpretation of Computer Programs*, MIT Press, 1996.

[Aho et al., 1986] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[Aksit et al., 1992] Mehmet Aksit, Lodewijk Bergmans, and S. Vural, "An Object-Oriented Language-Database Integration Model: The Composition Filters Approach," *European Conference on Object-Oriented Programming (ECOOP)*, LNCS 615, Springer-Verlag, Utrecht, The Netherlands, June/July 1992, pp. 372-395.

[Aksit et al., 1994] Mehmet Aksit, Jan Bosch, William van der Sterren, and Lodewijk Bergmans, "Real-Time Specification Inheritance Anomalies and Real-Time Filters," *European Conference on Object-Oriented Programming (ECOOP)*, LNCS 821, Springer-Verlag, Bologna, Italy, July 1994, pp. 386-407.

[Anderson and Hickey, 1999] Kenneth R. Anderson and Timothy J. Hickey, "Reflecting Java into Scheme," *Proceedings of Reflection '99: Metalevel Architectures and Reflection,* LNCS 1616, Springer-Verlag, Saint-Malo, France, July 1999, pp. 154-174.

[AOSD, 2002] http://aosd.net

[Astley et al., 2001] Mark Astley, Daniel Sturman, and Gul Agha, "Customizable Middleware for Modular Distributed Software," *Communications of the ACM*, May 2001, pp. 99-107.

[Aycock, 1998] John Aycock, "Compiling Little Languages in Python," *Proceedings of the 7th International Python Conference*, Houston, Texas, November 1998, pp. 69-77.

[Backus et al., 1957] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt, "The FORTRAN Automatic Coding System," *Western Joint Computer Conference*, 1957, pp. 188-198.

[Badros, 2000] Greg Badros, "JavaML: A Markup Language for Java Source Code," *Ninth International World Wide Web Conference*, Amsterdam, The Netherlands, May 2000, pp. 159-177.

[Barkmeyer and Lubell, 2001] Edward Barkmeyer and Joshua Lubell, "XML Representation of EXPRESS Models and Data," *ICSE Workshop on XML Technologies and Software Engineering*, Toronto, Ontario, Canada, May 2001.

[Barnes and Pandey, 1999] J. Fritz Barnes and Raju Pandey, "CacheL: Language Support for Customizable Caching Policies," *Proceedings of the Fourth International Web Caching Workshop*, San Diego, California, March 1999.

[Barstow, 1985] David Barstow, "Domain-Specific Automatic Programming," *IEEE Transactions on Software Engineering*, November 1985, pp. 1321-1336.

[Batory et al., 1994] Don Batory, Jeff Thomas, and Marty Sirkin, "Reengineering a Complex Application Using a Scalable Data Structure Compiler," *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, New Orleans, Louisiana, December 1994, pp. 111-120.

[Batory and Geraci, 1997] Don Batory and Bart J. Geraci, "Composition Validation and Subjectivity in GenVoca Generators," *IEEE Transactions on Software Engineerin*g, February 1997, pp. 67-82.

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

[Batory et al., 2000] Don Batory, Gang Chen, Eric Robertson, and Tao Wang, "Design Wizards and Visual Programming Environments for GenVoca Generators," *IEEE Transactions on Software Engineering*, May 2000, pp. 441-452.

[Baxter, 1992] Ira D. Baxter, "Design Maintenance Systems," *Communications of the ACM*, April 1992, pp. 73-89.

[Baxter, 2001] Ira D. Baxter, "DMS: A Tool for Automating Software Quality Enhancement," available at Semantic Designs (http://www.semdesigns.com), 2001.

[Bergmans and Aksit, 2001] Lodewijk Bergmans and Mehmet Aksit, "Composing Crosscutting Concerns using Composition Filters," *Communications of the ACM*, October 2001, pp. 51-57.

[Bentley, 1986] Jon Bentley, "Programming Pearls: Little Languages," *Communications of the ACM*, August 1986, pp. 711-721.

[Bettin, 2001] Joern Bettin, "A Language to Describe Software Texture in Abstract Design Models and Implementation," *OOPSLA Workshop on Domain-Specific Visual Languages*, Tampa, Florida, October 2001, pp. 1-10.

[Bézivin, 2001] Jean Bézivin, "From Object Composition to Model Transformation with the MDA," *Technology of Object-Oriented Languages and Systems (TOOLS),* Santa Barbara, California, August 2001.

[Bevington, 1997] David Bevington, editor, *The Complete Works of Shakespeare*, Addison-Wesley, 1997.

[Biggerstaff, 1998] Ted Biggerstaff, "A Perspective on Generative Reuse," *Annals of Software Engineering*, Vol. 5, 1998, pp. 169-226.

[Bloch, 2001] Joshua Bloch, *Effective Java Programming Language Guide*, Addison-Wesley, 2001.

[Bonachea et al., 1999] Dan Bonachea, Kathleen Fisher, Anne Rogers, and Frederick Smith, "Hancock: A Language for Processing Very Large-Scale Data," *USENIX Conference on Domain-Specific Languages*, Austin, Texas, October 1999, pp. 163-176.

[Booch et al., 1998] Grady Booch, Ivar Jacobson, James Rumbaugh, *The Unified Modeling Language User Guide*, Addison-Wesley, 1998.

[Booch, 2001] Grady Booch, "Through the Looking Glass," *Software Development Magazine*, July 2001, pp. 49-51.

[Bobrow et al., 1993] Daniel G. Bobrow, Richard Gabriel, and Jon L. White, "CLOS in Context: The Shape of the Design Space," A. Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*, 1993, pp. 29-61.

[Bracha and Cook, 1990] Gilad Bracha and William Cook, "Mixin-based Inheritance," *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA),* Ottawa, Canada, October 1990, pp. 308-311.

[Brooks, 1995] Fred Brooks, *The Mythical Man-Month*, Addison-Wesley, 1995.

[Bryant and Feldt, 2001] Avi Bryant and Robert Feldt, "AspectR - Simple Aspect-Oriented Programming in Ruby," http://aspectr.sourceforge.net

[Burns, 1786] Robert Burns, "To A Louse, On Seeing One on a Lady's Bonnet, at Church," *Poems, Chiefly in the Scottish Dialect*, Kilmarnock, 1786.

[Cardone, 1999] Richard Cardone, "On the Relationship of Aspect-Oriented Programming and GenVoca," *Workshop on Institutionalizing Software Reuse*, Austin, Texas, January 1999.

[Carley and Stewart, 2001] Thomas Carley and David Stewart, "Visual Aspect-Oriented Programming of Resource Constrained Real-Time Embedded Systems Using the Port-Based Object Model of Computation," *OOPSLA Workshop on Domain-Specific Visual Languages*, Tampa, Florida, October 2001, pp. 39-48.

[Carroll, 1872] Lewis Carroll, *Through the Looking Glass*, 1872.

[Chandy and Lamport, 1985] K. Mani Chandy and Leslie Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems*, February 1985, pp. 63-75.

[Chavez and de Lucena, 2001] Christina von Flach G. Chavez and Carlos J. P. de Lucena, "Design-level Support for Aspect-Oriented Software Development," *OOPSLA Workshop on Advanced Separation of Concerns*, Minneapolis, Minnesota, October 2001.

[Chiba and Masuda, 1993] Shigeru Chiba and Takashi Masuda, "Designing an Extensible Distributed Language with a Metalevel Architecture," *European Conference on Object-Oriented Programming (ECOOP)*, LNCS 707, Springer-Verlag, Kaiserslautern, Germany, July 1993, pp. 482-501.

[Chrysler and Escobar, 2000] John Chrysler and Thomas Escobar, *2000 Masonry Codes and Specifications*, CRC Press, 2000.

[Clarke, 2002] Siobhán Clarke, "Extending Standard UML with Model Composition Semantics," *Science of Computer Programming*, May 2002.

[Clarke et al., 1999] Siobhán 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 (OOPSLA)*, Denver, Colorado, November 1999, pp. 325-339.

[Clarke and Walker, 2001] Siobhán Clarke and Robert J. Walker, "Composition Patterns: An Approach to Designing Reusable Aspects," *International Conference on Software Engineering (ICSE)*, Toronto, Ontario, Canada, May 2001, pp. 5-14.

[Clauß, 2001] Matthias Clauß, "Generic Modeling Using UML Extensions for Variability," *OOPSLA Workshop on Domain-Specific Visual Languages*, Tampa, Florida, October 2001, pp. 11-18.

[Clavel, 2000] Manuel Clavel, *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*, CSLI Publications, 2000.

[Cleaveland, 1988] J. Craig Cleaveland, "Building Application Generators," *IEEE Software*, July 1988, pp. 25-33.

[Cleaveland, 2001] J. Craig Cleaveland, "Separating Concerns of Modeling from Artifact Generation Using XML," *OOPSLA Workshop on Domain-Specific Visual Languages*, Tampa, Florida, October 2001, pp. 83-86.

[Clements and Northrop, 2001] Paul Clements and Linda Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001.

[Coady et al., 2001a] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn, "Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code," *Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, Vienna, Austria, September 2001, pp. 78-88.

[Coady et al., 2001b] Yvonne Coady, Gregor Kiczales, Mike Feeley, Norm Hutchinson, and Joon Suan Ong, "Structuring Operating System Aspects," *Communications of the ACM*, October 2001, pp. 79-82.

[Constantinides et al., 2000] Constantinos Constantinides, Atef Bader, Tzilla Elrad, P. Netinant, and Mohamed Fayad, "Designing an Aspect-Oriented Framework in an Object-Oriented Environment," *ACM Computing Surveys*, March 2000.

[Covey, 1990] Stephen R. Covey, *The 7 Habits of Highly Effective People*, Simon & Schuster, 1990.

[Craig, 2000] Iain Craig, *The Interpretation of Object-Oriented Programming Languages*, Springer-Verlag, 2000.

[Czarnecki and Eisenecker, 1999] Krzysztof Czarnecki and Ulrich Eiseneker, "Components and Generative Programming," *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE '99),* Toulouse, France, September 1999, pp. 2-19.

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

[de Alwis, 2001] Brian de Alwis, "Apostle: Aspect Programming in Smalltalk," http://www.cs.ubc.ca/~bsd/apostle/

[de Moor et al., 1999] Oege de Moor, Simon Peyton-Jones, and Eric Van Wyk, "Aspect-Oriented Compilers," *In First International Symposium on Generative and Component-Based Software Engineering*, Erfurt, Germany, September 1999, pp. 121-133.

[De Volder and D'Hondt, 1999] Kris De Volder and Theo D'Hondt, "Aspect-Oriented Logic Meta Programming," *Proceedings of Reflection '99: Metalevel Architectures and Reflection,* LNCS 1616, Springer-Verlag, Saint-Malo, France, July 1999, pp. 250-272.

[Daft et al., 1987] Richard Daft, Kristen Skivington, and Mark Sharfman, *Cases and Applications: Organizational Theory*, West Wadsworth, 1987.

[Date, 1999] C.J. Date, *An Introduction to Database Systems*, Addison-Wesley, 1999.

[Delisle and Garlan, 1990] Norman Delisle and David Garlan, "Applying Formal Specification to Industrial Problems: A Specification of an Oscilloscope," *IEEE Software*, September 1990, pp. 29-36.

[Dessler, 1986] Gary Dessler, *Organization Theory: Integrating Structure and Behavior*, Prentice-Hall, 1986.

[Devanbu, 1999] Premkumar Devanbu, "GENOA – A Customizable, Front-end Retargetable Source Code Analysis Framework," *ACM Transactions on Software Engineering and Methodology*, April 1999, pp. 177-212.

[Dijkstra, 1968] Edsger Dijkstra, "Go To Statement Considered Harmful," letter to the editor, *Communications of the ACM*, March 1968, pp. 147-148.

[Dijkstra, 1976] Edsger Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.

[Dijkstra and Scholten, 1980] Edsger Dijkstra and C.S. Scholten, "Termination Detection for Diffusing Computations," *Information Processing Letters,* August 1980, pp. 1-4.

[DOME] http://www.htc.honeywell.com/dome/

[Einstein, 1950] Albert Einstein, *Out of My Later Years*, The Philosophical Library, 1950.

[Elrad et al., 2001] Tzilla Elrad, Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher, "Discussing Aspects of AOP," *Communications of the ACM*, October 2001, pp. 33-38.

[Faith, 1997] Rickard Edward Faith, *Debugging Programs After Structure-Changing Transformations*, Ph.D. Dissertation, Department of Computer Science, The University of North Carolina at Chapel Hill, 1997.

[Faith et al., 1997] Rickard E. Faith, Lars S. Nyland, and Jan F. Prins, "Khepera: A System for Rapid Implementation of Domain-Specific Languages," *USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997, pp. 243-255.

[Fayad et al., 1999] Mohamed Fayad, Douglas Schmidt, and Ralph Johnson, *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, John Wiley and Sons, 1999.

[Fayad, 2000] Mohamed Fayad, "Introduction to the Computing Surveys' Electronic Symposium on Object-Oriented Application Frameworks," *ACM Computing Surveys*, March 2000.

[Fernández et al., 1999] Mary Fernández, Dan Suciu, and Igor Tatarinov, "Declarative Specification of Data-intensive Web Sites," *USENIX Conference on Domain-Specific Languages*, Austin, Texas, October 1999, pp. 135-148.

[Filman and Friedman, 2000] Robert Filman and Dan Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness," *OOPSLA Workshop on Advanced Separation of Concerns*, Minneapolis, Minnesota, October 2000.

[Filman, 2001] Robert Filman, "What is Aspect-Oriented Programming, Revisited," *ICSE Workshop on Advanced Separation of Concerns*, Toronto, Ontario, Canada, May 2001.

[Filman et al., 2002] Robert Fillman, Stuart Barrett, Diana Lee, and Ted Linden, "Inserting Ilities by Controlling Communications," *Communications of the ACM*, January 2002, pp. 116-122.

[Floyd, 1979] Robert W. Floyd, "The Paradigms of Programming," *Communications of the ACM*, August 1979, pp. 455-460.

[Forman and Danforth, 1999] Ira R. Forman and Scott H. Danforth, *Putting Metaclasses to Work*, Addison-Wesley, 1999.

[Fuller, 1981] Buckminister Fuller, *Critical Path*, St. Martin's Press, 1981.

[Gabriel, 1995] Richard P. Gabriel, "The Column Without a Name: Software Development as Science, Art and Engineering," *C++ Report*, July/August 1995.

[Gabriel and Goldman, 2000] Richard P. Gabriel and Ron Goldman, "Mob Software: The Erotic Life of Code," *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Keynote Address, Minneapolis, Minnesota, October 19, 2000.

[Gagnon, 1998] Etienne Gagnon, "SableCC: An Object-Oriented Compiler Framework," Master's Thesis, School of Computer Science, McGill University, Montreal, March 1998.

[Gal et al., 2002] Andreas Gal, Wolfgang Schröder-Preikschat, and Olaf Spinczyk, "On Aspect-Orientation in Distributed Real-Time Dependable Systems," *IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, San Diego, Califorinia, January 2002.

[Gamma et al., 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Georgescu, 2002] Cristian Georgescu, "Code Generation Templates Using XML and XSL," *C/C++ Users Journal*, January 2002, pp. 6-19.

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

[Gianpaolo et al., 1998] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta, "Exploiting an Event-Based Infrastructure to Develop Complex Distributed Systems," *International Conference on Software Engineering (ICSE)*, Kyoto, Japan, April 1998, pp. 261-270.

[Gogolla, 2001] Martin Gogolla, "Using OCL for Defining Precise, Domain-Specific UML Stereotypes," *6th Australian Workshop on Requirements Engineering (AWRE),* Sydney, Australia, November 2001, pp. 51-60.

[Gray and Schach, 2000] Jeff Gray and Steve Schach, "Constraint Animation Using an Object-Oriented Declarative Language," *Proceedings of the 38th Annual ACM SE Conference*, Clemson, South Carolina, April 2000, pp. 1-10.

[Gray et al., 2000] Jeff Gray, Ted Bapty, and Sandeep Neema, "Aspectifying Constraints in Model-Integrated Computing," *OOPSLA Workshop on Advanced Separation of Concerns*, Minneapolis, Minnesota, October 2000.

[Gray, 2001a] Jeff Gray, "Using Software Component Generators to Construct a Metaweaver Framework," *International Conference on Software Engineering (ICSE)*, Toronto, Ontario, Canada, May 2001, pp. 789-790.

[Gray, 2001b] Jeff Gray, "A Framework for Creating Aspect Weavers," *Doctoral Symposium: OOPSLA '01 Companion to Proceedings*, Tampa, Florida, October 2001.

[Gray et al., 2001a] Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck, "Handling Crosscutting Constraints in Domain-Specific Modeling," *Communications of the ACM*, October 2001, pp. 87-93.

[Gray et al., 2001b] Jeff Gray, Ted Bapty, and Sandeep Neema, "An Example of Constraint Weaving in Domain-Specific Modeling," *OOPSLA Workshop on Domain-Specific Visual Languages*, Tampa, Florida, October 2001, pp. 49-56.

[Griswold et al., 2001] William G. Griswold, Jimmy J. Yuan, and Yoshikiyo Kato, "Exploiting the Map Metaphor in a Tool for Software Evolution," *International Conference on Software Engineering (ICSE)*, Toronto, Ontario, Canada, May 2001, pp. 265-274.

[Grundy, 2000] John Grundy, "Multi-Perspective Specification, Design and Implementation of Software Components Using Aspects," *International Journal of Software and Knowledge Engineering*, December 2000, pp. 713-734.

[Gudmundson and Kiczales, 2001] Stephan Gudmundson and Gregor Kiczales, "Addressing Practical Software Development Issues in AspectJ with a Pointcut Interface," *ECOOP Workshop on Advanced Separation of Concerns*, Budapest, Hungary, June 2001.

[Gulwani et al., 2001] Sumit Gulwani, Aasha Tarachandani, Deepak Gupta, Dheeraj Sanghi, Luciano Barreto, Gilles Muller, and Charles Consel, "WebCaL - A Domain-Specific Language for Web Caching," *Computer Communications*, February 2001, pp. 191-201

[Hall, 1998] Richard Hall, *Organizations: Structure, Process, and Outcomes*, Prentice-Hall, 1998.

[Hannemann and Kiczales, 2001] Jan Hannemann and Gregor Kiczales, "Overcoming the Prevalent Decomposition in Legacy Code," *ICSE Workshop on Advanced Separation of Concerns*, Toronto, Ontario, Canada, May 2001.

[Harrison and Ossher, 1990] William Harrison and Harold Ossher, "Subdivided Procedures: A Language Extension Supporting Extensible Programming," *International Conference on Computer Languages*, New Orleans, Louisiana, March 1990, pp. 190-197.

[Harrison et al., 1997] Timothy Harrison, David Levine, and Douglas C. Schmidt, "The Design and Performance of a Real-Time CORBA Event Service," *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Atlanta, Georgia, October 1997, pp. 184-200.

[Herndon and Berzins, 1988] Robert M. Herndon and Valdis Berzins, "The Realizable Benefits of a Language Prototyping Language," *IEEE Transactions on Software Engineering*, June 1988, pp. 803-809.

[Herrero et al., 2000] Jose Herrero, Fernando Sanchez, Fabiola Lucio, and Migeul Torro, "Introducing Separation of Aspects at Design Time*," ECOOP Workshop on Aspects and Dimensions of Concerns*, Cannes, France, June 2000.

[Hilliard, 1999] Rich Hilliard, "Views and Viewpoints in Software Systems Architecture," *First Working IFIP Conference on Software Architecture*, San Antonio, Texas, 1999.

[Hilsdale et al., 2001] Erik Hilsdale, Gregor Kiczales, Bill Griswold, Wes Isberg, Mik Kersten, and Jeffrey Palm, "Tutorial: Aspect-Oriented Programming with AspectJ," *Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, Vienna, Austria, September 2001.

[Hirschfield, 2001] Robert Hirschfield, "AspectS – AOP with Squeak," *OOPSLA Workshop on Advanced Separation of Concerns*, Tampa, Florida, October 2001.

[Ho et al., 2002] Wai-Meng Ho, Jean-Marc Jezequel, Francois Pennaneac'h, and Noel Plouzeau, "A Toolkit for Weaving Aspect-Oriented UML Designs," *First International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, April 2002.

[Hobatr and Malloy, 2001] Chanika Hobatr and Brian Malloy, "The Design of an OCL Query-Based Debugger for C++," *ACM Symposium on Applied Computing (SAC)*, Las Vegas, Nevada, March 2001, pp. 658-662.

[Hoffman and Weiss, 2001] Daniel Hoffman and David Weiss, editors, *Software Fundamentals – Collected Papers by David L. Parnas*, Addison-Wesley, 2001.

[Hoftstadter, 1979] Douglas R. Hofstadter, *Gödel, Escher, Bach*, Random House, 1979.

[Horowitz et al., 1985] Ellis Horowitz, Alfons Kemper, and Balaji Narasimhan, "A Survey of Application Generators," *IEEE Software*, January 1985, pp. 40-54.

[Hunleth et al., 2001] Frank Hunleth, Ron Cytron, and Chris Gill, "Building Customized Middleware Using Aspect-Oriented Programming," *OOPSLA Workshop on Advanced Separation of Concerns*, Tampa, Florida, October 2001.

[Hunt and Thomas, 2000] Andrew Hunt and David Thomas, *The Pragmatic Programmer*, Addison-Wesley, 2000.

[Hunt and Thomas, 2002] Andy Hunt and Dave Thomas, "Software Archaeology," *IEEE Software*, March/April 2002, pp. 20-22.

[Hußman et al., 2000] Heinrich Hußman, Brigit Demuth, and Frank Finger, "Modular Architecture for a Toolset Supporting OCL," *UML 2000: Advancing the Standard*, LNCS 1939, Springer-Verlag, York, UK, October 2000, pp. 278-293.

[IEEE 1471, 2000] *IEEE Standard 1471-2000: Recommended Practice for Architectural Description of Software-Intensive Systems*, The Institute for Electrical and Electronics Engineers, Inc., October 2000.

[Jackson, 1990] Michael Jackson, "Some Complexities in Computer-Based Systems and Their Implications for System Development," *International Conference on Computer Systems and Software Engineering*, Tel-Aviv, Israel, May 1990, pp. 344-351.

[Johnson, 1997] Ralph E. Johnson, "Frameworks = (Components + Patterns)," *Communications of the ACM*, October 1997, pp. 39-42.

[Karr et al., 2001] David Karr, Craig Rodrigues, Joseph Loyall, Richard Schantz, Yamuna Krishnamurthy, Irfan Pyarali, and Douglas Schmidt, "Application of the QuO Quality-of-Service Framework to a Distributed Video Application," *International Symposium on Distributed Objects and Applications*, Rome, Italy, September 2001.

[Karsai, 1995] Gábor Karsai, "A Configurable Visual Programming Environment: A Tool for Domain-Specific Programming," *IEEE Computer*, March 1995, pp. 36-44.

[Karsai et al., 1997] Gábor Karsai, Janos Sztipanovits, Ákos Lédeczi, and Michael Moore, "Model-Integrated System Development: Models, Architecture, and Process," *21st International Computer Software and Application Conference (COMPSAC),* Bethesda, Maryland, August 1997, pp. 176-181.

[Karsai, 1999] Gábor Karsai, "Structured Specification of Model Interpreters," *International Conference on Engineering of Computer-Based Systems (ECBS)*, Nashville, Tennessee, March 1999, pp. 84-90.

[Karsai, 2000] Gábor Karsai, "Why XSL is Not Suitable for Semantic Translation," ISIS Technical Research Note, April 2000.

[Karsai and Gray, 2000] Gábor Karsai and Jeff Gray, "Component Generation Technology for Semantic Tool Integration," *IEEE Aerospace Conference*, Big Sky, Montana, March 2000.

[Katz and Gil, 1999] Shmuel Katz and Joseph Gil, "Aspects and Superimpositions," *ECOOP Workshop on Aspect-Oriented Programming*, Lisbon, Portugal, June 1999.

[Kersten and Murphy, 1999] Mik Kersten and Gail C. Murphy, "Atlas: A Case Study in Building a Web-based Learning Environment Using Aspect-Oriented Programming," *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Denver, Colorado, November 1999, pp. 340-352.

[Kiczales et al., 1991] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

[Kiczales, 1992] Gregor Kiczales, "Towards a New Model of Abstraction in the Engineering of Software," *Proceedings of the International Workshop on New Models for Software Architectures (IMSA): Reflection and Metalevel Architecture*, Tokyo, Japan, November 1992, pp. 1-11.

[Kiczales et al., 1992] Gregor Kiczales, John Lamping, Luis H. Rodriguez Jr., and Erik Ruf, "Macros that Reach Out and Touch Somewhere," Internal Technical Report, Embedded Computation Area, Xerox PARC, 1992.

[Kiczales et al., 1993] Gregor Kiczales, J. Michael Ashley, Luis Rodriguez, Amin Vahdat, and Daniel G. Bobrow, "Metaobject Protocols: Why We Want Them and What Else Can They Do?" A. Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*, 1993, pp. 101-118.

[Kiczales, 1996] Gregor Kiczales, "Beyond the Black Box: Open Implementation," *IEEE Software*, January 1996, pp. 8-11.

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

[Kiczales, 2001] Gregor Kiczales, "Aspect-Oriented Programming: The Fun Has Just Begun," *Software Design and Productivity Coordinating Group – Workshop on New Visions for Software Design and Productivity: Research and Applications*, Nashville, Tennessee, December 2001.

[Kiczales et al., 2001a] Gregor Kiczales, Eric Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold, "An Overview of AspectJ," *European Conference on Object-Oriented Programming (ECOOP)*, LNCS 2072, Springer-Verlag, Budapest, Hungary, June 2001, pp. 327-353.

[Kiczales et al., 2001b] Gregor Kiczales, Eric Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold, "Getting Started with AspectJ," *Communications of the ACM*, October 2001, pp. 59-65.

[Kieburtz et al., 1996] Richard B. Kieburtz, Laura McKinney, Jeffrey M. Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino P. Oliva, Tim Sheard, Ira Smith, and Lisa Walton, "A Software Engineering Experiment in Software Component Generation," *International Conference on Software Engineering (ICSE)*, Berlin, Germany, March 1996, pp. 542-552.

[Kircher, 2002] Michael Kircher, "Eager Evaluation," *European Conference on Pattern Languages of Programs*, Kloster Irsee, Germany, July 2002.

[Knuth, 1984] Donald Knuth, "Literate Programming," *The Computer Journal*, May 1984, pp. 97-111.

[Lam, 2002] John Lam, "Cross-Language Load-Time Aspect Weaving on Microsoft's Common Language Runtime," Demonstration, *First International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, April 2002.

[Lämmel and Verhoef, 2001] Ralf Lämmel and Chris Verhoef, "Cracking the 500-Language Problem," *IEEE Software*, November/ December 2001, pp. 78-88.

[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*, November 2001, pp. 44-51.

[Lee and Zachary, 1995] Arthur H. Lee and Joseph L. Zachary, "Reflections on Metaprogramming," *IEEE Transactions on Software Engineering*, November 1995, pp. 883-893.

[Lee, 2001] Edward Lee, "Overview of the Ptolemy Project," Technical Memorandum UCB/ERL M01/11, March 6, 2001.

[Lego, 2002] http://www.lego.com/eng/info/profile.asp

[Lewis, 1967] C.S. Lewis, *Studies in Words*, 2nd ed., Cambridge University Press, Cambridge, England, 1967.

[Lewis, 1995] Ted Lewis, ed., *Object-Oriented Application Frameworks*, Manning Publications, 1995.

[Lieberherr and Holland, 1989] Karl Lieberherr and Ian Holland, "Assuring Good Style for Object-Oriented Programs," *IEEE Software*, September 1989, pp. 38-48.

[Lieberherr, 1996] Karl Lieberherr, *Adaptive Object-Oriented Software*, International Thomson Publishing, 1996.

[Lieberherr et al., 1999] Karl Lieberherr, David Lorenz, and Mira Mezini, "Programming with Aspectual Components," NU-CCS-99-01, College of Computer Science, Northeastern University, March 1999.

[Lieberherr et al., 2001] Karl Lieberherr, Doug Orleans, and Johan Ovlinger, "Aspect-Oriented Programming with Adaptive Methods," *Communications of the ACM*, October 2001, pp. 39-41.

[Lieberman, 1986] Henry Lieberman, "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems," *Object-Oriented Programming Systems, Languages and Applications (OOPSLA),* Portland, Oregon, November 1986, pp. 214-223.

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

[Long et al., 1998] Earl Long, Amit Misra, and Janos Sztipanovits, "Increasing Productivity at Saturn," *IEEE Computer*, August 1998, pp. 35-43.

[Lopes, 1997] Cristina Lopes, *D: A Language Framework for Distributed Programming*, Ph.D. Dissertation, College of Computer Science, Northeastern University, November 1997.

[Loyall et al., 2001] Joseph Loyall, Richard Schantz, John Zinky, Partha Pal, Richard Shapiro, Craig Rodrigues, Michael Atighetchi, David Karr, Jeanna Gossett, and Christopher Gill, "Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications," *IEEE International Conference on Distributed Computing Systems (ICDCS-21)*, Phoenix, Arizona, April 2001.

[Maes, 1987] Pattie Maes, "Concepts and Experiments in Computational Reflection," *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Orlando, Florida, December 1987, pp. 147-155.

[Maes, 1988] Pattie Maes, "Issues In Computational Reflection," P. Maes and D. Nardi, editors, *Metalevel Architectures and Reflection*, Elsevier Science, 1988, pp. 21-35.

[Maguire, 1994] Steve Maguire, *Debugging the Development Process*, Microsoft Press, 1994.

[Mahrenholz, 2002] Daniel Mahrenholz, Olaf Spinczyk, and Wolfgang Schröder-Preikschat, "Program Instrumentation for Debugging and Monitoring with AspectC++," *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Washington, DC, April 2002.

[Maroti et al., 2002] Miklos Maroti, Ákos Lédeczi, Arpad Bakay, Jeff Gray, and Gábor Karsai, "Type Hierarchies in Modeling and Metamodeling Languages," in preparation, 2002.

[Meyer, 1997] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall, New Jersey, 1997.

[Meyer, 2000] Erica Meyer, *Cascading Style Sheets: The Definitive Guide*, O'Reilly & Associates, 2000.

[Michels, 1915] Robert Michels, *Political Parties: The Sociological Study of the Oligarchical Tendencies of Modern Democracy*, translated by Eden and Cedar Paul, Batoche Books, 1915.

[Miller, 2001] Sandra Kay Miller, "Aspect-Oriented Programming Takes Aim at Software Complexity," *IEEE Computer*, April 2001, pp. 18-21.

[Moore et al., 2000] Michael Moore, Saeed Monemi, and Jianfeng Wang, "Integrating Information Systems In Electric Utilities," *IEEE International Conference on Systems, Man, and Cybernetics*, Nashville, Tennessee, October 2000.

[Murphy et al., 1999] Gail C. Murphy, Robert J. Walker, and Elisa L.A. Baniassad, "Evaluating Emerging Software Development Technologies: Lessons Learned from Assessing Aspect-Oriented Programming," *IEEE Transactions on Software Engineering*, July/August 1999, pp. 438-455.

[Murphy et al., 2001] Gail C. Murphy, Albert Lai, Robert J. Walker, and Martin P. Robillard, "Separating Features in Source Code: An Exploratory Study," *International Conference on Software Engineering (ICSE)*, Toronto, Ontario, Canada, May 2001, pp. 275-284.

[Narasimhan et al., 1999] Priya Narasimhan, Louise Moser, and P.M. Melliar-Smith, "Using Interceptors to Enhance CORBA," *IEEE Computer*, July 1999, pp. 62-68.

[Neema, 2001] Sandeep Neema, *System Level Synthesis of Adaptive Computing Systems*, Ph.D. Dissertation, Vanderbilt University, Department of Electrical Engineering and Computer Science, May 2001.

[Neema and Lédeczi, 2001] Sandeep Neema and Ákos Lédeczi, "Constraint Guided Self-Adaptation," *International Workshop on Self-Adaptive Software*, Balatonfured, Hungary, May 2001.

[Nelson et al., 2001] Torsten Nelson, Donald Cowan, and Paulo Alencar, "Supporting Formal Verification of Crosscutting Concerns," *Reflection 2001: The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, LNCS 2192, Springer-Verlag, Kyoto, Japan, September 2001, pp. 153-169.

[Newton, 1676] Sir Isaac Newton, Letter to Robert Hooke, February 5, 1676.

[Nordberg, 2001] Martin Nordberg, "Aspect-Oriented Dependency Inversion," *OOPSLA Workshop on Advanced Separation of Concerns*, Tampa, Florida, October 2001.

[Nordstrom et al., 1999] Greg Nordstrom, Janos Sztipanovits, Gábor Karsai, and Ákos Lédeczi, "Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments," *International Conference on Engineering of Computer-Based Systems (ECBS)*, Nashville, Tennessee, April 1999, pp. 68-74.

[Nuseibeh et al., 1994] Basher Nuseibeh, Jeff Kramer, and Anthony Finkelstein, "A Framework for Expressing the Relationship Between Multiple Views in Requirements Specification," *IEEE Transactions on Software Engineering,* October 1994, pp. 760-773.

[Ossher et al., 1996] Harold Ossher, Matthew Kaplan, A. Katz, William Harrison, and Vincent Kruskal, "Specifying Subject-Oriented Composition," *Theory and Practice of Object Systems*, vol. 2(3), 1996, pp. 179-202.

[Ossher and Tarr, 2001] Harold Ossher and Peri Tarr, "Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software," *Communications of the ACM*, October 2001, pp. 43-50.

[Ovlinger and Wand, 1999] Johan Ovlinger and Mitchell Wand, "A Language for Specifying Recursive Traversals of Object Structures," *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Denver, Colorado, November 1999, pp. 70-81.

[Parnas, 1972] David Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, December 1972, pp. 1053-1058.

[Parnas, 1976] David Parnas, "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering*, January 1976, pp. 1-9.

[Parnas, 1999] Nancy Eickelman, "ACM Fellow: David Lorge Parnas," *ACM Software Engineering Notes*, May 1999, pp. 10-14.

[Parr, 1993] Terrence J. Parr, *Language Translation Using PCCTS and C++*, Automata Publishing Company, 1993.

[Partsch and Steinbrüggen, 1983] H. Partsch and R. Steinbrüggen, "Program Transformation Systems," *ACM Computing Surveys*, September 1993, pp. 199-236.

[Perlis, 1982] Alan Perlis, "Epigrams on Programming," *ACM SIGPLAN Notices*, September 1982, pp. 7-13.

[Perrow, 1986] Charles Perrow, *Complex Organizations: A Critical Essay*, McGraw-Hill, 1986.

[Polya, 1957] George Polya, *How to Solve It*, Princeton University Press, 1957.

[Pu et al., 1997] Calton Pu, Andrew Black, Crispin Cowan, and Jonathan Walpole, "Microlanguages for Operating System Specialization," *First ACM SIGPLAN Workshop on Domain-Specific Languages*, Paris, France, January 1997, pp. 49-57.

[Rao, 1991] Ramana Rao, "Implementational Reflection in Silica," *European Conference on Object-Oriented Programming (ECOOP)*, LNCS 512, Springer-Verlag, Geneva, Switzerland, July 1991, pp. 251-266.

[Rashid and Pulvermueller, 2000] Awais Rashid and Elke Pulvermueller, "From Object-Oriented to Aspect-Oriented Databases," *Proceedings of the 11th International Conference on Database and Expert Systems Applications*, September 2000, London, UK, pp. 125-134.

[Rashid, 2001] Awais Rashid, "A Hybrid Approach to Separation of Concerns: The Story of SADES," *Reflection 2001: The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, LNCS 2192, Springer-Verlag, Kyoto, Japan, September 2001, pp. 231-249.

[Rashid, 2002] Awais Rashid, "Weaving Aspects in a Persistent Environment," *ACM SIGPLAN Notices*, February 2002, pp. 36-44.

[Robertson and Brady, 1999] Paul Robertson and J. Michael Brady, "Adaptive Image Analysis for Aerial Surveillance," *IEEE Intelligent Systems*, May/June 1999, pp. 30-36.

[Robillard and Murphy, 2002] Martin Robillard and Gail Murphy, "Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies," *International Conference on Software Engineering (ICSE)*, Buenos Aires, Argentina, May 2002.

[Schach, 2002] Stephen R. Schach, *Object-Oriented and Classical Software Engineering,* 5th ed., McGraw-Hill, 2002.

[Schmidt et al., 2000] Doug Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, Wiley and Sons, 2000.

[Schonger et al., 2002] Stefan Schonger, Elke Pulvermueller, and Stefan Sarstedt, "Aspect-Oriented Programming and Component Weaving: Using XML Representations of Abstract Syntax Trees," *Second German Workshop on Aspect-Oriented Software Development*, Bonn, Germany, February 2002.

[Sharp, 1998] David Sharp, "Reducing Avionics Software Cost Through Component Based Product-Line Development," *Software Technology Conference*, Salt Lake City, Utah, April 1998.

[Shukla et al., 2002] Dharma Shukla, Simon Fell, and Chris Sells, "Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse," *MSDN Magazine*, March 2002, pp. 60-68.

[Siegel, 2000] Jon Siegel, *CORBA 3 Fundamentals and Programming*, John Wiley & Sons, 2000.

[Simon et al., 1950] Herbert Simon, Donald Smithburg, and Victor Thompson, *Public Administration*, Alfred Knopf Publishing, 1950.

[Simon, 1996] Herbert Simon, *The Sciences of the Artificial*, The MIT Press, 1996.

[Simonyi, 1996] Charles Simonyi, "Intentional Programming: Innovation in the Legacy Age," Presented at *IFIP WG 2.1*, June 1996.

[Simonyi, 2001] Charles Simonyi, "Intentional Programming: Asymptotic Fun?" *Software Design and Productivity Coordinating Group – Workshop on New Visions for Software Design and Productivity: Research and Applications*, Nashville, Tennessee, December 2001.

[Smaragdakis and Batory, 1997] Yannis Smaragdakis and Don Batory, "DiSTiL: A Transformation Library for Data Structures," *USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997, pp. 257-270

[Smaragdakis and Batory, 2000] Yannis Smaragdakis and Don Batory, "Application Generators," J. Webster (ed.), *Encyclopedia of Electrical and Electronics Engineering*, John Wiley and Sons, 2000.

[Smith, 1776] Adam Smith, *An Inquiry into the Nature and Causes of the Wealth of Nations*, republished in Edwin Cannan's annotated edition, 1904, Methuen and Co.; first edition, 1776.

[Smith, 1982] Brian Smith, "Reflection and Semantics in Procedural Languages," Technical Report 272, Massachusetts Institute of Technology, Laboratory for Computer Science, 1982.

[Sobel and Friedman, 1996] Jonathan M. Sobel and Daniel P. Friedman, "An Introduction to Reflection-Oriented Programming," *Reflection '96*, San Francisco, California, April 1996.

[Sommerville and Sawyer, 1997] Ian Sommerville and Peter Sawyer, "Viewpoints: Principles, Problems and a Practical Approach to Requirements Engineering," *Annals of Software Engineering*, March 1997, pp. 101-130.

[Steele and Sussman, 1978] Guy Lewis Steele, Jr., and Gerald Jay Sussman, "The Art of the Interpreter, or the Modularity Complex (Parts Zero, One, and Two)," MIT Artificial Intelligence Memo 453, May 1978.

[Steele, 1990] Guy L. Steele, Jr., *Common Lisp: The Language*, Digital Press, 1990.

[Steele, 1998] Guy Steele, "Growing a Language," *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Keynote Address, Vancouver, British Columbia, Canada, October 22, 1998.

[Stevens et al., 1972] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine, "Structured Design," *IBM Systems Journal*, Vol. 13(2), 1974, pp. 115-139.

[SUIF2, 2000] *The SUIF 2 Compiler System*, http://suif.stanford.edu/suif/suif2/

[Sullivan, 2001] Gregory T. Sullivan, "Aspect-Oriented Programming using Reflection and Metaobject Protocols," *Communications of the ACM*, October 2001, pp. 95-97.

[Sutton and Rouvellou, 2001] Stanley M. Sutton and Isabelle Rouvellou, "Issues in Design and Implementation of a Concern-Space Modeling Schema," *ICSE Workshop on Advanced Separation of Concerns*, Toronto, Ontario, Canada, May 2001.

[Sztipanovits and Karsai, 1997] Janos Sztipanovits and Gábor Karsai, "Model-Integrated Computing," *IEEE Computer*, April 1997, pp. 10-12.

[Sztipanovits et al., 1998] Janos Sztipanovits, Gábor Karsai, and Ted Bapty, "Self-Adaptive Software for Signal Processing," *Comunications of the ACM*, May 1998, pp. 66-73.

[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 (ICSE)*, Los Angeles, California, May 1999, pp. 107-119.

[Tekinerdogan, 2000] Bedir Tekinerdogan, *Synthesis-Based Software Architecture Design*, Ph.D. Dissertation, Department of Computer Science, University of Twente, 2000.

[Tidwell, 2001] Doug Tidwell, *XSLT*, O'Reilly and Associates, 2001.

[Tolvanen and Kelly, 2000] Juha-Pekka Tolvanen and Steve Kelly, "Visual Domain-Specific Modeling: Benefits and Experiences of Using metaCASE Tools," *ECOOP Workshop on Model Engineering,* Cannes, France, June 2000.

[Tristram, 2001] Claire Tristram, "The Technology Review Ten: Untangling Code," *MIT Technology Review*, January 2001.

[Tsay et al., 2000] Jeff Tsay, Christopher Highlands, and Edward Lee, "A Code Generation Framework for Java Component-Based Designs," *International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, San Jose, California, November 2000.

[Vahid and Givargis, 2001] Frank Vahid and Tony Givargis, "Platform Tuning for Embedded Systems Design," *IEEE Computer*, March 2001, pp. 112-114.

[van Deursen and Knit, 1997] Arie van Deursen and Paul Klint, "Little Languages: Little Maintenance?" *First ACM SIGPLAN Workshop on Domain-Specific Languages,* Paris, France, January 1997, pp. 109-127.

[van Deursen et al., 2000] Arie van Deursen, Paul Klint, and Joost Visser, "Domain-Specific Languages: An Annotated Bibliography," *ACM SIGPLAN Notices*, June 2000, pp. 26-36.

[Van Wyk, 2000] Eric Van Wyk, "Domain-Specific Meta Languages," *ACM Symposium on Applied Computing*, Como, Italy, March 2000, pp. 799-803.

[Viega and Voas, 2000] John Viega and Jeffrey Voas, "Can Aspect-Oriented Programming Lead to More Reliable Software?" *IEEE Software*, November/December 2000, pp. 19-21.

[Walker et al., 1999] Robert J. Walker, Elisa L.A. Baniassad, and Gail C Murphy, "An Initial Assessment of Aspect-Oriented Programming," *International Conference on Software Engineering (ICSE)*, Los Angeles, California, May 1999, pp. 120-130.

[Wang et al., 2000] Nanbor Wang, Doug Schmidt, and Carlos O'Ryan, "Overview of the CORBA Component Model," *Component-Based Software Engineering: Putting the Pieces Together*, George Heineman and William Councill, editors, Addison-Wesley, 2001.

[Wang et al., 1997] Daniel Wang, Andrew Appel, Jeff Korn, and Chris Serra, "The Zephyr Abstract Syntax Description Language," *USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997, pp. 213-228.

[Warmer and Kleppe, 1999] Jos Warmer and Anneke Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1999.

[Weber, 1946] Max Weber, "Bureaucracy," in Hans Gerth and C. Wright Mills, eds., *From Max Weber*, Oxford University Press, 1946.

[Wegner, 1976] Peter Wegner, "Programming Languages – The First 25 Years," *IEEE Transactions on Computers*, December 1976, pp. 1207-1225.

[Wirth, 1974] Niklaus Wirth, "On the Design of Programming Languages," *Proceedings of the IFIP Congress*, 1974, pp. 386-93.

[Wittgenstein, 1961] Ludwig Wittgenstein, *Tractatus Logico-Philosophicus*, translated by D.F. Pears and B.F. McGuinness, Routledge and Kegan Paul, 1961.

[Wulf and Shaw, 1973] William Wulf and Mary Shaw, "Global Variables Considered Harmful," *ACM SIGPLAN Notices*, February 1973, pp. 28-34.

[Zou and Kontogiannis, 2001] Ying Zou and Kostas Kontogiannis, "A Framework for Migrating Procedural Code to Object-Oriented Platforms," *Proceedings of the 8th Asia-Pacific Software Engineering Conference*, Macau SAR, China, December 2001.