DISSERTATION

IMPROVING SOFTWARE MAINTAINABILITY THROUGH ASPECTUALIZATION

Submitted by

Michael Mortensen

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2009

COLORADO STATE UNIVERSITY

May 19, 2009

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY MICHAEL MORTENSEN ENTITLED IMPROVING SOFTWARE MAINTAINABILITY THROUGH ASPECTUALIZATION BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work

[illegible]

Tom Chen, Ph.D., Committee Member

[illegible]

L. Darrell Whitley, Ph.D., Committee Member

Sudipto Ghosh

Sudipto Ghosh, Ph.D., Adviser

[illegible]

James M. Bieman, Ph.D., Co-Adviser

[illegible]

L. Darrell Whitley, Ph.D., Department Chair

ABSTRACT OF DISSERTATION

IMPROVING SOFTWARE MAINTAINABILITY THROUGH ASPECTUALIZATION

The primary claimed benefits of aspect-oriented programming (AOP) are that it improves the understandability and maintainability of software applications by modularizing cross-cutting concerns. Before there is widespread adoption of AOP, developers need further evidence of the actual benefits as well as costs. Applying AOP techniques to refactor legacy applications is one way to evaluate costs and benefits.

Aspect-based refactoring, called aspectualization, involves moving program code that implements cross-cutting concerns into aspects. Such refactoring can potentially improve the maintainability of legacy systems. Long compilation and weave times, and the lack of an appropriate testing methodology are two challenges to the aspectualization of large legacy systems. We propose an iterative test driven approach for creating and introducing aspects. The approach uses mock systems that enable aspect developers to quickly experiment with different pointcuts and advice, and reduce the compile and weave times. The approach also uses weave analysis, regression testing, and code coverage analysis to test the aspects. We developed several tools for unit and integration testing. We demonstrate the test driven approach in the context of large industrial C++ systems, and we provide guidelines for mock system creation.

This research examines the effects on maintainability of replacing cross-cutting concerns with aspects in three industrial applications. We study several revisions of each application, identifying cross-cutting concerns in the initial revision, and also cross-cutting concerns that are added in later revisions. Aspectualization improved maintainability by reducing code size and improving both change locality and concern diffusion. Costs include the effort required for application refactoring and aspect creation, as well as a small decrease in performance.

Michael Mortensen
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Summer 2009

ACKNOWLEDGEMENTS

I am thankful for my wife and five children, who have been patient and supportive throughout my studies at CSU. They have made many sacrifices and helped me keep moving forward. They have helped me to keep everything in perspective and to make time for fun family activities. I am grateful to my parents for instilling in me the importance of education, and am grateful to my wife's parents for their enthusiasm for my studies.

I am grateful to my employer, Hewlett-Packard, for supporting me as a part-time Ph.D. student. Many co-workers and managers have provided feedback, expressed interest, and been supportive of me throughout this process. A former manager, Alva Barney, supported my travel to and participation in conferences that helped me present and receive feedback for my initial research results. As a researcher who is interested in using AOP rather than creating a new aspect-oriented language, I want to thank Olaf Spinczyk, Matthias Urban, and Daniel Lohmann for creating and providing feedback on the AspectC++ language.

I have had excellent teachers and mentors at Colorado State University. I want to thank committee members Dr. Tom Chen and Dr. Darrell Whitley for their valuable feedback and suggestions. I am grateful to my co-advisor, Dr. James Bieman, for his guidance on this research, particularly his help with creating and evaluating a software study of this kind. I am deeply indebted to Dr. Sudipto Ghosh for his patience throughout the process: helping me refine my topic, sharpening the focus on the key technical areas for the research, and his guidance in performing and writing about the research.

DEDICATION

For my family.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Aspect-oriented programming (AOP) [40] provides a new construct, an aspect, to modularize scattered and tangled *cross-cutting* code. A concern is cross-cutting if it is called by or affects A cross-cutting concern is a feature, such as logging or synchronization, that is called by or affects many separate

Typically an aspect combines *advice* – functionality to be woven into primary code – and a *point-cut*, which identifies the locations in the primary code, called *joinpoints*, where the advice is executed. Advice can be specified as before, after, or around advice: *before advice* executes before the joinpoint, *after advice* executes after the joinpoint, and *around advice* executes instead of the joinpoint but can execute the original joinpoint. The aspects are *woven* into the primary code by a preprocessor, compiler, or run-time system.

Aspectualization is the process of refactoring an application by moving code that implements cross-cutting concerns into aspects. Aspectualization is intended to improve design structure by modularizing cross-cutting code concerns [32]. We propose a test driven approach to aspectualize legacy systems. The approach addresses the challenges of long compile and weave times and lack of systematic testing techniques. We use mock systems to quickly prototype and validate aspects before weaving the aspects into a large system. Our approach tests aspects both at the unit level (with mock systems) and the integration level (with the whole system). We analyze the weaving process and apply coverage criteria to test the aspects, and use existing system regression tests for integration testing. Our tools leverage information generated during weaving to automate this approach. We demonstrate the approach with examples of aspectualizing large C++ systems using AspectC++ [1].

Aspectualization is intended to benefit the understandability and maintainability of the application. The primary goal of our study is to evaluate the effects of aspects on program maintainability. A secondary goal is to measure other costs of adopting AOP, including performance, testability, and defect introduction. A key goal of our study is to evaluate the effectiveness of using mock systems as a test-driven approach for aspectualizing legacy systems.

Maintenance of legacy applications consumes more time and resources than any other part of the software lifecycle [75]. This research evaluates the costs and benefits of refactoring existing legacy software with aspects. Replacing a concern that is scattered in many code locations with a single aspect can potentially reduce the number of changes during maintenance. For example, features such as caching, logging, and tracing typically require code to be scattered across many classes in many files, but this scattered code can be modularized as a single concern in one file using aspects. However, there are consequences associated with using aspects. Aspect pointcuts may be fragile, relying on naming conventions or program structures that can change over time. Maintenance changes that conflict with constructs used by pointcuts may introduce defects [41]. There are also one-time costs from restructuring a program to use aspects, including the cost of creating the aspects and removing the scattered code that is replaced by aspects.

[1]`http://www.aspectc.org`

Prior studies of costs and benefits of aspectualization focus on systems created as research projects [42], on only one revision [17, 42], or on a preselected set of aspects [10]. In this research we evaluate the maintenance of three legacy industrial applications by refactoring multiple cross-cutting concerns and follow the evolution across several revisions. In addition, we also refactor additional cross-cutting concerns that occur only in later revisions of the applications.

We compare the aspectualized and original version of the applications as they evolve over six revisions in one application, seven in another application, and three in the third application. Our study demonstrates the types of cross-cutting concerns that occur in existing applications, and measures the costs and benefits of re-engineering these applications to adopt aspects. Because the applications were developed in the same application domain, some of the identified aspects were used in more than one application. For those aspects, we compare the benefits (e.g., code savings) in each application. For each aspect, we report the size of each mock system and time spent developing it. We also describe guidelines for creating mock systems during aspectualization.

Chapter 2

Background

We provide a brief overview of topics, including aspect-oriented programming, object-oriented frameworks, refactoring and test-driven development.

2.1 Aspect-oriented Programming

Aspect-oriented programming has emerged as an approach for designing and implementing complex systems, particularly for handling cross-cutting concerns that affect many modules. Over three decades ago, Parnas advocated modularity for the sake of changeability, independent development, and comprehensibility [62]. To illustrate the benefits of modularity, Parnas showed two different ways to divide the same system into modules. His work also illustrates a potential limitation of modular systems: designers have to choose how (from several possibilities) to modularize a system, and doing so forces them to model everything else in terms of this dominant decomposition. Tarr et al. [68] call this problem "the tyranny of the dominant decomposition".

Kiczales et al. [38] state that "the central idea of AOP is that while the hierarchical modularity mechanisms of object-oriented languages are extremely useful, they are inherently unable to modularize all concerns of interest in complex systems". Aspect-oriented systems extend procedural and object-oriented systems to explicitly model system concerns that crosscut the modularity of the system, with independent core concerns implemented in the existing language structure of classes and procedures.

The most popular aspect-oriented programming language is AspectJ, which is an aspect-oriented extension to the Java programming language [44]. AspectC++ is based on the syntax and semantics of AspectJ but uses the C++ language [49]. Key features of aspect-oriented languages include pointcuts, advice, introductions, and dynamic context capturing. Each of these features will be briefly described.

Aspects typically contain constructs, known as advice, that are invoked at specific points of execution. Advice is analogous to methods of a class, but rather than being invoked directly, advice is associated with parts of the underlying C++ program (such as method calls or constructor invocations) [39]. There are three kinds of advice: before advice, after advice, and around advice. Before and after advice run, as the names suggest, before or after the associated method/constructor call, but around advice is called *instead* of the associated method, although around advice can choose to call the associated method using `proceed` [18].

The parts of the underlying program that can have advice associated with them are known as joinpoints. The joinpoints of a C++ program available to an aspect in AspectC++ include method and constructor invocations (at either the call site or execution site). In addition, AspectC++ provides reflection mechanisms to allow run-time examination of the environment in which advice is being run [72]. A pointcut expression in an aspect specifies the joinpoints that a particular piece of advice will be associated with [49]. Pointcuts provide a form of "quantification" over the underlying program, since they allow specifying many joinpoints using a single expression that can use regular expressions (e.g., all methods matching `Queue.get*`) as well as class hierarchies (all methods matching `get*` that are part of the `Queue` class or a `Queue` subclass) [22].

Filman and Friedman [22] defined aspect-oriented programming in terms of quantification and obliviousness, stating that "For true AOP, we want our system to work with oblivious programmers – ones who don't have to expend any additional effort to make the AOP mechanism work." However, experience in using aspect-oriented programming has uncovered problems with obliviousness. One example of a lack of obliviousness is the occurrence of fragile pointcuts [41], which are pointcuts with high name-based coupling between aspects and core concerns. Faults related to fragile pointcuts may be introduced by non-local changes during maintenance.

In addition to modifying an underlying program through advice, aspects can introduce additional methods and data members to classes. New data members can track additional object-specific information (e.g., mutually exclusive locking state). Another example application of aspects is adding shadows to a graphical display system, and associating a `PointShadow` object with each `Point` object [78]. Method introductions can define completely new members, or override an inherited method. If a class is a base class for other classes in a system, changes to its virtual (inherited) methods will affect the class and its child classes. The effect on the system of a method definition (or redefinition) in a class hierarchy is termed binding (or dynamic) interference [66].

While most AspectC++ features can be realized by compile time analysis and code generation during a process known as weaving [39], AspectC++ also allows for specifying control-flow specific matches. The `cflow` construct is "evaluated at runtime and returns all joinpoints where the control flow is below a specific code joinpoint"; this is done using the call stack (control flow) of the program. Example uses of `cflow` include discriminating between calls to a method from inside a part of an application and associating advice code with the outermost call to a recursive function [49].

Development aspects, such as tracing or debugging, are used when testing and debugging software but are not part of a final shipped product [37]. Aspects that are used in production, such as logging or reporting errors in method parameters, may be *observers* or *spectators* since they do not modify the program state and since the program still functions without them [15]. By contrast, other aspects are *assistants* that provide essential functionality of a delivered application [15].

2.2 Object-oriented Frameworks

The three legacy applications we aspectualized use a common object-oriented framework. An object-oriented framework is a set of related, cooperating classes targeted toward solving a problem in a particular domain. The framework classes define responsibilities, collaboration relationships between classes, and the thread of control [25]. Object-oriented frameworks have been in wide use since the 1980s and consist of collections of collaborating, extensible class hierarchies. Fayad and Schmidt list four main benefits of using frameworks in building systems: modularity, reusability, extensibility, and inversion of control [63].

Middleware is often provided to application designers as an object-oriented framework. Zhang and Jacobsen [76] identify five major architectural elements of middleware systems, and then define aspects of middleware systems as "abstractions or implementations that crosscut any of these major architectural components". They refactored a CORBA middleware system to show that the middleware services can be provided as an aspect-based system [76]. Unlike Zhang and Jacobsen, our work evaluates the effectiveness of using an aspects in conjunction with the original (unmodified) framework rather than refactoring the framework. Also, our research studies the impact over time to the framework-based applications.

2.3 Refactoring

Refactoring is a process of restructuring software to improve its design. The focus is on improving modularity, understandability, and maintainability through a series of small behavior-preserving steps (which are termed refactorings). Because refactoring emphasizes behavior-preserving steps, testing is a critical step to check for unexpected changes in the system. Problems in existing code that are candidates for refactoring are called *smells*; examples include duplicated code, long methods, and parallel inheritance hierarchies. These code smells are eliminated by applying one of the more than 75 refactorings presented by Fowler [23].

Just as Opdyke[61], Beck [8], and Fowler [23] have cataloged common refactorings, AOP techniques can also be used to refactor existing systems. Advocates of aspect-oriented programming have enumerated AOP-based refactorings and evaluating the associated benefits and costs [43]. Coady and Kiczales [17] demonstrated the benefits of using aspects in operating system code by implementing four modules as aspects in an early version of FreeBSD and then observing the changes to those modules as they introduced the changes from two subsequent versions.

Hannemann and Kiczales [32] implemented the original 23 design patterns [25] in Java and AspectJ and found that 17 of the patterns had improved code locality, reusability, composability, and unpluggability. Hannemann and Kiczales define unpluggability and pluggability as the ability to easily switch between using and not using a particular part of a design pattern in a system. A design is both pluggable and unpluggable if a component can easily be used or left out without major changes. Hannemann and Kiczales use locality to indicate that all code for a concern is implemented an aspect. Reusability indicates that an aspect can be used in multiple instances of a design pattern, while composability means that a class may participate in multiple aspect patterns.

They implemented tools to automate some of these refactorings, and demonstrated these tools on the JHotDraw application, which has over 240 user defined types and 15,000 lines of code [30]. More recently, Garcia et al. [26] also measured the impact of using aspects to implement the original 23 design patterns, and found that most design patterns had better structural software properties, such as cohesion and coupling.

The aspect-oriented design patterns previously described [26, 32] were evaluated on a single system, not over time as changes were made. Coady and Kiczales did not single out design patterns in their research on changes to operating system code over time, but they used aspects and measure them over time, and reported that change proneness was improved by using aspects [17]. Our research uses a time-based approach like Coady and Kiczales but we apply it to a set of framework-based applications that change over time.

2.4 Test Driven Development

In order to test code during development, approaches such as Test Driven Development (TDD) advocate creating unit tests before the functional code is written [24, 35]. Creating unit tests up front provides an executable specification for each module and ensures that all code has tests to verify functionality. Using tests to drive development also avoids scope creep – the tendency to add extra functionality that is not needed – since the focus of development is on writing only the code needed for tests to pass. The unit tests are added to the system as annotated test classes and test methods. Test frameworks such as Cpp-Unit [1] and JUnit [2] are used to specify and execute unit tests. Before refactoring legacy code, unit tests should be created to avoid introducing errors [23].

To emulate a complex system dependency, test driven development may use a mock object (or *mock*) in place of the real object [5]. Mock objects are similar to stubs used in testing, but emulate a class or interface, and may provide some basic checking, such as the validity of argument values.

We use two key ideas from test driven development: mock objects and using tests to drive development. The use of mock objects provides a context for creating and testing aspects. It is easier to weave a candidate aspect into a mock object than into a large system. Creating the mock system for testing aspects is similar in intent to creating test classes and methods when using test driven development. By defining a way to unit test aspects, we can have confidence in aspects when we introduce them into a legacy system.

Chapter 3

Related Work

Other researchers are studying the challenges and benefits of using aspects. In this chapter, we present related work on aspects and maintainability, aspect mining, and aspect testing.

3.1 Aspects and Maintainability

Coady and Kiczales [17] evaluated aspects and modularity by refactoring four crosscutting concerns as aspects in revision 2 of FreeBSD. They reported several change metrics, such as the number of source code locations, functions, files, and subdirectories that required modifications in the original and aspectualized versions of the code. They also reported finding additional benefits, including more localized change and reduced redundancy of scattered concern code. Our use of the source code repository of an existing application is similar to theirs, but we consider three applications rather than one, and we follow the changes across more revisions. We use similar measures, although we do not include directory change locality. Coady and Kiczales measured the number of source code locations (i.e. blocks of code) that change, which would correspond to changes in concern diffusion that we measured. In addition, we do not restrict the focus to the set of aspects identified in the first revision, but also look for additional aspects in subsequent revisions.

Kulesza et al. [42] created two versions of the same design as part of their research, where one is a Java-based object-oriented application and the other uses Java and AspectJ. To create a second revision, they carry out a set of maintenance tasks, comparing the changes made in the three applications. They reported that aspects improve maintainability by improving cohesion and by reducing the number of changes. Our study differs by using legacy applications as candidates for aspectualization, and also by evaluating the effects over many revisions rather than two. In addition, the revision differences we consider are based on actual source code changes and not high level design changes. Since we use legacy applications, our results also indicate what types of aspects occur in real applications, which avoids the bias of creating an application after having been exposed to aspect-oriented programming.

Bruntink et al. [10] refactored a cross-cutting concern that implements the *return code idiom* in a large (15 million LOC) application. However, their focus is on the variability of this idiom and the challenges that occur during aspectualization. Rather than focusing on a single concern in one application, in this research we focus on multiple aspects across multiple revisions of legacy applications.

Ceccato and Tonella [69] extended the object-oriented metric suite of Chidamber and Kemerer [14] for aspect-oriented software. These metrics differ from ours by focusing on coupling at the class and aspect level, while we focus on differences in measures such as size and change-proneness. In addition, our study comprises three legacy applications over multiple revisions, while their validation of the metrics was performed using two implementations of the observer pattern.

Griswold et al. [29] refactored HyperCast, a large Java application for multicast networks, using AspectJ. They encountered difficulty in specifying pointcuts that would match the state-machine of HyperCast because they needed to specify multiple points within a method. They reported that many pointcuts were too tightly coupled to names, so that changes to primary classes would break them. They reported two types of development problems. First, the tight coupling between aspects and method names prevented the development of aspects in parallel with primary code refactoring because the aspects could only be developed after inspecting the core concerns. Second, they encountered cases where joinpoints were not accessible because the language (AspectJ) supports specifying joinpoints at the method call level and data member level, but not at the `if` or `switch` statement level [67]. They propose the use of cross-cutting interfaces to improve the modularity of aspect-oriented applications, avoid fragile pointcuts, and enable parallel development of aspects and classes.

Figueiredo et al. [21] studied the benefits of AOP in the context of software product lines applications that are deployed on different hardware platforms. They consider two product lines and evaluate several revisions. Although they also evaluate modularity and find that aspects provide benefits during maintenance, they focus on design stability of the product lines – how change impact, modularity, and dependencies between components are affected when providing variations of the same application for different hardware targets (where some features are enabled and others are disabled). They also consider multiple revisions, and report that aspects had positive impact. However, in the context of product lines, they reported that aspects provided better design stability only for optional or alternative features, and did not perform as well for required features.

Bartsch and Harrison [6] created two online shopping applications, and had groups of subjects perform maintenance tasks on one of two separate applications, which were intended to be equally difficult to modify. Subjects performed tasks on one of the applications and completed a survey, which included information about how long certain tasks took. Bartsch and Harrison reported that the results appeared to slightly favor the object-oriented approach over the aspect-oriented approach. Although the test subjects were software professionals, the application was not an industrial application and the same tasks were evaluated repeatedly, rather than evaluating maintenance over multiple revisions.

Tsang, Clarke, and Baniassad [71] extended the Chidamber and Kemerer metrics and compared two real-time applications, one creating using real-time Java extensions and the other created with Java and AspectJ. They did not aspectualize an application, but instead compared metrics of two applications to determine strengths and weaknesses of aspect-oriented programming. They found that aspects improved modularity by reducing coupling and cohesion. However, aspects increased metrics such as weighted metrics per class, since a method in an object-oriented application often corresponded to a method plus associated advice in the aspect-oriented application. Our study uses metrics to compare three applications, but we compared an aspectualized application to its original. We focus on maintainability metrics such as size, change-proneness, and concern diffusion, and did not specifically measure coupling and cohesion.

Hoffman and Eugster [34] aspectualized three Java applications, measuring coupling, cohesion, size, concern diffusion, and the number of reusable operations. Although their study considered multiple applications, they did not focus on maintainability over multiple revisions. Instead, they compared Java, AspectJ, and a new language mechanism they proposed, explicit join points, in order to provide reusable aspects that were modular without the problems found with obliviousness. Their study focuses on evaluating and improving aspect-oriented languages, while we focus on how aspectualization can improve maintainability.

3.2 Aspect Mining and Refactoring

Aspectualizing involves finding crosscutting code, in a process called *aspect mining*, followed by refactoring.

Coady and Kiczales [17] refactor legacy operating systems written in C to use aspects. Two main criteria identify the aspects: intent and program structure. They report that using aspects helps to localize changes, reduces redundancy, and improves modularity. These benefits are realized with negligible performance impact. They found that aspects helped avoid *brittleness*, where a single functional change required many changes or introduced bugs in other parts of the system [18].

Some system-wide properties, particularly extra-functional properties such as performance, synchronization or memory usage, occur because of architectural decisions that affect all modules; they are emergent properties that arise out of the collection of components. Lohmann et al. [50] propose the use of domain analysis, which produces feature diagrams, which guide the design of an architecture-neutral system. This approach allows aspects to be woven across multiple modules to configure non-functional properties.

Hannemann, Murphy, and Kiczales [33] refactor cross-cutting concerns by identifying roles performed by system components. Roles are mapped to design patterns and aspects using a tool that automates the transformation of the system from Java to AspectJ. Rather than identify component roles, we examine the code for scattered code fragments.

One can use aspects to modularize framework-related code in applications that use frameworks. Our intent is similar to that of Ghosh et al. [27, 28], who differentiate between business logic and code that interacts with middleware services, and implement middleware-related code with aspects. In a similar manner, as we analyze framework-based code in separate applications, we identify aspects based on common infrastructure tasks, such as logging and event-handling. A study redesigning the PURE operating system with AspectC++ explored multiple aspect-oriented changes [52]. Better solutions were obtained when the components and aspects were designed together, since pointcuts in the aspects need to refer to structures within the components.

Tonella and Ceccato [69] empirically assess aspectizable interfaces in the Java Standard Library. They use aspects to implement secondary concerns of classes and interfaces such as serializability or participation in a role of a design pattern (e.g., observable). They report that using aspects improves maintainability and understandability. We also seek to implement secondary concerns as aspects, but our refactoring is done within applications rather than in a framework or library.

Marin, Moonen, and van Deursen [53] identify and use fine-grained *refactoring types* to identify and classify refactorings. A refactoring type documents a concern's intent, its typical idiomatic implementation without aspects, and an aspect-oriented mechanism to implement the concern. We also identify aspects based on intent and idiomatic implementation.

Clone detection techniques [7] can also identify cross-cutting concerns. Bruntink et al. [13] evaluate the effectiveness of clone detection software in identifying cross-cutting concerns in legacy C software, and report that error-handling code and parameter checking code were easiest to automatically identify.

Older programming languages, such as C, do not explicitly support exceptions. Instead, they typically rely on an idiomatic approach for signaling and handling exceptions. One common idiomatic approach is the *return code idiom* [12], in which a special return code signals that an exception has occurred. Even though C++ supports exceptions, C++ code may rely on legacy code and libraries that are written in C and use the return code idiom. Common programming idioms, such as the return code idiom, result in code scattered throughout an application to check the return values of function calls, such as C system calls. An aspect allows replacement of this checking code with advice that executes after the function calls and throw an exception whenever a return code indicates an error.

3.3 Challenges of Testing Aspects

Douence et al. [19] explore techniques to reason about or specify aspect-oriented systems to better understand the effects of aspects on the system. In general, complete verification of aspect behavior is not practical. Thus, we focus on improved testing techniques.

Alexander, Bieman, and Andrews [3] describe a key problem related to testing aspects: aspects depend on weaving and do not exist independently, and are often tightly coupled to the context to which they are woven. Thus, aspects cannot be unit tested in isolation, but can only be tested in conjunction with the core concerns that they are woven with. We weave aspects with the core concerns in the mock system, and use the mock system method calls to provide unit testing of the woven functionality.

Aspect-oriented programming can also introduce new faults, by way of faulty advice code or faulty pointcuts [3]. Existing AOP testing approaches focus on aspect-specific faults [45, 54], or on coverage criteria to provide adequate testing of aspects in the context of a system. Proposed coverage criteria for aspects are based on dataflow coverage [77], path coverage [46], and state-based coverage [74]. Dataflow and path coverage require program analysis that is beyond the scope of our work. Our legacy systems do not have state diagrams to guide state-based testing. However, we do measure coverage of joinpoints matched by a pointcut, as described in Section 4.2.

Zhou, Richardson, and Ziv [79] use an incremental testing approach in which classes are unit tested, aspects are tested with some classes, and then aspects are integrated one at a time with the full system. Test case selection forces the execution of specific aspects. Using our approach, iterative test cycles are applied to the mock system rather than the full system. Rapid iterations are achieved because the mock system and aspects can be compiled and woven in a small fraction of the time required to compile and weave the full system.

Lesiecki [47] advocates delegating advice functionality to classes, so that the classes used by advice can be unit tested directly. This is similar to the language approach of JAML [51], which implements aspects as classes that are woven by XML specifications. Lesiecki uses mock objects and mock targets to help unit test aspects and uses visual markup in the Eclipse Integrated Development Environment (IDE) to verify that pointcuts affected the expected program points. A mock target is similar to our concept of a mock system. However, a mock target is created from an aspect to unit test pointcut matching. By contrast, our mock systems are created from the real system based on how we expect aspects to be used in that system.

Chapter 4

Aspectualization using Mock Systems

We use the following steps [59] to aspectualize the original applications:

1. Identify a cross-cutting concern that can be refactored as an aspect.

2. Create a small mock system to iteratively develop and unit test a prototype aspect. The mock system mimics the program structures in the application (or real system) that the aspect must interact with.

3. Refactor the application to use the aspect by removing duplicate or cross-cutting code from the application and then weaving the aspect.

4. Conduct integration testing of the refactored application by executing the regression tests.

Figure 4.1 illustrates the approach. In the second step, developers use the mock system to iteratively experiment with alternative aspect pointcut specifications and advices to test that they correctly modularized the cross-cutting concerns. The pointcut specifications must have the correct strength [3] so that they match all (and only) the desired joinpoints in the mock system. The intent is for the pointcut to match the desired joinpoints in the real system. Mock systems contain a small `main()` function, as well as class and function stubs to provide targets for pointcuts and to provide just enough structure and functionality for the advice to interact with. In contrast to the application, which may have thousands of lines of code, mock systems often contain fewer than 100 lines of code.

Creating and debugging aspects often requires developers to iteratively identify, develop, integrate, and test aspects. Thus, the steps may be repeated as needed. Problems encountered during integration may result in changing an aspect and re-testing using the mock system. We describe each of the four steps in Sections 4.1 through 4.4 below.

4.1 Identifying Aspects in Legacy Systems

The legacy systems we aspectualized consist of three VLSI CAD applications which are all based on a VLSI CAD framework [56]. We identify aspects in the system to factor out scattered identical or similar code, to enable fast debugging, and to provide automatic enforcement of system-wide policies.

Like Coady and Kiczales [17], we use intent and structure as a primary means to identify aspects. We look for features that crosscut modules or provide the means (intent) to deal with crosscutting concerns (such as callbacks and mixins). For example, we identify policies based on the design intent of a base class and scattered code in methods of its sub-classes. We also refactor system-wide concerns that affect performance and architectural decisions such as caching and system configuration, following the approach of Lohmann et al. [50].

We use aspects to modularize cross-cutting application code that uses framework methods and data structures. Since our example systems are framework-based, we seek candidate aspects such as code repeated when using the framework, or common idioms associated with parts of the framework.

Due to a lack of appropriate tools for C++ code, we identify aspect candidates using manual inspection of source code, simple textual approaches such as `grep`, and

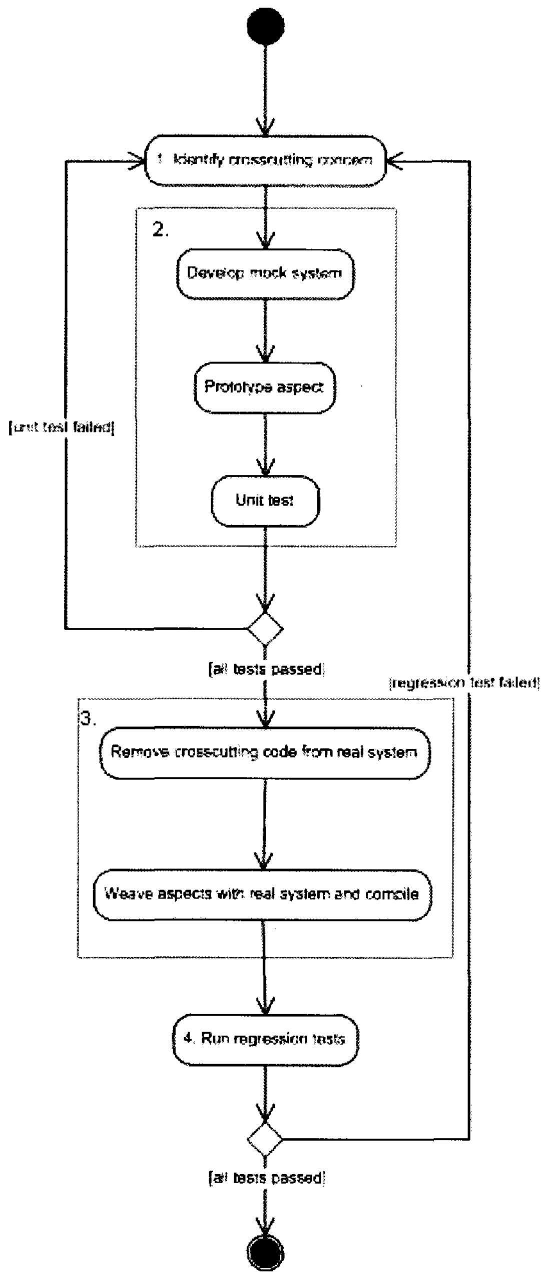

Figure 4.1: Steps for our Mock System-based Approach.

a tool we developed that identifies concerns based on word sets after function and method calls. We describe our concern identifier in Chapter 5.

Because cross-cutting code typically involves many files and classes, code browsers and Unix utilities such as `grep` help to identify similar or related code. Aspect-mining tools that fully parse a program are a better long term approach [31]. In addition to using our own concern identifier tool, we use the CCFinder [36] clone detection tool.

4.2 Using Mock Systems to Create and Unit Test Aspects

In this step we aim to produce aspects with pointcut specifications and advice that will work correctly in the real system. Each identified cross-cutting concern is implemented as an aspect.

A mock system consists of functions and classes that implement a simple program with similar program structure and behavior as the real system, but on a much smaller scale: hundreds of lines of code (LOC) instead of tens of thousands. A mock system contains joinpoints that mimic the real system. The pointcuts are defined to match the mock system structure. We create the mock system by copying or implementing the subset of classes, methods, and functions in the real system that are relevant to an aspect. The methods and functions need only implement enough functionality for the mock system to run the test. The mock system may use assertions to aid in unit testing the aspect. Guidelines for this process are in Chapter 7.

The overall goal of unit testing is to test that (1) the pointcuts have the correct strength and (2) the advice is correct. During test execution of the woven mock system, we aim to achieve joinpoint coverage and mock system statement coverage. *Joinpoint coverage* requires executing each joinpoint that is matched by each aspect. Thus, joinpoint coverage focuses on testing each aspect in all of the contexts where it is woven. We use statement coverage to identify any mock system code that was not executed.

To meet the goal of correct pointcut strength, we analyze the weave results to identify unused advice. In addition, for each advice of each aspect, we annotate some methods or functions in the mock system to indicate whether or not they should be advised. We use four types of annotations: `ADVISED`, `NOT_ADVISED`, `ADVISED_BY(name)`, and `NOT_ADVISED_BY(name)`. These annotations express the design intent — whether or not a core concern is expected to have advice that matches it. The `name` argument can be used to indicate a specific aspect that should or should not advise a method. We check whether the annotated methods had the expected advice (or lack or advice), depending on the annotation. One advantage that our annotations provide is that they are checked right after weaving, and do not depend on running the mock system.

We use three support tools: weave analysis, advice instrumentation, and coverage measurement. Weave analysis evaluates pointcut strength, while advice instrumentation and coverage analysis check that advice is tested in all contexts (joinpoints), supporting the goal of specifying correct advice. These tools are described in Chapter 5.

4.3 Removing Cross-Cutting Code

Once we complete unit testing of the woven mock system, we apply the aspects to the real system. Refactoring involves removing scattered code, and may also involve restructuring or renaming core concerns so that pointcut statements can match the desired joinpoints in the program. The aspects are then woven with the refactored system.

In most cases, the aspect pointcut does not need to be changed. However, some aspects, such as caching, define a pointcut as a list of all functions to cache. When we use such aspects with the real system, the pointcut must change to reflect each cached method from the real system.

4.4 Integration Testing of Refactored System

This step tests whether or not aspectualizing the system introduces new faults by running existing regression test suites. We do not seek 100% statement coverage of the real systems, since the regression tests do not achieve complete coverage on our legacy systems even without aspects. We use joinpoint coverage to verify that advice that we are adding has been tested in all execution contexts, and add regression tests if needed to achieve joinpoint coverage.

If a regression test fails, we determine the root cause of the failure. Suspected root causes can be simulated in the mock system by emulating the system context that contained a fault or that exposed a fault in the aspect. This allows us to observe the erroneous behavior in the mock system and fix it before we modify, weave, and compile the real system. During integration testing, any unused advice is reported as an error. In addition, annotations (such as `Advised` and `NotAdvised`) can be inserted in the real system to check that aspects advise the intended core concerns.

4.5 Mock System Examples

We used mock systems to aspectualize 14 aspects from three legacy systems at Hewlett-Packard. In this section we briefly describe two of legacy systems, and then describe four of the aspects and the mock systems used to develop and test them. We use these four aspects as a running example to illustrate our approach.

4.5.1 The Legacy Systems

The `ErcChecker` is a C++ application that consists of approximately 64,400 LOC. It performs 59 different electrical checks. Each electrical check implements a set of virtual methods as a subclass of an abstract class ErcQuery. We use aspects to modularize the enforcement of two design policies related to the ErcQuery subclasses. We use an aspect to implement caching for functions that calculate electrical properties of objects in the ErcChecker. The original code had caching functionality scattered in each of these functions to improve performance.

The PowerAnalyzer is a C++ application containing 13,900 LOC that is used to estimate power dissipation of electrical circuits. It consists of three smaller tools and a libPower library with some common code. Although the PowerAnalyzer uses an object-oriented framework, and defines and uses C++ classes, much of it is written as procedural functions rather than classes and methods.

Both the PowerAnalyzer and ErcChecker make calls to the object-oriented framework API to create an in-memory graph of instances of classes from the framework. The graph represents circuit elements (e.g. transistors and capacitors) and the connectivity between those elements (called nets or nodes).

4.5.2 ErcChecker Policies

The ErcChecker has two policies that each electrical check must implement. The policies represent a set of features that each electrical check is supposed to provide, which are implemented as scattered code in the createQueries() method of each ErcQuery subclass.

The first policy aspect, QueryConfig, provides run-time configuration, which allows users to disable queries at run-time via an editable configuration file. Each createQueries() method has similar code with the following structure:

```
static void CLASS::createQueries() {
 if(ErcFet::getQueryConfig("CLASS")==eOff){
  //run-time user config disabled this query
  return;
 }
 //create and evaluate query objects...
```

Calls to `ErcFet::getQueryConfig()` are almost identical in each subclass, with the word `CLASS` above being replaced by the actual name of each subclass. Because C++ lacks run-time reflection, the `createQueries()` method uses the class name as a literal string when calling `getQueryConfig()`.

This policy is always implemented as scattered code within the `createQueries()` method of each `ErcQuery` subclass. Hence, the pointcut should just be `%::create-Queries()`, with the AspectC++ wildcard (%) used to match `createQueries()` in all subclasses. The advice uses the scattered call (the call to `getQueryConfig()`) and either proceed to the `createQueries()` body or return without executing it.

The second policy aspect, `QueryPolicy`, implements three of the six conceptual steps needed by each query, but with significant variation between the queries. These steps are:

1. Call framework methods to identify needed circuit data.

2. For each relevant part of the circuit, create an instance of the query class associated with the check.

3. Call the `executeQuery()` method on the query object from step two.

4. Add queries that result in a failure or warning to a container class.

5. Write the results of `executeQuery()` to a log file.

6. Delete queries that did not result in a failure or warning.

Although the first three steps vary significantly between the different subclasses, steps four through six use the same set of method calls and always follow a call to `executeQuery()`. Thus, a single aspect can implement steps four through six using `after` advice to provide the same functionality. Since steps four through six always follow a call to `executeQuery()`, the aspect uses `executeQuery()` as the pointcut.

4.5.2.1 Using the Mock System for Aspect Creation and Testing

Since both policy aspects use the same class hierarchy, the mock system models the `ErcQuery` base class and its subclasses. We created a mock `ErcQuery` class and four subclasses with method stubs based on different types of checks, such as transistor-based checks and net-based checks. In addition, a driver file (`main.cc`) creates query objects, calls the `createQueries()` and `executeQuery()` methods, and reports success or failure. The `LevelManager` singleton class stores all objects related to failures and reports this information, so we need a mock class for the `LevelManager`.

The subclasses in the mock system contain only the method call to be advised and the system methods used by that method and the advice. In the mock system, the sub-classes of `ErcQuery` emulate both types of query behavior: failing due to circuit errors and passing due to error-free circuits.

With the mock system in place, we created and tested the `QueryConfig` and `QueryPolicy` aspects. The `QueryConfig` advice executes around each call to the `createQueries()` method, and extracts the class name from the joinpoint information available through AspectC++ joinpoint API. The `QueryConfig` implementation is shown below:

```
aspect QueryConfig {
   pointcut createQuery()=execution("% %%::createQueries(...)");
    advice createQuery() : around() {
     string jpName = JoinPoint::signature();
     int first_space = jpName.find(' ');
     int scope_operator = jpName.find("::");
      string className=jpName.substr( first_space+1,
          scope_operator-first_space-1);
     if(ErcFet::getQueryConfig(className)==eOff)
        return;  //user config exists, SKIP
     tjp->proceed();
   }
};
```

The `QueryConfig` prevents the call to `createQueries()` by only calling `proceed()` when a configuration does not disable a query.

The `QueryPolicy` aspect uses after advice to implement the last three steps of the query policy for each subclass. Its AspectC++ implementation is shown below:

```
aspect QueryPolicy {
   pointcut exec_query(ErcQuery *query) =
      execution("% %::executeQuery(...)") && that(query);
   advice exec_query(query) : after(ErcQuery *query) {
       if(gReportAll || query->errorGenerated()) {
          LevelManager::addQuery(query);
          gLog->log() << "Query error: type: "
                      << query->getName()
                      << " start element: "
                      << query->getStartName()
                      << query->getSummary() << endmsg;
          query->logQueryDetails();
       }
       else {
          gLog->log() << "Query ok: "
                      << query->getName()
                      << endmsg;
          query->logQueryDetails();
          delete query;
       }
   }
};
```

The `QueryPolicy` aspect uses the `errorGenerated()` method to determine if the query that called `executeQuery()` found an error. If `errorGenerated()` returns true, then the query is added to the `LevelManager`, which stores all circuit failures so that the user can view them. If `errorGenerated()` returns false, the advice deletes the query.

Unit testing for both aspects was driven by code in the mock system that created electrical query objects using our `ErcQuery` mock classes. We annotated the `createQueries()` method of each `ErcQuery` subclass to check that the pointcut matched. Statement coverage of the mock system found dead code and an unused mock class method, enabling us to make changes to achieve 100% statement coverage of the mock system.

4.5.2.2 Refactoring and Integration Testing

Introducing the aspects requires removing all code that is duplicated by the aspect. Since the pointcuts matched methods (`executeQuery()` and `createQueries()`) that are already in the real system, we did not rename or restructure the code to provide targets for aspects weaving. We encountered three challenges when refactoring to use the query behavior policy.

The first challenge was that replacing custom text (scattered in each query) with an aspect changed the output of the program. Regression tests that rely on output logs fail due to the now standardized output. Although the standardization should improve the maintainability of the `ErcChecker`, it does require a one-time update of the expected test output files.

The second challenge results from the *asymmetric* nature of aspect-oriented refactoring: removing the scattered code must be done for each subclass (typically manually), while the aspect is woven automatically into all matching subclasses. The `QueryPolicy` aspect deletes query objects that do not detect electrical errors (step 6 in section 4.5.2). During refactoring, if the object deletion code is not removed from the core concern, both the core concern and the aspect try to delete the same object, resulting in a memory fault. When we manually removed scattered code, we failed to remove the deletion code from one `ErcQuery` subclass. Finding the root cause of this defect in the real system was difficult because the defect results from an interaction between the aspect and underlying system, and the woven code can be hard to understand. To confirm the suspected root cause of this fault, we created another `ErcQuery` class in the mock system that deliberately called `delete` outside the aspect to recreate the memory fault.

The third challenge was that we did not anticipate some necessary changes to the real system and aspect when we created the mock system and the aspect. During refactoring, we realized that 18 of the 59 `ErcQuery` subclasses printed out some additional information between steps five and six (on p. 25), which were implemented by the advice [56]. In order for the aspect to work with all subclasses of `ErcQuery`, we (1) added an empty method to the base class, (2) refined the method to contain the logging statements in the subclasses that needed this feature, and (3) modified the advice to call the new method. We made these change to the `ErcQuery` class hierarchy and aspect in the mock system. We tested the changes in the mock system, made the changes to the real system, and continued refactoring the real system.

4.5.3 ErcChecker Caching Aspect

Caching is a common example in the AOP literature of a candidate aspect since its implementation is similar across all cached functions [44, 48]. We can identify cached functions since they use a local static set or static map. The `ErcChecker` contains 38 functions that implement similar caching code, but are in various classes and do not have a common naming convention. The aspect pointcut specifies a list of all the functions to be cached, while the advice provides the caching functionality.

4.5.3.1 Using Mock Systems for Aspect Creation and Testing

We created one mock system to test the caching aspect's functionality, and another mock system to evaluate performance. The first mock system contains methods that have the same types of parameters and return values as the functions to be cached in the real system. The mock system methods are short (1-4 lines) and return a value based on the argument. For example, to test caching of methods with a `bcmNet` pointer parameter, we can use the `GetNetValue()` method below.

```
int GetNetValue(bcmNet *n)    /*AOP=ADVISED*/
{
    return n->GetName().length();
  }
```

In the mock system, we can call `GetNetValue()` with different `bcmNet` instances and check for the correct return value. Then, with caching added, we can check that we still get the correct return values.

For aspect-oriented caching, we checked several requirements using the mock systems. First, the cache must function properly. Second, the caching aspect must work correctly with different data types. Third, the cache should not introduce any noticeable performance penalty; in fact, caching should improve performance. Fourth, we needed a way to determine if a cached function was actually using previously stored values (i.e. being called repeatedly with the same value), since unnecessary caching adds overhead without improved performance.

The first mock system focused on functional behavior and modeled pointers to complex data types and different scalar types. The mock system imported the `BlockData` component from the `ErcChecker`. The `BlockData` component populates the framework with data, enabling caching to be tested with framework pointers. We explored a number of alternative caching implementations [55] using C++ templates and inheritance.

We created caching aspects for procedural functions and object-oriented methods. For methods, the hash key stored is the object invoking the method, while for procedural functions the hash key is the function parameter. The caching aspect for procedural functions is shown below. It stores the first argument to the cached function (`tjp->arg(0)`) and the return value (`tjp->result()`). The static map defined in the aspect uses AspectC++ type definitions. For example, `JoinPoint::Arg<0>::Type` is the data type for the first argument to the parameter of the function that matched the pointcut.

```
  aspect AbstractFunctionCache {
      pointcut virtual ExecAroundArgToResult() = 0;
      advice ExecAroundArgToResult() : around() {
         JoinPoint::Result *result_ptr = NULL;
         static map <
           typename JoinPoint::Arg<0>::Type,
           typename JoinPoint::Result > theResults;
         JoinPoint::Arg<0>::Type *arg_ptr =
             (JoinPoint::Arg<0>::Type*) tjp->arg(0);
         JoinPoint::Arg<0>::Type arg = *arg_ptr;
         if( theResults.count( arg ) ) {
            //already have the answer, return it
            result_ptr = tjp->result();
            *result_ptr = theResults [ arg ];
         } else {
            //proceed and store the answer
            tjp->proceed();
        result_ptr = tjp->result();
            theResults [ arg ] = *result_ptr;
         }
      }
   };
```

The `AbstractFunctionCache` aspect defines a virtual pointcut. To use the cache, a concrete aspect extends the `AbstractFunctionCache` and defines the pointcut as a list of functions to be cached. Using the mock system, we tested that the pointcut matched intended functions. We tested that the advice avoided recomputation when calls used the same arguments. The mock system contained math functions (e.g., `Square()` and `SquareRoot()`) for which a concrete aspect was created that cached their values. Although the pointcut was not the same in the mock system and real system, the abstract aspect with its virtual pointcut is the same in the mock and real systems.

Because C++ supports an object-oriented style and a procedural style, the mock system had class-based method calls and procedure calls, and the aspects were developed to provide both types of caching. Our approach enabled developers to easily switch between a simple cache or a slightly slower cache that reported on cache usage for each function so we could measure if caching was actually saving computation [55].

We created a second mock system to compare the performance of the AspectC++ approach to the original C++ cache. Using command-line parameters, we could specify which caching implementation to use and the number of times to execute the cached function. Because the execution of the second mock system only executed one set of caching benchmarks, we were able to use the `Linux time` command to measure the CPU time for each type of caching.

4.5.3.2 Refactoring and Integration Testing

The only change needed to weave the caching aspects with the full system was to expand the pointcut of the concrete caching aspects to match all cached functions. We changed the pointcut by using the names of the previously identified cached functions.

When we removed the original caching code from methods in the real system, we added an annotation to indicate that the method should be advised and added it to the list of functions in the pointcut. The weave analyzer checked that the pointcuts defined when refactoring matched the intended functions.

4.5.4 PowerAnalyzer Debugging Aspect

The first aspect we used in the `PowerAnalyzer` was a development aspect to aid in debugging a case of abnormal termination. A framework method in the `Power-Analyzer` called `exit()` after indicating that a null pointer had been encountered. Calling `exit()` limits visibility when using a debugger such as `gdb` because the program terminates without preserving any system state. By contrast, using `assert` also exits a program, but creates a core file with the state of the program so that the call stack and program state can be analyzed.

The error message listed the name of the framework iterator method name. However, the method was in a base class that was inherited by several subclasses, so there were many candidate calls in the application that may have triggered the error. These calls represented possible locations of the defect and cross-cut 18 locations in four files.

The prior approach to debugging such problems involved adding print statements around all calls that could have triggered the error. This required modifying all 18 locations in four files, finding the defect and fixing it, and removing the 18 modifications from the four files. This process is tedious and error prone. A single aspect can automatically provide the same tracing, using the framework iterator initialization as the pointcut. The advice uses AspectC++ features to print the context of each iterator call.

4.5.4.1 Using the Mock System for Aspect Creation and Testing

The mock system for the debug tracing aspect calls different types of framework iterators as well as similarly named methods that should not be matched by the `CadTrace` aspect pointcut. We checked that the aspect prints out tracing statements only before the intended iterator calls. We also reused the `BlockData` component from the `PowerAnalyzer` to load framework data so that we could call framework iterators in the mock system.

We used our annotations to indicate which methods in the mock system should be advised and which should not. The weave analyzer and joinpoint coverage data checked that the aspect matched only the desired framework calls.

Since all the iterators initially call a `Reset()` method, the aspect used `before` advice associated with `Reset()` iterator methods.

```
aspect CadTrace {
   advice call("% %Iter::Reset(...)\")
   : before() {
      cerr << "call Iter::Reset for"
```

```
                << JoinPoint::signature() << " at jpid: "
                << JoinPoint::JPID << endl;
        }
    };
```

The CadTrace aspect uses the AspectC++ joinpoint method, JoinPoint::signature(), to print out which iterator is called.

4.5.4.2 Refactoring and Integration testing

A development aspect for tracing calls does not require refactoring the core concerns, so integration only requires weaving the aspect into the application code. The aspect worked correctly with the libPower library on the first try and the call that triggered the framework error was located. After weaving the aspect with the PowerAnalyzer, we ran the test that was producing the program exit failure.

4.5.5 PowerAnalyzer Timing Aspect

The second PowerAnalyzer aspect modularizes code that reports the status of the application and writes to a log file. Because execution times for VLSI CAD software can be long (hours or even days), a common extra-functional concern is to write time stamps and elapsed time to a log file. The PowerAnalyzer uses a TimeEvent class, which contains a method to reset the elapsed time and a method to return the elapsed time as a string suitable for writing to a log file.

The aspect instantiates a TimeEvent object within the advice body and uses the AspectC++ joinpoint API to print the context in which the TimeEvent object is being used. Since the TimeEvent is used within different functions, function names must be used as pointcut targets. In order to avoid enumerating all functions that should be associated with the TimeEvent, we decided to rename methods to begin with tmr if they should have the timing functionality. This enables the aspect to use a pointcut with a wildcard to indicate "all functions beginning with tmr".

Modularizing the code for capturing and recording timer information into a single aspect provides flexibility if the `TimeEvent` interface changes. In addition, if a different timing module were substituted, only the aspect would need to change, rather than scattered code in the `PowerAnalyzer`.

4.5.5.1 Using the Mock System for Aspect Creation and Testing

The mock system for the `Timer` aspect has two methods beginning with `tmr` to match the pointcut and two other methods that do not match this pattern. The two methods that begin with the pointcut pattern have `ADVISED` annotations, while the other two have `NOT_ADVISED` annotations. One `tmr` method calls the other to test that timing works correctly with nested calls. The method bodies contain only print statements to show program flow and calls to a system function (`sleep()`) to insert delays that are measured by the `TimeEvent` class. The mock system does not rely on framework components, but uses the `TimeEvent` module, which already existed in the `Power-Analyzer`.

The `Timer` aspect uses around advice. The aspect instantiates a `TimeEvent` object to record the time, proceeds with the original function call, and then accesses the `TimeEvent` object to calculate and write the elapsed time.

```
aspect Timer {
   pointcut tmr() = call("% tmr%(...)"));
   advice tmr() : around() {
      TimeEvent timer;      //set up timer
      timer.Reset();
      tjp->proceed();   // execute advised method
      //write out the time used
      PrintI(911, "Time around %s: (%s)\n",
             JoinPoint::signature(),
             timer.CheckTime());
      PowerMessage::WriteBuffers();
   }
};
```

The mock system allowed us to quickly test the pointcut. We changed the pointcut twice as we corrected problems we encountered when advising nested calls. We also used the mock system to test how the advice instantiated the `TimeEvent` module and how timer log messages were formatted.

4.5.5.2 Refactoring and Integration Testing

AspectC++ relies on name-based pointcuts to weave in advice. Even though similar code to instantiate and use `TimeEvent` objects exists at many locations, there is no common structure or naming convention for the pointcut to match. To refactor the `PowerAnalyzer`, functions that used the `TimeEvent` class had the `TimeEvent` instance and calls removed, and were renamed from `FunctionName` to `tmrFunctionName` to match the pointcut specification.

One challenge was that some of the application code was written as large, procedural functions, including a 500 line function with 15 separate uses of the `TimeEvent` module. These separate uses were either loop statements or conceptually separate code blocks. For this function, we first used Extract Method refactoring [23][1]. Creating a function with a name that begins with `tmr` allowed capturing the joinpoint in AspectC++. By using a meaningful function name, we could pass a single signature to the `TimeEvent` module instead of a separate descriptive argument for each `TimeEvent` call. For consistency, we applied the same aspect across all three executables of the `PowerAnalyzer`.

Using name-based pointcuts results in tight coupling that can cause problems during maintenance due to name changes in functions [70]. There is tight coupling between the `Timer` aspect and the naming convention of methods. If a new function

[1]This refactoring step states: "Turn the fragment into a method whose name explains the purpose of the method."

is added that should use the timer, it must begin with `tmr` or it will not have the `Timer` functionality woven in. In addition, if someone changes one of the names of the functions so that it no longer begins with `tmr`, time logging will no longer occur for that function. If a function that does not need timing information is created with a name that begins with `tmr`, that function will match the pointcut and have `Timer` advice associated with it. Since `PowerAnalyzer` regression tests focus on functionality and not the time taken by system functions, an error associated with timing might not be immediately detected. Our annotations can be used in the real system to report when a change in the system or a pointcut change causes a pointcut to not match the intended joinpoint.

We refactored the application so that the `TimeEvent` module is only invoked by the advice of the `Timer` aspect. To enforce this design decision, we added additional advice [55] that uses compile-time assertions in C++ [4] to trigger a compilation error if direct calls to the `TimeEvent` module are re-added in the core concerns.

Chapter 5

Tools

Four tools developed for this research support our aspectualization approach:

1. *Concern identifier* searches for repeated word sets that may indicate crosscutting concerns.
2. *Weave analysis* checks for unused advice or advised methods without advice.
3. *Advice instrumentation* supports coverage analysis.
4. *Aspect and system coverage measurement* measures joinpoint coverage and statement coverage of the mock system.

While the testing concepts are general and could be applied to other languages (such as AspectJ), our unit and system testing tools support AspectC++, and they leverage features of the AspectC++ weaver. The AspectC++ weaver writes information about the weave to an XML file for use by IDEs such as Eclipse[1]. The XML weave file contains information on aspect advice and joinpoints, and the methods, functions, and advice associated with each joinpoint [60].

[1]`http://www.eclipse.org`

5.1 Concern Identifier

Similar code that occurs in several locations of an application following a function call may represent a crosscutting concern. We remove the scattered code and use it as the basis for advice. The pointcut uses the function call to advise the original location of the scattered code. The concern identifier tokenizes the lines of code that immediately follow each function call, creating a list of words (variable names, function and method names, and keywords) that follow the function call. The list of words is sorted alphabetically into a word set. The tool compares the word sets that follow a function call to the word sets that follow that same function call in other program locations. Function calls that have word sets with a high percentage of common words are flagged as potential aspects.

The concern identifier finds cross-cutting concerns, such as the `Excepter` aspect described in Chapter 6, in which error handling was implemented with a function call (e.g. `fopen()`) followed by code that checked the return value of the call. Since code that followed `fopen()` was similar throughout the application, the word sets were identified as similar and the error handling code was refactored as an aspect.

5.2 Weave Analyzer

The weave analyzer parses the XML weave file, which identifies the line numbers of joinpoints matched by each pointcut. Next, it reads the core concern source code to find the line numbers that have annotations indicating where advice should and should not be woven. Weave analysis checks for our four types of annotations: `ADVISED`, `NOT_ADVISED`, `ADVISED_BY(name)`, and `NOT_ADVISED_BY(name)`.

By comparing line number information from the XML weave file with the lines that have annotations, the weave analyzer identifies lines of code with one of the following annotation violations:

1. Lines with an ADVISED annotation that are not matched by any pointcut.

2. Lines with an ADVISED_BY(name) annotation that are not matched by a pointcut of the named aspect.

3. Lines with a NOT_ADVISED annotation that are matched by any pointcut.

4. Lines with a NOT_ADVISED_BY(name) annotation that are matched by a pointcut of the named aspect.

For each of these annotation violations, the tool prints the line of source code and the preceding and succeeding lines to provide context. Checking for annotation violations helps identify pointcut strength errors by flagging pointcuts that do not match the developer's intent. The NOT_ADVISED annotations identify pointcuts that are too weak, matching methods the designer did not intend. The ADVISED annotations identify pointcuts that are too strong (restrictive), missing intended joinpoints.

In addition to checking annotations, the weave analyzer reports any advice whose pointcuts match *no* program joinpoints as an error. Unused advice indicates a pointcut error.

Example output of the location and body of unused advice that was identified by weave analysis of the ErcChecker mock system is shown below:

```
========================
We have UNUSED advice:
   Advice: aspect:0 fid:1 line:18 id:0
   type:after lines: 4
========================
File: LogMath.ah aspect: LogExecution  lines: 18-21
      advice call("% mth%(...)") : after()
      {
         cerr << " AFTER calling "
              << JoinPoint::signature()
              << endl;
      }
```

When a method with an ADVISED annotation does not have a matching pointcut, the tool prints the line of source code along with the preceding and succeeding lines. Example output for a such method is shown below:

```
========================
We have UNADVISED code:  did not find a
Pointcut for Line 13 of file ./main.cc
which was specified as ADVISED
========================

   void run_checks() /* AOP=ADVISED */
   {
```

5.3 Advice Instrumenter

As part of our build process, we instrument advice to enable the measurement of joinpoint coverage. During execution of the mock or real system, we gather information about which aspects were executed, and, for each each aspect, which joinpoints are executed. To gather this data, we preprocess the aspects before weaving, and for each advice body we insert a call to a C++ macro, TEST_TRACE, which we define. This macro produces the following: the aspect filename, the source code line number of the advice body, and the joinpoint identifier where the aspect is executing.

The aspect filename is determined in C++ by the C++ preprocessor directive __FILE__, which is replaced at compile time by the actual name of the file containing the directive. In AspectC++, aspect file names end .ah; file names are not changed during the weave.

The source code line number of the advice body is inserted as an argument to the macro call that is added to the advice. Although C++ contains a directive to emit the actual line number of a statement, __LINE__, the number is determined at compile-time. Since AspectC++ uses source-to-source weaving, the line numbers emitted by the __LINE__ directive are based on the woven code rather than on the source code. The coverage analyzer (described in section 5.4) uses the XML weave data, which refers to pre-weave source numbers. Thus, we cannot use the `__LINE__` directive. Instead, the advice instrumenter embeds the pre-woven line number as a parameter to the `TEST_TRACE` macro.

To obtain the joinpoint identifier, the `TEST_TRACE` macro uses an AspectC++ construct: `JoinPoint::JPID`, which is an integer that can be accessed within advice code. The integer serves as the joinpoint identifier that was written to the XML weave file.

5.4 Joinpoint Coverage Analyzer

We measure joinpoint coverage during both mock and system testing. System joinpoint coverage data is calculated during regression runs from the instrumented code. The generated data includes the original, pre-weave source code line number of the advice and the joinpoint ID (available in AspectC++ advice as `JoinPoint::JPID`). This data is cross-referenced to the XML weave file to identify any advised joinpoints that were not executed.

Existing statement coverage tools can check coverage of all mock system code during unit testing. We use `gcov`[2] on the woven code, and generate coverage data for each source file. Statement coverage produced by `gcov` identifies missed core concern code in the mock system. Since the mock system is designed to emulate interactions between the aspect and real system and to call methods that will be advised, we use this coverage to check that we are actually testing all the interactions.

The `gcov` output is a copy of the mock system code that marks how many times each line was executed. Our joinpoint coverage analyzer prints out any missed mock system statements from this `gcov` output file. Sample output is shown below:

[2]`http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

```
File QueryPolicy.ah was covered
File main.cc was covered
File query.h had 2 missed lines:
 ###:   63:this->__exec_old_executeQuery();
 ###:   67:inline void __exec_old_execute
            std::cerr << "Executing query
File erc_main.cc had 2 missed lines:
 ###:   64:    cerr << "No LevelManager
 ###:   65:    return;
TotalLinesMissed = 4
```

The results above indicate that two code fragments were not used. The first unused code fragment reported (associated with lines 63 and 67 of `query.h`) refers to a method in the base class of a mock hierarchy. The method was overridden by every child class and thus was never used. We fixed this by converting it into a pure virtual method in the base class. The second unused code fragment (lines 64–65 of `erc_main.cc`) represents error handling code that was not tested. We covered this fragment by adding code to the mock system to test the error-handling code. Because the mock system is small and is created to test aspects, we aim for 100% statement coverage of the mock system. When using `gcov` and `g++` in the mock system, we do not enable compiler optimizations, avoiding known interactions problems between `gcov` coverage and `g++` optimization.

Figure 5.1 shows the data read in and produced by three of the tools – weave analyzer, advice instrumenter, and coverage analyzer – and how the tools use data produced by the AspectC++ weaver. The advice instrumenter reads an aspect and adds a call to `TEST_TRACE`, with the '`#`' value indicating that an actual line number would be inserted. The weave analyzer reads the C++ source code and XML weave file, and is run after the code is woven. The coverage analyzer reads the XML weave file and output of the woven system to identify advised joinpoints that were not executed.

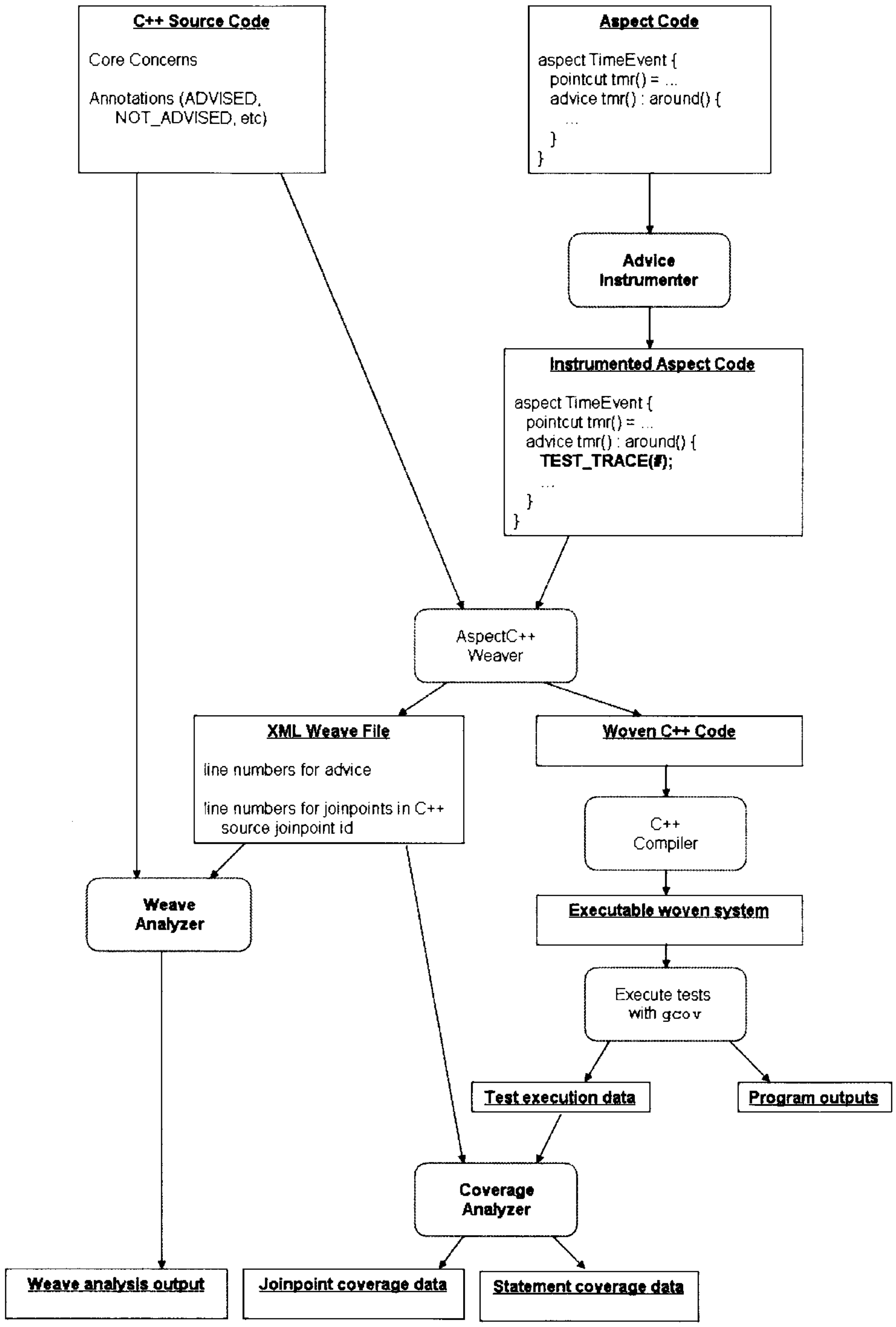

Figure 5.1: Using the Weave Analyzer, Advice Instrumenter, and Coverage Analyzer During Refactoring.

5.5 Discussion

The AspectC++ weaver may eventually be extended by its developers to identify unused advice as an error when weaving. Even if such an extended weaver were available, our tools provide other capabilities not offered by the AspectC++ or AspectJ weavers. Our four compile-time assertions enable developers to indicate where they do and do not expect advice, so that pointcut errors can be caught immediately after weaving. These annotations help when creating a pointcut.

Laddad [44] shows how to use the `declare error` construct in AspectJ to compare two pointcuts (i.e., an old one and a potential new one) to catch errors when a pointcut is changed. His approach does not help when a pointcut is initially created, nor does his approach compare the matched pointcuts to source code annotations that capture a developer's intent and expectation.

Chapter 6

Legacy Systems and Aspects

In this chapter, we describe the three legacy systems we used for the study. We also describe the cross-cutting concerns that were refactored as aspects.

6.1 Legacy Systems

The study subjects are three proprietary VLSI CAD applications of Hewlett-Packard: `InstanceDrivers`, `PowerAnalyzer`, and `ErcChecker`. All three are C++ applications that are based on an object-oriented C++ framework developed at Hewlett-Packard. The framework contains classes and method calls for use in the VLSI CAD domain to provide basic functionality (e.g., read circuit files and create graphs of connectivity) so that applications can focus on specific tasks (e.g., analyze power consumption and identify certain types of circuits). We refactored six cross-cutting concerns as aspects in two applications and seven in the third. There are fourteen distinct aspects in all.

The initial version of the `InstanceDrivers` application consists of 1,600 lines of code (LOC). The final revision has 3,300 LOC. This application identifies the instances (transistors) that *drive* electrical nets, providing electrical current.

The initial version of the `PowerAnalyzer` application contains 13,900 LOC and 16,600 LOC in the final revision. `PowerAnalyzer` estimates power dissipation of electric circuits. It consists of 3 smaller tools and a library of common code that the tools use.

The ErcChecker implements electrical circuit checks such as checking for proper transistor ratios between the pull-up and pull-down transistors of an inverter, checking for fan-out limits, and checking for drive strength problems. In order to understand (and ultimately correct) the violation, the circuit designer needs access to contextual circuit data, which the ErcChecker displays and writes to a log file. We aspectualized three revisions of the core application code, which has 51,600 LOC in the initial revision and 64,400 LOC in the final revision.

6.2 Concerns

We identified six cross-cutting concerns as aspects in the InstanceDrivers application: Excepter, CheckFwArgs, Tracing, Caching, and CadTrace and Singleton. We identified seven cross-cutting concerns in the PowerAnalyzer application: Excepter, Timer, CheckFwArgs, FwErrs, FetTypeChkr, ViewCache, and UnitCvrt. We identified seven cross-cutting concerns as aspects in the ErcChecker: Excepter, CheckFwArgs, FwErrs, Caching, ErcTracing, QueryPolicy, and QueryConfig.

Two aspects, CheckFwArgs and Excepter, are used by all three applications, and two others, Caching and FwErrs, are used by two applications. We created 14 aspects between the three applications.

Table 6.1 shows the applications associated with each aspect. The first column lists the aspect. The second column contains an I if the aspect was used in InstanceDrivers, a P if the aspect was used in PowerAnalyzer, and an E if the aspect was used in ErcChecker. The first letter used indicates the application for which the aspect was first created.

The CheckFwArgs and Excepter aspects were originally developed for the InstanceDrivers application. Reusing them in the PowerAnalyzer and ErcChecker applications required only a one line change in the pointcut of each aspect. The FwErrs aspect uses framework method names, which are the same in all applications,

Table 6.1: Aspect and Systems

Aspect	Used In System
Caching	E,I
CadTrace	I
CheckFwArgs	I,P,E
ErcTracing	E
Excepter	I,P,E
FetTypeChkr	P
FwErrs	P,E
QueryConfig	E
QueryPolicy	E
Singleton	I
Timer	P
Tracing	I
UnitCvrt	P
ViewCache	P

and so required no changes when reused in the `ErcChecker`. The `Caching` aspect was originally developed for `ErcChecker` and used in the `PowerAnalyzer`. Aspect inheritance is used to define the application-specific pointcut. We describe each of the aspects below except for the `Caching`, `CadTrace`, `QueryPolicy`, `QueryConfig`, and `Timer` aspects, which we described in Chapter 4.

6.2.1 CheckFwArgs

The `InstanceDrivers` application contains a utility layer that implements application-specific algorithms for framework objects. This layer calls framework API methods using framework object pointers that are passed in as parameters to the utility layer functions. Object pointers should be checked to confirm that they are not null to prevent fatal run-time errors. Null pointer checking is performed in some of the functions in the original application. The `CheckFwArgs` aspect removes duplicate checking code and improves safety by automating this check consistently across all callsites.

We used template meta-programming and the AspectC++ reflection API to implement the aspect, so that it can handle all framework utility methods even though they have different numbers of arguments and use several different framework and non-framework types as parameters. The aspect uses around advice, which returns if a null object is found, preventing the use of a null pointer with framework methods. The pointcut advises a group of functions without enumerating each one. We grouped the functions into a class as static methods, had the class inherit from an empty mixin class (C++ allows multiple inheritance), and created a pointcut that specified all the methods of sub-classes of the mixin class. We chose the mixin name so that it would indicate that an aspect was advising this code, which communicates the relationship between the primary code and an aspect to developers in a less oblivious way, similar to annotation-based weaving. We also created a C++ macro that redirected existing calls to the original function names to the static class methods of the same name.

6.2.2 ErcTracing

The `ErcTracing` aspect provides detailed tracing for the `ErcChecker`, replacing scattered inconsistent code. We had already developed the `Tracing` aspect, but because the `Tracing` aspect uses class data rather than global variables, we did not use aspect inheritance. The `ErcTracing` aspect contains seven advice blocks for different components in the `ErcChecker`. Each advice block uses a different global variable to indicate when tracing that subsection of code should be enabled. There are minor differences between the advice bodies, but all seven reuse the library of template metaprogramming code from the `Tracing` aspect to print method arguments.

6.2.3 Excepter

The `Excepter` aspect replaces similar code that checks return values from functions such as `getenv()` and `fopen()`. These functions use the return code idiom [11], i.e., they signal errors through return codes. The `Excepter` aspect provides a more modular approach for consistently handling errors than the scattered checks [57]. It uses after advice to check the return value of each function call. If the return code is null, the advice throws an exception. The aspect contains a second advice that provides a try/catch block around `main`. The try/catch block catches the exception thrown by the first advice and exits. If developers want to catch the error in the function call itself instead of relying on `main` to exit, they must manually add a try/catch block around that function call to catch the exception thrown by the first advice.

Both the `Excepter` and `CheckFwArgs` aspects provide error checking, which is an extra-functional concern. Such concerns relate to characteristics such as dependability, configurability, and performance [50].

6.2.4 FetTypeChkr

The `FetTypeChkr` replaces parameter checking that is done in the `FetType` class of the `PowerAnalyzer` application. The aspect uses the pointcut pattern-matching capability in AspectC++ to advise all methods of `FetType` that have an integer as the first parameter. The aspect advice validates that the range does not exceed the maximum value for the class, which is stored as a class attribute. The aspect replaces scattered checks on several methods with a single advice body to perform the same checking, thereby modularizing pre-condition contract checking for the class. This aspect is similar to design-by-contract checking aspects that have been explored by several researchers [9, 29, 73].

6.2.5 FwErrs

The `FwErrs` aspect refactors cross-cutting code for handling framework-specific failures. We identified this aspect based on idioms in code that detects these errors and handles system shutdown due to fatal framework errors. This aspect differs from the `Excepter` aspect, which is for non-framework system calls in C or C++ where the response to errors depends on the context of use. The `FwErrs` aspect, like the `Excepter` aspect, uses around advice to throw an exception when a return code from an advised method indicates an error. The aspectualized application relies on the aspect advice for `main` to catch the exception and exit.

6.2.6 Singleton

The `Singleton` aspect [32] modularizes similar code for a class that has multiple constructors for various contexts, but which all trigger the creation and sharing of a single instance. This aspect uses around advice with the core concern's static creation methods to ensure that a constructor is only called when the global instance has not already been created.

6.2.7 Tracing

The `Tracing` aspect refactors a concern commonly reported in the aspect-oriented programming literature – program tracing [65]. The `InstanceDrivers` application contains classes that have optional verbose output enabled via command-line arguments. The verbose mode causes method calls and their return values to be reported, as well as calls and return values of any functions called from within the traced methods. Besides being scattered throughout the class, such tracing code is often inconsistent. The `Tracing` aspect's around advice uses template-metaprogramming in order to process and print out parameter values and return values regardless of the number of parameters, parameter types, or return type.

6.2.8 UnitCvrt

For many scientific applications, units may be represented using different internal formats but must be written out with a standard unit of measure, such as nanometers or picofarads. The `UnitCvrt` aspect replaces scattered code that converts CAD property data into a standard unit of measure before printing the data in reports.

The CAD property data is stored as property objects that are attached to electrical net objects. Not all properties need their values converted to standard measures; only properties that store the length and width of a transistor need to be converted. This aspect uses around advice to intercept all calls to the `GetValue()` method of the `Property` class. The advice always calls `proceed()` so that the method is executed. Next, the advice calls the property's `GetName()` method and if required, the value is converted into a standard unit of measure.

6.2.9 ViewCache

The `ViewCache` aspect provides caching of different data views in the `PowerAnalyzer` application. We could not extend the `Caching` aspect [55] from `InstanceDrivers` to create the `ViewCache` because of differences in caching between the two applications. The `ViewCache` does not cache a value for a function or method call like the `Caching` aspect. Instead, it provides primary code methods with the capability to avoid reloading the necessary component views (e.g., schematic view, electrical view, and layout view) when those views already exist within the loaded framework data. The `ViewCache` does not aspectualize application-specific loading of component views performed during framework initialization, but accesses the framework data only to check what views are already loaded. The `ViewCache` aspect uses around advice to replace scattered code that determines if the different views are already loaded into memory during certain operations.

Chapter 7

Mock System Creation Guidelines

A key feature of our approach is the use of a mock system. In the ideal case, a mock system will have a structure that allows aspects to be moved *without change* from the mock system to the actual system for integration testing. Thus, an important question is how to create a useful mock system. Based on our experience with the legacy applications, we developed a list of mock system creational patterns that can aid the developer of a mock system. Although we provide specific examples for each pattern, the patterns are described in general terms and represent solutions to common problems in developing mock systems.

The patterns are based on characteristics of both the aspects and the real system. All of these patterns assume developers have identified potential joinpoints of aspects in the real system, which will be used as a guide when creating the mock system.

7.1 Create Mock Methods for Spectator Aspects

Spectator aspects are defined by Clifton and Leavens [16] as aspects that do not change the behavior of advised modules. Faults in spectators result in incorrect system behavior (e.g. missing or incorrect logging), but do not change the advised core concern.

7.1.1 Motivation

Because spectators do not rely on the internal state of the methods and classes they advise, we use method stubs: methods with empty or trivial bodies. Spectators are validated by ensuring that the pointcut matches the expected joinpoints and that the advice functionality executes correctly.

7.1.2 Mechanics

Create method stubs with naming conventions such that the pointcut will match in the mock system and the real system.

7.1.3 Example

The `Timer` aspect in the PowerAnalyzer is a spectator that adds timing information without affecting the power analysis performed. An error in the aspect may result in incorrect times, and pointcut strength faults may result in the wrong methods being timed, but the functionality of the PowerAnalyzer is not affected.

For the `Timer` aspect, methods to be timed must begin with `tmr` to match the pointcut: `call("% tmr%(...)̈));`. We created the mock system by writing method stubs that match this calling convention, which are called in `main`.

7.2 Create Simple Functional Mock Methods for Non-functional Concerns

Non-functional aspects [50] modularize cross-cutting concerns that improve non-functional characteristics such as performance or dependability without changing the existing functionality. Unlike spectators, faults in non-functional aspects can change the observed behavior of the advised concerns.

7.2.1 Motivation

We use mock methods that provide simple functionality when validating a non-functional aspect. This differs from the mock method stubs in Section 7.1 because non-functional concerns, such as caching, must not only advise the right joinpoints without introducing defects to the advised methods, but also must implement the cross-cutting concern without changing the existing functionality.

7.2.2 Mechanics

1. Create mock classes and methods with the same parameter types and return types but with simpler functionality.
2. Invoke the mock method with different parameter values from the `main` function to validate the non-functional property.

A mock system for caching values can create a function that operates on the same types. Statement coverage (or even compile-time enabled print statements) can be used to ensure the advice is used. For caching, we want to ensure that the advice executes correctly when it should avoid a pre-computed method call and when it should not. Unit testing can explicitly call a function with the same value multiple times and ensure the output is always correct.

7.2.3 Example

The caching aspect in the ErcChecker should improve performance without affecting functionality, but faulty advice can affect program modules. In addition, since caching is intended to improve performance, we want to validate that the cache is used whenever a function is called with a value that should already be cached.

In the ErcChecker, the cached methods had object pointers as parameters and returned object pointers and scalars as values. An example method is to calculate the fanout (integer) given an electrical net (bcmNet*). Thus, a mock system for caching with templates needs to validate that caching works correctly for various data types and that subsequent calls with the same input values are not recomputed.

We want functions that work on the same types, but are simple to compute and validate. Rather than calculating fanout of an electrical net, we can create a function (GetNetLength) that returns the length of the net's signal name.

```
int GetNetLength(bcmNet *n)
{
  return n->GetName().length();
}
```

We also create a similar function (GetWrongNetLength) with a different value, that is cached:

```
int GetWrongNetLength(bcmNet *n)
{
  return n->GetName().length()+10;
}
```

These functions need to be called in an interleaved fashion to validate that they are being cached separately, and that they return the correct value in each case. An example from the mock system is shown below:

```
//code that sets up the framework context
bcmCell *cell = GetTopCell();
//find 3 net objects...
bcmNet *net1 = FindNet( cell, "VDD" );
bcmNet *net2 = FindNet( cell, "GND" );
bcmNet *net3 = FindNet( cell, "clock" );

//make sure GetNetLength works,
//when not cached (first call)
//and when cached (second call)
assert ( GetNetLength(net1) == 3);
assert ( GetNetLength(net1) == 3);

//interleave calls to GetNetLength and
```

```
// GetNetLength to make sure they are not
// 'mixing' caches from different functions
// together
assert ( GetNetLength(net3) == 5);
assert ( GetWrongNetLength(net3) == 10);
assert ( GetWrongNetLength(net3) == 10);
assert ( GetNetLength(net3) == 5);
```

Code coverage tools, compile-time enabled debug output, or similar techniques can be used to ensure that the cache is actually used. In addition, the caching aspect can use an interface to the actual caching template so that a fast cache can be used as well as a slower cache that records hits and misses [55]. Calls to other data types that are cached were also included in the mock system.

7.3 Reuse Real System Components

Because the full system exists, components that are needed in the mock can sometimes be directly used.

7.3.1 Motivation

Often there is code in a large system, such as framework code, that is necessary to establish the initial state of the system before any advised methods are called. For example, a graphical system might have a common set of methods to create an environment for OpenGL or for a GUI windowing system. Our CAD applications require reading a netlist into memory before most framework methods may be called. We can import system components that are necessary for system initialization and copy in the small code sections that must be called to use these components. This avoids creating mock classes for large or complex components but still enables the mock system to have some actual functionality.

7.3.2 Mechanics

1. Use components from the main system that provide essential system components or services for the mock system. In C++, this can be done by including a header file and linking against a system or framework library.

2. Copy small code sections (e.g. less than 100 KLOC) that contain boilerplate code for using imported components.

7.3.3 Example

Aspects in the ErcChecker and PowerAnalyzer that rely on API calls or objects from HP's internal C++ framework were first created in mock systems. These mock systems reused a singleton class, `BlockData`, used by applications to provide a simple interface to framework methods for loading and initializing design data.

In the mock system, we included the `BlockData` class header and copied the code that uses it to initialize the system. Importing this framework component enables framework-related aspects to interact with framework objects in the mock system.

7.4 Emulate the Callsite Context for Joinpoints

The mock system should create a callsite context that is similar to the expected joinpoints in the real system.

7.4.1 Motivation

Callsite context refers to information that the aspect uses from the joinpoint, including method parameters and call flow information. In addition, changes to the control flow such as exceptions, method calls, and recursive calls should be identified.

The mock system should have methods that are similar to the real system in terms of number of arguments (zero, some, many) and argument types (e.g., simple scalar types, user-defined types, pointers, and templatized types such as STL containers.).

Aspects that have the potential for recursive advice invocation should be modeled in the mock system to ensure that the aspect structure and method calls are safe in such a context. Two common ways for recursion to occur are having an advised method call another method advised by the same aspect and advising a recursive method. The mock system can create intra-method calls and recursive calls so that accidental infinite recursion, a common mistake in adopting AOP [44] is avoided.

Aspects that throw or catch exceptions may be changing exceptions seen by callers as well as the resulting control flow. Thrown or caught exceptions can be simulated in the mock system.

7.4.2 Mechanics

1. Identify calling context passed in from the joinpoint, including parameter values and parameter types, especially templates and user-defined types. Create and advise mock methods with the same parameter types.

2. If advice can throw or catch exceptions, then model that in the mock system.

3. Identify call flow information (such as `cflow`) and recursion that exists in the real system and emulate it in the mock system.

7.4.3 Example

In the PowerAnalyzer, functions that were timed called lower level functions that were also timed. We modeled this structure in the mock system to ensure that the timing information for each function was correct and that the nested around advice of the aspect did not lead to problems. In the CachingAspect, we used the mock system to verify that the templatized cache used by the advice worked with a wide variety of built-in and user-defined data types.

7.5 Provide Functionality used by Advice

If an aspect's advice makes calls to external components or other parts of the system, then this functional context needs to be present in the mock system. Advice functional context is different than the pointcut context (Section 7.4), since the advice context refers to components used by the advice.

7.5.1 Motivation

To execute and validate the advice, the mock system must provide the methods called by the advice as well as any data structures used by the advice. A Logging or Tracing aspect, for example, may instantiate and use a Logger object, which must be present in the mock system. A policy aspect that refactors scattered code must have enough of the system data structures emulated for the advice to run in a meaningful way. The components needed by the advice may either be mock classes or, like components necessary for system initialization (Section 7.3), may be imported from the real system.

7.5.2 Mechanics

1. Identify any components called by advice.
2. Identify any data structures (classes, pointers to certain object types) used by the advice.
3. Create mock components or reuse components so that advice functionality can be validated.

7.5.3 Example

The advice of the `ErcChecker`'s `QueryPolicy` aspect performs logging of `ErcQuery` information, deletes `ErcQuery` objects that did not find electrical errors, and adds

`ErcQuery` objects that detect electrical failures to a container class (`LevelManager`). In order for the mock system to have enough functionality to validate the advice, the mock system needs to (1) create `ErcQuery` objects, (2) provide a `Logging` class that the advice uses, and (3) provide a mock `LevelManager` class. We created mock classes for all three of these requirements, implementing only the methods of the mock classes that are called.

Providing these enables the aspect to be behaviorally validated in the mock system. Using the same names for the `Logging` class and the mock container class (`LevelManager`) enables the aspect to function within the real system without changes to the advice. The `QueryPolicy` aspect affected more than fifty classes. During system refactoring, some issues were found that required changes to what system methods the advice used. We first emulated these changes in the mock system, then continued refactoring the real system.

7.6 Check Potential Pointcut Strength Faults

Faulty pointcuts that match too many or too few locations can be difficult to debug. By creating annotations in the mock system for methods that should and should not match a pointcut, we can guard against some of these faults.

7.6.1 Motivation

Pointcut strength faults [3] are particularly difficult to test for. One advantage that our annotations provide is that they are checked right after weaving, and do not depend on running the mock system. The mock system is created with annotated methods that are intended to be matched by an aspect as well as annotated methods that should not be matched. If multiple aspects exist in a mock system, our annotation approach (see Section 5.2) allows us to specify the specific aspect that should or should not advise a method.

7.6.2 Mechanics

1. Create names that should not match the pointcut but are similar, such as similar naming conventions or patterns.

2. Create namespaces in C++ or user-defined types that have similar naming conventions as pointcuts.

3. Model class hierarchies in the mock system to validate pointcut matching in base classes and subclasses.

4. Use annotations (Advised, NotAdvised) to check for incorrect pointcuts immediately after weaving.

7.6.3 Example

For pointcuts that use regular expressions such as

```
call("\% \%Iter\:\:Reset(...)\")
```

we should create multiple method calls that we intend to match (e.g., `InstIter::Reset`, `NetIter::Reset`).

In addition, namespaces can affect whether pointcuts match, since `%` in AspectC++ only matches one level (all classes) and not two levels of naming (all namespaces and all levels).

Annotations indicate and check that advice was not woven to some call sites and was woven to others, so that errors are caught immediately after weaving. Calls to the `Reset` method of an framework iterator were annotated to indicate the name of the aspect that should advise them:

```
while( (n=netIterPtr->Next()) ) {/*AOP=ADVISED_BY(PowerOnlyFilter)*/
```

The `ADVISED_BY(PowerOnlyFilter)` annotation is checked at weave-time to ensure that this line did have an associated advice joinpoint from the `PowerOnlyFilter` aspect. We have also used annotations to indicate functions that should not have any advice:

```
    void lookAtInsts(bcmCell *cell) /* AOP=NOADVICE */
```

These types of annotations can also be used in the real system, and serve as compile-time assertions that guard against accidentally changing the affect of pointcuts through either advice changes or core concern changes.

Aspects that interact with class hierarchies should be tested within a mock class hierarchy that, like method calls, includes join points that should and should not match. With annotations, we can ensure that a pointcut does match both mock base classes and mock derived classes. This can catch pointcuts that only match a base class when they should work with all classes, and can identify pointcuts that are too broadly defined.

7.7 Emulate Aspect-Aspect Interactions

When multiple aspects are used in a system, we can emulate the ways that we anticipate them interacting.

7.7.1 Motivation

Aspects may interact with one another by many means, including advising the same method and introducing members or methods to the same class [20]. These intended interactions should be modeled in a mock system by creating the conditions (e.g., overlapping pointcuts and introductions) that are anticipated in the full system.

Although the full system may contain interactions that were not anticipated, using the mock system can ensure that the aspects can work together correctly in at least some situations. Like unit testing, this provides a small environment to validate basic functionality before large-scale integration.

7.7.2 Mechanics

1. Identify and emulate aspects that share joinpoints.

2. After weaving aspects with the main system, use analysis of weave results to identify any unexpected shared joinpoints that should also be tested within the mock system.

7.7.3 Example

Although our aspects for these systems were non-overlapping, we did consider implementing caching with two overlapping aspects: one for function caching and one for hit/miss analysis of the cache.

The two aspects were created with identical pointcuts so that both would advise the same methods. In the mock system, we realized that order of execution was important, because both aspects used `proceed`, and if the caching aspect executed first and had stored the value already, it returned without calling proceed, which prevented the other aspect from intercepting the call as well.

One benefit of a mock system for testing aspect interference is that a small system can be set up with specific call orders. One drawback of mock systems is that there may be many complex scenarios that are not anticipated in the mock system.

Chapter 8

Evaluation of Mock Systems

We evaluate the costs and benefits of using mock systems in terms of our experiences. We evaluate the process in terms of four evaluation questions. The goal of the mock system evaluation is to answer the following research questions:

1. Can mock systems be developed for aspects that will be woven with legacy systems?

2. What costs are incurred in creating a mock system?

3. Does using mock systems save time when creating an aspect requires multiple iterations of our approach?

4. Did the aspects created using the mock system require changes to work with the real system?

We use version 1.0pre3 of AspectC++, which is a source-to-source weaver: weaving C++ code is followed by compiling the woven code. Our development environment for our tools is based on Linux and uses version 3.2.3 of the g++ compiler. We use version 3.2.3 of gcov (which depends on features of the g++ compiler) for measuring statement coverage.

Since we were able to create a mock system for each aspect, the answer to the first question was always "yes".

For each aspect, we answer the second question by reporting the lines of code and time required to create it. Lines of code includes only new code created for the mock system, not components reused from the real systems. We report time spent creating the mock system, either by writing new code or reusing existing components.

For the third question, we compare the time spent on creating the mock system to the compilation and weave time saved by using the mock system when multiple iterations were needed to create an aspect. We answer the fourth question by describing any changes made to pointcuts or advice when moving the aspects from the mock system to the real system.

8.1 Cost of Developing Aspects with Mock Systems

We show mock system cost measurements for the initial creation of each aspect in Table 8.1. The first column lists the aspect. The second column shows the size of the aspect (LOC). The third column shows the total time in minutes for all iterations used to create the aspect. The fourth column indicates the size (LOC) of the mock system. The last column shows the number of iterations it took us to create the aspect. The mock system creation time includes the time to create the aspect and mock system, and to iteratively test and refine the aspect using the mock system. If an aspect can be reused in more than one application, we do not repeat the mock system process, thereby decreasing the cost of future use of the aspect.

The `Caching` aspect required the largest mock system and took the most iterations because of the complexity of the caching functionality. It was created for a separate application, `ErcChecker`, which required caching of both procedural functions and object-based method calls. In addition, we created several specialized caches through aspect inheritance that can also track cache hit rates and cache usage to report performance [55]. Thus, there are 130 lines of code in the set of four caches for reuse

Table 8.1: Mock System Costs Per Aspect

Aspect	Aspect LOC	Mock System Creation Time (minutes)	Mock LOC	Mock Iterations
Caching	130	65	600	15
CadTrace	60	30	1	1
CheckFwArgs	29	50	76	3
ErcTracing	108	30	30	2
Excepter	28	50	125	5
FetTypeChkr	11	40	60	3
FwErrs	16	15	25	1
QueryPolicy	10	60	160	10
QueryConfig	14	30	20	3
Singleton	5	20	28	3
Timer	12	30	50	4
Tracing	62	90	150	8
UnitCvrt	26	75	122	5
ViewCache	4	15	33	1

through aspect inheritance. The concrete realization has only three lines of aspect code to select the cache and specify the pointcut. When the `Caching` aspect was reused in `InstanceDrivers`, it had 33 LOC (30 for the base aspect and 3 for the pointcut). The mock system was reused and thus, had no new costs.

The `ViewCache` aspect is a simplified type of cache. The experience with creating the `Caching` aspect enabled the rapid development of `ViewCache`; only one iteration was needed.

The `Singleton` aspect is also structurally similar to the `Caching` aspect. The `Singleton` aspect advises a static creation method and avoids executing it more than once. Our experience with the `Caching` aspect helped us create the `Singleton` aspect with just three iterations in 20 minutes.

The `Tracing` aspect took more time to create and more iterations than all but the `Caching` and `ErcTracing` aspect. The greater effort was due to the complex nature of the aspect, which uses C++ templates within the advice to check an arbitrary number of parameters. In addition, virtual methods were used so that the advice could print out objects of many different data types.

The `ErcTracing` aspect was created after `Tracing`. The mock system is larger, because the aspect contains seven advice blocks that advise different code components. Fewer iterations were required than for `Tracing` because of our prior experience developing `Tracing`.

We also used templates that call virtual methods to create the `CheckFwArgs` aspect. Since we had already used the technique for `Tracing`, we could create the advice faster for `CheckFwArgs`. However, developing the pointcut required several iterations because of the complexity of using template metaprogramming and mixins.

Developing the `Excepter` aspect required the third most iterations. The aspect must be woven with many functions that return several different data types (e.g., `bool`, `int`, and `int*`). In addition, we explored approaches for using aspects to locally catch exceptions for non-fatal cases [57].

The `FwErrs` aspect is similar to `Excepter`, which allowed us to develop `FwErrs` with fewer iterations. Since `FwErrs` advises just a few framework methods rather than several different functions, the mock system was also smaller.

For the `Timer` aspect, we created a mock system with function names beginning with `tmr`, and ensured that the advice correctly logged execution times of the functions. Several iterations were required to get the functionality implemented correctly. The `PowerAnalyzer` had nested timed functions, which caused the early versions of our aspect code to incorrectly reset the start time for the outer method when invoking the nested method. We emulated the nested function structure in our mock system and made changes in the aspect to correctly handle this case.

The pointcut for the `FetTypeChkr` aspect uses the class name and the type of the first argument so that only methods whose first argument was of type `int` would be matched. The mock system required 60 lines, most of which were in a class with the same name as the one in the real application. We needed two iterations to correctly define the pointcut because we had not previously used pointcuts that matched both names and argument types. A third iteration completed the advice implementation.

Because the `UnitCvrt` aspect advises functions that use the property objects defined in the VLSI framework, its mock system uses code from the VLSI framework. The mock system creates several framework objects. Some of them have properties with names that indicate a unit conversion, and thus, must be advised by the aspect. To ensure that the aspect did not apply unit conversion to unintended objects, the mock system created objects with property names that did not indicate unit conversion, and also objects with no properties added. We defined the correct pointcut in the first iteration. However, it took four iterations to create advice that would correctly and efficiently convert units of properties. This was a challenge because the properties are stored as strings. The advice must convert the string to a floating point number. Rather than storing the converted value as a string, the advice cached the converted value for each object. The next time the program tried to access a property of the object, the value was retrieved from the cache.

The `QueryPolicy` and `QueryConfig` aspects modularize cross-cutting code specific to the `ErcQuery` class, which is part of only the `ErcChecker`. In the mock system for `QueryPolicy`, we model the class hierarchy of the `ErcQuery` class. In addition, a driver file (`main.cc`) creates and executes query objects, storing results data internally. Once we had created the mock system for `QueryPolicy`, we were able to use it with only minor changes as the mock system for creating and testing the `QueryConfig` aspect.

Three factors influenced the amount of effort required to create and test an aspect using a mock system. First, aspects for which we needed to create larger mock systems tended to require more time to develop. We created larger mock systems when an aspect needed to be woven into more structures (e.g., several sub-classes of a common base class) or interacted with the application in a way that required more functionality than empty stubs. Mock system development time can be reduced by reusing system code when possible [59]. Second, the aspects vary in size and complexity. For example, because the `Caching` aspect used inheritance and cached values of methods and functions using a variety of data types, creating it took longer than `FwErrs`, which used no inheritance and checked the return value of the advised functions. At times, there is tension between aspect complexity and refactoring cost. For example, renaming methods to begin with `tmr` simplified the pointcut of the `Timer` aspect, but required more refactoring cost. By contrast, using mixins and inheritance with the pointcut added time to the development of the `CheckFwArgs` aspect, but reduced the amount of refactoring required. Third, gaining experience with features of AspectC++ reduced the time required for aspects that used those features. Creation times for the `ViewCache`, `FwErrs`, and `Singleton` aspects were all reduced because we became familiar with the AspectC++ features needed.

8.2 Time Saved using Mock Systems

Table 8.2 summarizes our time cost and savings data for each aspect. The 'Mock System Creation Time' column contains the time (in minutes) spent creating the mock systems, while the 'Application Weave Time' is how long weaving and compiling took in the original application that first used the aspect. Each mock system was able to be compiled and woven in one minute or less. The 'Mock Iterations' column is the number of iterations used to create the aspect. The 'Time saved' column is the total time saved in the aspect creation by using mock systems. We calculated this by multiplying the time saved for a single iteration -- Application Weave Time minus 1 (since the mock systems compile and weave in 1 minute or less) -- by the number of iterations ('Mock Iterations'), and then subtracting the cost of creating the mock system ('Mock System Creation Time'). For some aspects, such as `CadTrace`, the negative value indicates that there was not a time savings.

Table 8.2: Mock Systems Time Savings

Aspect	Mock System Creation Time (minutes)	Application Weave Time (minutes)	Mock Iterations	Time Saved (minutes)
Caching	65	25	15	295
CadTrace	30	6	1	-25
CheckFwArgs	50	6	3	-35
ErcTracing	30	25	2	18
Excepter	50	6	5	-25
FetTypeChkr	40	15	3	2
FwErrs	15	15	1	-1
QueryPolicy	60	25	10	180
QueryConfig	30	25	3	42
Singleton	20	6	3	-5
Timer	30	9	4	2
Tracing	90	6	8	-50
UnitCvrt	75	15	5	-5
ViewCache	15	15	1	-1

Because the `ErcChecker` has the longest compilation and weave times, the aspects created for this application (`Tracing`, `Caching`, `QueryPolicy`, and `QueryConfig`) all showed a time savings for using mock systems. This savings varied from 18 minutes to 295 minutes depending on the number of iterations needed to create the aspect.

Using mock systems for the `PowerAnalyzer` aspects (`Timer`, `FwErrs`, `FetTypeChkr`, `ViewCache`, `UnitCvrt`, and `CadTrace`) either resulted in a small savings of 2 minutes or cost of 25 minutes. Weave times was not the same for `PowerAnalyzer` aspects because not all aspects were woven into all of the subsystems of the `PowerAnalyzer`. Using mock systems for the `InstanceDrivers` aspects (`Excepter`, `CheckFwArgs`, `Tracing`, and `Singleton`) never saved time. Cost of using mock systems ranged from 5 to 50 minutes.

The time savings depends on the compilation and weave times of the original system and on the number of iterations required to create an aspect. For example, we used many iterations for the `Caching` aspect as we explored and evaluated different caching strategies. For the `QueryPolicy` aspects, the number of iterations results from the large number of classes involved in the refactoring and faults encountered during the refactoring.

8.3 Changes Made to Aspects for the Real System

Mock systems are intended to allow aspects to be developed and tested quickly and then moved without change to the target application. Here, we report on any changes that were made when moving aspects to the application or in later revisions of the application. Aspect changes can be of two types: pointcut changes and advice changes.

Table 8.3 lists each aspect in column one followed by the the kind of pointcut used in column two. Column three indicates the type of changes made, if any, to an aspect after it had been used in an application. The fourth and fifth column indicate the number of aspect defects or refactoring defects associated with an aspect.

Five types of pointcuts were used: pattern, list, mixin, class name, and method name. A pattern pointcut uses a regular expression, such as `tmr*` to indicate where advice is woven. A list pointcut enumerates each function or method that is advised. A mixin pointcut specifies the name of a mixin class, which advice targets inherit from. The mixin class provides no functionality to the class, but enables easy pointcut specification. A class name pointcut specifies all methods of some class are advised. A method name pointcut specifies a single method name, although the method may be part of many classes in a class hierarchy, such as the `executeQuery()` method in the `ErcQuery` subclasses of the `ErcChecker`.

Defects were made during the aspectualization of the `Timer` aspect and `Excepter` aspect. Both of these aspects required significant changes to be made to the appli-

Table 8.3: Post-mock Aspect Changes

Aspect	Pointcut	Changes	Aspect defects	Refactoring defects
Caching	List	pointcut	0	0
CadTrace	Pattern	none	0	0
CheckFwArgs	Mixin	advice	1	0
ErcTracing	Pattern	pointcut	0	0
Excepter	List	pointcut;advice	2	1
FetTypeChkr	Class Name	none	0	0
FwErrs	Method Name	none	0	0
QueryPolicy	Method Name	none	0	3
QueryConfig	Method Name	none	0	0
Singleton	Method Name	none	0	0
Timer	Pattern	none	0	1
Tracing	Class Name	none	2	0
UnitCvrt	Method Name	none	0	0
ViewCache	Method Name	none	0	0

cation when using the aspect. Thus, in addition to requiring more refactoring effort, aspects that require many source code changes have a higher risk of introducing defects during these changes.

8.4 Mock Systems Discussion

The mock systems for the four aspects were all small, with the compilation and weave times being dramatically less (one minute versus up to 25 minutes) than in the real system. None of the mock systems were difficult to create, with all taking an hour or less. The total time spent creating mock systems for the `ErcChecker`'s caching aspect was 65 minutes, but this was because two mock systems were created.

Even when mock systems do not save development time, they provide a more controlled environment for testing the aspect, just as traditional unit testing can focus more on a function or class before integration testing. For example, using a mock system for evaluating caching focuses solely on caching rather than testing the caching aspect as part of the full system. Other types of testing, such as performance or stress testing, can be done with mock systems, such as our testing of caching behavior and performance.

One additional cost of mock systems is that as an aspect evolves during development, any changes need to be mirrored and validated in the mock system or the mock system becomes stale. However, the mock system changes can be used in regression testing to validate that an aspect still functions as expected when either the aspect or system structure change.

One goal of mock systems is to create aspects that do not require change when woven with the real system. The only aspects that required pointcut changes were those that used an explicit list of functions to advise. If AspectC++ supported annotation-based weaving, the caching aspect pointcut change could have been avoided. Weaving based on annotations is different from our own annotations, which are used to check the weave results. Instead, languages such as AspectJ allow developers to use annotations as weave targets. While this requires the annotations to be inserted at all join points, it avoids depending on function names or naming conventions. If AspectC++ supported annotations, we could use annotation-based pointcuts for some aspects, and the mock system and real system could both contain annotations.

We do not have tools that help automate the creation of mock systems. Although tools can potentially extract classes or interfaces from the real system, engineering judgment is required when deciding what classes and methods are needed in the mock system.

8.5 Threats to Validity

The evaluation of the new approach demonstrates that it can be applied to legacy systems. However, like most case studies, it is difficult to generalize from a small-scale study. Thus, there are threats to external validity.

This study applied the aspectualization process to only three legacy systems. The legacy systems were not selected randomly, which limits external validity. Certainly, different results are likely when applying the process to other systems. Still this study demonstrates that the process can work, and the use of mock systems can lower costs and help to find errors.

Because of the limited nature of the evaluation, there are threats to internal validity, which is concerned with whether the new approach is actually responsible for the time savings and for revealing faults. One concern is that all of the aspect development and refactoring was performed by the same subject. This subject is also one of the developers of the approach, and clearly understands and believes in the approach. Others would not be biased, and might choose different aspects or implement them differently. In addition, the subject had development experience with the `ErcChecker`, and this potentially sped up aspect identification, refactoring, and mock systems development. For the `PowerAnalyzer`, although the developer did not participate in the design or development of the system, he did have some limited knowledge of the code based on making changes during maintenance.

Construct validity focuses on whether the measures used represent the intent of the study. We reported on whether or not aspect pointcuts or advice changed when moving from the mock system to the real system, since this is one way of measuring if the mock structure is similar enough to the real system. Other approaches might use structural code metrics or defects found to measure how effective mock systems are at providing an adequate environment for developing aspects. Time savings is a key dependent variable; it is based on compilation and weave times and how many iterations were used in the mock system to create an aspect. This is a reasonable way to measure effort. However, all iterations may not require an equal amount of time to complete.

Chapter 9

Evaluation of Maintainability

The costs of mock system-based aspectualization include the costs related to identifying aspects, developing mock systems, performing unit testing, creating aspect code, and changing primary code. If we used a different approach for aspectualization, the costs related to creating aspect code and changing primary code would be the same. We reported the cost of developing mock systems in Chapter 8 but the cost of identifying aspects is outside the scope of this dissertation. Thus, in this chapter, our focus is on the cost of aspect code creation and primary code changes.

Each application has a source code repository that contains the version history of each source file. For each application, we start at revision 1, and identify and refactor cross-cutting concerns as aspects so that we have revision 1-aop (aspectualized version of revision 1). We report the cost incurred when aspectualizing the first revision of an application. We compare the original and aspectualized code for any revision of the software, using metrics such as size, execution time, memory usage, and test coverage.

Next, we examine the changes made between the current revision (i.e. revision 1) and the next revision (i.e. revision 2), making the functionally equivalent changes in aspectualized revision 1 (i.e. 1-aop) to create revision 2-aop. Some of the changes made between revisions may also correspond to cross-cutting concerns that could be refactored as aspects. Thus, in addition to the aspects identified in the initial revision, aspects may also be identified based on the changes occurring after the first revision. We compare change-related metrics when going from revision N to revision N+1 on the original and aspectualized branch of the code. We follow this process over several revisions so that costs and benefits can be measured in terms of long-term maintenance.

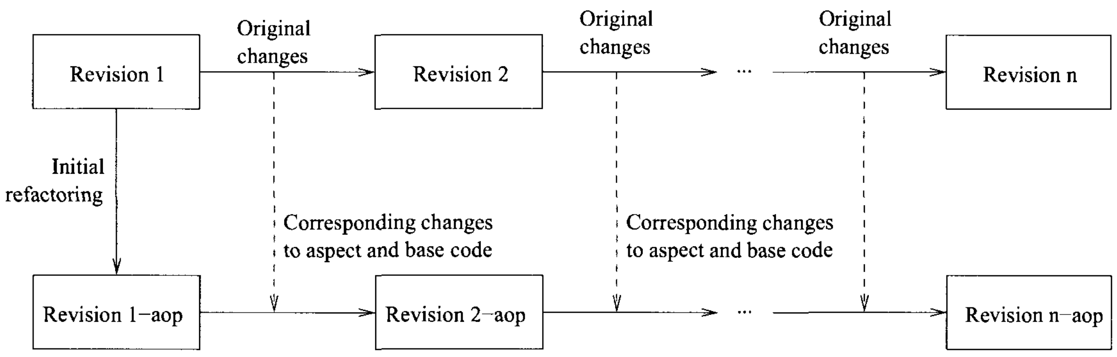

Figure 9.1: Evaluating Aspectualization using Revision History

Figure 9.1 shows the process. The rectangles represent revisions and the label, "Original changes", between revisions represent the changes that were made in the original application. Corresponding changes are made in the corresponding revisions of the aspectualized version.

9.1 Initial Refactoring Change Impact

We consider the code added, modified, or deleted when the aspect is *first* introduced to the application. The InstanceDrivers data is shown in Table 9.1. The first column shows the aspect name. The second column shows the size of the aspect (LOC), which is not a refactoring cost, but is shown to contrast with the changes made to the primary code for each aspect. The next three columns (Additions, Changes, Deletions) show how many lines were added, changed, or deleted in the original application during aspectualization. The last column is the total number of lines changed – the sum of the number of lines added, modified, or deleted.

Table 9.1: Instance Drivers: Source Code Changes Made

Aspect Name	Aspect Size(LOC)	Changes to primary code (LOC)			
		Additions	Changes	Deletions	Total
Caching	3	0	0	18	18
CheckFwArgs	29	13	7	0	20
Excepter	28	0	0	1	1
Singleton	5	0	0	11	11
Tracing	62	0	0	16	16
Total	127	13	7	46	66

For the `InstanceDrivers` application, 66 lines were changed, of which, 46 (more than half) were deletions. Four of the aspects required only deletions. The `CheckFwArgs` aspect needed the most additions because its pointcut required that the advised functions be refactored as static methods of an advised class. The number of deletions is greater than the size of the aspect for most aspects.

The `Caching` aspect uses aspect inheritance to reuse a more general caching aspect from a collection of abstract caching aspects that we had created for a separate application [55]. Although the base aspect size is 30 LOC, the table shows the size (3 LOC) of the application-specific derived aspect.

The largest aspect, `Tracing`, required more lines to implement than the number of lines removed (16). However, during subsequent revisions, the use of `Tracing` resulted in 66 lines being removed by the last revision.

The `Excepter` aspect allowed us to delete one line in the `InstanceDrivers` code because the error checking it provided was only implemented as a line check in one location in `InstanceDrivers`. The `CheckFwArgs` removed no lines because none of the checks it performs were implemented in the initial revision.

Table 9.2 shows the changes made to the `PowerAnalyzer` during aspectualization. As with Table 9.1, we list each aspect, its size, the number of additions, changes, and deletions, and the total number of changes.

Table 9.2: PowerAnalyzer: Source Code Changes Made

Aspect Name	Aspect Size(LOC)	Changes to primary code(LOC)			
		Additions	Changes	Deletions	Total
CadTrace	8	0	0	0	0
Excepter	28	17	57	112	186
CheckFwArgs	29	18	4	81	103
Timer	12	34	20	79	133
FwErrs	16	0	0	16	16
FetTypeChkr	11	0	0	18	18
ViewCache	4	0	1	21	22
Total	123	69	82	327	478

The `Timer` aspect required the most additions and the second most changes. The additions represent the lines that we added when we used the Extract Method Refactoring approach to refactor blocks of code as new functions so that these functions could begin with the `tmr` prefix. The changes represent the modifications we made to the code that called existing functions that were renamed.

The `Excepter` aspect required the most changes when code for checking and handling return values was removed, but it also resulted in the largest number of deleted lines of code from the `PowerAnalyzer`.

The lines added or modified in `PowerAnalyzer` for the `Excepter` aspect represent cases where we added local try/catch blocks to prevent applications from exiting on errors. The additions for the `CheckFwArgs` aspect are for converting the advised functions to static methods of classes. The `CadTrace` aspect does not replace any application code, but instead adds tracing information used to debug an application error. During normal use, it is disabled.

The `ViewCache` aspect required a change to one line of code and the removal of 21 lines, while `FetTypeChkr` and `FwErrs` each deleted fewer than twenty lines of code without requiring any change to the original application. The `UnitCvrt` aspect is not shown in Table 9.2 because the concern it represents was not implemented in the initial revision. Instead, the concern was introduced during the transition from revision two to revision three.

Table 9.3 shows the changes made to the `ErcChecker` during aspectualization. We list each aspect, its size, the number of additions, changes, and deletions, and the total number of changes.

Table 9.3: ErcChecker: Source Code Changes Made

Aspect Name	Aspect Size(LOC)	Changes to primary code(LOC)			
		Additions	Changes	Deletions	Total
Caching	130	0	0	184	184
CheckFwArgs	29	0	3	33	36
Excepter	28	16	11	24	51
ErcTracing	108	0	0	1057	1057
FwErrs	16	0	0	24	24
QueryPolicy	10	73	217	335	625
QueryConfig	14	0	0	57	57
Total	335	89	231	1714	2034

The `QueryPolicy` aspect enabled many deletions by aspectualizing common policy code for the `ErcQuery` class. The changes and additions required during refactoring varied among the different subclasses of `Erc Policy`[56]. For 15 subclasses, we only made deletions. We added a method to 18 subclasses to consistently manage memory. The subclass consistency is necessary because the `QueryPolicy` aspect applies the same policy advice to all subclasses. In addition, six subclasses perform electrical checking in multiple phases, requiring that more than one location within the class be refactored.

Both the `QueryConfig` and `FwErrs` aspect enabled deleting about the same number of lines as the size of the aspect and required no changes and additions. The `CheckFwArgs` also removed approximately the same number of lines of code as the size of the aspect; the 3 lines added were to move functions into classes as static methods so that the aspect pointcut use the class name rather than a list of functions.

The Caching aspect was the largest aspect used in the ErcChecker, but allowed more deletions than all but the ErcTracing aspect. The ErcTracing aspect provided the largest removal of core concern code of any aspect in the three applications.

9.2 Size

We compare the size of the original and aspectualized applications. We also show the *equivalent* application size, which represents the size the original application would have if concerns that were not fully or consistently implemented in the primary code had been included. In InstanceDrivers, these concerns were related to the Excepter, Tracing, and CheckFwArgs aspects. In the PowerAnalyzer, these concerns were related to the Excepter, CadTrace and CheckFwArgs aspects. The concerns implemented inconsistently in ErcChecker were CheckFwArgs, Excepter, and ErcTracing.

Using the size of the equivalent implementation allows us to compare the aspectualized solution to the amount of code that would have been needed to achieve the same level of checking and tracing in the original application.

In the equivalent implementation, the code would be added at each location advised by the aspect. Thus, we calculated the equivalent size by multiplying the number of joinpoints advised by the aspect with the number of lines of code required in C++ to implement the advice. The size of the aspectualized application includes the refactored C++ code, the aspects, and any support code that is invoked from within the aspect advice.

The data for the six InstanceDrivers revisions is shown in Table 9.4. Each row in Table 9.4 represents a revision, with the revision number shown in column one. The second column shows the size of the original C++ source code, and the third column shows the equivalent size. The 'AOP' column shows the size of the aspectualized application. The last column is the size of the aspectualized application (AOP) minus

Table 9.4: InstanceDrivers: Code Size (LOC)

Revision	Original	Equivalent	AOP	AOP savings
1	1619	1972	1924	48 (2.4%)
2	1672	2025	1974	51 (2.5%)
3	1823	2209	2130	79 (3.5%)
4	2967	3473	3236	237 (6.7%)
5	3008	3544	3269	269 (7.5%)
6	3155	3743	3395	348 (9.3%)

the size of the equivalent application. This number represents the reduction in size achieved by using aspects and shows that this size reduction grew over time as the size of the application grew.

Table 9.5: PowerAnalyzer: Code Size (LOC)

Revision	Original	Equivalent	AOP	AOP savings
1	13,931	14,066	13,734	332 (2.4%)
2	14,861	15,000	14,638	362 (2.4%)
3	15,183	15,322	14,959	363 (2.4%)
4	15,356	15,499	15,128	371 (2.3%)
5	15,727	15,876	15,482	394 (2.5%)
6	16,412	16,567	16,118	449 (2.7%)
7	16,600	16,755	16,294	461 (2.8%)

Table 9.5 contains the data for the `PowerAnalyzer`. Unlike `InstanceDrivers`, the concerns implemented as aspects were all present in the initial revision of the `PowerAnalyzer`. The last column, 'AOP savings', is cumulative: if 332 lines are removed due to aspectualization in revision 1 and 30 lines are removed due to aspects in revision 2, then revision 2 will show the aggregate savings of 362 lines of code.

Some cross-cutting concerns, such as tracing, had new code added at each revision of the original application. Thus, aspectualization results in code savings with each revision because the changes are not scattered but are restricted to the `Tracing` aspect. This increased savings in later revisions was particularly apparent for `InstanceDrivers` with the `Tracing` aspect.

In the PowerAnalyzer, 72% (332/461 lines) of the AOP savings were realized with the initial revision. The initial benefit was much larger for PowerAnalyzer 332 lines) than for InstanceDrivers (48 lines); aspectualization saved more code in the initial revision than the size of the aspects themselves.

For the CheckFwArgs and Tracing aspects, we created a 200 LOC library that provides all the type-specific printing and checking methods needed to handle all the datatypes advised. The library that can be accessed by any datatype-based aspect. Its development represents a one time cost for these aspects. The library made the aspectualized InstanceDrivers application larger than the original version, but with increased functionality.

The data for three revisions of ErcChecker is shown in Table 9.6. In the initial revision, no checks were originally provided. There were three cases where a return code was not checked; we corrected this using the Excepter aspect. There were 320 methods in the classes that used framework pointers without checking for null pointers; using CheckFwArgs fixed them. In revision four, we identified four checks that were missing in the original code; two were fixed by using the Excepter aspect and two were fixed by using the CheckFwArgs aspect.

Table 9.6: ErcChecker: Code Size (LOC)

Revision	Original	Equivalent	AOP	AOP savings
1	51,692	52,664	50,492	2172 (4.1%)
2	54,137	55,132	52,845	2287 (4.1%)
3	64,145	65,154	62,608	2546 (3.9%)

Like PowerAnalyzer, most of the AOP savings is realized in revision 1 of the ErcChecker. The ratio is even more pronounced for the ErcChecker, with 85% (2172/2546 lines) of the saving from revision 1. Aspectualization removed more code for the ErcChecker, which was by far the largest application. In terms of percentage saved, ErcChecker had a large percentage saved than PowerAnalyzer but saved less than half of the percentage in the final revision of InstanceDrivers.

9.3 Performance

We present the data for woven code size along with memory and execution time performance data in Tables 9.7 and 9.8. The first column lists the revision. The second column shows the multiplicative increase in the number of lines of the woven code. For example, 4.3x means that the woven code is 4.3 times larger than the original code. The third column shows the percent increase in compiled object code. The fourth column shows the percent increase in regression test execution time. The fifth column shows the percent increase in memory usage.

Table 9.7: InstanceDrivers: AOP Performance Data

Revision	Code Increase	ObjCode Increase	Run Time Increase	Memory Increase
1	4.3x	+54%	+5.1%	+2.4%
3	4.4x	+91%	+5.0%	+2.1%
4	3.9x	+119%	+14.0%	+11.1%
6	4.2x	+123%	+18.0%	+15.0%

Two factors explain the increases in `InstanceDrivers` memory usage and execution time for later revisions. First, more aspects were added to the application. Revision six has more aspect code woven in than revision one. Second, the regression tests evolved with the application. In the fourth revision, tests were added that had tracing enabled by default unlike some of the previous tests. Most of our aspects result in only a small increase in woven code size and have negligible effects on execution time and memory usage. However, the `Tracing` and `CheckFwArgs` aspects increase execution time and memory usage due to the overhead of additional method calls to check and print method parameters. The size of the compiled object code increased until it was more than double (+123%) the original size. This *code bloat* is similar to what can occur when C++ programs with templates are compiled and is much larger than the increase in execution time or memory usage.

The `PowerAnalyzer` data is shown in Table 9.8. The increases in `PowerAnalyzer` execution time, memory usage, and object size were less with later revisions, which we

Table 9.8: PowerAnalyzer: AOP Performance Data

Revision	Code Increase	ObjCode Increase	Run Time Increase	Memory Increase
1	1.78x	+30%	+10.0%	+14.0%
3	1.80x	+29%	+9.1%	+12.1%
5	1.79x	+29%	+9.0%	+12.0%
7	1.80x	+29%	+9.0%	+12.0%

attribute to the non-aspect code growing faster than the amount of aspect code in later revisions. Thus, by revision seven, the aspects made up a smaller overall percentage of the code. The increase in the code size of the `PowerAnalyzer` was less than half the increase of `InstanceDrivers`. This difference was due to the greater use of aspects that employ template metaprogramming to recursively expand argument lists into individual typed values. Although template metaprogramming led to high code bloat in the `InstanceDrivers`, it was used in aspects (`Tracing` and `CheckFwArgs`) that are called infrequently during application regression tests, and have minimal impact on performance. The `PowerAnalyzer` aspects, such as `Excepter`, intercept calls that are made frequently during regression tests, which causes a higher performance overhead.

In the `PowerAnalyzer`, memory usage and execution times both consistently increased by about 10%, primarily due to the consistent error checking implemented in the `Excepter`, `CheckFwArgs`, `FetTypeChkr`, and `FwErrs` aspects. The execution time also increased because the `CheckFwArgs` aspect processes each parameter; framework pointers are checked and non-framework parameters are ignored. Calling the empty virtual 'no-op' function for non-framework parameters also introduces some overhead. These checks provide improved error detection and traceability without an order of magnitude increase in execution time and memory use, and reduce code size by eliminating scattered code to provide the same checks.

When we performed the original regression, test data for `ErcChecker` was no longer accessible to us for proprietary reasons. Thus, we only present woven code size and object code size data for the `ErcChecker`, which is shown in Table 9.9.

Table 9.9: ErcChecker: AOP Woven Size Data

Revision	Code Increase	ObjCode Increase
1	4.8x	+125%
2	4.7x	+123%
3	5.1x	+136%

The `ErcChecker`'s increases in woven code size and compiled object size are similar to `InstanceDrivers`. The `ErcChecker` makes extensive use of template-based aspects, as does, `InstanceDrivers`, which increases both woven code size and object code size.

We present joinpoint data for `InstanceDrivers`, `PowerAnalyzer` and `ErcChecker` in Table 9.10, using "n/a" for aspects that were not used in an application, and thus, are not applicable. This data is from the last revision of each application, thus all aspects were present in at least one of the applications. We see from the table that the `Excepter` was woven into more places in the `PowerAnalyzer` than the `Instance-Drivers` and `ErcChecker`. This explains why it had a greater impact on execution time and memory usage of the `PowerAnalyzer`.

The `Tracing` aspect advises many joinpoints in `InstanceDrivers`, resulting in a large increase in the woven code size. Similarly, the `ErcTracing` aspect and `CheckFwArgs` aspects were woven in many joinpoints of `ErcChecker`. Analysis of `ErcChecker` individual regression runs confirmed that when only the `Tracing` aspect was disabled, average memory and execution time increases were only between 2-4% rather than 9-10%. Since tracing is only enabled in rare cases for debugging program or circuit problems, the average case for `InstanceDrivers` is the 2-4% increase. Since the verbose mode is only used for debugging, we assert that the larger performance increase in that case is acceptable because the aspect provides more thorough debug output.

Table 9.10: Joinpoint Data for all Applications

Aspect	Instance-Drivers Joinpoints	Power-Analyzer Joinpoints	ErcChecker Joinpoints
Caching	3	n/a	27
CadTrace	n/a	8	n/a
CheckFwArgs	9	54	338
ErcTracing	n/a	n/a	211
Excepter	11	91	11
FetTypeChkr	n/a	14	n/a
FwErrs	n/a	9	12
QueryPolicy	n/a	n/a	45
QueryConfig	n/a	n/a	58
Singleton	4	n/a	n/a
Timer	n/a	34	n/a
Tracing	115	n/a	n/a
UnitCvrt	n/a	10	n/a
ViewCache	n/a	10	n/a

9.4 Change Locality

We measure change locality by comparing the number of modules and files that we modified when going from revision `N` to revision `N+1`. A lower number indicates better change locality. We also counted the number of lines changed to go from one version to the next.

Each row in Table 9.11 represents moving from one revision to the next one. For example, '1-2' in the 'Revisions' column means changes when moving from revision one to revision two. The 'Lines' and 'Lines(AOP)' columns show how many lines were modified in the original application and in the aspectualized application, respectively. Similarly, the 'Modules' and 'Modules(AOP)' columns compare how many modules were modified. Last, the 'Files' and 'Files(AOP)' columns compare how many source files were modified in the original and aspectualized application. The table shows that better change locality (i.e. lower number of modules and files changed) generally corresponds with fewer lines changed.

Table 9.11: InstanceDrivers: Change Locality

Revisions	Lines	Lines (AOP)	Modules	Modules (AOP)	Files	Files (AOP)
1-2	56	53	3	2	3	2
2-3	148	149	8	8	9	9
3-4	1155	1125	26	26	14	14
4-5	59	49	10	8	9	8
5-6	158	109	13	9	5	5

The increase in lines changed when going from revision two to revision three was caused by the implementation choice in the `CheckFwArgs` aspect: we refactored the functions needing their framework pointers to be checked to static class methods, and added an additional wrapper class to hide this change from the rest of the application. This increase was small, and had the original code used methods of a class rather than functions, the AOP cost would be less. For all other revision changes, the aspectualized `InstanceDrivers` application had fewer line changes.

Aspects improved module change locality in three revisions and file change locality in two revisions. Module and file change locality were never worsened by aspectualization. However, even when there were improvements, they tended to be small. This was because each new revision included changes in several core and crosscutting concerns at once. While aspects localized the changes in cross-cutting concerns, the changes in the core concerns still had to be implemented in several modules and files.

The `PowerAnalyzer` change locality data is shown in Table 9.12. As in the `InstanceDrivers`, change locality either showed improvement or remained the same.

The `ErcChecker` change locality data is shown in Table 9.13. As in the `InstanceDrivers` and `PowerAnalyzer` applications, aspectualization provided a small decrease in module and file change locality. The small improvements were due to changes avoided because of the `ErcTracer` aspect. Like the `Timer` aspect in `InstanceDrivers`, the concerns aspectualized by `QueryPolicy` and QueryConfig had

Table 9.12: PowerAnalyzer: Change Locality

Revisions	Lines	Lines (AOP)	Modules	Modules (AOP)	Files	Files (AOP)
1-2	940	879	78	75	24	23
2-3	322	320	6	5	4	3
3-4	173	169	21	19	15	13
4-5	371	354	55	53	16	16
5-6	685	636	77	75	21	21
6-7	188	178	7	5	5	5

few changes during the evolution of the `ErcChecker`. Thus, these did not decrease module and file change locality.

Table 9.13: ErcChecker: Change Locality

Revs	Lines	Lines (AOP)	Modules	Modules (AOP)	Files	Files (AOP)
1-2	4,638	4,546	112	109	47	46
2-3	16,967	16,722	317	310	105	103

9.5 Concern Diffusion

In Table 9.14 we list each aspect used in `InstanceDrivers`, the number of concern switches removed by the aspect, and any new concern switches that occur when aspects are used. Thus, although we do not measure concern diffusion over lines of code (CDLOC) directly, we measure the difference in CDLOC using the final revision of each application to manually compare the concern switch differences. The numbers in parentheses represent concern diffusion values when we consider equivalent functionality.

The `CheckFwArgs` aspect lists two values in the reduction column: 0 and 18. The value, 0, reflects that no checking was done in the original code, hence no concern switching actually occurred. The value, 18, represents the number of concern switches (2 switches per method in 9 methods) that would have occurred if the concern had been implemented in every place (equivalent functionality). The addition of two new

Table 9.14: InstanceDrivers: Concern Diffusion

Aspect	Concern Switch Reduction	New Concern Switches
Caching	16	0
CheckFwArgs	0 (18)	2
Excepter	16	0
Singleton	6	0
Tracing	78	0
Total	116(134)	2

concern switches reflects the restructuring that we performed in the core concern to allow us to use a simple pointcut as described in Section 6.2.1.

For the `InstanceDrivers` application, even without considering equivalent functionality, 116 concern switches were removed and only two were added.

Table 9.15: PowerAnalyzer: Concern Diffusion

Aspect	Concern Switch Reduction	New Concern Switches
CadTrace	72	0
CheckFwArgs	84	8
Excepter	98	30
FwErrs	18	0
FetTypeChkr	28	0
Timer	68	0
UnitCvrt	38	0
ViewCache	20	0
Total	426	38

We show the reduction in concern diffusion of the final revision of `PowerAnalyzer` in Table 9.15. Overall, concern diffusion decreases in `PowerAnalyzer`. The `CadTrace` reduction reflects that the equivalent amount of debugging would have resulted in 2 switches before and 2 switches after each of the 18 joinpoints. The `Excepter` concern has a net decrease of 68 concern switches. At the locations where 30 concern switches were added, a concern switch was also removed at the same location. Adding and removing a concern switch was done when aspectualizing error handling. In the original application, errors in functions (such as `fopen`) returned error codes that would be checked as follows:

```
fp = fopen(filename,mode);
if(fp==NULL) {
   //error handling code
}
//back to regular code
```

This shows two concern switches: (1) error-handling, for when the variable `fp` is NULL, and (2) the primary code concern for which the file pointer `fp` will be used.

For applications to catch errors locally, the developer must add a try/catch block around the function (`fopen`) rather than checking the return value, as shown below:

```
try {
   fp = fopen(filename,mode);
}
catch(anErrorOccurred e) {
   //error handling code
}
```

This represents two concern switches (switching to and from error handling code in the catch block). The net reduction in concern switches is 316 because Table 9.15 shows 354 switches were reduced and 38 were added.

We show the reduction in concern diffusion of the final revision of `ErcChecker` in Table 9.16.

The `CheckFwArgs` and `Excepter` aspects list two values in the reduction column. The smaller value reflects the concern switches reduced in the original code. The larger parenthesized value represents the number of concern switches that would have occurred if the concern had been implemented in every place (equivalent functionality).

Each aspect reduced the number of concern switches. The `QueryPolicy` aspect required adding a new virtual method to the `ErcQuery` base class, and 14 of the 58

Table 9.16: ErcChecker: Concern Diffusion

Aspect	Concern Switch Reduction	New Concern Switches
Caching	216	0
CheckFwArgs	24 (38)	0
Excepter	14 (668)	8
ErcTracing	598	0
FwErrs	12	0
QueryPolicy	78	57
QueryConfig	114	0
Total	1,056 (1,724)	65

classes had to override this method. The changes to the base class and 14 sub-classes are where the 57 new concern switches were added. The `Excepter` class reduced 14 switches, but 8 new concern switches were added as local try/catch blocks to handle some errors locally rather than exiting from the application. All other aspects reduced concern switches without adding new concern switches with a total of 1,020 concern switches reduced in the original application.

9.6 Test Coverage

We describe the challenges and trends we observed when we gathered test coverage data for four revisions of `InstanceDrivers` and PowerAnalyzer. As previously mentioned, regression data was not available for `ErcChecker`.

In Table 9.17 we show statement coverage data for four revisions of `Instance-Drivers`. The first column lists the revision, while the second and third column show missed lines of code and total lines for the original application, and the fifth and sixth columns show missed and total lines of woven code of the aspectualized application. The fourth and seventh columns show the percentage of missed lines of code for the original and aspectualized applications. For this study, we used the existing regression tests as is; we did not change them to try and achieve 100% coverage in the original or aspectualized applications.

Table 9.17: InstanceDrivers: Statement Coverage

Revision	Missed lines	LOC	Percentage	Missed lines(AOP)	LOC (AOP)	Percentage (AOP)
1	102	1619	6%	624	7082	8.9%
3	116	1767	7%	685	8040	8.6%
4	294	2967	10%	428	11578	4.0%
6	320	3155	10%	538	13140	4.0%

From Table 9.17, we observe that the aspectualized application always had more lines of missed code. The missed lines correspond to the primary code that did not get executed as well as the aspect code. If statements in the primary code are not executed by the test inputs, then any associated joinpoint-specific code will also be missed. Moreover, the woven code can also contain unreachable statements. Each joinpoint has methods, such as `Joinpoint::signature()`, that can be used in the advice body of the aspect. If these joinpoint methods are not used by any advice, they are unreachable and will be marked as not covered by statement coverage tools.

Table 9.17 reflects two significant changes that occurred in revision four of `InstanceDrivers`. The first change was that more than 1000 lines of code were added, the largest one-time increase for any revision. Secondly, because of the changes in the original code, the regression tests were updated by the original developers to test additional functionality, including more testing of the verbose-mode functionality, which provided much better testing for the `Tracing` aspect. Although the number of lines missed in revision four was still greater in the aspectualized version, the aspectualized version had a lower percentage of misses because previously missed `Tracing` aspect code was now being tested.

Test data for four revisions of `PowerAnalyzer` is shown in Table 9.18. When we performed the study, about half of the original regression test data for `PowerAnalyzer` was no longer accessible to us for proprietary reasons. The missing regression test cases lower the coverage. Because our focus was on the aspectualization, attempting

Table 9.18: PowerAnalyzer: Statement Coverage

Revision	Missed	LOC	Percentage	Missed (AOP)	LOC (AOP)	Percentage (AOP)
1	2435	13,931	17.4%	3500	24,744	14.1%
3	2592	15,183	17.1%	3811	27,292	14.0%
5	2790	15,727	17.7%	3995	28,151	14.2%
7	3205	16,600	19.3%	4467	29,760	15.0%

to obtain the old test cases to construct new test cases was beyond the scope of this study.

We were able to run `gcov` with the aspectualized application. However, the woven code contains new code generated through aspect weaving and an extra level of indirection for the advice to intercept and advise methods and functions. This made the report of missed lines more difficult to understand.

Table 9.19: InstanceDrivers: Joinpoint Coverage

Revision	Covered Joinpoints	Total Joinpoints	Percentage
1	54	77	70%
3	54	84	64%
4	85	114	75%
6	107	145	74%

Table 9.19 shows joinpoint coverage for the same revisions for which we reported statement coverage. The first column contains the revision number, the second column shows the number of covered (executed) joinpoints, while the third column shows the total number of joinpoints. The last column (Percentage) is the percentage of joinpoints covered. At revision three, the joinpoint coverage decreased because there were new joinpoints in the application which were not executed by the regression tests, even though the old joinpoints were covered. At revision four, when the regression tests were improved so that the `Tracing` aspect was executed more often, we see an increase in joinpoint coverage.

Table 9.20: PowerAnalyzer: Joinpoint Coverage

Revision	Covered Joinpoints	Total Joinpoints	Percentage
1	93	155	60%
3	113	183	62%
5	122	192	64%
7	134	211	64%

Table 9.20 shows joinpoint coverage for the PowerAnalyzer. Unlike Instance-Drivers, no changes occurred in the regression test suite of PowerAnalyzer between revisions. Thus, joinpoint coverage percentage for PowerAnalyzer remained about the same across all revisions.

9.7 Defect Tracking

We are interested in understanding what types of defects might be avoided with aspects, as well as what types of defects occur when using aspects or aspectualizing a application. For each revision, Table 9.21 shows the number of defects found in the original InstanceDrivers during aspectualization and the number of defects that were introduced (and fixed) when aspectualizing the application.

Table 9.21: InstanceDrivers: Defects

Revision	Defects in Original Application	Defects Introduced When Aspectualizing Application
1	2	2
2	0	0
3	0	1
4	2	0
5	0	0
6	0	0

In the original InstanceDrivers application, the two defects found in revision one were both failures to check the return code of getenv() calls for errors. These were fixed by using the Excepter aspect. One defect in revision four was also related to not checking `getenv()` error codes. The second defect was that the caching mechanism, which was manually implemented, did not cache the result in one case. This defect resulted in lower performance but did not give incorrect results. This was fixed when we used our `Caching` aspect.

The first defect that we introduced in revision one was caused by the `Tracing` aspect, which accessed a class member variable when it advised a static method resulting in a null pointer access. The defect was fixed by making sure that we checked the '`this`' pointer of the advised object in the aspect so that member data was only inspected on non-static methods.

The second defect we introduced in revision one was that the template-based C++ support code for checking methods with an arbitrary number of parameters did not work with zero-argument methods. We tested the aspect with a variety of parameter types using a small mock system [59], but this testing did not include a zero argument method.

At revision three, the woven code would not compile. The reason was that the `CheckFwArgs` aspect called a method that had to be defined and overridden for each framework pointer type used. The use of a new framework pointer as an argument caused a compilation error; we corrected this by defining the method for the newly used type.

We show the number of `PowerAnalyzer` defects in Table 9.22. The defect found using the `CadTrace` aspect is not included, because the defect is not removed or avoided using aspects; instead, the aspect was used to avoid adding additional code to find the source of a defect.

In revision one, of the 21 original defects, 12 resulted from failure to check return values, while nine defects resulted from failure to check for null framework parameters. We removed these defects when we used the `Excepter` and `CheckFwArgs`. In revisions two and four, the original code contained defects due to missed unit conversions, which

Table 9.22: PowerAnalyzer: Defects

Revision	Defects in Original Application	Defects Introduced When Aspectualizing Application
1	21	2
2	1	0
3	0	0
4	2	0
5	2	0
6	2	1
7	0	0

were fixed when we used the `UnitCvrt` aspect. Also, in revision four the original code had a defect in the way the `TimeEvent` class was used. We fixed that by using the `Timer` aspect. Revisions five and six each contained two defects from failing to check for null framework parameters.

We introduced two defects when we aspectualized revision one. One defect was due to an incorrect pointer reference when we added a method for the `Timer` aspect, while the other was due to incorrect try/catch logic added to handle an exception locally in conjunction with the `Excepter` aspect. We introduced a defect in revision six where one of the methods that was supposed to be advised by the `Excepter` aspect was not matched by the pointcut. We modified the aspect pointcut to include the new function.

Table 9.23: ErcChecker: Defects

Revision	Defects in Original Application	Defects Introduced When Aspectualizing Application
1	325	2
2	8	1
3	5	1

In revision one, we removed two defects from the `ErcChecker` when aspectualizing: one `ErcQuery` that did not write its results to a log file, and one `ErcQuery` that could not be disabled with run-time configuration commands. We fixed the first defect by using the `QueryPolicy` aspect and fixed the second using the `QueryConfig` aspect. We identified three function calls whose return value was not checked; using the `Excepter` aspect provides these checks in the aspectualized version. We found 320 methods that used framework pointers without checking for null, which we corrected by using the `CheckFwArgs` aspect. We introduced two aspect-related defects in revision one. In both cases, we failed to remove code from the application that was replaced by the aspect's advice, resulting in both the aspect and the core concern attempting to free memory.

In revision two, aspectualization removed eight defects from the original application. The body of an `ErcQuery` class method missed the deletion of a query object from memory. We removed this defect by using the `QueryPolicy` aspect. We identified seven missing null pointer checks in revision two. The `Excepter` aspect handled two of them and `CheckFwArgs` handled five. We encountered one aspectualization error in revision two where new functions did not match the `ErcTracing` aspect's pointcut, disabling tracing for those functions.

We identified five defects in the original application in revision three. One defect resulted from failure to check for framework initialization errors; using the `FwErrs` aspect provided this check. We found two defects where the checking provided by the `CheckFwArgs` aspect was missing, and two defects where the checking performed by the `Excepter` aspect was missing. We encountered one aspectualization error in revision three: the `Excepter` aspect did not advise a method that we expected it to advise. We detected this omission using our own weave analysis tool because the number of joinpoints advised did not change as expected.

Eight of the nine aspectualization defects found in the `PowerAnalyzer`, `Instance-Drivers`, and `ErcChecker` can be categorized into three fault types we identified in previous work [58]:

- Incorrect advice behavior: two defects from revision one of `InstanceDrivers`; one defect in revision two of `ErcChecker`.
- Advice syntax errors: one defect from revision three of `InstanceDrivers`.
- Pointcut too weak: one defect in revision six of `PowerAnalyzer`; one defect in revision three of `ErcChecker`.
- Accidental code duplication: two defects found in revision one of `ErcChecker`.

We also encountered a new fault type: faults introduced in the core concern during aspectualization (two defects in revision one of `PowerAnalyzer`). These faults differ from 'Accidental code duplication' because they involve errors introduced when using aspects, rather than failing to reduce code replaced by aspects. We observed that aspectualization removed or avoided more defects than were introduced, and our regression tests identified the aspectualization defects listed above.

9.8 Discussion

For all three applications, using aspects requires a time investment to develop aspects and refactor the original application. However, more than half of the refactoring changes were deletions, and the overall effect was to reduce source code size. There were increases in execution time, memory size, and compiled object size. The largest increase was for object size, which indicates potential memory and execution time increases. However, for these applications, object code size itself was not critical, since large increases in object code size resulted in small increases in memory size and execution time. Memory and execution time increases were caused by the safety checks in the aspectualized version that the original application omits. Had the original application implemented the same checks, the increases in execution time and memory usage would have been smaller. In addition, aspects that perform these checks can easily be enabled or disabled by a change in the aspect, while modifying the equivalent checking code in the original application would require many changes.

Change locality was improved by using aspects. In general, aspectualization makes it possible to avoid many changes in the core concerns. However, in our study, we found that it caused two kinds of changes to the aspects or support code. First, for aspects such as the `Excepter`, whose pointcuts consisted of a list of functions, we needed to update the pointcut in subsequent revisions with additional function names to advise the new functions. Aspects, such as `CheckFwArgs`, whose pointcut was based on application code being part of a class or namespace, require that the core concerns adhere to that naming convention. Thus, new functions added to the core concern, which needed to be checked by the `CheckFwArgs` aspect, had to be enclosed within a class or namespace so that the pointcut matched them.

Concern diffusion was reduced as aspects provided better modularity than the original object-oriented implementation. The use of aspects removed defects that occurred when a concern was inconsistently implemented in multiple locations.

The benefits from using aspects in the three applications varied. For example, the `ErcChecker` had the largest reduction in source code size when revision one was refactored. By contrast, `InstanceDrivers` had little initial code savings, but later revisions saved more code because several cross-cutting changes were avoided in the aspect-oriented implementation. Both the `ErcChecker` and the `PowerAnalyzer` realized more than half of the code savings during aspectualization of revision one. The potential benefit of aspectualization depends on the amount of cross-cutting code in the original application. The final revision of `PowerAnalyzer` had 16,286 lines of code, with aspectualization providing a savings of 397 (2%). The final revision of the `ErcChecker` contained 62,608 lines of code, with aspectualization providing a savings of 2546 lines (4.1%). The final revision of `InstanceDrivers` had the highest percentage of cross-cutting code savings, with 348 lines saved out of 3395 (10%).

Much of the code in these applications is procedural C-style code with some object-oriented code, which may be a factor in explaining the low percentage of cross-cutting code that was found. Object-oriented code has more structure than older C-style code, including class names, package names, and more uses of interfaces, which can be used in identifying aspects and creating pointcuts. The structural relationships found in object-oriented design patterns also facilitate aspectualization [32]. The `ErcChecker` has more object-oriented code than the other two applications, which was used by some aspects, such as the `QueryPolicy` and `QueryConfig` aspects.

In general, aspects that allow deleting the most lines of code do so by advising many joinpoints. For team-based development, one potential challenge may be in communicating these aspectual relationships between the advice and the joinpoints. As Griswold et al. [29] report, the obliviousness that name-based pointcuts provide may complicate maintenance and parallel development of aspects and core concerns. For our study, since one developer performed the aspectualization and maintenance between revisions, this was not a challenge, but it is a potential challenge when refactoring legacy applications that are being maintained by a large team.

9.9 Threats to Validity

Our evaluation demonstrates that maintainability can be improved with aspectualization. However, like most case studies, it is difficult to generalize from a study of three applications. Thus, there are threats to external validity. Some of the threats to validity mentioned in Section 8.5 are threats here: three applications not selected randomly, one subject performing the studies, biases because the subject believes in aspect-oriented programming, and bias in what aspects were identified in the systems

In addition, there are threats to internal validity when performing a maintainability study such as this. The analysis of changes between revisions could have been influenced by the prior experience with the applications. When identifying aspects from the first revision of a legacy application, a developer may be biased toward parts of an application that he knows were more change-prone, and may be biased against those parts of the application that were not change prone. The approach in this study was to only make changes that were necessary for using aspects; another approach would be to favor more extensive refactoring (e.g., adding more object-orientation to the primary code) before aspectualization.

The developer who created the aspects reported the defects avoided by aspects and the defects caused by aspects. A different developer might find different defects in the original application and insert different defects during refactoring. In addition, any aspect developer might be unaware of defects in his or her aspectualized application that would be found by another developer or if the test coverage were higher.

Using legacy applications to perform the study may pose problems. Legacy software often reflects the decisions made based on the language used and tools available at the time of developing the software. Legacy code may be dated by a particular design style, such as the use or lack of design patterns, which may affect what types of refactoring can be performed. Some changes, such as those related to a concern that crosscuts many files, may not have been made in the original code because of the cost; however, had the original implementation used aspects from the beginning, such changes would have been less costly.

Using legacy proprietary software can make it difficult to access the artifacts that are required to perform a study. Our study of regression test coverage for the `PowerAnalyzer` shows low coverage data because we did not have access to half of the original regression test suites. We did not have access to `ErcChecker` regression tests, which could mask some aspectualization defects.

How we use source code repositories of legacy applications presents an additional threat to internal validity. The revisions we considered are major releases across a large set of files. In between two major releases (coarse-grained revisions) of an application, two files might have different revision histories – one might not have changed at all, while the other might have had several file revisions. For files that have multiple changes between revisions, a module in a file might have undergone several changes, only one of which is related to a cross-cutting concern. When that concern is refactored as an aspect, there will be a reduction in the amount of change in that module. However, since there are other changes in the module and the associated file, aspectualization alone is unable to reduce file and module change locality. A more fine-grained approach would consider each file's individual revision history. Using fine-grained revisions for each file is likely to affect change locality results because each change would be smaller, and some changes would be related only to cross-cutting concerns.

We used the *unplanned* changes that occurred in the legacy applications' revision history. We did not design the original application or the subsequent revisions to be explicitly adaptable for upcoming changes in later revisions. We selected aspects by identifying cross-cutting concerns in revision one and by identifying cross-cutting concerns in the code changes between subsequent revisions. A different approach would be to aspectualize a set of modules within an application that are expected to be more change-prone. In our study, the aspects themselves changed little or not at all. For example, the `TimeEvent` class used by the `Timer` aspect of `InstanceDrivers` did not change. Had the type of timing data or core concerns that collected this data changed, the `Timer` aspect itself would have required changes. However, our study did not encounter this type of change.

Construct validity focuses on whether the measures used represent the intent of the study. We compared the original and aspectualized applications over the existing revision history, using code-based metrics such as lines of code, number of modules and files modified, and code concern diffusion. This is one approach for studying the effects of aspect-oriented technology on maintainability. Other researchers have created two separate applications [42, 29], one designed with aspects and one designed without aspects, rather than modifying the original application to incorporate aspects. Maintenance was simulated by implementing features defined in use cases. We identified cross-cutting concerns from source code and did not use requirements or design documents. A different approach would be to define aspects from design documents or indications of designer intent, which might identify aspects that are more abstract and less focused on scattered, similar code.

Our results are dependent on the features and capabilities of AspectC++, which depend on the features of C++. AspectC++ is based on the design and goals of AspectJ [64], but differences between C++ and Java are reflected in AspectC++. For example, as C++ lacks reflection, access to parameter type information is provided through a static, compile-time API. Language differences between C++ and Java, such as pointers, memory management, and operator overloading affect how design patterns such as singleton can be implemented, and cause AspectC++ not to support some features, such as getter and setter accesses of class attributes. Studies using a different primary code language and a different aspect-oriented language may encounter different challenges and benefits from aspectualization.

Chapter 10

Conclusions and Future Work

We developed a test driven approach for creating and unit testing aspects with mock systems, and for performing integration testing with these same aspects in the context of large legacy systems. We built tools for advice instrumentation, weave analysis, and coverage measurement that support our approach.

We demonstrate our approach by refactoring three large legacy C++ applications. We also provide guidelines for the creation of mock systems. We show how using mock systems helps overcome the challenges of long compilation and weave times when refactoring large legacy systems. For the larger of the three systems, using a mock system saved between three and five hours for each aspect. Using mock systems and the tools helps validate aspects, explore different implementations, and identify pointcut and advice faults.

Aspectualization enabled cross-cutting concerns to be implemented in a more modular way. Aspects improved maintainability in all applications as indicated by improved change locality and reduced concern diffusion. In addition, aspectualization reduced overall source code size and reduced the number of defects in the applications. Costs of aspectualization include the time and effort spent refactoring, an increase in execution time of 5%-18.0%, and increased memory usage of 2.4%-12.0%.

The benefits of using aspects increased over subsequent revisions of the applications, suggesting that aspectualizing code that is being actively maintained can have an increasing benefit over time. Most of the costs of aspectualization occur in the first revision that refactors with aspects. Part of the refactoring cost can also be viewed as a benefit — more than half of our changes were deletions of scattered, similar code. Such refactoring reduces the size of the code and improves modularity. After aspectualization, future revisions of the three applications required fewer changes that were more localized.

The reduction of code size depends on the amount of cross-cutting code in the original application. The percentage reduction in `InstanceDrivers` (10%) was 2.5 times the percentage reduction in `ErcChecker` (3.9%), and was five times the percentage reduction in `PowerAnalyzer` (2%). For all three applications, the amount of code removed was always greater than the amount of code added or changed during aspectualization. Even when the reduction in size is small, aspects modularize scattered code into a single module, which reduces the number of locations to modify in the future when that concern changes. For example, the tracing concern in `InstanceDrivers` changed in several revisions as more tracing was added to the application. These changes were unneeded when an aspect was used. Applications that contain more cross-cutting code would realize more benefit from aspectualization, as would applications where the cross-cutting code is change-prone. The `ErcChecker` shows the largest total code savings, but as a percentage was in the middle.

Software quality was improved by aspectualization because some defects were removed or avoided. Although some testing challenges remain, we were able to use statement coverage and joinpoint coverage to identify untested code and help validate the changes made during aspectualization. We were also able to use our weave analysis tools and annotations to identify problems with pointcut strength.

10.1 Future Work

Future work could explore four areas related to this research: testing, mock systems, tool development, and maintainability studies.

10.1.1 Testing

Testing more complex aspects leads to an open question: are there coverage criteria or measures that would be more effective than statement coverage or joinpoint coverage without being too cumbersome? Future work could explore coverage criteria and testing techniques for aspect-aspect interactions and for aspects that change static properties such as inheritance hierarchies.

Our existing coverage-based approach could be extended to consider statement coverage within advice bodies. Since we gather statement coverage information from woven code, we would have to reverse-engineer the weaving process to match it to the original advice body. In addition, even if we knew that all advice statements were executed, we might want to know if all advice statements were executed at each join point.

10.1.2 Mock Systems

Mock systems could also be used to unit test other advice features, such as introductions, class hierarchy changes, exception softening, and advice precedence. We expect that introductions and hierarchy changes can be unit tested on mock systems that will then be modified by the aspect.

More studies are also needed on using mock systems to evaluate adding many aspects to a system, particularly when there will be intended aspect-aspect interactions. In addition, strategies for effectively detecting or mitigating unintended aspect-aspect interactions need to be developed. In our preliminary studies we saw that mock systems can help isolate aspect-aspect interactions by easily creating structures based on method names and class structures where multiple aspects interact in the small. Our weave analyzer could be extended to report joinpoints advised by multiple aspects so that developers are aware of them and test aspect interactions in a mock system.

Additional research could explore ways to validate that a mock system is similar enough to the actual system. This is an important question, since unit testing with a mock system assumes the mock system provides a useful abstraction of the real system.

Another interesting question is the effect of crosscutting interfaces [29] on mock systems. A crosscutting interface (XPI) uses a semantic description of intent and a pointcut designator signature to specify the dependencies between core concerns and aspects. Clearly defined interfaces for aspects might allow mock systems to be created to match the same XPI.

Although mock systems are beneficial during refactoring of legacy systems, mock systems will need to be maintained with the aspects and system in order to remain relevant. A benefit of mock system maintenance is that mock systems can be used for regression testing of aspects to validate aspect and system changes during evolution. Understanding the costs and benefits of a mock system over time is an important extension of this work. In particular, future work could investigate co-maintenance of mock systems with the legacy system.

Tools could be developed to create mock systems based on XPIs or on the proposed aspect pointcuts. Automating the creation of mock systems could reduce their cost. In addition, tools could then be used to update the mock system as the aspect or core concerns evolve.

10.1.3 Tool Development

Test tools, such as our advice instrumenter, weave analyzer, and coverage analyzer could be developed for other aspect-oriented languages, such as AspectJ. Our concern identifier could be extended to look for other idioms, rather than solely focusing on word sets that follow function calls.

Improving the incremental weaving and compilation capabilities of AspectC++ would help mitigate some problems we have described. However, during aspect development both pointcuts and advice are frequently changing, requiring a full reweave and recompilation of the systems since aspect features (such as wildcards in pointcuts) require full system analysis.

Aspect weavers could be extended to identify one of the errors detected by our static checking tools: unused advice. In addition, our compile-time assertions about where advice is expected (`Advised`, `NotAdvised`) could be provided by AspectC++ or AspectJ so that developers can more easily identify pointcut strength errors.

10.1.4 Maintainability Studies

Various other maintainability studies can be performed:

- A study could consider the benefits of aspectualization when an aspectualized concern changes. For example, we could compare the changes required if the `TimeEvent` class changed in `InstanceDrivers`.

- In our research, the applications changed but the underlying object-oriented framework remained static. A study could determine if aspectualization helps limit changes to an application when the underlying object-oriented framework changes. These changes could be simple version changes in a framework or much more complex changes, such as moving to a completely different framework.

- A study could compare the effect of anticipated changes versus unanticipated changes on aspectualized legacy systems.

- A study might identify cross-cutting concerns using higher level abstractions such as design documents and requirements rather than those found directly in the code.
- Instead of refactoring legacy code, one may study the evolution of applications that used AOP from the beginning.
- Use applications or open source code repositories to do more fine-grained studies. For example, we could look at each revision of a single file or set of files, instead of our more coarse-grained approach of using application releases where some files have been revised several times between releases.
- Identify the kinds of structural changes that make it easier to aspectualize an application. For example, adding namespaces and grouping functions into a class allowed us to define simpler pointcuts as well as minimize changes to the primary code.

REFERENCES

[1] CppUnit website. `http://sourceforge.net/projects/cppunit`.

[2] JUnit website. `http://www.junit.org`.

[3] Roger T. Alexander, James M. Bieman, and Anneliese A. Andrews. Towards the Systematic Testing of Aspect-Oriented Programs. *Technical Report CS-4-105, Department of Computer Science, Colorado State University*, March 2004.

[4] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied.* AW C++ in Depth Series. Addison Wesley, January 2001.

[5] Dave Astels. *Test Driven development: A Practical Guide.* Prentice Hall Professional Technical Reference, 2003.

[6] Marc Bartsch and Rachel Harrison. An exploratory study of the effect of aspect-oriented programming on maintainability. *Software Quality Journal*, 16(1):23–44, 2008.

[7] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the 14th IEEE International Conference on Software Maintenance*, page 368, Washington, DC, USA, 1998. IEEE Computer Society.

[8] Kent Beck. Make it run, make it right: Design through refactoring. *The Smalltalk Report*, 6(4):19–24, January 1997.

[9] Lionel C. Briand, W. J. Dzidek, and Yvan Labiche. Instrumenting contracts with aspect-oriented programming to increase observability and support debugging. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 687–690, Washington, DC, USA, 2005. IEEE Computer Society.

[10] Magiel Bruntink, Arie van Deursen, Maja D'Hondt, and Tom Tourwé. Simple crosscutting concerns are not so simple: analysing variability in large-scale idioms-based implementations. In *AOSD '07: Proceedings of the 6th International Conference on Aspect-oriented Software Development*, pages 199–211, New York, NY, USA, 2007. ACM.

[11] Magiel Bruntink, Arie van Deursen, and Tom Tourwe. Isolating idiomatic crosscutting concerns. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 37–46, Washington, DC, USA, 2005. IEEE Computer Society.

[12] Magiel Bruntink, Arie van Deursen, and Tom Tourwé. Discovering faults in idiom-based exception handling. In *ICSE '06: Proceeding of the 28th International Conference on Software Engineering*, pages 242–251, New York, NY, USA, 2006. ACM Press.

[13] Magiel Bruntink, Arie van Deursen, Remco van Engelen, and Tom Tourw. On the use of clone detection for identifying crosscutting concern code. *IEEE Transactions on Software Engineering*, 31(10):804–818, 2005.

[14] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[15] Curtis Clifton and Gary T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In Gary T. Leavens and Ron Cytron, editors, *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2002*, number 02-06 in Technical Report, pages 33–44. Department of Computer Science, Iowa State University, April 2002.

[16] Curtis Clifton and Gary T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. In Lodewijk Bergmans, Johan Brichau, Peri Tarr, and Erik Ernst, editors, *SPLAT: Software Engineering Properties of Languages for Aspect Technologies*, Mar 2003.

[17] Yvonne Coady and Gregor Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In Mehmet Akşit, editor, *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD-2003)*, pages 50–59. ACM Press, March 2003.

[18] Yvonne Coady, Gregor Kiczales, Joon Suan Ong, Andrew Warfield, and Michael Feeley. Brittle Systems will Break — Not Bend: Can AOP Help? . In *Proceedings of the 10th ACM SIGOPS European Workshop on Operating Systems.* ACM Press, September 2002.

[19] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In Karl Lieberherr, editor, *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD-2004)*, pages 141–150. ACM Press, March 2004.

[20] Pascal Durr, Tom Staijen, Lodewijk Bergmans, and Mehmet Aksit. Reasoning about semantic conflicts between aspects. In Kris Gybels, Maja D'Hondt, Istvan Nagy, and Remi Douence, editors, *2nd European Interactive Workshop on Aspects in Software (EIWAS'05)*, September 2005.

[21] Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sergio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho, and Francisco Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 261–270, New York, NY, USA, 2008. ACM.

[22] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, Boston, 2005.

[23] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, August 1999.

[24] Erich Gamma and Kent Beck. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, July 1998.

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

[26] Alessandro Garcia, Cludio Sant'Anna, Eduardo Figueiredo, Uira Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing design patterns with aspects: a quantitative study. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 3–14, New York, NY, USA, 2005. ACM Press.

[27] Sudipto Ghosh, Robert France, Devond Simmonds, Abhijit Bare, Brahmila Kamalakar, Roopashree P. Shankar, Gagan Tandon, Peter Vile, and Shuxin Yin. A middleware transparent approach to developing distributed applications. *Software Practice and Experience*, 35(12):1131–1159, October 2005.

[28] Sudipto Ghosh and Brahmila Kamalakar. An aspect-oriented approach to developing middleware-based applications. In *OOPLA and GPCE Workshop on Model Driven Software Development*, Vancouver, Canada, October 2004. Position paper.

[29] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 23(1):51–60, 2006.

[30] J. Hannemann, Gail Murphy, and Gregor Kiczales. Role-based refactoring of cross-cutting concerns. In *4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, pages 135–146, March 2005.

[31] Jan Hannemann and Gregor Kiczales. Overcoming the prevalent decomposition in legacy code. In Peri Tarr and Harold Ossher, editors, *Workshop on Advanced Separation of Concerns in Software Engineering (ICSE 2001)*, May 2001.

[32] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA-02)*, volume 37, 11 of *ACM SIGPLAN Notices*, pages 161–173, New York, November 4–8 2002. ACM Press.

[33] Jan Hannemann, Gail Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. In Peri Tarr, editor, *Proceedings 4th International Conference on Aspect-Oriented Software Development (AOSD-2005)*, pages 135–146. ACM Press, March 2005.

[34] Kevin Hoffman and Patrick Eugster. Towards reusable components with aspects: an empirical study on modularity and obliviousness. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 91–100, New York, NY, USA, 2008. ACM.

[35] David Janzen and Hossein Saiedian. Test-driven development: Concepts, taxonomy, and future direction. *Computer*, 38(9):43–50, 2005.

[36] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[37] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[38] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of European Conference on Object-Oriented Programming 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

[39] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. Technical Report SPL97-008 P9710042, Xerox PARC, February 1997.

[40] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, June 1997.

[41] Christian Koppen and Maximilian Störzer. PCDiff: Attacking the fragile pointcut problem. In Kris Gybels, Stefan Hanenberg, Stephan Herrmann, and Jan Wloka, editors, *European Interactive Workshop on Aspects in Software (EIWAS)*, September 2004.

[42] Uir Kulesza, Cludio Sant'Anna, Alessandro Garcia, Roberta Coelho, Arndt von Staa, and Carlos Lucena. Quantifying the effects of aspect-oriented programming: A maintenance study. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 223–233, Washington, DC, USA, 2006. IEEE Computer Society.

[43] Ramnivas Laddad. Aspect-oriented refactoring part 1: Overview and process. Technical report, TheServerSide.com, 2003. `http://www.theserverside.com/resources/article.jsp?l=AspectOrientedRefactoringPart1`.

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

[45] Otavio Lemos, Fabiano Ferrari, Paulo Masiero, and Critina Videira Lopes. Testing aspect-oriented programming pointcut descriptors. In *Proceedings of the 2nd Workshop on Testing Aspect-Oriented programs, in conjunction with the International Symposium on Software Testing and Analysis (ISSTA06)*, July 2006.

[46] Otavio Augusto Lazzarini Lemos, Jose Carlos Maldonado, and Paulo Cesar Masiero. Structural unit testing of aspectj programs. In *2005 Workshop on Testing Aspect-Oriented Programs (held in conjunction with AOSD 2005)*, March 2005.

[47] Nick Lesiecki. Unit test your aspects. Technical report, Java Technology Zone for IBM's Developer Works, November 2005. `http://www-128.ibm.com/developerworks/java/library/j-aopwork11/`.

[48] Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In Gabor Karsai and Eelco Visser, editors, *Proceedings of the Third Conference on Generative Programming and Component Engineering*, volume 3286 of *Springer-Verlag Lecture Notes in Computer Science*, pages 55–74. Springer, October 2004.

[49] Daniel Lohmann, Andreas Gal, and Olaf Spinczyk. Aspect-Oriented Programming with C++ and AspectC++ (Tutorial). In Mira Mezini, editor, *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD-2005).* ACM Press, March 2005.

[50] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schrder-Preikschat. On the configuration of non-functional properties in operating system product lines. In David H. Lorenz and Yvonne Coady, editors, *ACP4IS: Aspects, Components, and Patterns for Infrastructure Software*, March 2005.

[51] Cristina Videira Lopes and Trung Chi Ngo. Unittesting aspectual behavior. In *2005 Workshop on Testing Aspect-Oriented Programs (held in conjunction with AOSD 2005)*, March 2005.

[52] Daniel Mahrenholz, Olaf Spinczyk, Andreas Gal, and Wolfgang Schrder-Preikschat. An aspect-orientied implementation of interrupt synchronization in the pure operating system family. In *Proceedings of the 5th ECOOP Workshop on Object Orientation and Operating Systems*, Malaga, Spain, June 2002.

[53] Marius Marin, Leon Moonen, and Arie van Deursen. An approach to aspect refactoring based on crosscutting concern types. In Martin Robillard, editor, *MACS '05: Proceedings of the 2005 workshop on Modeling and analysis of concerns in software*, pages 1–5. ACM Press, May 2005.

[54] Michael Mortensen and Roger T. Alexander. An approach for adequate testing of AspectJ programs. In *2005 Workshop on Testing Aspect-Oriented Programs (held in conjunction with AOSD 2005)*, March 2005.

[55] Michael Mortensen and Sudipto Ghosh. Creating pluggable and reusable non-functional aspects in AspectC++. In *ACP4IS '06: Proceedings of the 5th Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 1–7, Bonn, Germany, 2006.

[56] Michael Mortensen and Sudipto Ghosh. Using aspects with object-oriented frameworks. In *AOSD '06: 5th International Conference on Aspect-oriented Software Development Industry Track*, pages 9–17, Bonn, Germany, March 2006.

[57] Michael Mortensen and Sudipto Ghosh. Refactoring idiomatic exception handling in C++: Throwing and catching exceptions with aspects. In *AOSD '07: 6th International Conference on Aspect-oriented Software Development Industry Track*, pages 9–15, Vancouver, British Columbia, Canada, March 2007.

[58] Michael Mortensen, Sudipto Ghosh, and James Bieman. Testing during refactoring: Adding aspects to legacy systems. In *Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE 06)*, Raleigh, North Carolina, USA.

[59] Michael Mortensen, Sudipto Ghosh, and James Bieman. A test driven approach for aspectualizing legacy software using mock systems. In *Information and Software Technology*, volume 50, pages 621–640. Elsevier, 2008.

[60] Olaf Spinczyk and pure-systems GmbH. *Documentation: AspectC++ Compiler Manual*, May 2005. `http://www.aspectc.org/fileadmin/documentation/ac-compilerman.pdf`.

[61] William Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992. `ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z`.

[62] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[63] D. Schmidt and M. Fayad. Object-oriented application frameworks. *Communications of the ACM*, 10(40):32–38, 1997.

[64] Olaf Spinczyk, Andreas Gal, and Wolfgang Schrder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[65] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. AspectC++: An AOP extension for C++. In *Software Developers Journal*, number 7, pages 68–74. Software-Sydawnicto, Warsaw, Poland.

[66] Maximilian Störzer. Analysis of AspectJ programs. In Boris Bachmendo, Stefan Hanenberg, Stephan Herrmann, and Günter Kniesel, editors, *3rd Workshop on Aspect-Oriented Software Development (AOSD-GI) of the SIG Object-Oriented Software Development, German Informatics Society*, March 2003.

[67] Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 166–175, New York, NY, USA, 2005. ACM.

[68] Peri Tarr, Harold Ossher, Stanley M. Sutton, Jr., and William Harrison. N degrees of separation: Multi-dimensional separation of concerns. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 37–61. Addison-Wesley, Boston, 2005.

[69] Paolo Tonella and Mariano Ceccato. Refactoring the aspectizable interfaces: An empirical assessment. *IEEE Transactions on Software Engineering*, 31(10):819–832, 2005.

[70] Tom Tourwé, Johan Brichau, and Kris Gybels. On the existence of the AOSD-evolution paradox. In Lodewijk Bergmans, Johan Brichau, Peri Tarr, and Erik Ernst, editors, *SPLAT: Software Engineering Properties of Languages for Aspect Technologies*, Mar 2003.

[71] Shiu Lun Tsang, Siobhan Clarke, and Elisa Baniassad. An evaluation of aspect-oriented programming for Java-based real-time systems development. In *Proceedings of The 7th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, pages 291–300, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[72] Matthias Urban and Olaf Spinczyk. AspectC++ language reference. June 2004. `http://aspectc.org/fileadmin/documentation/ac-languageref.pdf`.

[73] Dean Wampler. Contract4J for design by contract in Java: Design pattern-like protocols and aspect interfaces. In *ACP4IS '06: Proceedings of the 5th Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 27–30, Bonn, Germany, 2006.

[74] Dianxiang Xu and Wiefing Xu. State-based incremental testing of aspect-oriented programs. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD 2006)*, March 2006.

[75] Jos Pablo Zagal, Ral Santelices Ahus, and Miguel Nussbaum Voehl. Maintenance-oriented design and development: A case study. *IEEE Software*, 19(4):100–106, 2002.

[76] Charles Zhang and Hans-Arno Jacobsen. Refactoring middleware with aspects. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1058–1073, November 2003.

[77] J. Zhao. Unit testing for aspect-oriented programs. Technical Report SE-141-6, Information Processing Society of Japan (IPSJ), May 2003. `http://www.fit.ac.jp/~zhao/pub/ps/ipsj-tr-se-141-6.pdf`.

[78] J. Zhao and M. Rinard. Pipa: A behavioral interface specification language for AspectJ. In *Proceedings of Fundamental Approaches to Software Engineering (FASE'2003), LNCS 2621*, pages 150–165. Springer-Verlag, April 2003.

[79] Yuewei Zhou, Debra Richardson, and Hadar Ziv. Towards a practical approach to test aspect-oriented software. In *TECOS 2004: Workshop on Testing Componhent-Based Systems, Net.Object Days 2004*, September 2004.