A Test Driven Approach for Aspectualizing Legacy Software Using Mock Systems

Michael Mortensen

Hewlett-Packard, 3404 E. Harmony Rd MS 88, Fort Collins, CO, 80528

Sudipto Ghosh, James M. Bieman

Computer Science Department, Colorado State University, Fort Collins, CO, 80523-1873

Abstract

Aspect-based refactoring, called aspectualization, involves moving program code that implements cross-cutting concerns into aspects. Such refactoring can 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.

Key words: mock systems, aspect oriented programming, legacy systems, refactoring, testing

1 Introduction

Aspect-oriented programming provides a new construct, an aspect, that can modularize scattered and tangled (termed crosscutting) code. Replacing cross-

Email addresses: `mike.mortensen@hp.com` (Michael Mortensen), `ghosh@cs.colostate.edu`, `bieman@cs.colostate.edu` (Sudipto Ghosh, James M. Bieman).

cutting code with aspects can improve the design structure of legacy systems [16,34]. The use of aspects in a refactored *aspectualized* system helps modularize cross-cutting code concerns. As with any refactoring technique, a systematic approach to testing is needed to check that new faults are not introduced [11].

In an aspect-oriented program, cross-cutting concerns are modularized into *aspects*. The aspects are *woven* into the primary code by a preprocessor, compiler, or run-time system. An aspect includes *advice*, which is the functionality to be woven in, and a *point-cut*, which identifies the locations in the primary code, called *joinpoints*, where the advice is inserted. Advice specifies when it executes: *before advice* executes before the joinpoint, *after advice* executes after the joinpoint, and *around advice* executes instead of a joinpoint [19]. Around advice can either bypass or execute the code represented at the joinpoint. Aspect languages such as AspectJ [3] and AspectC++[1] [12] offer a set of pattern-matching mechanisms to match advice to joinpoints. We use AspectC++ in this work, since we are refactoring legacy C++ systems.

This paper introduces the use of mock systems to aid in aspectualization of a system. Developers develop and test aspects within the small mock system, which can be quickly woven and compiled. After testing aspects in the mock system, we weave them with the real system. Our approach uses the testing of aspects with mock systems as unit testing and the testing of the real system as a form of integration testing. We apply coverage criteria to test the aspects, and use existing system regression tests for integration testing. We use information generated during weaving to automate this approach, and we demonstrate the approach by aspectualizing large C++ systems using AspectC++.

The remainder of this paper is structured as follows. Section 2 summarizes related background work. We present our test driven approach in Section 3 and describe new tools to find faults and analyze coverage in Section 4. In Section 5, we describe aspects that we developed to refactor legacy systems using mock systems. Section 6 presents a set of guidelines for creating mock systems. We evaluate the feasibility of our approach in Section 7. Conclusions and future work are described in Section 8.

2 Background

We summarize related work on aspect mining and refactoring, test-driven development, and testing of aspect-oriented programs.

[1] www.aspectc.org

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

Coady and Kiczales [8] 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.

Lohmann et al. [26] use aspects to implement cross-cutting architectural decisions, such as synchronization policies and power usage that affect many modules in a system, to improve modularity and allow architectural flexibility.

Hannemann, Murphy, and Kiczales [17] 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 into an AspectJ-based system. Rather than identify component roles, we examine the code for scattered code fragments.

One use for aspects is to modularize framework-related code in applications that use frameworks. Our intent is similar to that of Ghosh et al. [13], who differentiate between business logic and code that interacts with middleware services, and implement middleware-related code with aspects.

Tonella and Ceccato [34] 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 [28] 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 [5] can also identify cross-cutting concerns. Bruntink et al. [6] 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.

2.2 Test Driven Development

Test Driven Development (TDD) is a software process in which unit tests are written before the functional code is written. A key goal of TDD is that all code has associated tests. To emulate a complex system dependency, test driven development may use a mock object (or *mock*) in place of the real object [4]. 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 the use of tests to drive refactoring. We extend the concept of mock objects to create mock systems, providing a context for creating and unit testing aspects. We weave aspects into the mock systems to develop them through iterative unit testing, which increases confidence in aspects introduced into a real system.

2.3 Testing Aspect-Oriented Programs

Douence et al. [9] 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 [1] 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 use mock systems, weaving aspects with the mock concerns and using mock concern 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 [1]. Existing AOP testing approaches focus on aspect-specific faults [22,29], 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 [37], path coverage [23], and state-based coverage [36]. 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 3.2.

Zhou, Richardson, and Ziv [38] 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 [24] 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 [27], 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.

3 Approach

We use the following steps in our aspectualization approach:

(1) Identify cross-cutting concerns and duplicated code in the legacy system that can be realized as an aspect.
(2) Create a small mock system to iteratively develop and unit test a prototype aspect.
(3) Refactor the real system to use the aspect by removing duplicate or cross-cutting code from the system and then weaving the aspect.
(4) Conduct integration testing of the refactored system by running the regression tests.

Figure 1 illustrates the approach. During the first two steps, developers use the mock system to experiment with several aspect pointcut specifications and advice to test that they correctly modularize the cross-cutting concerns. The pointcut specifications must have the correct strength [1] so that they match all (and only) the desired joinpoints in the mock system.

The mock system mimics the structures that the aspect will interact with in the real system so that the pointcut will also work in the real system. We typically create a mock system for each aspect. The advice must specify correct behavior. The aspect is woven with the mock system so that we can validate advice behavior. After step three is performed, developers need to test the refactored system using pre-existing regression tests.

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 within the mock system.

3.1 Identifying Aspects in Legacy Systems

The legacy systems described in this paper consist of two VLSI CAD applications which are both based on a VLSI CAD framework [31]. 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 [8], 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. [26].

Just as Ghosh et al. [13] differentiate between application-specific code and middleware-related code, 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 (some tools support refactoring of Java code [17,34]), we primarily rely on manual inspection of source code and simple textual approaches such as `grep` to identify aspect candidates. Developing automated aspect mining tools is not the goal of our work.

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 [15]. We have begun experimenting with clone detection software, as Bruntink et al. [6] have done, to evaluate code clones as potential aspects. Our initial efforts involve the use of CCFinder[18] to identify code clones in the `PowerAnalyzer`.

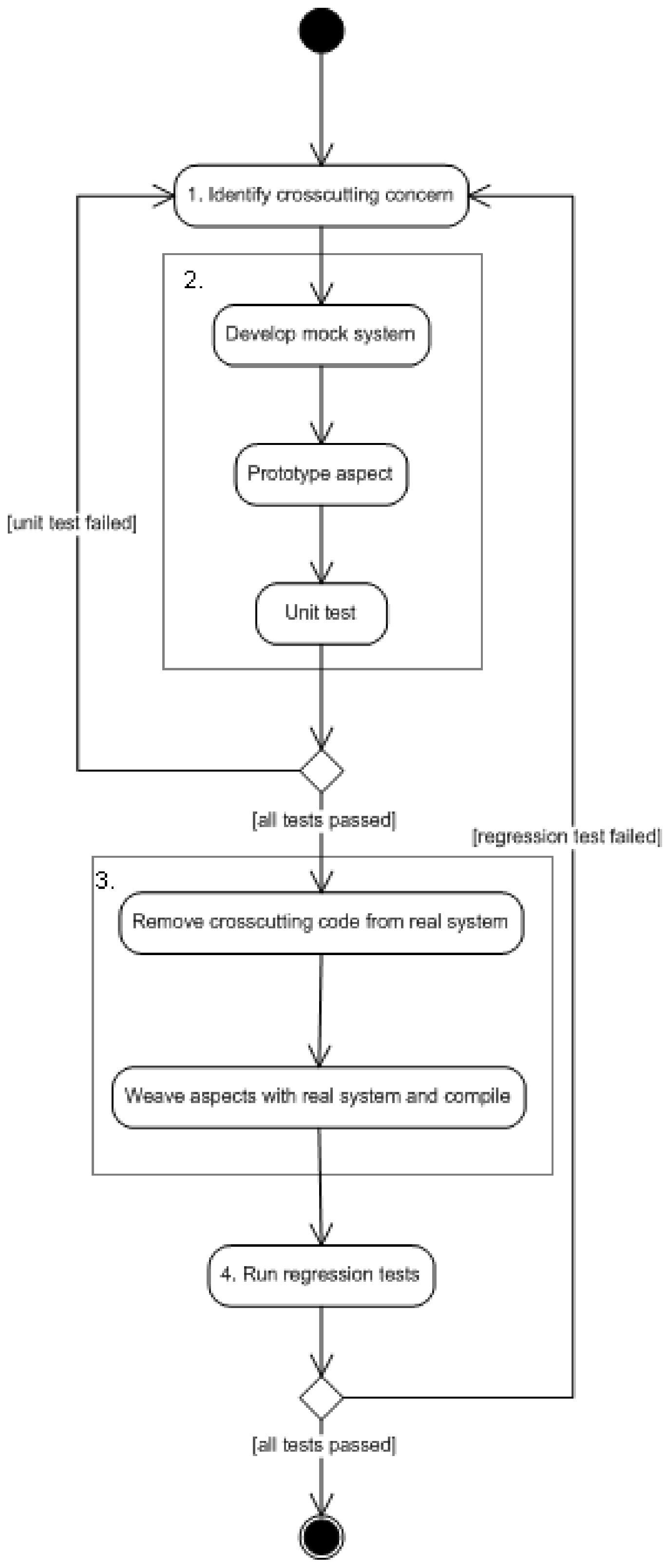

Fig. 1. Steps for our Approach.

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 Section 6.

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.

3.3 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.

The goal of unit testing with the mock system is to avoid changing an aspect during integration testing. Some aspects, such as caching, define a pointcut as a list of all functions to cache. When using the aspect with the real system, this pointcut must change to reflect each cached method from the real system, but the advice can be tested within the mock system.

3.4 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 Tools

We developed three tools to support our approach:

(1) The *Weave analyzer* checks for unused advice and for annotation violations.
(2) The *Advice instrumenter* supports coverage analysis.
(3) The *Coverage analyzer* measures joinpoint coverage and statement coverage of the mock system.

Our unit and system testing tools support AspectC++, and they leverage features of the AspectC++ weaver. The AspectC++ weaver writes information about the weave results to an XML file for use by IDEs such as Eclipse[2]. The XML weave file lists the pointcuts and advice for each aspect, identifying each with a unique numerical identifier. The weave file lists all source code joinpoints (by line number) that are matched by each pointcut [33].

4.1 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:

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

```
========================
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 */
   {
```

4.2 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 4.3) 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.

4.3 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 code during unit testing. We use `gcov`[3] on the woven code, which generates a coverage file 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 statements in the mock code.

The `gcov` output is a copy of the mock system code that marks how many times each line was executed. We process this file to ignore AspectC++-specific aspect definition code that `gcov` incorrectly analyzes. Our coverage analyzer prints out any missed mock system statements from this `gcov` output file. Sample output is shown below:

```
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
```

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

```
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 2 shows the data read in and produced by our tools, 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.

4.4 Related tool approaches

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 annotations 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. We also believe they should help during maintenance, since changes to core concerns could change the matching joinpoints.

Laddad [20] 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.

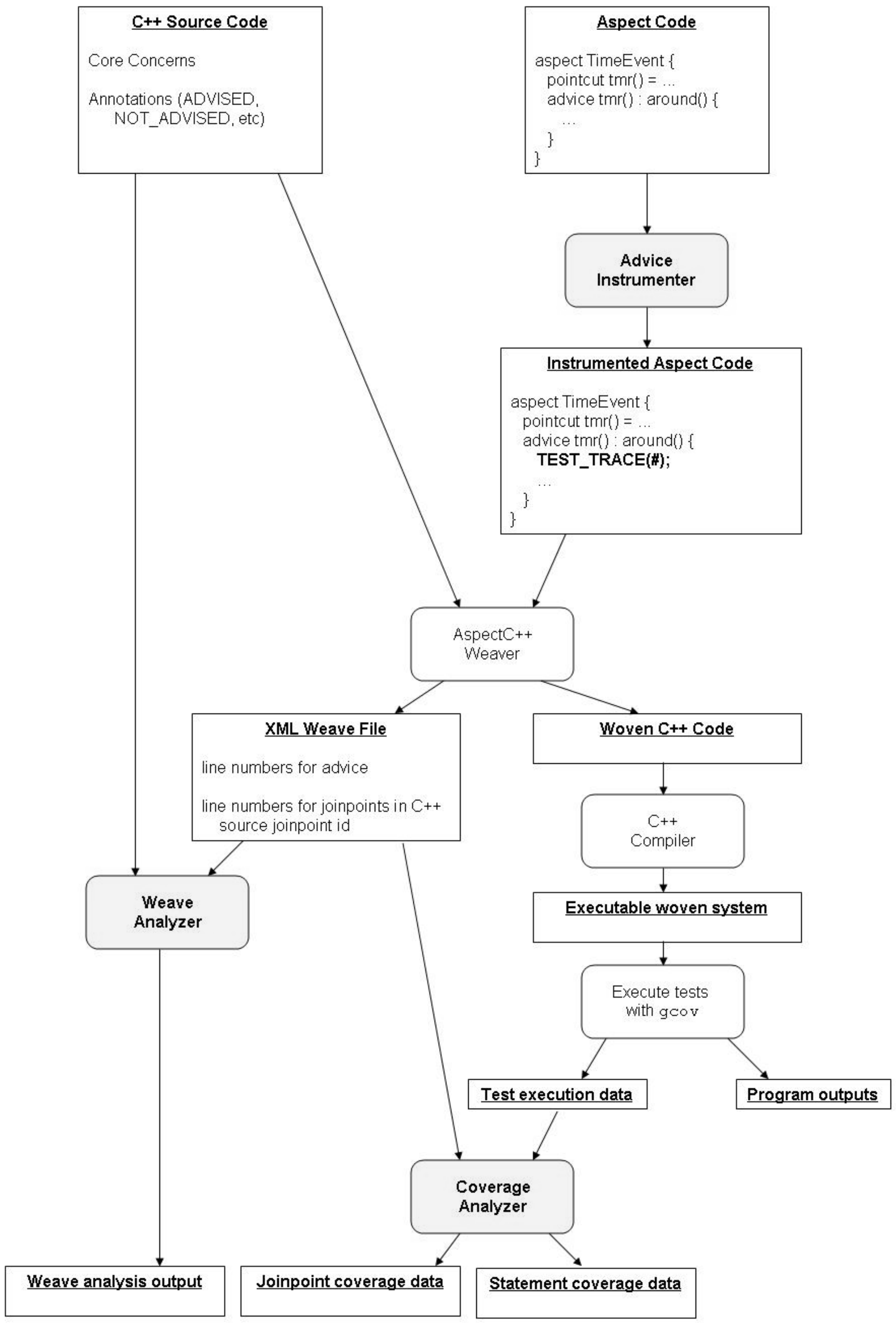

Fig. 2. Using the Tools During Refactoring.

5 Example Mock Systems and Aspects

We used mock systems to aspectualize two legacy systems at Hewlett-Packard. We now describe the aspects and mock systems and use them as a running example and for evaluating our approach. In this section we first describe the two legacy systems. Next, we describe four types of aspects and the mock systems used to develop and test them. Our evaluation results are presented in Section 7.

5.1 The Legacy Systems

The `ErcChecker` is a C++ application that consists of approximately 80,000 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 12,000 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).

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, `QueryConfigAspect`, provides run-time configuration, which allows users to disable queries at run-time via a configuration file that users can edit. 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 `%::createQueries()`, with the AspectC++ wildcard (%) used to match `createQueries()` in all subclasses. The advice can use 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 as `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.

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 `QueryConfigAspect` and `QueryPolicy` aspects. The `QueryConfigAspect` advice executes around each call to the `createQueries()` method, and extracts the class name from the joinpoint information available in AspectC++. The `QueryConfigAspect` implementation is shown below:

```
aspect QueryConfigAspect {
   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 `QueryConfigAspect` 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.

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 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. Once the root cause was suspected, 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 creating 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, which were implemented by the advice [31]. 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.

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 [21,25]. 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.

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 whose argument is a `bcmNet` pointer, 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 had several requirements that we wanted to check using the mock systems. First, the cache needed to function properly. Second, we wanted our caching aspect to work correctly with different data types. Third, we did not want the cache to introduce any noticeable performance penalty; in fact, caching should improve performance. Fourth, we needed a way to measure 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 [30] 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. In the mock system we verified that the pointcut matched intended functions. We verified 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 [30].

We also created a second mock system to compare the performance overhead of the AspectC++ approach to the original C++ caching code. We used mock statement coverage to check that the cached functions in the mock system were called. Joinpoint coverage checked that the function and method caching aspects were executed at all matching join points.

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 caching 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.

5.4 PowerAnalyzer debugging aspect

The first aspect used in the `PowerAnalyzer` was a development aspect to aid in debugging a case of abnormal termination. During one user's run, the `PowerAnalyzer` aborted a run with a message from a framework class that a null pointer had been encountered. Unfortunately, the framework method called `exit()`, which forced the `PowerAnalyzer` to exit without creating a core file that stores program state and stack trace information [4]. Calling `exit()` limits visibility when using a debugger such as `gdb`.

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 can use AspectC++ features to print the context of each iterator call.

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

[4] 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.

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 have advice and which should not. The aspect was created, and 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++ method, `JoinPoint::signature()`, to print out which iterator is called. The `JoinPoint::JPID` value is used with the XML weave file to determine the callsite context — the location in the core concern that called the iterator.

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.

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 run-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 `Timer` 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 encapsulates the `Timer` within the advice body and uses the AspectC++ joinpoint API to print the context in which the `Timer` is being used. Since the `Timer` 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 `Timer`, 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 the `Timer` 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`.

5.5.1 Using the mock system for aspect creation and testing

The mock system for the `TimeEvent` 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 `Timer` class. The mock system does not rely on framework components, but uses the `Timer` module, which already existed in the `PowerAnalyzer`.

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

```
aspect TimeEvent {
   pointcut tmr() = call("% tmr%(...)");
   advice tmr() : around() {
      Timer 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 make changes to the pointcut (changing it twice) as we corrected problems we encountered when advising nested calls. We also used the mock system to test how the advice instantiated the `Timer` module and how timer log messages were formatted.

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 Timer objects exists at many locations, there was no common structure or naming convention for the pointcut to match. To refactor the `PowerAnalyzer`, functions that used the `Timer` class had the `Timer` 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 `Timer` module. These separate uses were either loop statements or conceptually separate code blocks. For this function, we first used Extract Method refactoring [11][5]. 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 `Timer` module instead of a separate descriptive argument for each `Timer` 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 [35]. There is tight coupling between the `TimeEvent` aspect and the naming convention of methods. If a new function 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 `TimeEvent` 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 `Timer` module is only directly called from the advice of the `TimeEvent` aspect. To enforce this design decision, we also added additional advice to the `TimeEvent` aspect so that a direct

[5] This refactoring step states: “Turn the fragment into a method whose name explains the purpose of the method.”

use of the `Timer` will result in an error when the woven code is compiled. AspectC++ does not have a `declare error` statement like AspectJ. However, Alexandrescu [2] shows how to create compile-time assertions in C++ by defining a C++ template that is instantiated with a boolean value, and only defined for only one boolean value (e.g. `true`), so that a compile time error occurs whenever the template is instantiated with a `false` value. We created additional advice [30] for the `PowerAnalyzer` that uses a C++ template that fails whenever it is woven around core concern calls to the `Timer` class, ensuring that the `Timer` is only called from our `TimeEvent` advice.

6 Mock System Creation

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 real system for integration testing. Thus, an important question is how to create a mock system.

Based on our experience with legacy applications, we developed a preliminary list of mock system creational patterns that can aid the developer of a mock system. As more experience is gained with using mock systems, we expect the catalog of patterns to grow.

The patterns are based on characteristics of both the aspects and the real system. The mock system must provide an environment that emulates the structure and functionality for pointcuts, advice, and other aspect-based features to function.

6.1 Create mock methods for spectator aspects

Spectator aspects are defined by Clifton and Leavens [7] 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.

Motivation: Because spectators do not rely on the internal state of the methods and classes they advise, we use method stubs in the mock system. Stubs are 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.

Step:

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

Example: The `TimeEvent` aspect in the `PowerAnalyzer` is a spectator that adds timing information without affecting the power analysis. 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 `TimeEvent` 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; the stubs are called in `main()`.

6.2 Create simple functional mock methods for non-functional concerns

Non-functional aspects [26] 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.

Motivation: We use mock methods to provide simple functionality when validating a non-functional aspect. This differs from the motivation in Section 6.1 because non-functional concerns, such as caching, must not only advise the right joinpoints but also must implement the cross-cutting concern without changing the existing functionality.

Steps:

(1) Create mock class methods with the same parameter types and return types as the methods in the real system, but with simpler functionality.
(2) Invoke the mock method with several different parameter values from the `main()` function to validate the non-functional property.

A mock system for the caching aspect can use a function that operates on the same data types used in the real system. Joinpoint coverage checks that the advice is used. Statement coverage of the woven mock system can be used to check that all advice statements are executed. For caching, we want to test that the advice executes correctly: on the first call it should use `proceed()` and store the result, while subsequent calls with the same parameter should return the cached value without calling `proceed()` to re-execute the method. Unit tests can call a function with the same value multiple times and check for the correct output.

Example: The caching aspect in the ErcChecker should improve performance without affecting functionality, but faulty advice can affect program modules.

In the mock system, we created functions that work on the same types, but are simple to compute and validate. For example, rather than calculating fanout of an electrical net, we created 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 were called in an interleaved fashion to validate that the aspect created separate caches for each, and that the advised functions returned 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);
```

In the mock system, we created functions and methods using other framework pointers, such as `bcmCell` and `bcmInstance`. We also tested caching of built in types (e.g. `float`) using mathematical functions including `Square()` and `SquareRoot()`.

6.3 Reuse real system components

Components needed in the mock system can sometimes be directly obtained from the existing real system.

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.

Steps:

(1) Identify components methods and classes in the real system that provide essential functionality in the mock system.
(2) In the mock system, import the necessary components. In C++, this can be done by including a header file and linking against a system or framework library.
(3) Copy small code sections that contain boilerplate code for using imported components.

Example: To use framework calls in mock systems for the `ErcChecker` and `PowerAnalyzer`, we reused a singleton class, `BlockData`, that handled initialing and loading framework design data. In the mock system, we included the `BlockData` class header and copied the code that uses it to initialize framework data in the mock system.

6.4 Emulate the callsite context for joinpoints

The developer creates callsite contexts in the mock system that are similar to the expected joinpoints in the real system. Callsite context includes information that the aspect uses from the joinpoint, including method parameters and call flow information.

Motivation: Changes to the control flow such as exceptions, method calls, and recursive calls should be identified and tested in the mock system. Creating appropriate callsite information for method calls in the mock system includes using a similar number of arguments and argument types (e.g., simple scalar types, user-defined types, pointers, and templatized types such as STL containers).

In emulating the control flow context of the real system, particular attention should be given to potential causes of infinite recursion, which Laddad identifies as a common mistake in adopting AOP [21]. For example, an aspect whose pointcut may match a call within its own advice may lead to unintended infinite recursion unless the pointcut is constrained (e.g., through a more restrictive pattern or using AspectC++ directives such as `within` or `cflow`). In addition, aspects may advise recursive methods in the real system. Developers can create intra-method calls and recursive calls in the mock system to test the advice with existing recursion.

Aspects that throw or catch exceptions may be changing the resulting control flow and the exceptions seen by callers. Throwing and catching exceptions can be tested in the mock system.

Steps:

(1) Identify callsite 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 catches exceptions thrown by the real system, throw these exceptions in the mock system.
(3) If advice throws exceptions, catch these exceptions in the mock system.
(4) Identify call flow information (such as `cflow`) and recursion that exists in the real system and emulate it in the mock system.

Example: In the `PowerAnalyzer`, functions that were timed were nested, calling other functions that were also timed. We modeled this structure in the mock system to check that (1) the timing information for each function was correct, (2) the nested timed calls reported correct timing information, and (3) the nested calls did not lead to infinite recursion. In the caching aspect, 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.

6.5 Provide functionality used by advice

System functionality used by advice is imported into or emulated by the mock system.

Motivation: To test the advice, the mock system must provide the methods called by the advice and 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. To test an aspect that modularizes a system policy, the mock system must provide functionality that the advice depends upon. We can use existing system components in the mock system or create new mock classes for components that are large or difficult to import.

Steps:

(1) Identify any system components called from 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.

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 in the `ErcChecker` mock system.

By using the same names for the `Logging` class and the mock container class (`LevelManager`), we used the aspect in the `ErcChecker` without changing the methods called. During system refactoring, we identified an additional method that needed to be called from the advice. We first emulated the changes in the mock system, and then continued refactoring the real system.

6.6 Use annotations in the mock system to 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 check the weave results for incorrect pointcuts.

Motivation: Pointcut strength faults [1] are particularly difficult to test for. 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 are woven with a mock system, our annotation approach (see Sections 3.2 and 4.1) would allow us to specify the specific aspect that should or should not advise a method. By choosing the class and method names in the mock system based on those in the real system, we increase our confidence that the pointcuts will choose correct joinpoints in the real system.

Steps:

(1) Create namespaces, classes, and names that should not match the pointcut but are similar (e.g, same prefix to class names or namespace names).
(2) Create namespaces 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) Add annotations to indicate methods that should be advised and which should not be advised.

Example: For regular expression types of pointcuts 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).

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 weave analyzer checks that the `ADVISED_BY(PowerOnlyFilter)` annotation did have an associated advice joinpoint from the `PowerOnlyFilter` aspect. We also used annotations to indicate functions that should not have any advice:

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

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

Motivation: Aspects may interact with one another by many means, including advising the same method and introducing members or methods to the same class [10]. These intended interactions should be modeled in a mock system by creating the conditions (e.g., overlapping joinpoints) that are anticipated in the real system. For example, we can create a mock class method that will match two aspects and test the combined behavior.

Although the real system may contain unanticipated aspect-aspect interactions, the mock system tests that the aspects work together correctly in at least some situations. Like unit testing, this provides an environment to validate basic functionality before large-scale integration.

Steps:

(1) Identify joinpoints in the real system that will be matched by more than one aspect.
(2) Create the mock system to contain joinpoints advised by these aspects.
(3) After weaving aspects with the real system, use analysis of weave results to identify any unexpected shared joinpoints that should also be tested within the mock system.

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 tested together in the first caching mock system. The aspects had identical pointcuts to advise the same methods. Because the pointcuts were the same, we did not have to change the mock system to test the interaction of multiple aspects. Both aspects were woven with the mock system. During testing, we found that order of execution was important because both aspects used `proceed()`. If the caching aspect executed first and had stored the value already, it returned without calling `proceed()`, which prevented the other aspect from advising the call as well. We chose not to use this two aspect version of caching; thus, it was not tested in the performance-oriented mock system.

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

7 Evaluation

In Section 5 we describe the process of creating aspects and mock systems to aspectualize two legacy systems. In this section 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. Then we examine threats to validity.

7.1 Evaluation Questions

The goal of the 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.

7.2 ErcChecker policy aspects

7.2.1 What costs are incurred in creating a mock system?

We created the mock system in one hour by inspecting the classes and components called by the `ErcQuery` hierarchy. The mock system had a small class hierarchy based on the `ErcQuery` class hierarchy. The mock hierarchy contained four classes and had a depth of inheritance of two. The mock system also implemented classes that are used by the mock system hierarchy or by advice. For the `ErcChecker` policy aspects, the mock system we created contained 160 LOC.

7.2.2 Does using mock systems save time when an aspect requires repeated cycles to create?

Weaving takes 15 minutes. Compiling the woven code takes 10 minutes. Thus, making and testing small changes in pointcuts or advice requires 25 minutes. The mock system can be compiled and woven in one minute.

Due to the wide scope of refactoring (59 subclasses to consider) [31] and errors in the initial aspect advice, and because several faults were encountered doing refactoring [32], 10 iterations were needed during the creation of the aspect. Since the time saved when weaving and compiling by using a mock system is 24 minutes, the compilation and weave savings were 240 minutes. The mock took 60 minutes to create, so the total savings of the mock systems approach was 180 minutes.

7.2.3 Did the aspects created in the mock system require changes to work in the real system?

We discovered a required change in the `ErcQuery` hierarchy for the aspect to work with all 59 subclasses. This also required a one line addition to the advice. A more thorough inspection of the `ErcChecker` could have identified this before creating the mock system. The pointcut did not have to be changed when the aspect was woven with the real system.

7.3 ErcChecker caching aspects

7.3.1 What costs are incurred in creating a mock system?

We created two mock systems. The first mock system focused on functional behavior and modeled pointers to complex data types and different scalar types. The mock system consisted of 400 lines of C++ code and also imported the `BlockData` component. Creating it took 45 minutes. Using the mock system, a prototype caching aspect could be woven, compiled, and run in less than a minute.

Once we developed the caching aspect, we created a second mock system for measuring performance. We compared the performance overhead of the AspectC++ caching using around advice to the overhead of the scattered C++ caching code. This mock system contained 200 lines of C++ code and did not import the `BlockData` component. Creating it took 20 minutes.

7.3.2 Does using mock systems save time when an aspect requires repeated cycles to create?

We used 15 iterations to create the caching aspect to evaluate different approaches. Weaving and compiling the `ErcChecker` would have dramatically slowed down our ability to test and evaluate the aspect since compiling and weaving the `ErcChecker` requires 25 minutes. For 15 iterations to create the caching aspect, using mock systems saved 24 minutes per iteration, for a total of 360 minutes, minus the mock development time (80 minutes). Thus, total savings was 280 minutes (4.67 hours).

7.3.3 Did the aspects created in the mock system require changes to work in the real system?

The only changed needed in the aspect was in the pointcut definition. In the concrete aspect, it listed all functions to be cached. Changing the pointcut was done by using the names of caching functions we had identified previously. The abstract aspect that contained the caching advice code did not change.

7.4 PowerAnalyzer Tracing Aspect

7.4.1 What costs are incurred in creating a mock system?

We reused the `BlockData` component and created 60 lines of C++ code that made calls to different types of framework iterators, and similarly named methods that should not match. Creating the mock system took 30 minutes.

7.4.2 Does using mock systems save time when an aspect requires repeated cycles to create?

Once the aspect was created, weaving and compiling it within the mock system took less than a minute. The CadTrace aspect was woven with the `libPower` component of the `PowerAnalyzer`. Weave time and compilation time total 4 minutes.

Since this aspect worked correctly on the first iteration, there was no time savings from compiling and weaving the aspect with a smaller system during development. Using the mock system introduced a 30 minute cost.

7.4.3 Did the aspects created in the mock system require changes to work in the real system?

The aspect was woven with the `PowerAnalyzer` without changes to the pointcut or advice.

7.5 PowerAnalyzer Timing Aspect

7.5.1 What costs are incurred in creating a mock system?

The mock system provided similar naming conventions as the `PowerAnalyzer`. We created two methods that matched the proposed pointcut naming convention and two that did not, and had one advised method call the other. The mock system contained 50 LOC and was created in 30 minutes.

7.5.2 Does using mock systems save time when an aspect requires repeated cycles to create?

Each of `PowerAnalyzer` executables takes three minutes to weave and compile, so nine minutes are required to weave the aspect with the full system. Weaving the aspect with the mock system, compiling and running together took under a minute.

Iterative aspect development was important because we changed the pointcut twice as we experimented with pointcuts and advice to handle nested calls advised by the `TimeEvent` aspect. The advice body also went through two iterations to finalize the format and information logged. Making these changes using the mock system provided feedback in less than one minute.

The four iterations we used to create the aspect took only four minutes of run time in the mock system. Had the full system been used, these iterations would have taken 36 minutes. The compile/weave time savings (32 minutes) minus the mock creation time (30 minutes) means that the total time saved was 2 minutes.

7.5.3 Did the aspects created in the mock system require changes to work in the real system?

No, the aspect was woven without change with the `PowerAnalyzer` because we did not find any issues that required going back to the mock system for evaluating further changes.

7.6 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.

Table 1 summarizes our data for each aspect. The 'Iterations' column is the number of iterations used to create the aspect. The 'Mock LOC' shows the size (lines of code) of the mock system, while 'Mock Creation Time' is the time (in minutes) spent creating the mock systems. The 'Changes' column indicates if changes had to be made in the aspects when moving them from the mock system to the real system. The 'Time saved' column is the time saved in the aspect creation iterations minus the time to create the mock system. The negative value for the `CadTrace` aspect indicates a net cost to use the mock system.

For the `ErcChecker`, which used 10–15 iterations to create aspects, there was a clear time savings. For the `PowerAnalyzer`, the time savings is 2 minutes

Table 1
Summary of mock systems data for each aspect.

Aspect	Iterations	Mock Creation Time	Mock LOC	Changes	Time saved
QueryPolicy	10	60 min	160	advice	3 hrs
Caching	15	65 min	600	pointcut	4.67 hrs
CadTrace	1	30 min	60	none	-0.5 hrs
TimeEvent	4	30 min	50	none	2 mins

for the TimeEvent aspect and negative (a cost) for the `CadTrace` aspect.

Use of the mock system did not save time when aspectualizing the `PowerAnalyzer`. However, the use of a mock system to aspectualize the larger `ErcChecker` did save significant time. This savings was due to the longer compile times and the increased number of iterations. For caching, this was due to many different caching strategies being considered and evaluated. 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. One hypothesis that we propose is mock systems will provide a greater time savings when aspectualizing large systems. In addition, mock systems will provide greater cost benefits for aspects that use more complex advice and hence require more iterations to create and validate.

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.

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 judgement is required when deciding what classes and methods are needed in the mock system. In our work, we quickly created mock systems and were not motivated to develop automated tools.

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 two 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.

8 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. Similar tools could be developed for other aspect-oriented languages, such as AspectJ.

We demonstrate our approach by refactoring two 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 two 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.

The aspects enable cross-cutting concerns, such as specific design policies, caching, debug tracing, and timing to be implemented in a more modular way. A development aspect that is not a permanent part of the program supports a pluggable approach, in which weaving, not manual code changes, is used to enable or disable tracing.

We continue to evaluate our test driven approach as these programs are refactored. We are studying how mock systems can also be used to 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. We are exploring other legacy C++ systems within Hewlett-Packard that use the same framework as these applications do.

One goal of mock systems is to create aspects that do not require change when woven with the real system. 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.

Testing more complex aspects leads to an open question: are there criteria or measures that should be used 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 [14] on mock systems. Crosscutting interfaces (XPIs) use 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.

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.

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. Understanding the costs and benefits of a mock system over time is an important extension of this work.

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.

References

[1] 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.

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

[3] AspectJ project.
http://www.eclipse.org/aspectj/.

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

[5] 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 International Conference on Software Maintenance*, page 368, Washington, DC, USA, 1998. IEEE Computer Society.

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

[7] Curtis Clifton and Gary T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. In *AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies*, March 2003.

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

[9] 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.

[10] 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.

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

[12] Andreas Gal, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. AspectC++: Language proposal and prototype implementation. In Kris De Volder, Maurice Glandrup, Siobhán Clarke, and Robert Filman, editors, *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001)*, October 2001.

[13] 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.

[14] 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.

[15] 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.

[16] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press, 2002.

[17] Jan Hannemann, Gail Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. In Peri Tarr, editor, *Proceedings of the 4th International*

Conference on Aspect-Oriented Software Development (AOSD-2005), pages 135–146. ACM Press, March 2005.

[18] 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.

[19] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *11th European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[20] Ramnivas Laddad. Aspect-oriented refactoring. Technical report, The ServerSide.com, 2003.

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

[22] Otavio Augusto Lazzarini Lemos, Fabiano Cutigi Ferrari, Paulo Cesar Masiero, and Cristina Videira Lopes. Testing aspect-oriented programming pointcut descriptors. In *WTAOP '06: Proceedings of the 2nd workshop on Testing aspect-oriented programs*, pages 33–38, New York, NY, USA, 2006. ACM Press.

[23] 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.

[24] Nick Lesiecki. Unit test your aspects. Technical report, Java Technology Zone for IBM's Developer Works, November 2005.

[25] 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, *Proc. Generative Programming and Component Engineering: Third International Conference*, volume 3286 of *Springer-Verlag Lecture Notes in Computer Science*, pages 55–74. Springer, October 2004.

[26] 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.

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

[28] 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.

[29] 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.

[30] Michael Mortensen and Sudipto Ghosh. Creating pluggable and reusable non-functional aspects in AspectC++. In *Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 1–7, Bonn, Germany, March 20 2006.

[31] 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, March 2006.

[32] Michael Mortensen, Sudipto Ghosh, and James Bieman. Testing during refactoring: Adding aspects to legacy systems. In *17th International Symposium on Software Reliability Engineering (ISSRE 06)*, November 2006.

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

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

[35] Tom Tourwé, Johan Brichau, and Kris Gybels. On the existence of the AOSD-evolution paradox. In *AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies*, March 2003.

[36] 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.

[37] Jianjun Zhao. Unit testing for aspect-oriented programs. Technical Report SE-141-6, Information Processing Society of Japan (IPSJ), May 2003.

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