Adequate Testing of Aspect-Oriented Programs*

Michael Mortensen
Hewlett-Packard, Colorado State University
3404 E. Harmony Rd MS 32
Fort Collins, CO 80528 USA
1-970-898-0686
mmo@fc.hp.com

Roger T. Alexander
Colorado State University
Department of Computer Science
601 S. Howes Street
Fort Collins, Colorado 80523 USA
1-970-491-7026
rta@cs.colostate.edu

ABSTRACT
Aspect-oriented programming supports the separation of concerns into traditional core concerns and cross-cutting aspects. Aspects typically contain new code fragments that are introduced to the system (such as advice or introductions) and a means of quantification that specifies where these code fragments are to be inserted. Although powerful, aspect-oriented programming includes new sources for program faults. These faults may be due to errors in the aspect-based code fragments or in the quantification directives. To address both sources of error, we propose combining two traditional techniques to adequately test aspect-oriented software: white box coverage and mutation testing. We use static analysis of an aspect within a system to guide in the selection of white box criteria for an aspect (such as statement coverage, context coverage, and def-use coverage) that will determine if a test suite is adequately testing the new code fragments. In addition, we provide a set of mutation operators to evaluate if a test suite is sufficiently sensitive to find errors in pointcuts and aspect precedence. Together, coverage criteria and mutation testing provide a framework for defining adequate testing of aspect-oriented programs.

Categories and Subject Descriptors
D.2.5 **[Testing and Debugging]**: Testing tools D.3.3 [**Language Constructs and Features**]: Control structures, Data types and structures, Inheritance, and Polymorphism.

General Terms
Reliability, Languages, Verification.

Keywords
Aspect-orientation, AspectJ, Coverage testing, Mutation testing

1. INTRODUCTION
Aspect-Oriented Programming supports separating concerns into traditional structures (classes, methods) and cross-cutting aspects [12]. Numerous examples have demonstrated the potential for reducing explicit coupling and increasing modularity of the core concerns in such areas as logging, persistence, and security. Aspect-oriented programming accomplishes this by modularizing code fragments with quantification over the underlying syntax tree [10] so that these fragments may be automatically inserted into a system using an *aspect weaver*.

While providing great flexibility and power, both an aspect's code fragments and its quantification directives are sources for new types of programming faults. Testing aspect-oriented systems must consider faults due to either the fragments or where they are inserted through pointcuts (the quantification). In addition, the types of code inserted by an aspect can be very different, so different aspects may require different testing strategies. For example, a *method introduction* is very different than *before advice*.

Alexander, Bieman, and Andrews [2] point out the need for a systematic approach to testing aspects with the following questions: "How do we adequately test aspect-oriented programs? How do we know that our testing and quality objectives have been attained?" These questions are difficult, they point out, because of the following issues:

- **Aspects do not exist independently.** We must analyze an aspect in the context of its use in a program. Even if a particular aspect has been used before, a new program supplies a new use context that must be re-verified, just as inheritance forces retesting of unchanged, inherited methods [17].
- **Aspects can be tightly coupled to their woven context.** Although systems can be designed so that core concerns are oblivious to aspects [10], aspects are often tightly coupled to properties of the underlying system, such as method names, class names, and return types [3]. This can complicate aspect comprehension and maintenance.
- **Control and data dependencies are not readily apparent from the source code of aspects or classes.** Aspects may change existing dependencies as well as introducing new dependencies.
- **Faults may result from emergent properties arising from how the core concerns and aspects interact.** Examples include side effects or aspect precedence.

Many researchers are investigating techniques for reasoning about aspect-oriented systems, with the goal of understanding and verifying the effects of aspects on the underlying primary concerns as well as aspect interactions [3][5][7][9][19][21]. Although many types of errors can be detected, in general we cannot prove an aspect-oriented program is correct, and so testing complements formal and informal approaches to verification.

Fault-based testing focuses on testing specific kinds of faults that are more likely to exist. Two basic assumptions of fault-based testing are the Competent Programmer Hypothesis and the Coupling Effect [16]. The Competent Programmer Hypothesis assumes that the program may be incorrect, but will only differ by relatively simple faults. According to Offutt[16], the coupling effect states that a test suite that detects all simple faults is sensitive enough to detect more complex faults.

We propose to apply fault-based testing to aspect-oriented programs using both coverage and mutation techniques. Coverage criteria provide a definition of test adequacy in terms of testing statements, branches, def-use pairs, etc, and are helpful in determining if testing has met a measurable goal of "enough testing". Mutation testing injects faults into an existing system to see if the test suite is sensitive enough to detect common faults. While the ideas presented would be applicable to various aspect-oriented programming languages, we will use AspectJ as the basis of discussion.

The rest of the paper is organized as follows. Section 2 considers different types of coverage criteria and what AspectJ structures would motivate their use. Section 3 presents a mutation-based approach for assessing a test suite relative to where aspect code is inserted. Example applications of coverage and mutation testing are provided in section 4, followed by a discussion section. Conclusions and future work are discussed in section 6.

2. COVERAGE CRITERIA FOR ASPECT-ORIENTED PROGRAMS

The primary goal of coverage criteria is to measure the amount of testing that has been done against some measure or goal. For aspect-oriented testing, coverage criteria is intended to ensure that aspect code fragments (e.g. advice statements) are tested.

2.1 Coverage Criteria

A test requirement is a specific feature to be satisfied or covered during testing, such as reaching a statement for statement coverage. A testing criterion is a rule or set of rules used to evaluate a set of test cases (known as a test suite). Testing can be evaluated in terms of coverage, which is the percentage of test requirements from a test criterion that are satisfied. The expected output of a test is the result if software behaves correctly. Expected outputs may be determined internally (such as the return value of a function) or externally (by examining the output or log file produced by a program) [1].

Testing criteria can be used to measure the quality of an existing test suite, or they can be used to guide the generation of test cases. Testing criteria can be compared using a subsumption hierarchy. Criterion A subsumes criterion B if each test set that satisfies A also satisfies B [1]. Criteria at higher levels in a subsumption hierarchy are viewed as having more testing power, but have a higher cost. In particular, more powerful criteria may have some test requirements that are not possible, such as infeasible def-use pairs or infeasible paths.

Some common coverage criteria (in order of subsumption from least powerful to most powerful) include statement coverage (every statement in a program must be executed at least once), branch coverage (every branch of a program must evaluate to both true and false at least once), and condition coverage (each condition in a branch must also evaluate to both true and false at least once) [15].

Dataflow testing focuses on testing subpaths between the definition of a variable to points where that variable is used. If the use is a predicate use, then tests must traverse subpaths to both branches of the predicate. Def-use coverage is a dataflow approach to testing. A *def* occurs when a variable is assigned a value. A *use* occurs when a variables is used as part of a computation or predicate. A *def-use pair* occurs when we have a path from a def to a use without an intervening def (the path is termed a def-clear path). Rapps and Weyuker show that All-Paths subsumes all-def-use paths, which subsumes all-branches [18].

Harrold and Rothermel have extended dataflow testing for handling object-oriented classes, by testing def-use pairs at the method level, inter-method level, and class level [11]. With inheritance and polymorphism in object-oriented systems, the def-use pairs that occur between method calls on an object can depend on which class in a class hierarchy an object is bound to. Alexander and Offutt [1] present a criteria for testing these indirect coupling relationships. For example, All-Coupling-Sequences requires that if there is an object *o* in a client with methods *m()* and *n()* that could, depending on the type, have an indirect def-use coupling (that is, *m()* sets a state variable which *n()* uses), then we must test this sequence at least once. A stronger criterion is All-Poly-Classes, which requires that each class in a hierarchy that could have a def-use pair between the coupled methods be tested.

2.2 Aspect Variations

One challenge with selecting coverage criteria for aspects is that aspects can have very different effects on the core concerns and the system itself. Some aspects, such as logging, may be strongly orthogonal in that they do not interact with the behavior or state of the core concerns in any way [8]. Other aspects, such as introducing a data member that does not directly interact with the core concern, may be weakly orthogonal. Weakly orthogonal aspects may unnoticeable to clients of the core concern in some programs, but in some contexts (such as serialization) the extra data member may cause system changes [8].

In addition, aspects may have internal state or utilize object state. An aspect can be considered stateful if it has its own state that is defined and used in join points, or if it modifies or uses the state of other objects during a sequence of join points [9]. Zhao and Rinard have extended existing system dependence analysis to AspectJ [22], and Zhao has used this as a basis to extend Harrold and Rothermel's approach to handle dataflow testing of AspectJ programs [23].

Method introduction by an aspect into a class hierarchy can change the dynamic binding of an object's call. Storzer and Krinke call this binding interference, and analyze the inheritance structure and method introductions to determine what calls could be potentially changed [19][20].

2.3 A candidate fault model for Aspect-oriented programs

Alexander, Bieman, and Andrews have defined an initial candidate fault model for AOPs with six classes of AOP-specific faults [2]. These are in addition to faults that can (previously) exist in object-oriented systems such as Java [13].

2.3.1 Incorrect strength in pointcut patterns
If a pointcut is too weak, it will select unintended join points that should not have had aspect code fragments added. If a pointcut is too strong, some join points will be unintentionally missed.

2.3.2 Incorrect aspect precedence
For systems with multiple aspects, either the default or the specific aspect precedence may be incorrect, resuling in incorrect behavior. This can be problematic for stateful aspects that are reading or writing common state variables in the core concern.

2.3.3 Failure to establish expected postconditions
Clients of a system will expect postconditions to hold even in the presence of aspects, as is required for behavioral inheritance (sub-typing).

2.3.4 Failure to preserve state invariants
Aspect-based changes introduced to the core concerns, either directly or indirectly, should not violate state invariants.

2.3.5 Incorrect focus of control flow
In addition to using pointcuts or introductions to statically determine where aspect code fragements are inserted, aspects can use dynamic context such as *cflowbelow* to specify when advice should be activated. Incorrect use of dynamic context can lead to errors or performance issues, such as jumping aspects [Brichau].

2.3.6 Incorrect changes in control dependencies
Aspects may introduce dataflow or control flow changes in the primary concern, which can change control dependencies. While such changes are often the goal of the aspect, they are potential sources of faults.

2.4 Aspect Coverage Criteria
Based on the body of coverage criteria research overviewed thus far, we propose a set of coverage criteria as well as guidelines for when – depending on aspect structure – they are appropriate.

2.4.1 Statement coverage
An aspect code fragment is covered by statement coverage if every path through the fragment is executed at least once after being woven into a program. As with statement coverage for procedural and object-oriented programs, statement coverage is not particularly strong but serves as a lower bound of test coverage. Clearly we should strive to test all code, recognizing that exceptions to this should be rare.

2.4.2 Insertion coverage
Insertion coverage means testing each aspect code fragment at each point it is woven into the program. Before or after advice should be tested with each method with which it can be associated. An introduced data member should be tested with each method that sets or uses it. An introduced method should be tested with each class that could use it (either directly or through inheritance) in the same manner as all-poly-classes coverage [1]. This is particularly important for advice or introductions that can modify program state or affect control or data dependencies.

2.4.3 Context coverage
Context coverage extends insertion coverage to test an aspect code fragment in each place it is used. For a piece of before or after advice, for example, this ensures that the advice is tested wherever the associated method is called. This level of coverage, while expensive, is appropriate for advice that not only changes state, but makes changes depending on the current program state or context in which it is called. For a method introduction, this means testing the method with each class in each place in which it can be used, which is equivalent to all-poly-uses coverage in OO testing.

2.4.4 Def-use coverage
For aspects, def-use coverage depends on the type of aspect code fragment. For a data member introduction, this would be testing def-use pairs within methods or advice or between methods and advice. For method introduction, this means testing indirect def-use pairs between the introduced method and other methods where both def and use the same state variable. For advice, this means testing def/use pairs within advice, between different advice fragments, between advice and methods, and between methods where the control flow has changed due to advice control flow changes (such as around advice). Def-use coverage is not meaningful for aspects (such as logging) that do not def any aspect or core concern state variables.

2.5 Selecting coverage criteria for aspects
Statement coverage would be at the ‘bottom’ of the subsumption hierarchy, and would be subsumed by insertion coverage, which is likewise subsumed by context coverage. Since an introduced method or advice may define a variable that is not used, in some cases def-use coverage of interactions between aspects and core concerns may not subsume all context coverage (e.g. focusing on testing only def-use pairs may omit valid paths unrelated to the aspect code). However, an aspect code fragment that defines a variable but does not use it should be investigated as a potential anomaly. Traditionally def-use coverage subsumes branch testing because *all* variables are included in the analysis, whereas here we are focusing on variables from aspect code fragments.

For the purpose of selecting an appropriate coverage criterion, it is necessary to first perform static analysis on the aspect and system in which it is used. Selecting test criteria is always a trade-off between confidence in our testing and the cost of that testing. Designating an appropriate criterion is not meant to imply exhaustive coverage, but rather to match the relative complexity (or amount of possible interaction) of an aspect with a level of testing that will provide confidence in the tests performed.

Based on existing research in aspect-oriented progamming and analysis [8][9], we categorize aspects as follows:

- Orthogonal (either dependent or independent)
- Altering (whether directly or indirectly)
- Stateful

Orthogonal aspects do not change control or data dependencies in the system; an example would be logging [8]. After weaving, orthogonal aspects may be (control or data) *dependent* on primary concerns or *independent* if the inserted statements execute without regard to core concern state.

An *altering* aspect changes control flow or data flow of a system. *Directly altering* aspects include around advice and dynamic binding interference (which can occur due to method introduction or class hierarchy changing). *Indirectly altering* aspects introduce dataflow changes that may change state variables or predicate values that affect control flow. For example, before advice may modify an object that is passed as a parameter to a method in a way that changes method behavior.

A *stateful* aspect has behavior that depends on one or more state variables, such as an aspect data member or introduced object state variable [9]. A stateful aspect can be orthogonal (if the state is internal to the aspect or does not interact with the core concern), or can be altering.

We propose that testing criteria for an aspect be selected based on the properties identified above (orthogonal, altering, and stateful). Since testing resources (time, effort) are finite, a goal of testing is to identify a criteria that will not be too difficult while still testing the code in a meaningful way.

Because they do not interact directly with the core concern and don't depend on state, aspects that are orthogonal and non-stateful are adequately tested with statement coverage. Orthogonal aspects with state should use insertion coverage to ensure that aspect code fragments are tested with each program location.

A minimum baseline for altering aspects should be insertion coverage, but more appropriate choices would be either context coverage or def-use coverage. Aspects with complex dataflow interactions might seem best suited for def-use coverage, but complex dataflow interactions may lead to infeasible or difficult to execute def/use pairs. Simpler dataflow relationships such as caching method return values to improve efficiency can be easily tested with def-use coverage to validate that both inserting and retrieving a value into the aspect cache work together appropriately.

2.5.1 Coverage criteria and the fault model

The first two faults in the AOP fault model presented earlier related to pointcut strength and aspect precedence, which govern where code is woven in. Coverage testing cannot test for missing code and may not detect code that is inserted in the wrong place (e.g. through incorrect pointcut strength).

The other four faults can be the result of incorrect aspect code fragments (such as faulty advice). The different coverage criteria provide a means for testing these fragments in an appropriately for a given aspect.

Statement coverage is the weakest, and may only find faults if the fault occurs in all weave contexts. Insertion coverage, context coverage, and def-use coverage provide progressively stronger coverage criteria for testing for the presence of faults.

Context coverage could detect incorrect focus of control flow, but to be complete def-use coverage ensure that context considered all state-based decisions through def-use pairs. Def-use testing also corresponds best to incorrect changes in control dependencies.

3. ASPECT MUTATION TESTING

Mutation testing is used to insert faults and evaluate the effectiveness of a test suite. For aspect-oriented programs, we use mutation testing to focus on faults related to advice fragment insertion. By mutating pointcuts and precedence, we can evaluate the effectiveness of the testsuite at finding the first two faults: incorrect strength in pointcut patterns and incorrect aspect precedence.

3.1 Mutation Testing

Mutation testing takes a program and test suite, and introduces faults via mutation operators. The resulting, modified programs are referred to as *mutants*. A mutant is considered *dead* if a test from the test suite distinguishes the output from the expected output. A test that does so is termed *effective* and is said to *kill* the mutant [16].

If some mutants are still alive, a tester can attempt to kill them by introducing new tests to the suite. If a mutant can be shown to be *functionally equivalent* to the original program, it is not counted in the mutation score. Functional equivalence is usually shown by hand, during the process of trying to kill mutants. A *mutation score* is calculated as the ratio of dead mutants over the total number of (non-equivalent) mutants [16]. The mutation score provides a measure of the extent of testing; while the live mutants pinpoint specific inadequacies in the test suite [14].

3.2 Mutation Faults and Operators

We focus mutation operators only on the process of inserting aspect code fragments. Thus, these operators are related to the first two faults from the fault model presented earlier. Mutation operators are inserted using static analysis of the aspect code fragments, pointcuts, and the resultant weave. We list the operators along with short abbreviations for reference. Note that one thing that we do not mutate is the presence or absence of dynamic context operators such as *cflowbelow*, since that can lead to infinite recursion in some aspects.

3.2.1 Pointcut strengthening (PCS)

This operator decreases the number of matched join pointjoin points by strengthening the pointcut. This can be done in several ways. For class hierarchy operators (such as *Type+.method()*), we can change the name of *Type* to a child type. For a pointcut name that matches multiple methods (such as *get*(..)*) we can change it to one of the matched methods (e.g. *getX(..)*). For a pointcut name that matches on multiple argument types, we can select the argument type of one of the matches. In addition, we can strengthen the pointcut so that it matches no join points (or equivalently make the advice empty). One benefit of this last mutation is that we validate if the test suite is sensitive to this pointcut at all.

3.2.2 Pointcut weakening (PCW)

The PCW operator is the oppositve of pointcut strengthening. For class hierarchy operators we can move the name of the *Type* up in the hierarchy. For pointcut names, we can change the pointcut name or type matching to be less restrictive. We can also perform the most extreme PCW operation: making the pointcut match all possible join points in a class or package. As with strengthening a pointcut to match nothing, this detects if the test suite is sensitive at all to the pointcut.

3.2.3 Precedence changing (PRC)

By changing the aspect precedence, we can determine the test suite (and system) sensitivity to aspect advice orderings. In order to maximize benefits, mutation testing typically selectively applies operators to only some parts of the system [14]. For PRC, we propose that this operator only be used with mutually interfering aspects [7] since we would not expect aspect precedence to matter otherwise.

4. APPLYING COVERAGE AND MUTATION TESTING

In order to illustrate the aspect coverage criteria and aspect mutation operators, we present several examples drawn from existing research. These examples have been modified to focus on the coverage or mutation operators. We use a numbering scheme that spans all examples so that we simply indicate a line number.

In addition, we have added a test suite for each test case and, if needed, a static *main* method. For simplicity, we assume that the test correctness is determined by examining the program output. For simplicity of demonstration our *test suites* are simply a sequence of calls in main rather than multiple invocations of the program with different values.

4.1 Optimizing using a caching aspect

We have adapted the factorial optimizer example presented by Alexander, Bieman, and Andrews [2], and put it in the context of a larger class (more than one static method) in order to demonstrate pointcut faults. The static method *MathFunctions.main* provides a simple test suite. The *MathFunctions* class is shown in Figure 1.

```
1.   public class MathFunctions {
2.    public static long factorial (int n) {
3.    if(n==0) {
4.      return 1;
5.    }
6.    else {
7.      return n* factorial(n-1);
8.    }
9.    public static long random_seed(int  n) {…}
10.   public static long random(long n) {
11.     //body omitted, returns a number in the range [0,n-1]
12.     //according to a pseudo-random algorithm that is
13.     //deterministic based on a seed value passed in
14.     //with random_seed()
15.   }
16.   public static main(String[] args) {
17.     System.out.println(“3! Is “ + factorial(3));
18.     System.out.println(“5! Is “ + factorial(5));
19.     System.out.println(“2! Is “ + factorial(2));
20.
21.     random_seed(5);
22.     System.out.println(“ random = “ + random(50000));
23.     System.out.println(“ random = “ + random(50000));
24.   }
25.  }
```

Figure 1. MathFunctions class and suite

The expected output (assuming some pseudorandom values for the random call based on the seed) for the program when executing the test suite embedded in *MathFunctions.main* is:

```
3! Is 6

5! Is 120

2! Is 2

random = 223202

random = 437293
```

The aspect for optimizing the factorial call is shown in Figure 2. It uses *cflowbelow* so that it only caches values around the top level call and not to lower level recursive calls.

```
26.  public aspect OptimizeFactorial {
27.    private Map _factorialCache = new HashMap();
28.    pointcut factorialOp(int n) :
29.        call(long *.factorial(int)) && args(n);
30.    pointcut topLevelFactorialOp(int n) :
31.        factorialOp(n) && !cflowbelow(factorialOp);
32.    long around(int n) : factorialOp(n) {
33.      Object cachedValue =
34.        _factorialCache.get(new Integer(n));
35.     if(cachedValue != null) {
36.         return ((Long) cachedValue).longValue();
37.     }
38.     return proceed(n);
39.    }
40.    after(int n) returning(long result)
41.        : topLevelFactorialOp(n) {
42.      _factorialCache.put(new Integer(n),
43.                        new Long(result));
44.    }
45.  }
46.
```

Figure 2. Factorial Optimizer Aspect

Weaving in the aspect in Figure 2 and rerunning *MathFunctions.main* results in the same output. Since the *OptimizeFactorial* aspect isn’t supposed to change behavior, this is a good start. However, analysis of the code for this test suite indicates that the around advice never uses a *cachedValue*, since no top level factorial calls were repeated with the same value. The after advice does get executed each factorial call (on lines 17-19).

The inadequacy of the existing tests illustrates how an aspect that does not change the observed behavior may need additional tests. To achieve statement coverage of the around advice, we can add a single statement to line 20:

```
System.out.println(“3! Is “ + factorial(3));
```

One of the benefits of coverage analysis is that we are able to gain a measure of observability not available by observing program outputs. For this simple example, no additional tests are required to test insertion coverage or context coverage. Insertion coverage is only meaningful if a pointcut selects multiple methods (from one or more classes). Context coverage is only meaningful is a method that has associated aspect code is called in different contexts.

This aspect is stateful, and we can apply def-use coverage. Each method call to factorial has both around and after advice, so that the woven code of a factorial call has – to the caller -- this structure (but not this actual representation in AspectJ):

```
long NewFactorial(int n) {
  Object cachedValue =
    _factorialCache.get(new Integer(n));
  if(cachedValue!=null) {
   return ((Long) cachedValue).longValue();
  long result = factorial(n);
  _factorialCache.put(new Integer(n),new Long(result));
  return result;
}
```

A sequence of calls to *factorial* can have a def-use pair between the put from one call to the get of a subsequent call. One challenge with def-use testing is that there will be chains of *potential* def-use pairs for the simple program of Figure 1 (with the added line 20). For example, if we treat *factorialCache* as a single def-use variable (which is a common, but pessimistic approach for collections and arrays), line 17 triggers a def to *factorialCache* that may be used on lines 18, 19, and 20. Only the def-use pair between the call on line 17 and the call on line 20 actually occurs.

Now that we have updated the test suite based on coverage analysis, we consider mutation analysis. Precedence changing does not apply to this example, because there is only one aspect. Pointcut strengthening (PCS) and pointcut weakening (PCW) can be used to generate mutants.

We apply PCS by creating mutant M_1, which changes the *factorialOp* pointcut so that it no longer matches any code (this is straightforward to do, since we can change the class name or method name to something that does not exist in the program). No test call to factorial can detect the change to M_1; however, this is because M_1 is functionally equivalent (since the absence of the advice doesn't change behavior).

For mutant M_2, we apply PCW by changing the pointcut to match any name with the required signature for the around:

```
pointcut factorialOp(int n) :
  call(* *.*(int)) && args(n);
```

We made as much of the pointcut specifier '*' as possible, but the argument *(int n)* is used by the advice, so we left its type and name intact. This pointcut does not match the call to *random*, because *random* has a *long* parameter, but it does match the call to *random_seed*. The result will be that the around advice will already have a cached value for the parameter 5, and will therefore not proceed with the actual call to *random_seed*, which will result in the calls to *random* giving different values. Mutant M_2 is dead because it the test suite will detect the change. Had the parameter types been identical for *random* and *factorial*, they also could have interacted in a detectable way.

4.2 Enforcing constraints with an aspect

The next example is adapted from an example by Zhao and Rinard [21] of using aspects to enforce constraints on a Point class. We have made slight changes to the code, omitted the Pipa annotations, and added a *main* method that serves as the test suite.

```
47.  public class Point {
48.    int x,y;
49.    public Point(int _x, int _y) {x = _x; y = _y;}
50.    public void setX(int newx)
51.    {       x = newx;     }
52.    public void setY(int newy)
53.    {      y = newy;      }
54.    public int getX() { return x;}
55.    public int getY() { return y;}
56.    public void printPosition() {
57.      System.out.println("Point at ("+x+","+y+")");
58.    }
59.    public static main(String[] args) {
60.      Point p1 = new Point(3,3);
61.      p1.setX(5);
62.      p1.setY(0);
63.      Point p2 = new Point(-1,-1);
64.      p2.setX(3);
65.      p2.setY(-1);
66.      p1.printPosition();
67.      p2.printPosition();
68.    }
```

Figure 3. Simple Point class

The *PointBoundsConditions* aspect is a combination of two aspects in Zhao and Rinard's paper and provides pre-condition (before) checks and a post-condition (after) check. It is shown in Figure 4. We assume that *MIN_X* and *MIN_Y* are 0, and that *MAX_X* and *MAX_Y* are some large value, such as the max int value.

```
69.  aspect PointBoundsConditions {
70.    before(int x) : call( void Point.setX(int)) && args(x)
71.    {
72.      if(x < MIN_X || x > MAX_X)
73.        throw new RunTimeException();
74.    }
75.    before(int y) : call( void Point.setY(int)) && args(y)
76.    {
77.      if(y < MIN_Y || y > MAX_Y)
78.        throw new RunTimeException();
79.    }
80.    after(Point p, int x) : call( void Point.setX(int))
81.      && target(p) && args(x) {
82.      if(p.getX() != x)
83.        throw new RunTimeException();
84.    }
85.    after(Point p, int x) : call( void Point.setY(int))
86.      && target(p) && args(y) {
87.      if(p.getY() != y)
88.        throw new RunTimeException();
89.    }
90.  }
```

Figure 4. PointBoundsConditions aspect

The *PointBoundsConditions* aspect does not maintain state (either directly or indirectly) but it does check the internal object state after each *setX* and *setY* method call. It alters the behavior as seen by the client or program output, since it prevents some method calls from completing, throwing a *RunTimeException* instead.

The output of the program before the aspects are added is:

```
Point at (5,0)
Point at (3,-1)
```

After aspect weaving, the output is:

```
Point at (5,0)
Run Time Exception at line….
```

In this case the default test cases provide statement coverage for the advice lines 72-73. The *setY* method never throws an exception, so line 77 is covered by not line 78. Lines 87-88 are not executed. We can analyze the code and determine that these lines are not reachable (we can never have the after exception in this program since *setX* and *setY* either set the value correctly or the program throws an exception). Automatically determining unreachability is an undecidable problem in general, and is one of the challenges of coverage testing (and why 100% coverage is not always achievable).

As before, insertion coverage and context coverage do not add to statement coverage due to the simplicity of the system. Each time *setX* or *setY* is called, we have a def-use pair, from the setting of the *Point* data member (*x* or *y*) to the use of the *Point* data member in the advice. The def-use pair is always covered by the same tests that provide statement coverage, so def-use coverage does not add to the strength of testing that advice fragment.

With only one aspect, we can apply both pointcut strengthening (PCS) and weakening (PCW) operators. We generate the following mutants:

M_1 – by removing the before setX() advice using PCS

M_2 – by removing the before setY() advice using PCS

M_3 – by removing the after setX() advice using PCS

M_4 – by removing the after setY() advice using PCS

We do not apply PCW because the advice uses the method argument and object state, which makes it difficult to apply the advice to other methods. Mutants M_2 and M_4 are killed by the test program since it calls *setY* with a negative value. Mutants M_1 and M_3 are not functionally equivalent to the original program, and indicate an insufficient test suite, which can be remedied by adding addition method calls to the end of the *main* method.

```
p2.setX(-3);
p1.printPosition();
p2.printPosition();
```

An interesting alternative to the before advice is to ensure that the *x* and *y* values are in a designated range instead of throwing an exception, as shown in Figure 5.

```
91.  aspect PointBoundsConditions {
92.    before(int x) : call( void Point.setX(int)) && args(x)
93.    {
94.       if(x < MIN_X) x = MIN_X;
95.       if(x > MAX_X) x = MAX_X;
96.     }
97.     before(int y) : call( void Point.setY(int)) && args(y)
98.    {
99.       if(y < MIN_Y) y = MIN_Y;
100.      if(y > MAX_Y) y = MAX_Y;
101.    }
102.    // after advice as previously defined on lines 80-89
103.    after(Point p, int x) : call( void Point.setX(int))
104.      && target(p) && args(x) {
105.      if(p.getX() != x)
106.         throw new RunTimeException();
107.    }
108.    after(Point p, int x) : call( void Point.setY(int))
109.      && target(p) && args(y) {
110.      if(p.getY() != y)
111.         throw new RunTimeException();
112.    }
113. }
```

Figure 5. Revised PointBoundsConditions aspect

The program output is now different, since the before advice for the *setY()* method does not throw exceptions, but now the after advice for the *setY()* method throws an exception at a different location because the result of *setY()* did not set the data value to the parameter.

This changes coverage testing in several ways. Previously, the before *setY()* advice was covered by a single out of range test case, but now we must test a value below *MIN_Y* and a value above *MAX_Y*. In addition, the after advice is now reachable and needs to be tested for methods *setX()* and *setY()*.

These two versions of *PointsBoundsChecking* show that advice bodies can interact through the state of a common object method call. Changes in advice may require changes to test suites in order to achieve adequate testing.

4.3 Testing aspect introduction

The third example is from Zhao and Rinard [22] and uses the same Point class but a *PointShadowProtocal* aspect, which uses a *Shadow* object that is introduced.

```
114. class Shadow {
115.   public static final int offset = 10;
116.   public int x,y;
117.   Shadow(int _x, int _y) {
118.     x = _x;
119.     y = _y;
120.   }
121.   public void printPosition() {
122.     System.out.println("Shadow at ("+x+","+y+")");
123.   }
124. }
125. aspect PointShadowProtocol {
126.   private int shadowCount = 0;
127.   public static int getShadowCount() {
128.       return PointShadowProtocol.
129.           aspectOf().shadowCount;
130.   }
131.   private Shadow Point.shadow;
132.   public static void associate(Point p, Shadow s) {
133.      p.shadow = s;
134.   }
135.   public static Shadow getShadow(Point p) {
136.      return p.shadow;
137.   }
138.
139.   pointcut setting(int x, int y, Point p) :
140.       args(x,y) && call(Point.new(int,int));
141.   pointcut settingX(Point p) :
142.       target(p) && call(void Point.setX(int));
143.   pointcut settingY(Point p) :
144.       target(p) && call(void Point.setY(int));
145.
146.   after(int x, int y, Point p) returning : setting(x,y,p) {
147.       associate(p,s);
148.       shadowCount++;
149.   }
150.   after(Point p) : settingX(p) {
151.      Shadow s = getShadow(p);
152.       s.x = p.getX() + Shadow.offset;
153.   }
154.   after(Point p) : settingY(p) {
155.      Shadow s = getShadow(p);
156.      s.y = p.getY() + Shadow.offset;
157.   }
158.   after(Point) : call( void Point.printPosition())
159.       && target(p) {
160.       Shadow s = getShadow(p);
161.       s.printPosition(); //print the shadow info too
162.   }
163. }
```

Figure 6. PointShadowProtocal aspect

The *PointShadowProtocol* aspect is stateful since it modifies and uses the Shadow object that it introduces to all Point objects. It does change the output of a program after weaving, since any call to *Point.printPosition()* will then call the associated *Shadow* object's *printPosition()* as well. For testing *Point* and *Shadow* together we re-show *Point.main()* below.

```
164. public static main(String[] args) {
165.    Point p1 = new Point(3,3);
166.    p1.setX(5);
167.    p1.setY(0);
168.    Point p2 = new Point(-1,-1);
169.    p2.setX(3);
170.    p2.setY(-1);
171.    p1.printPosition();
172.    p2.printPosition();
173. }
```

Figure 7. Point.main method

With just the *PointShadowProtocol* woven in, we consider coverage and mutation. The after *setX()*, after *setY()*, after construction advice, and after *Shadow.printPosition()* are all covered by this simple set of test calls. This test case does not have a complex enough structure for the insertion or context criteria to be different.

There are def-use pairs that occur through the advice, as well as between advice and methods. These include:

- A shadow def occurs after a Point constructor (e.g. line 2) and a use of that shadow occurs whenever the point's *setX()* or *setY()* methods are called (e.g. lines 3-4).
- A call to a *setX()* followed by a call to *printPosition()* on the same object results in a definition from the setting method being used in the *printPosition*, of both the *Point*'s internal state and the *Shadow*'s internal state.

These def-use pairs are a subset of the def-use pairs identified by Zhao and Rinard [23], since we only consider def-use pairs between actual method sequences in a client program, while they consider all def-use pairs based on all possible method sequences. From a testing point of view, considering all possible method sequence calls is more powerful, but may result in many sequence calls that are semantically invalid (such as popping an empty stack).

We can apply mutation testing to the pointcut of each advice fragment, resulting in the following mutants:

M_1 –by removing the after setting advice using PCS

M_2 –by removing the after settingX advice using PCS

M_3 –by removing the after settingY advice using PCS

M_4 –by removing the after printPosition advice using PCS

Mutant M_1 will result in subsequence advice calls attempting to reference a null Shadow variable, which will result in early termination and an exception being thrown. Mutant M_2 will cause the shadow to have an incorrect *x* value, which will be visible in the output. Mutant M_3 will likewise cause the shadow to have an incorrect *y* value. The effect of M_4 will be no shadow output that will also be detectable. All four mutants are killed by the existing test suite.

4.4 Interfering aspects and aspect precedence

If the *PointShadowProtocol* aspect were combined with an aspect similar to *PointBoundsConditions*, but which used after advice to set the values of a *Point*'s *x* and *y* values to be within a range (0…*MAX*), then aspect precedence becomes an issue.

If the *PointShadowProtocol* after advice is executed before *PointBoundsCondition* advice that changes the *Point x* and *y* values, then the *Shadow x* and *y* values will be based on a different set of *Point* values than the final point.

Statement coverage will still measure if both aspect's after advice is executed. In this case, neither insertion advice or context advice will be stronger, but what we really want to test is if **both** advice fragments were activated for the same method call. This is an area for further research.

Mutation testing will include the precedence changing operator (PRC), and will create two mutants to represent the two orders of these two aspects. One mutant will match the default behavior (since it will be functionally equivalent), while the other mutant will be killed. Regardless of which behavior turns out to be the default, one value of this set of mutants is that it shows that, for this system, aspect precedence does matter. Besides evaluating the test suite, this will likely result the expected aspect precedence being added to the system.

4.5 Calling Context

To demonstrate the effects of calling context, consider a slightly different math library, with a static Factorial method, a static Permute method that uses Factorial, and a static Root method that returns a square root.

```
174.  public class MathLib {
175.  public static long Factorial(long n)
176.  {
177.     if(n==1) return 1;
178.     else return n*Factorial(n-1);
179.  }
180.  public static double Root(long n)
181.  {
182.     // return the square root by some method…
183.     return root;
184.  }
185.  public static long Permute(long n, long r)
186.  {
187.     // P(n,r) = n! / (n-r)!
188.     return Factorial(n) / Factorial(n-r);
189.  }
190.
191.  public static void main(String[] args)
192.  {
193.     System.out.println("3!  is " + Factorial(3));
194.     System.out.println("P(3,2) is " + Permute(3,2));
195.     System.out.println("Root of 6 is " + Root(6));
196.
197.     System.out.println("P(2,3) is " + Permute(2,3));
198.  }
199. }
```

Figure 8. Another MathLib class

For the given test suite defined in *MathLib.main*, the *Factorial* static method gets called at three locations: recursively at line 178, twice from *Permute* at line 188, and by the test harness (acting as a client) at line 193.

Suppose that the *Factorial* and *Root* method do not handle negative inputs, and a client wishes to do that using an aspect rather than directly modifying code. One approach could be:

```
200. aspect Bounds {
201.    pointcut FactOrRootOp(long n) :
202.     (call(* *.Factorial(long)) ||
203.       call(* *.Root(long))) && args(n);
204.
205.  long around(long n) : FactOrRootOp(n) {
206.      if(n<0) {
207.         return 0;
208.        }
209.        return proceed(n);
210.    }
211. }
```

Figure 9. Bounds aspect for MathLib

The pointcut will match two methods, *Factorial* and *Root*. Statement coverage would simply require that the around advice code fragment be tested once, with either method. Insertion coverage would require testing it with each associated method (*Factorial* and *Root*) but at any call site. Context coverage would require testing the around advice fragment at each call site in the program, which would be four calls for *Factorial* listed above and the call to *Root* on line 195. The value of a stronger criteria (context coverage) is shown by the fact that the around advice fixes some possible program errors, but having *Factorial* return 0 can still lead to errors in some contexts – such as the denominator of *Permute*. This could be handled by changing the advice to return 1 or by modifying the pointcut to be sensitive to call location (e.g. using *cflowbelow*).

The mutation operators PCW and PCS can sometimes be performed as text manipulations on the class type or method type, but this example illustrates that another possible aplication would be removing part of a complex logical statement. In this case we create four mutants – one whose pointcut matches all *MathLib* methods, one whose pointcut matches only *Factorial*, one whose pointcut matches only *Root*, and one whose pointcut matches no methods.

5. DISCUSSION

The preceding examples demonstrate possible benefits from both coverage testing and mutation testing. Adequate coverage testing can be used to ensure advice that is semantics preserving is actually executing. In addition, coverage shows what advice is not getting used in a particular context.

Mutation testing can determine if a test suite is sufficiently sensitive to faults in pointcut strength and aspect precedence. Mutants that cannot be killed point to weaknesses in the test suite or aspects that may (regardless of intent) be orthogonal (since omitting them does not affect the system).

5.1 Tool Support

Although small examples can be examined by hand, clearly tool support is needed to help automate coverage testing. This will allow larger systems to be analyzed, and will allow developers to evaluate the effectiveness of such techniques.

Analysis of the Java core concerns, AspectJ aspects, and the resulting weave are necessary for steps in order to analyze and instrument AspectJ programs for coverage testing. While statement coverage is straightforward, insertion coverage, context coverage, and def-use coverage will require more detailed analysis of system traces to consider call stacks.

5.2 Aspect Coverage

We have only briefly considered dynamic interference due to hierarchy changes, method introduction, and interface introduction. Dynamic interference can lead to the same types of indirect def-use coupling as inheritance and polymorphism [1].

Another extension of def-use coverage would be to focus on defs and uses in the primary concern that occur due to aspect-induced control flow changes [2]. We also need to evaluate additional coverage criteria, such as executing all associated advice statements (from the same or multiple aspects) on a single method call.

5.3 Aspect Mutation

Two primary challenges for mutation testing are accurately simulating realistic faults, and limiting the number of mutants generated in large systems. Offutt [14] summarizes some common approaches: selective mutation to generate fewer mutants; mutant sampling to reduce how many mutants are actually run; and weak mutation that looks at internal state rather than program state to detect mutants.

The mutation operators presented focus on advice pointcut strength and aspect precedence. There are many different ways to mutate a pointcut beyond simple name changes, such as changing the logical operators (||, &&) and argument type changes. Another mutation approach might be to swap before and after advice that are associated with a common pointcut. Additionally, developing an approach to mutating pointcuts that uses target objects, arguments, and object state in a semantically meaningfully way would provide a richer set of mutants.

For aspects that change class hierarchy, we can perform mutation by changing aspect to move the class to a different level. We can also remove the operator so that the class is at its original hierarchy level.

Member introduction mutation was not considered in this paper, but one approach might be apply traditional scalar mutation operators (add 1, subtract 1, etc.). For introduction of Java collections, we can apply the five mutation operators described in by Bieman, Ghosh, and Alexander [4]:

1. Make the collection empty.
2. Remove an element from the collection.
3. Add an element to the collection.
4. Mutate elements within the collection.
5. Reorder some elements within the collection.

5.4 Experimental Validation

With adequate tool support, we can test many different aspect-oriented systems to experimentally measure the effectiveness of these criteria for particular systems. Since results from small programs created by hand for the purpose of testing and analysis may not have wide applicability, we need to apply our framework to a wide range or larger aspect-oriented systems.

Unlike procedural and object-oriented systems, we don't have a large repository of large AspectJ systems. However, as more researchers and practictioners develop systems in AspectJ, more and larger systems will be available for analysis.

This will be important for evaluating coverage testing. In large industrial systems, 100% statement coverage is rarely achieved. In addition, we will want to understand the overhead of measuring coverage with an aspect-oriented language.

Mutation testing has had limited use in part because of the large number of possible mutants to test [14]. Our approach applies mutation to only part of the system (such as pointcut strength and precedence), which may help mitigate this. Experimental studies and additional research in aspect-oriented systems can be used to validate our fault model, identify other mutation operators based on additional faults, and measure the cost of applying these mutation operators to large systems.

6. CONCLUSIONS AND FUTURE WORK

We have presented a new framework for aspect-oriented test adequacy that combines coverage testing and mutation testing. While our research is early – that is, it has not been experimentally validated – our proposed approach is novel in testing based on faults that can occur in both aspect code fragments and the quantification statements. We have demonstrated the application and benefits of this new approach on small AspectJ programs.

We have identified the need for an integrated set of tools to analyze AspectJ programs, instrument and gather coverage criteria, and generate and test program mutants. Our current and future work include developing these tools and using them as the basis for experimental validation of the proposed framework. Based on experimental validation and continued research, we plan to further refine and develop coverage criteria and mutation operators. We also want to explore the application of existing techniques for achieving testing goals with smaller mutant sets [14] to our aspect-specific mutation operators.

7. REFERENCES

[1] R. Alexander and J. Offutt. Criteria for Testing Polymorphic Relationships. in Eleventh IEEE International Symposium on Software Reliability Engineering (ISSRE '00). 2000. San Jose CA.

[2] R Alexander, J. Bieman, and A. Andrews. Towards the Systematic Testing of Aspect-Oriented Programs, AOSD 2005, submitted.

[3] L. Bergmans, Towards the Detection of Semantic Conflicts between Crosscutting Concerns, AAOS 2003 (Analysis of Aspect-Oriented Software), July 2003.

[4] J. Bieman, S. Ghosh, and R. Alexander, A Technique for Mutation of Java Objects, Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'2001) , IEEE Computer Society, San Diego CA, November 26-29 2001.

[5] L. Blair and M. Monga, Reasoning on AspectJ Programmes, GI-AOSDG 2003, Germany.

[6] H. Brichau, W. Meuter, and K. DeVolder, Jumping Aspects, Workshop on Advanced Separation of Concerns, ECOOP 2000, Cannes, France, 2000.

[7] L. Bussard, , L. Carver, E. Ernst, M. Jung, M. Robillard and A. Speck. "Safe Aspect Composition." Workshop on Aspects and Dimensions of Concern at ECOOP'2000, Cannes, France, June 2000.

[8] A. Colyer, A. Rashid, G. Blair, On the Separation of Concerns in Program Families. Technical Report Number: COMP-001-2004, http://www.comp.lancs.ac.uk/computing/aop/papers/COMP-001-2004.pdf.

[9] R. Douence, P. Fradet, and M. Sudholt, Composition, Reuse and Interaction Analysis of Stateful Aspects. AOSD 04, March 2004.

[10] R. Filman and D. Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness", Workshop on Advanced Separation of Concerns, OOPSLA 2000, October 2000, Minneapolis.

[11] M. Harrold and G. Rothermel. Performing Data Flow Testing on Classes. Proc. *ACM SIGSOFT Foundation of Software Engineering*, pp. 154-163, December 1994.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In *15th European Conference on Object-Oriented Programming*, Budapest, Hungary, 2001.

[13] J. Offutt, R. Alexander, Y. Wu, Q. Xiao, and C. Hutchinson, A Fault Model for Subtype Inheritance and Polymorphism, Proceedings of the 12th International Symposium on Software Reliability and Engineering (ISSRE01), IEEE Computer Society, Hong Kong, Nov 2001.

[14] J. Offutt and R. Untch, Mutation 2000: Uniting the Orthogonal. *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, 45--55, San Jose, CA, October 2000.

[15] J. Offutt and J. Voas. Subsumption of Condition Coverage Techniques by Mutation Testing. January 1996, ISSE-TR-96-01, http://www.isse.gmu.edu/techrep/1996/96_01_offutt.pdf.

[16] J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3--18, January 1992.

[17] D. Perry and G. Kaiser, Adequate Testing and Object-Oriented Programming. *Journal of Object-oriented Programming*, 1990: p. 13-19.

[18] S. Rapp and E. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367-375, Apr. 1985.

[19] M. Storzer, and J. Krinke, Inteference Analysis for AspectJ. Workshop on Foundations of Aspect-Oriented Languages (FOAL 2003) as part of AOSD 2003, March 2003.

[20] M. Storzer, Analysis of AspectJ Programs, 3rd German Workshop on Aspect-Oriented Software Development, Essen, Germany, March 2003.

[21] J. Zhao and M. Rinard, "Pipa: A Behavioral Interface Specification Language for AspectJ," Proceedings of Fundamental Approaches to Software Engineering (FASE'2003), LNCS 2621, pp.150-165, Springer-Verlag, April 2003.

[22] J. Zhao and M. Rinard, System Dependence Graph Construction for Aspect-Oriented Programs, MIT LCS Tech Report 891, March 2003, http://www.fit.ac.jp/~zhao/pub/ps/mit-lcs-tr-891.pdf.

[23] J. Zhao, Data-Flow-Based Unit Testing of Aspect-Oriented Programs. Proc. 27th Annual IEEE International Computer Software and Applications Conference (COMPSAC'2003), pp.188-197. Dallas, Texas, USA, November 2003.