

An Approach for Adequate Testing of AspectJ 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. This mechanism introduces new sources for program faults, 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: coverage and mutation testing. We use static analysis of an aspect within a system to guide in the selection of appropriate coverage criteria for aspect code fragments. 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 [10]. 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 [8] 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 and identify 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 [16].
- **Aspects can be tightly coupled to their woven context.** Although systems can be designed so that core concerns are oblivious to aspects [8], aspects are often tightly coupled to properties of the underlying system, such as method and class names, and return types [3].
- **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][4][5][7][18][19]. 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 [15]. The Competent Programmer Hypothesis assumes that the program may be incorrect, but will only differ by relatively simple faults. The coupling effect states that a test suite that detects all simple faults is sensitive enough to detect more complex faults [15].

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 for AspectJ. Section 3 presents aspect mutation. Example applications of coverage and mutation testing are provided in section 4. Conclusions and future work are discussed in section 5.

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.

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. Some common coverage criteria include statement coverage, branch coverage, condition coverage [14], and dataflow coverage [17].

Harrold and Rothermel have extended dataflow testing for handling object-oriented systems [9]. Alexander and Offutt [1] present criteria for testing the indirect coupling relationships in object-oriented programs that are due to inheritance and polymorphism. Zhao and Rinard have extended existing system dependence analysis to AspectJ [20], and Zhao has used this as a basis of dataflow testing of AspectJ programs [1].

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) might not interact with the behavior or state of the core concerns in any way [6] while others aspects introduce changes in the dynamic binding of an object’s call [18].

2.1 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 [12].

Incorrect strength in pointcut patterns will result in selecting unintended join points (if too weak) or in unintentionally missing pointcuts (if too strong).

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

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

Failure to preserve state invariants violates the integrity of the original class. Aspect-based changes introduced to the core concerns, either directly or indirectly, should not violate state invariants.

Incorrect focus of control flow can result in aspects that 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.

Incorrect changes in control dependencies can occur when aspects 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.

2.2 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.

2.2.1 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.

2.2.2 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.

2.2.3 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.

2.2.4 Def-use coverage

For aspects, def-use coverage depends on the type of aspect code fragment. Introductions as well as advice can introduce new def-use pairs related to the aspect code. 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).

2.3 Selecting coverage criteria for aspects

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 programming and analysis [6][7], we categorize aspects using the terms *orthogonal*, *altering*, and *stateful*.

Orthogonal aspects do not change control or data dependencies in the system; an example would be logging [6]. 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. A *stateful* aspect has behavior that depends on one or more state variables, such as an aspect data member or introduced object state variable [7].

Testing criteria for an aspect should be selected based on these properties. 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 can be easily tested with def-use coverage to validate that both inserting and retrieving a value into the aspect cache work together appropriately.

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 cannot detect missing code and might not detect code that woven incorrectly due to 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 and provide a means (but not a guarantee) of finding these types of faults.

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 [15].

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 [15]. The mutation score provides a measure of the extent of testing; while the live mutants pinpoint specific inadequacies in the test suite [13].

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 points by strengthening the pointcut. 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 we can change it to one of the matched methods. For a pointcut name that matches on multiple argument types, we can select the argument type of one of the matches. We can strengthen the pointcut so that it matches no join points to determine if the test suite is sensitive to this pointcut.

3.2.2 Pointcut weakening (PCW)

The PCW operator is the opposite 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 (to determine 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 [13]. For PRC, we propose that this operator only be used with mutually interfering aspects [5] 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 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, and our *test suites* are simply a sequence of calls in *main*.

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.     }
```

```
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. The around advice never uses a *cachedValue* because no top level factorial calls were repeated with the same value. The after advice does get executed for 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 if a method that has associated aspect code is called in different contexts.

Since this aspect is stateful, we consider def-use coverage. Each method call to factorial has both around and after advice, so that the woven code of a factorial call has a *use* of the cache and then, if a new result was computed, a *def* of the cache.

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 lines 17 and 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. M_2 is dead because if 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 [19] of using aspects to enforce constraints on a class. We have made slight changes to the code, omitted 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 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 (they 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).

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();
```

One alternative to the before advice is to change the x and y values are to be in some range rather than using exceptions. Mortensen and Alexander demonstrate that this changes both the program output and coverage for the given suite [11].

5. CONCLUSIONS AND FUTURE WORK

The preceding examples demonstrate some possible benefits from both coverage testing and mutation testing. Additional examples that include aspect introduction, interfering aspects, and aspect precedence have been provided by Mortensen and Alexander [11].

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.

Two primary challenges for mutation testing are accurately simulating realistic faults, and limiting the number of mutants generated in large systems. We restrict our use of mutation to pointcuts in order to restrict the number of mutants and because we feel that faults related to advice bodies may be found through coverage-based testing. We also need to validate our mutation operators to see if they correspond to real faults.

In this paper we have created mutants in an ad-hoc manner; automatic mutation of pointcuts needs more investigation. Mortensen and Alexander have also identified some uses of mutation for aspects that change hierarchy and that use collection classes as members [11].

We are developing an integrated set of tools to analyze AspectJ programs, instrument and gather coverage criteria, and generate and test program mutants. These tools will be used as the basis for experimental validation and refinement of the proposed framework.

6. 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, Technical Report CS-4-105, March 2004.
- [3] L. Bergmans, Towards the Detection of Semantic Conflicts between Crosscutting Concerns, AAOS 2003 (Analysis of Aspect-Oriented Software), July 2003.
- [4] L. Blair and M. Monga, Reasoning on AspectJ Programmes, GI-AOSDG 2003, Germany.
- [5] 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.
- [6] A. Colyer, A. Rashid, G. Blair, On the Separation of Concerns in Program Families. Technical Report COMP-001-2004, <http://www.comp.lancs.ac.uk/computing/aop/papers/COMP-001-2004.pdf>.
- [7] R. Douence, P. Fradet, and M. Sudholt, Composition, Reuse and Interaction Analysis of Stateful Aspects. AOSD 04, March 2004.
- [8] R. Filman and D. Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness", Workshop on Advanced Separation of Concerns, OOPSLA 2000, October 2000, Minneapolis.
- [9] M. Harrold and G. Rothermel. Performing Data Flow Testing on Classes. Proc. *ACM SIGSOFT Foundation of Software Engineering*, pp. 154-163, December 1994.
- [10] 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.
- [11] M. Mortensen and R. Alexander, Adequate Testing of Aspect-Oriented Programs, Department of Computer Science, Colorado State University, Fort Collins, CO, USA, Technical report CS 04-110, December 2004.
- [12] 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.
- [13] 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.
- [14] J. Offutt and J. Voas. Subsumption of Condition Coverage Techniques by Mutation Testing. Jan 1996, ISSE-TR-96-01, http://www.isse.gmu.edu/techrep/1996/96_01_offutt.pdf.
- [15] J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3--18, January 1992.
- [16] D. Perry and G. Kaiser, Adequate Testing and Object-Oriented Programming. *Journal of Object-oriented Programming*, 1990: p. 13-19.
- [17] 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.
- [18] M. Storzer, and J. Krinke, Inteferece Analysis for AspectJ. Workshop on Foundations of Aspect-Oriented Languages (FOAL 2003) as part of AOSD 2003, March 2003.
- [19] J. Zhao and M. Rinard, "Pipa: A Behavioral Interface Specification Language for AspectJ," Proceedings of Fundamental Approaches to Software Engineering, LNCS 2621, pp.150-165, Springer-Verlag, April 2003.
- [20] J. Zhao and M. Rinard, System Dependence Graph Construction for Aspect-Oriented Programs, MIT LCS Tech Report 891, March 2003.
- [21] 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.