A Joinpoint Coverage Measurement Tool for Evaluating the Effectiveness of Test Inputs for AspectJ Programs

Fadi Wedyan, Sudipto Ghosh
Department of Computer Science
Colorado State University
{*wedyan,ghosh*}@*cs.colostate.edu*

Abstract

Testing aspect-oriented programs is challenging in part because of the interactions between the aspects and the base classes with which the aspects are woven. Coverage metrics, such as joinpoint coverage, address faults resulting from the changes in base class control flow that may be introduced by the woven advices. Definitions of joinpoint coverage in the literature typically require counting the execution of aspects at each joinpoint. We present a tool for measuring joinpoint coverage from two perspectives: per advice, which measures the execution of the advice at each joinpoint it is woven into, and per class, which measures the execution of all the advices in each joinpoint in the class. This gives a more detailed measurement of joinpoint coverage and helps in identifying what more needs to be tested in both the base class and the aspect. The tool is based on AspectJ and Java bytecode, and thus, does not require any source code. We demonstrate the use of our tool to measure the joinpoint coverage of test inputs generated by Xie and Zhao's Aspectra framework.

Keywords: *AspectJ, aspect-oriented programming, Java bytecode, joinpoints, test coverage, test input generation*

1. Introduction

It is generally recognized that software systems contain both core concerns and crosscutting concerns, i.e., concerns that are spread throughout several modules of implementation. Aspect-Oriented Programming (AOP) [8] supports the modularization of crosscutting concerns by introducing a construct called *aspect*. An aspect contains three main components: (1) *pointcuts*, which intercept the execution of base code at a given joinpoint, (2) *introductions*, which add or alter features of the base code, and (3) *advices*, which provide implementations of the crosscutting concern.

The benefits of modularizing crosscutting concerns come with a cost. Aspect-oriented programs can fail because of faulty base classes, faulty aspects, or faulty interactions between the aspects and the base classes [1, 2, 14]. Of particular interest are the new types of faults resulting from the interaction of the aspects with the base classes. Aspects can introduce new methods and state variables to base classes, alter the base class state variables, and change the control flow of base class methods. Aspects are themselves affected by the base class methods.

Researchers have proposed new testing approaches that target aspect-oriented programs. New test input generation methods [23, 24] and new coverage criteria [3, 11, 15, 25] have been proposed specifically for aspect-oriented programs. However, little has been done to provide automated tools for measuring test coverage. Moreover, there is a need to measure the coverage resulting from the inputs generated by the proposed test input generation techniques.

In this paper, we present a tool that facilitates the measurement of joinpoint coverage achieved by a given test suite. Definitions of joinpoint coverage in the literature typically require counting the execution of aspects at each joinpoint. We present a tool for measuring joinpoint coverage from two perspectives: per advice, which measures the execution of the advice at each joinpoint it is woven into, and per class, which measures the execution of all the advices in each joinpoint in the class.

Xie and Zhao [23] state that "coverage criteria defined at the bytecode level are stronger than the same ones defined in the source code level". It is easy to see this for joinpoint coverage. A class with one adviced method can be covered with one execution of the method in the source code level. At the bytecode level, the same joinpoint coverage requires executing the advice in all calls of the adviced method.

Our tool works on the bytecode of the woven classes, aspects, and test driver. Thus, we do not need the source code of the tested programs or the test suite. This facilitates the testing of aspects developed by a third party and gives us the flexibility of evaluating different test generation approaches. We use the Bytecode Engineering Library

(BCEL) [5] to parse and instrument classes and derive joinpoint information. Our tool also provides a test runner to run the test suites and collect coverage measurements. The tool itself is written in Java and is based on AspectJ [20].

We use this tool to measure the coverage obtained by using Xie and Zhao's Aspectra framework [23]. In their paper, the goal was to achieve branch coverage in the aspects. We measure the joinpoint coverage achieved through their test cases on several applications.

The rest of the paper is organized as follows. Section 2 summarizes related work in the area of testing aspect-oriented programs. A description of the implementation of advice weaving in AspectJ is given in Section 3. Section 4 presents our joinpoint coverage measurement tool. We demonstrate the use of our tool in Section 5. Finally, conclusions and future work are discussed in Section 6.

2. Related Work

Alexander et al. [1] describe the key problems related to testing aspects: aspects depend on weaving, do not exist independently, and are often tightly coupled to the context to which they are woven. Faults in aspect-oriented systems include faulty advice code and faulty pointcuts [1, 15]. Proposed coverage criteria for aspects are based on dataflow coverage [25, 26], path coverage [10], and state-based coverage [24].

Work has begun in the area of test case generation (Xie and Zhao [23]; van Deursen et al. [21]), test selection (Zhou et al. [27]; Souter [19]), coverage and fault models (Alexander et al. [1, 2]; Lemos et al. [9]), and test frameworks (AJTE [1], aUnit [2]), and Jam1Unit [12].

Mortensen et al. [16] developed a two-step tool for measuring joinpoint coverage of AspectC++ programs. In the first step, they instrument the advice body before weaving so that the execution of the woven program generates a list of joinpoints that were covered. The second step reads a file created by the AspectC++ weaver that lists all joinpoints and compares it with the list of covered joinpoints in order to report any joinpoints that were not executed.

3. Background

We provide an overview of the functioning of the AspectJ weaver to help understand the functionality of our coverage tool. For a complete discussion, please refer to Haupt [6] and Hilsdale [7]. AspectJ [20] is an aspect-oriented extension of Java. The AspectJ compiler accepts both bytecode and source code of pure classes and aspects and produces pure Java bytecode [7]. When advices are woven, the weaver produces a method for each advice. The method parameters are the same as the advice parameters and might be extended with an object of type `thisJoinPoint`.

```
public aspect Aspect1 {
    void around (): call(void *.method1()) {
        System.out.println("Starting around");
        proceed ();
        System.out.println("Ending around");
    }
    before() : call(void *.method1()){
        System.out.println("starting Before");
    }
}
```

Figure 1. Sample AspectJ Aspect.

Consider the `before` advice shown in Figure 1. This advice is compiled into the method shown in Figure 2. Aspect methods always start with `ajc` followed by the advice kind and the aspect name. Each advice is given a reference name (the last word in the method name) and a serial number. The symbol `$` is used to concatenate the words that form the method name.

Around advices may use a special construct called `proceed` to continue with the control flow of the method in the matching joinpoint. In the aspect bytecode, two methods are constructed for the around advice, the first method corresponds to the proceed construct, which takes the advice arguments and an object of type `AroundClosure` that encapsulates the control flow of the matching method. The second method contains the around advice body and invokes the proceed method [7]. Figure 3 shows the signatures of the around methods of the around advice shown in Figure 1.

In the woven code of the classes, the AspectJ weaver inserts *static joinpoint shadows* in the locations of possible joinpoints [7]. Matching advices are invoked in the joinpoint shadows. If the joinpoint cannot be determined at compile time, the static joinpoint shadow is guarded by dynamic tests to make sure the joinpoint matches at runtime. These joinpoint shadows are called *residues* [7]. In the static joinpoint shadow and for advices of type `before` or `after`, the weaver creates an instance of the aspect using the static method `AspectOf()`. The actual advice invocation simply calls the method that represents the advice in the aspect bytecode. The code in Figure 4 shows the joinpoint shadow of the before advice in Figure 1.

The joinpoint shadow is inserted in the code that calls the advised method. Around advices are treated differently. The advices are directly inlined into the base class bytecode. At least two methods are added to the class bytecode for

[1]`http://www.ncfreak.com/asato/software/ajtest-en.html`
[2]`http://aunit.sourceforge.net/`

```
public void ajc$before$Aspect1$2$ba417cde();
  0  getstatic java.lang.System.out : java.io.PrintStream [40]
  3  ldc <String "starting Before"> [73]
  5  invokevirtual java.io.PrintStream.println(java.lang.String) : void [48]
  8  return
```

Figure 2. Bytecode of the `before` Advice in Figure 1.

```
public void ajc$around$Aspect1$1$ba417cde(
    org.aspectj.runtime.internal.AroundClosure ajc_aroundClosure);
//around advice body ...

static synthetic void ajc$around$Aspect1$1$ba417cdeproceed(
org.aspectj.runtime.internal.AroundClosure this) throws java.lang.Throwable;
```

Figure 3. Methods Signatures for the `around` Advice of Figure 1.

```
 0  invokestatic Aspect1.aspectOf() : Aspect1 [50]
 3  invokevirtual Aspect1.ajc$before$Aspect1$2$ba417cde() : void [53]
 6  aload_0
 7  invokevirtual classA.method1() : void [20]
```

Figure 4. Joinpoint Shadow of the `before` Advice in Figure 1.

every around advice. One wraps the actual method body (this corresponds to the `proceed()` method) and the other wraps the advice body and invokes the first method.

4. Joinpoint Coverage Measurement

Joinpoint coverage was suggested by Mortensen and Alexander [15] as *insertion coverage*. Lemos et al. [11] suggested a similar criterion called *all-crosscutting-node criterion*. Mortensen et al. [16] used the term joinpoint coverage to describe the same criterion. Regardless of the name or the phrasing of the definition, joinpoint coverage requires testing each advice by executing it at each matching joinpoint. A matching joinpoint, in the source code level, resides before, around, or after the advice. However, in the bytecode of a woven class, a static joinpoint shadow is added whenever a matching method is called. In other words, every call to an advised method will have a joinpoint. We follow the bytecode view of a join point. In other words, we require the execution of an advice in every joinpoint shadow in the woven classes.

In order to analyze the bytecode, we used the Apache BCEL library [5]. The BCEL library is used for the static analysis and dynamic creation or transformation of Java bytecode [4]. BCEL is already used by the AspectJ weaver to produce the woven classes.

Figure 5 shows the steps involved in using our tool and the components inside the tool. Our tool works in two

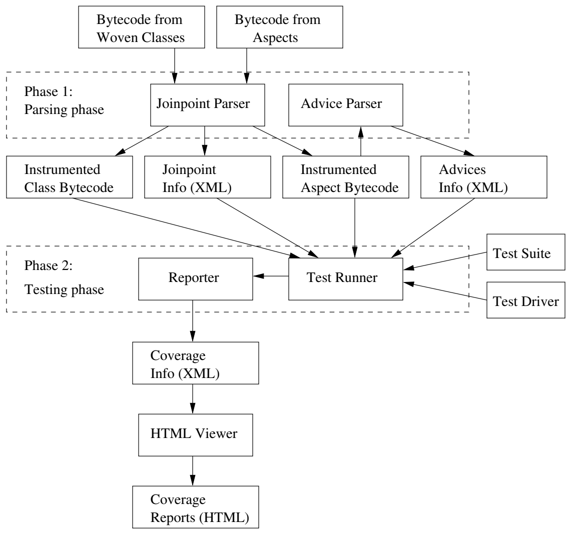
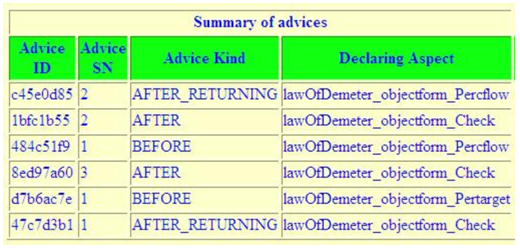


Figure 5. Joinpoint Coverage Tool.

phases. In phase one, the class bytecode is parsed to identify the joinpoints. When a joinpoint shadow is located, joinpoint information is saved in an XML file. Currently the gathered information includes the classname where the joinpoint resides, method name, line number, declaring aspect, advice serial number, and advice bytecode name. Saving the line number is necessary in order to distinguish between two or more joinpoints for the same advice in the same method. We also add an instruction that calls a method in a recording class that counts how many times a before or an after advice is executed. We do the same thing for the calls of methods that are created in the class containing the around advice body. Calling a counter from an outside class makes the instrumentation easier with fewer changes to the bytecode.

Calls to the proceed method are not instrumented since they do not invoke an advice. Residues are instrumented inside the guarding condition so that the joinpoint is verified as a real joinpoint only if the condition is satisfied at runtime. Note that aspect classes are also parsed since they might include joinpoints from other aspects.

We also parse the aspect bytecodes to identify the advices. Similar to joinpoints, advice information is saved in an XML file. The gathered information includes the declaring aspect name, advice kind, advice bytecode name, and advice serial number. Figure 6 shows the XML file contents for the advices found in the *LawOfDemeter* benchmark. Note that the dot (used for packaging) in the aspect name in the bytecode is replaced by an underscore.



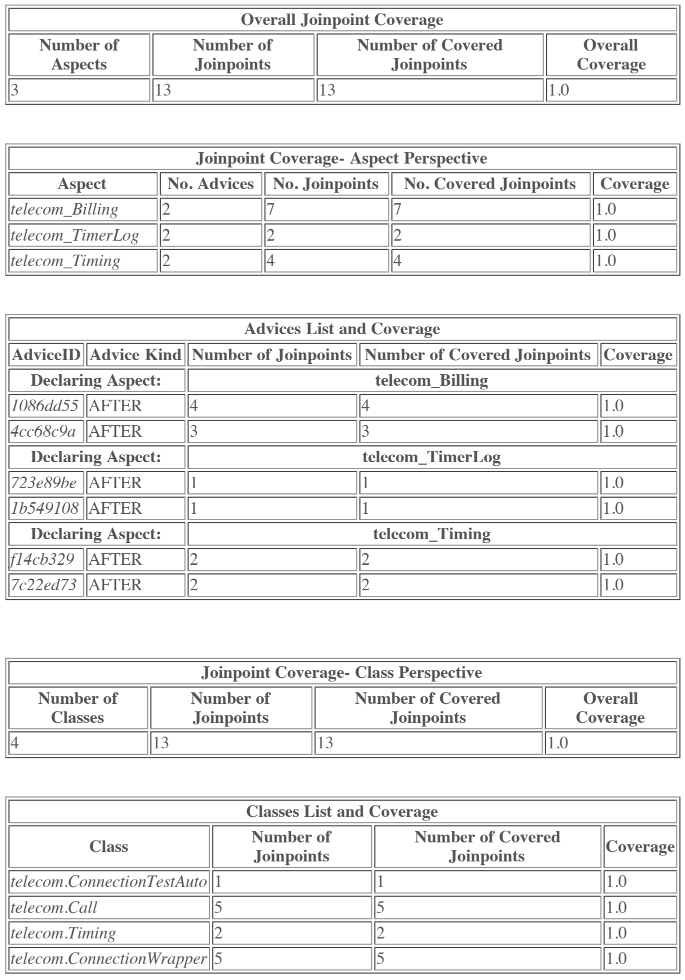
Summary of advices

Advice ID	Advice SN	Advice Kind	Declaring Aspect
c45e0d85	2	AFTER_RETURNING	lawOfDemeter_objectform_Percflow
1bfc1b55	2	AFTER	lawOfDemeter_objectform_Check
484c51f9	1	BEFORE	lawOfDemeter_objectform_Percflow
8ed97a60	3	AFTER	lawOfDemeter_objectform_Check
d7b6ac7e	1	BEFORE	lawOfDemeter_objectform_Pertarget
47c7d3b1	1	AFTER_RETURNING	lawOfDemeter_objectform_Check

Figure 6. Advice Information in the XML File.

In phase two, we use the test runner component to run the test driver of the test suite on the instrumented bytecode. Our tool is independent of the testing approach and no assumptions are made about the test suite. The runner component uses a reporter class that counts the covered joinpoints whenever an instrumented joinpoint shadow is executed. The reporter also computes the coverage obtained after the test inputs are run. Test coverage results are saved in an XML file. For each class, we save the list of joinpoints in the class, number of executed joinpoints, how many times each joinpoint is covered, and the overall joinpoint coverage. For advices, we report (in addition to the advice information mentioned earlier) the list of matching joinpoints, the coverage of each joinpoint, and the coverage of the advice. We also provide an *html viewer* to view the test results in a user friendly format.

In addition to providing joinpoint coverage data from two perspectives, we also display overall coverage results. Figure 7 shows a snapshot of the output produced by our tool with the overall joinpoint coverage and the aspect and class perspectives. We also compute the *aspect coverage* which measures how many of the matching join points of all advices in the aspect are executed. We also produce reports showing which joinpoints are not covered.



Overall Joinpoint Coverage

Number of Aspects	Number of Joinpoints	Number of Covered Joinpoints	Overall Coverage
3	13	13	1.0

Joinpoint Coverage- Aspect Perspective

Aspect	No. Advices	No. Joinpoints	No. Covered Joinpoints	Coverage
telecom_Billing	2	7	7	1.0
telecom_TimerLog	2	2	2	1.0
telecom_Timing	2	4	4	1.0

Advices List and Coverage

AdviceID	Advice Kind	Number of Joinpoints	Number of Covered Joinpoints	Coverage
Declaring Aspect:		**telecom_Billing**		
1086dd55	AFTER	4	4	1.0
4cc68c9a	AFTER	3	3	1.0
Declaring Aspect:		**telecom_TimerLog**		
723e89be	AFTER	1	1	1.0
1b549108	AFTER	1	1	1.0
Declaring Aspect:		**telecom_Timing**		
f14cb329	AFTER	2	2	1.0
7c22ed73	AFTER	2	2	1.0

Joinpoint Coverage- Class Perspective

Number of Classes	Number of Joinpoints	Number of Covered Joinpoints	Overall Coverage
4	13	13	1.0

Classes List and Coverage

Class	Number of Joinpoints	Number of Covered Joinpoints	Coverage
telecom.ConnectionTestAuto	1	1	1.0
telecom.Call	5	5	1.0
telecom.Timing	2	2	1.0
telecom.ConnectionWrapper	5	5	1.0

Figure 7. Joinpoint Coverage Screenshot.

5. Demonstration

Using our tool, we measured the joinpoint coverage achieved by the test inputs generated by Xie and Zhao's Aspectra framework [23][3]. We also showed that our coverage tool was able to recognize all the joinpoints from the bytecode.

[3]We would like to thank the authors of [23] for providing us with the test inputs.

5.1. Aspectra Framework

Xie and Zhao [23] developed a framework (called *Aspectra*) that automates the generation of test inputs for AspectJ programs with a focus on behaviors implemented in advices and intertype methods. Given an aspect to be tested, developers are required to construct base classes to produce woven classes. Aspectra synthesizes a wrapper class for each woven class. The purpose of the wrapping is to allow test generation tools to exercise advices related to *call* joinpoints and public non-advice aspect methods. The wrapper class has a wrapper method for each base-class public method, each method introduced to the base-class by the aspect, and each public non-advice method of the aspect. The wrapper class is compiled and the wrapper class, base-class and aspect are woven.

Given the wrapper classes, Aspectra uses two tools to generate test inputs for AspectJ programs. *Parasoft Jtest*[17] is a commercial object-oriented testing tool used to generate arguments for public methods. The second, Rostra [22], developed for testing OO programs, is used to put an object of the class under test into a particular state required by the test.

Aspectra is a *unit* testing approach for the aspects. Since aspects can not exist on their own, Aspectra provides automated wrapper classes to serve as scaffolding where advices are woven and, thus, can be tested. Aspectra does not aim at testing software that wants to take advantage of existing aspects but rather testing that the aspects function properly.

Test inputs provided by Aspectra will always have a 100% joinpoint coverage in the woven wrapper classes and 0% in other classes (which are actually not tested). To evaluate Aspectra, a unit testing approach for aspects, we want to make sure that all advices in the aspects are exercised. Thus, we measure only the joinpoints per advices. This illustrates the importance of providing two perspectives of measuring the joinpoint coverage.

5.2. Benchmarks

We tested one benchmark collected by the Sable Research Group in McGill University [18]. We also included the telecom and Bean example shipped with the AspectJ environment. Table 1 summarizes the main characteristics of the tested benchmarks.

Table 1. Benchmark Characteristics.

Benchmark	# Join Points	# Advices	# Aspects
ProdLine	1364	15	6
telecom	12	6	3
Bean	4	2	1

The *ProdLine* benchmark was developed by Lopez-Herrejon and Batory [13] as an illustrative example of how to use AOP in developing product-line architectures.

The *telecom* benchmark is a simple simulation of a telephony system in which customers make, accept, merge, and hangup both local and long distance calls. Telecom has two main aspects, *Timing* and *Billing*, and one logger aspect (*TimerLog*). The *TimerLog* matches joinpoints in the Timing aspect. Finally, the *Bean* benchmark adds Java bean functionality to a given class.

5.3. Results

The aspects in *ProdLine* declare intertype methods inside empty classes. That was challenging to our tool since all the joinpoints reside inside intertype methods. Calls to intertype methods inside the bytecode appear to be just like calls to the advice in a joinpoint shadow because of similar naming conventions. We addressed this issue by first identifying all the methods of the class with the help of the BCEL API. Thus, the tool could figure out which calls were to intertype methods and which ones were to advices. Our tool found all the joinpoints inside the intertype methods. Aspectra provides test cases for the class *vertex* and achieved 98.34% coverage (1344 out of 1364 joinpoints) for the advices that advise methods added to the class. The missing joinpoints were in the `main` method, which was not called from the tests. We do not have test inputs for the other classes. The *ProdLine* benchmark is challenging to any test generation approach since it requires generating the tests based on the woven classes only.

We ran the test inputs generated by Aspectra on the *telecom* benchmark. Aspectra provides a wrapper class for the *connection* class of the benchmark. Aspectra's test inputs covered all the advices. We also generated test inputs for the *call* class which uses the *connection* class to make phone calls. Our tool was able to find all the joinpoints in the classes as well as the covered joinpoints.

The *BoundPoint* aspect in the *Bean* benchmark declares two *around* advices. Aspectra provides test inputs for the *point* class. Aspectra covered the two advices and our tool successfully detected the joinpoints of the around advices in the class.

6. Conclusions and Future Work

We presented a tool for measuring the joinpoint coverage achieved by test inputs for AspectJ programs. Our tool analyzes the bytecode of the woven classes and aspects to extract joinpoint information. The tool also instruments the joinpoint shadows in the bytecode to keep track of which advices are executed when the tests are run.

We provide coverage measurements from two perspective: class perspective, by measuring the coverage in the woven classes, and aspects perspective, by measuring which advices were covered. This allows our tool to evaluate test inputs that target either the base code or the aspects.

We tested the tool with 3 benchmarks that have various types of joinpoints. The tool was able to locate all the joinpoints and keep track of the ones covered when the tests are run. We evaluated Aspectra and found that Aspectra did cover all the advices in the aspects.

We plan to extend our tool to measure other aspect-related coverage criteria, such as dataflow coverage between the elements in the base-code and aspect. We also intend to evaluate other test input generation techniques, both with respect to test coverage and fault detection ability.

References

[1] R. T. Alexander, J. M. Bieman, and A. 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] J. S. Bækken and R. T. Alexander. A candidate fault model for AspectJ pointcuts. In *17th International Symposium on Software Reliability Engineering (ISSRE 2006), 7-10 November 2006, Raleigh, NC*, pages 169–178, 2006.

[3] M. Ceccato and P. T. F. Ricca. Is AOP code easier or harder to test than OOP code? In *WTAOP: Proceedings of the 1st Workshop on Testing Aspect-Oriented Programs, held in conjunction with the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, 2005.

[4] M. Dahm. Byte Code Engineering with the JavaClass API. Technical Report B-17-98, Institut fur Informatik, Freie Universitat, Berlin, January 1999.

[5] M. Dahm and J. van Zyl. Byte code engineering library. http://jakarta.apache.org/bcel. June 2006.

[6] M. Haupt. *Virtual Machine Support for Aspect-Oriented Programming Languages*. PhD thesis, der Technischen Universitat Darmstadt, 12 2005.

[7] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, pages 26–35, 2004.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingier, and J. Irwin. Aspect Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Springer Verlag LNCS 1241*, Finland, June 1997.

[9] O. A. L. Lemos, F. C. Ferrari, P. C. Masiero, and C. V. 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.

[10] O. A. L. Lemos, J. C. Maldonado, and P. C. Masiero. Structural unit testing of aspectj programs. In *2005 Workshop on Testing Aspect-Oriented Programs (held in conju nction with AOSD 2005)*, March 2005.

[11] O. A. L. Lemos, A. M. R. Vincenzi, J. C. Maldonado, and P. C. Masiero. Control and data flow structural testing criteria for aspect-oriented programs. *Journal of Systems and Software*, 80(6):862–882, 2007.

[12] C. V. Lopes and T. Ngo. Unit testing aspectual behavior. In *WTAOP '05: Proceedings of the 1st workshop on Testing aspect-oriented programs*, 2005.

[13] R. Lopez-Herrejon and D. Batory. Using aspectj to implement product-lines: A case study. Technical report, University of Texas at Austin, 2002.

[14] N. McEachen and R. T. Alexander. Distributing classes with woven concerns: an exploration of potential fault scenarios. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development, AOSD 2005, Chicago, Illinois, USA,*, pages 192–200, 2005.

[15] M. Mortensen and R. 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.

[16] M. Mortensen, S. Ghosh, and J. M. Bieman. A test driven approach for aspectualizing legacy software using mock systems. *Information and Software Technology*, (to appear, available at `http://www.cs.colostate.edu/~ghosh/papers/ist2007.pdf`).

[17] ParaSoft Inc. `http://www.parasoft.com/`.

[18] Sable Research Group. AspectJ benchmarks. http://www.sable.mcgill.ca/benchmarks/. October 2004.

[19] A. L. Souter, D. Shepherd, and L. L. Pollock. Testing with respect to concerns. In *ICSM*, pages 54–. IEEE Computer Society, 2003.

[20] The AspectJ Team. Aspectj compiler 1.5.4. http://www.eclipse.org/aspectj/. December 2007.

[21] A. van Deursen, M. Marin, and L. Moonen. A systematic aspect-oriented refactoring and testing strategy, and its application to JHotDraw. Technical Report SEN-R0507, CWI Technical Report, 2005.

[22] T. Xie, D. Marinov, and D. Notkin. Rostra: a framework for detecting redundant object-oriented unit tests. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, pages 196–205, Sept 2004.

[23] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of AspectJ programs. In R. E. Filman, editor, *AOSD*, pages 190–201. ACM, 2006.

[24] W. Xu and D. Xu. State-based testing of integration aspects. In *WTAOP '06: Proceedings of the 2nd workshop on Testing aspect-oriented programs*, pages 7–14, New York, NY, USA, 2006. ACM Press.

[25] J. Zhao. Data-flow-based unit testing of aspect-oriented programs. In *Proceedings of the 27th International Computer Software and Applications Conference*, pages 188–197, Dallas, TX, USA, November 2003.

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

[27] Y. Zhou, H. Ziv, and D. J. Richardson. Towards a practical approach to test aspect-oriented software. In S. Beydeda, V. Gruhn, J. Mayer, R. Reussner, and F. Schweiggert, editors, *SOQUA/TECOS*, volume 58 of *LNI*, pages 1–16. GI, 2004.