# Perspectives on Automated Testing of Aspect-Oriented Programs

Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
xie@csc.ncsu.edu

Jianjun Zhao
Department of Computer Science
Shanghai Jiaotong University
Shanghai 200240, P.R. China
zhao-jj@cs.sjtu.edu.cn

## ABSTRACT

Aspect-oriented software development is gaining popularity with the adoption of aspect-oriented languages in writing programs. To reduce the manual effort in assuring the quality of aspect-oriented programs, we have developed a set of techniques and tools for automated testing of aspect-oriented programs. This position paper presents our perspectives on automated testing techniques from three dimensions: testing aspectual behavior or aspectual composition, unit tests or integration tests, and test-input generation or test oracles. We illustrate automated testing techniques primarily through the last dimension in the perspectives. By classifying these automated testing techniques in the perspectives, we provide better understanding of these techniques and identify future directions for automated testing of aspect-oriented programs. This position paper also presents a couple of new techniques that we propose based on the perspectives.

**Categories and Subject Descriptors:** D.2.5 [Testing and Debugging]: Testing tools

**General Terms:** Reliability.

**Keywords:** Software testing, Aspect-oriented software development, AspectJ

## 1. INTRODUCTION

In aspect-oriented software development (AOSD) [11, 22], separation of concerns in software development has been improved because techniques in AOSD facilitate modularizing crosscutting concerns of a software system and thus make it easier to maintain and evolve. Previous research in AOSD has focused primarily on the activities of software system design, problem analysis, and language implementation. Although testing is known to be a labor-intensive process that accounts for half the total cost of software development [6], research on testing in AOSD, especially automated testing, has not been sufficiently conducted. Although an aspect-oriented design or implementation can lead to a better system architecture or a disciplined coding style, it does not protect against mistakes made by developers during software development. To assure high software quality in AOSD, we need to develop automated testing techniques and tools for AOSD in order to reduce human effort in the software testing process.

Aspect-oriented programming languages, such as AspectJ [11], introduce some new language constructs (such as join points, advice, intertype declarations, and aspects) to the common object-oriented programming languages, such as Java. The behavior of an aspect in AspectJ programs can be categorized into two types [14]: *aspectual behavior* is behavior implemented in pieces of advice and *aspectual composition* is behavior implemented in pointcuts for composition between base and aspectual behavior.

When we treat an aspect as a unit, *unit tests* for an aspect are those tests that are created to test in isolation pieces of advice defined in the aspect. However, it is often difficult to manually or automatically construct the aspect's execution context in unit tests. When we treat an aspect as a unit and its affected classes also as units, *integration tests* for the aspect are those tests that are created to test the affected classes woven with the aspect. These integration tests can consist of the invocations of those methods affected by the aspect. These invocations eventually exercise the interaction between the aspect and the affected classes by invoking pieces of advice from the advice-call sites inserted within the affected classes.

In automated software testing, there are two major activities: *test-input generation* and *test oracles* [5]. Test-input generation generates test inputs for the program under test. After these test inputs are generated and executed, we need to have a way (called test oracles) to determine whether these test executions are correct. When there are no specifications for the program, relying on uncaught exceptions is limited in exposing faults other than robustness faults [7]. But it is also infeasible for developers to inspect the executions of a large number of generated test inputs. One feasible and practical way is test selection: using a tool to select a subset of test executions for inspection. Another way is runtime behavior checking against specifications: allowing developers to write specifications for the program and the specifications can be used to automatically check the executions of the generated test inputs.

In our past research, we have developed a set of techniques and tools for automated testing of aspect-oriented programs. In this position paper, we present our perspectives on classifying automated testing techniques from three preceding dimensions:

- testing aspectual behavior or aspectual composition
- unit tests or integration tests
- test-input generation or test oracles

Through the classification, we can gain better understanding of the existing automated testing techniques and identify future research directions by filling the identified gap in the research agenda. We illustrate the techniques primarily for aspect-oriented programs written in AspectJ [11], a popular aspect-oriented programming language. In our perspectives, we summarize the techniques and tools that we have developed in our priori work. In particular, we present the techniques of test-input generation based on a wrapper mechanism, which enables us to leverage the existing Java test-generation tools to generate test inputs for AspectJ programs [26], as well as test selection techniques for behavior checking based on branch and interaction coverage [26], dataflow coverage [30], data coverage [28], and mutation testing [3]. We also present behavior checking techniques based on specifications for aspect-oriented programs [31]. In addition, we propose a couple of new techniques in our perspectives: generating unit tests by leveraging existing AspectJ testing frameworks [14, 29] and automatic mock-object creation techniques [21], and generating mutants for advice body by leveraging muJava [15], a Java mutation testing tool.

## 2. OVERVIEW

We present our perspectives on automated testing of aspect-oriented programs in order to classify automated testing techniques. Table 1 shows the classification of major automated testing techniques. This section gives an overview of these techniques and their classifications, and the rest of this position paper gives the details of these techniques.

We classify techniques based on three dimensions:

- testing aspectual behavior or aspectual composition [14]. *Aspectual behavior* describes behavior implemented in pieces of advice. *Aspectual composition* describes behavior implemented in pointcuts for composition between base and aspectual behavior.
- unit tests or integration tests. *Unit tests* for an aspect are those tests that are created to test in isolation pieces of advice defined in the aspect. *Integration tests* for the aspect are those tests that are created to test the classes being woven with the aspect.
- test-input generation or test oracles. *Test-input generation* generates test inputs for either aspects directly (as unit tests) or woven classes (as integration tests). *Test oracles* [5] provide ways for checking the correctness of the executions of generated test inputs. Some techniques can be semi-automated, such as selecting a subset of test inputs for inspection, still requiring human inspection for correctness. Some techniques can be totally automated, such as runtime behavior checking against specifications when they are available.

JamlUnit [14] and AJTE [29] are two AspectJ unit testing frameworks for facilitating the creation of unit tests for aspects. These two frameworks themselves do not provide the feature of automated test-input generation, similar to the JUnit testing framework [10] for testing Java classes. Based on these two frameworks, we propose automated test-input generation techniques by leveraging existing Java test-generation tools, improving the techniques of directly feeding compiled aspect bytecode to the existing tools [27]. Both frameworks primarily focus on testing aspectual behavior but AJTE also allows developers to write assertions that check pointcut matching. Our APTE [2] approach automates the generation of such assertions. General speaking, this feature of checking pointcut matching can be seen as a (limited) way of testing aspectual composition.

Our Aspectra approach [26] allows developers to supply base classes for the aspect under test and then creates a wrapper for each woven class. Then the existing Java test-generation tools can be leveraged to generate integration tests for the woven class by treating the wrapper class as the class under test. Although Aspectra generates integration tests, its major objective is not to test aspectual composition but to test aspectual behavior, which is exercised eventually by the generated integration tests.

Various types of coverage information [32] can be used to select tests for testing aspectual behavior so that developers can focus their inspection efforts on these selected tests. For example, our test selection techniques use branch coverage [26] or dataflow coverage [30] inside the aspect under test. Our test selection techniques also use data coverage [28], which is concerned with the input values to pieces of advice defined in the aspect. We also propose a new wrapper technique for leveraging existing Java mutation testing tool [15] in mutation testing of aspectual behavior. Pipa [31] is a behavioral interface specification language for AspectJ. It is an extension to the Java Modeling Language [13], a behavioral interface specification language for Java. After developers write Pipa specifications for an aspect, we can automatically check test executions against the specifications. All these preceding techniques can be applied for either unit tests or integration tests.

Two existing types of coverage information can be used to select tests for testing aspectual composition. Our test selection techniques use interaction coverage [26], which characterizes the coverage of the call sites calling between methods in base classes and advice in an aspect. Our test selection techniques also use dataflow coverage [30] across aspects and base classes. In addition, our mutation testing techniques for pointcuts [3] can also be used to select tests for testing aspectual composition.

Based on Table 1, we can observe that we are short of test-input generation techniques for testing aspectual composition (shown by the empty entry in the third row, last column of the table). This observation leads us to propose to extend our Aspectra approach to fill this gap in future work. We can also observe that there exists no specification or property language for specifying properties of aspectual composition. This observation leads us to propose to develop languages to express properties of aspectual composition to fill this gap in future work. Note that for unit tests there is generally no automated technique for checking aspectual composition (shown by the empty entry in the last row, third column of the table), because unit tests are generally not created for testing aspectual composition.

In the rest of the position paper, we present the techniques listed in Table 1 in more details.

## 3. TEST-INPUT GENERATION

This section presents challenges and proposed techniques for generating unit tests and integration tests, respectively. In particular, unit-test generation needs to address the issue of constructing execution contexts for aspects and we propose to exploit two existing AspectJ unit testing frameworks and existing Java test-generation tools. Integration-test generation needs to address the issue of aspect weaving for test generation: visibility of woven methods needs to be provided to test-generation tools and unwanted weaving needs to be avoided. We propose a wrapper mechanism to address the issue by leveraging existing Java test-generation tools.

### 3.1 Unit-Test Generation

An aspect can be treated as a class and a piece of advice defined in the aspect can be treated as a method in the class. Then unit tests can be generated for the aspect and its advice. In fact, an AspectJ compiler such as ajc [1, 9] compiles aspect source into aspect-class bytecode and a piece of advice into a method in the class bytecode.

Our techniques [27] generate unit tests for aspects by feeding the complied aspect classes to existing Java test-generation tools such as Parasoft Jtest [19], JCrasher [7], Rostra [24], and Symstra [25], which can generate tests based on Java bytecode. However, these

| | | Unit tests | Integration tests |
|---|---|---|---|
| Test-input generation | Aspectual behavior | JamlUnit [14], AJTE [29] | Aspectra [26] |
| | Aspectual composition | AJTE [29], APTE [2] | |
| Test oracles | Aspectual | branch cov [26], dataflow cov [30] data cov [28], Pipa [31], muJava [15] | branch cov [26], dataflow cov [30] data cov [28], Pipa [31], muJava [15] |
| | Aspectual composition | | interaction cov [26] dataflow cov [30] pointcut mutation testing [3] |

**Table 1: Classification of automated testing techniques for aspect-oriented programs**

Java test generation tools based on bytecode are not able to generate meaningful tests for advice methods whose argument types include `JoinPoint` or `AroundClosure`. These two classes belong to AspectJ execution contexts and these Java test generation tools cannot create appropriate objects for these two classes.

To address the issues of constructing meaningful execution contexts, we propose to incorporate JamlUnit [14] and automatic mock-object creation [21]. JamlUnit uses mock objects [17] to emulate execution contexts such as join points. In particular, JamlUnit provides some helper classes, which allow developers to carefully construct environments as if certain actual join points in base classes are reached. Although JamlUnit has not provided features of automatic mock object creation, some existing automatic mock-object creation techniques [21] can be used in combination with JamlUnit. Note that these existing techniques capture the execution of existing tests and replay the executions to create mock objects; therefore, applying these techniques requires some initial integration tests to be generated for woven classes (described in the next section) and the integration-test executions are captured and replayed for creating mock objects in unit tests.

Another issue of generated unit tests is that the tests invoke methods (complied from advice) whose names are not human-readable. This issue makes it difficult for developers to inspect test executions for correctness (described in Section 4). The AJTE unit testing framework [29] creates a wrapper class for each aspect. For each piece of advice in the aspect, AJTE creates a method in the wrapper class; the method name is generated from keywords in the advice declaration, being more readable than the advice's method name compiled by ajc. Although this feature does not improve the test generation directly, the generated test inputs are more human-readable. AJTE also creates a method in the wrapper class for each pointcut. This method checks whether a given joinpoint matches the pointcut expression. Our APTE [2] approach automatically generates joinpoints collected from the base classes as arguments for this method in generated unit tests.

## 3.2 Integration-Test Generation

To get around the difficulties of constructing execution contexts for aspects, we can generate integration tests for base classes woven with the aspects and these integration tests eventually exercise aspectual behavior. In particular, given aspects, developers can construct appropriate base classes for the aspects and then use an AspectJ compiler such as ajc [1, 9] to weave aspects into the constructed base classes to produce woven classes in the form of bytecode. Then these woven classes can be fed to automatic Java test-generation tools such as Parasoft Jtest [19], JCrasher [7], Rostra [24], and Symstra [25], which can generate test inputs based on Java bytecode. However, three issues need to be addressed when these existing test-generation tools are leveraged to generate test inputs for AspectJ programs:

- When a piece of advice is related to `call` join points, the existing test-generation tools cannot execute the advice during its test-generation process, because the advice is to be woven in call sites, which are not available before test generation.
- Although we can use ajc [1, 9] to weave the generated tests with the aspect classes in order to execute advice related to `call` join points, the compilation can fail when the interfaces of woven classes contain intertype methods and the generated tests invoke these intertype methods. In addition, weaving the generated test classes with the aspect classes could introduce unwanted advice into the test classes.
- Public non-advice methods in aspect classes cannot be exercised by the test inputs generated for woven classes, because woven classes do not have any call sites of these public non-advice methods.

To address these issues, we have developed the Aspectra approach [26] to automatically synthesize a wrapper class for each constructed base class for aspects and then the wrapper class is fed to existing test-input generation tools for generating integration tests. In particular, there are six steps for generating test inputs based on the wrapper mechanism:

1. Compile and weave the base class and aspects into class bytecode using an AspectJ compiler such as ajc [1, 9].
2. Synthesize a wrapper class for the base class based on the woven class bytecode. Aspectra synthesizes a wrapper method for each public method in the base class. This wrapper method invokes the public method in the base class. In this wrapper class, Aspectra also synthesizes a wrapper method for each public intertype method woven into the base class. This wrapper method uses Java reflection [4] to invoke the intertype method; otherwise, the compilation in the third step can fail because intertype methods are not recognized by ajc before compilation. In a similar way, Aspectra also synthesizes a wrapper method for a public non-advice method in aspect classes. The wrapper method ensures that an intertype method or public non-advice method of aspect classes is tested by existing test-generation tools.
3. Compile and weave the base class, wrapper class, and aspects into class bytecode using ajc. This step ensures that the advice related to `call` join points is executed during the test-generation process, because the invocations to the advice are woven into the call sites (within the wrapper class) of public methods in the base class.
4. Clean up unwanted woven code in the woven wrapper class. Aspectra scans the bytecode of the woven wrapper class and

removes the woven code that are for advice related to `execution` join points. Note that Aspectra needs to keep the woven code that are for advice related to `call` join points.

5. Generate test inputs for the woven wrapper class using existing test-generation tools based on class bytecode, such as Parasoft Jtest [19], JCrasher [7], Rostra [24], and Symstra [25]. These tools export generated tests to test code, usually as a JUnit test class [10].
6. Compile the generated test class into class bytecode using a Java compiler [4]. Note that we do not use ajc to weave the exported test class with the wrapper class, base class, or aspects, because the weaving process can introduce unwanted woven code in the test class.

## 4. TEST SELECTION FOR BEHAVIOR INSPECTION

After a large number of unit or integration tests are automatically generated and executed, the correctness of these test executions is still unknown when there are no specifications for checking their correctness. But it is infeasible for developers to inspect such a large number of test executions. This section presents test selection techniques that select a subset of generated tests for inspection. These techniques are based on coverage information [32] such as branch, interaction, dataflow, and data coverage or based on mutation testing [3, 15]. A generated test input is selected if it achieves new coverage that is not achieved by already selected test inputs or it kills new mutants that are not killed by already selected test inputs.

### 4.1 Branch Coverage

Branch coverage [6] measures whether boolean expressions in control structures are evaluated to both true and false. Examples of control structures include if-statement, while-statement, switch-statement cases, and exception handlers. To determine whether a test exercises new aspectual behavior, we measure whether the test covers a previously uncovered branch in aspects of an AspectJ program. Branch coverage can be measured at the source code level or bytecode level. We choose to measure aspectual branch coverage at the bytecode level because the same piece of source code in aspects (e.g., source code in `around` advice) can be woven into multiple places in woven bytecode and covering these several places are often necessary for assuring high quality of the woven code.

When measuring aspectual branch coverage at the bytecode level, we have faced a couple of complications and developed techniques to address them in the tool implementation. First, we need to identify bytecode that is compiled from aspect source code. We scan the woven bytecode based on some characteristics of the woven bytecode produced by an AspectJ compiler. In our tool implementation, we identify a class (in the bytecode form) produced by ajc [1, 9] to be an *aspect class* if it has a method whose name starts with "`ajc$`". The methods in an aspect class are *aspect methods*. Because some new methods of a base class are also created by ajc for advice such as `around` advice, we identify a method in a non-aspect class also to be an aspect method if its name ends with "`$advice`". Then the branches within an aspect method are instrumented automatically and their coverage is measured at runtime. Note that in this research context, a method entry is considered as one branch; therefore, measuring method coverage is part of measuring branch coverage. By doing so, we can produce meaningful measurement results when there is no branch in aspect methods.

The second complication is that some branches woven in the bytecode of an aspect method can be infeasible to cover. We have inspected uncovered branches measured by our tool and determine whether some of these uncovered branches are inherently infeasible to cover. Then we improve our tool to exclude these infeasible branches for measurement.

### 4.2 Interaction Coverage

Branch coverage characterizes aspectual behavior but does not characterize aspectual composition, which is concerned with interaction between base classes and aspects. The interactions between base classes and aspects are primarily through the call sites between methods in base classes and advice in aspects; we call these call sites as *aspectual interactions*. To determine whether a test exercises new aspectual composition, we measure whether the test covers a previously uncovered aspectual interaction in an AspectJ program [26].

In particular, we classify four types of methods in AspectJ programs: advised methods, advice, intertype methods, and public non-advice methods. We call advice, intertype methods, and public non-advice methods as *aspect methods*. We particularly focus on aspectual interactions, which are between advised methods and aspect methods. An interaction from method $m_1$ to method $m_2$ is characterized by a call site in $m_1$'s body and this call site invokes $m_2$. We categorize aspectual interactions into the following two types:

- from advised methods to aspect methods (in short as *advised-aspect interaction*)
- from aspect methods to advised methods (in short as *aspect-advised interaction*)

Note that normally a piece of advice is supposed to be woven into at least one base class; thus, it is supposed to have at least one advised-aspect interaction. But in practice, a piece of advice may not have any advised-aspect interaction because of program errors or other practical reasons. As an extreme case, we may not have any advised-aspect interaction for an aspect because no aspects can be woven into provided base classes. Then we may measure the interaction coverage to be 100% although there is no interaction to cover at all. To address this issue and reflect the absence of composition for advice, we add into interaction coverage a special coverage: coverage of advice. With the augmented interaction coverage, we can measure 0% interaction coverage for the preceding extreme case. This situation is similar to where we consider a method entry as one branch in measuring branch coverage, as is described in the previous section.

### 4.3 Dataflow Coverage

Data flow testing [8, 12, 20] mainly focuses on testing the value assignment of each variable in a program by executing sub-paths from the assignment (*definition*) to some program points in which the variable is used (*use*). A *def-use pair* of a variable $v$ is an order pair $(d, u)$ where $d$ is a statement having a definition of $v$ and $u$ is a statement having a use of $v$, or some memory location bound to $v$, that can be reached by $d$ over some path in the program. A test covers a def-use pair if the execution of the test leads to traversal of a subpath from the definition to the use without any intervening redefinition of that memory location [8]. Dataflow coverage is used to select tests that cover new def-use pairs that have not been covered by previously selected tests.

Our test selection techniques support dataflow testing of aspects at three levels [30]. In each aspect, a *module* can be a piece of advice, an intertype method, or a normal method. We perform three levels of testing for an aspect: *intra-module*, *inter-module*, and *intra-aspect* testing. For an individual module such as a piece of advice, an intertype method, and a normal method, we perform intra-module testing. For a public module along with other modules that it calls in an aspect, we perform inter-module testing. For modules that can be accessed outside the aspect and can be invoked in any order by users of the aspect, we perform intra-aspect testing.

We use dataflow coverage to select particular def-use pairs within an aspect to cover. Corresponding to the three levels of testing for an aspect, we compute three types of def-use pairs: intra-module, inter-module, and intra-aspect def-use pairs for the aspect under test. We can define these three types of def-use pairs regarding an aspect by adapting the idea proposed by Harrold and Rothermel [8], which originally defined def-use pairs for testing of classes.

### 4.4 Data Coverage

Sometimes a program fault in an aspect can be value-sensitive: two test inputs may exercise the same path within the faulty aspect and one test input can expose the fault in the aspect but the other test input cannot. Recently *bounded-exhaustive testing* [16] has been proposed to test a program, especially one that has structurally complex inputs. Bounded exhaustive testing tests a program on all valid inputs up to a given bound. Basically bounded-exhaustive testing can be considered as a technique that adopts *data coverage* [18] and we can apply data coverage in selecting tests for testing aspectual behavior [28]: if a test covers a previously uncovered input-data value of advice or intertype method in aspects, then this test is selected.

There are two major problems to be addressed when we measure data coverage for AspectJ programs. One is to identify which methods belong to aspects and the other is to characterize input-data values for these methods because input-data values can involve the states of receiver objects and arguments. To address the first problem, similar to the techniques presented in Section 4.1, we identify two types of aspect methods in AspectJ programs: advice and intertype methods in aspect classes. To address the second problem, we represent each aspect-method execution with the actual method that was executed and a representation of the state (reachable from the receiver object and method arguments) at the beginning of the execution. We call such a state *method-entry state*.

### 4.5 Mutation Testing

To help select tests with respect to aspectual behavior, we perform mutation testing of the source code in aspects. We propose to leverage the existing Java mutation testing tool such as muJava [15]. Because muJava cannot directly analyze aspect code, we first refactor the body of each advice method or intertype method into a static method defined in another newly created normal Java class, and replace the advice or intertype method body with a call to the static method. Then we apply muJava on the newly created normal Java class, which includes the bodies of the statements originally defined in aspects. The resulting mutants together with the refactored aspect code can be used to perform mutation testing of aspect code.

To help select tests with respect to aspectual composition, we perform pointcut mutation testing [3], which requires generation of effective mutants, i.e., variations of the pointcut expression that resemble closely the original pointcut expression. We have developed a tool that serves the following purposes: generating relevant mutants and detecting equivalent mutants. Relevant mutants are those that are relevant to the original pointcut and resemble closely the original pointcut without being arbitrary strings. Equivalent mutants are those that are pointcut mutants that match the same set of join points as the original pointcut. In particular, our tool identifies join points that are matched by a pointcut expression, generates mutants of this pointcut expression, and identifies join points that are matched by these mutants. The mutants' matched join points are then compared with those of the original pointcut and these mutants are classified as different types of mutants for selection. When more than one mutant has the same set of join points, we select the mutant with the longest expression (in string length) for that particular set of join points. The classified mutants are ranked using a string similarity measure to help the developer choose a mutant that resembles closely the original pointcut. The developer could use designed test data (for the woven classes produced by aspect weaving) along with these mutants to perform mutation testing of pointcuts.

## 5. RUNTIME BEHAVIOR CHECKING WITH SPECIFICATIONS

In previous sections, we discussed ways for automated generation of unit and integration tests and selection of generated tests based on coverage information. However, manual effort is still needed to inspect the executions of the selected tests for correctness. In this section, we discuss how to check runtime program behavior (in particular, aspectual behavior) by further developing a behavioral interface specification language called Pipa for AspectJ. Developers can write Pipa specifications as test oracles for the aspects under test and use a runtime assertion checker to check the correctness of test executions.

Pipa [31] is a formal behavioral interface specification language (BISL) tailored to AspectJ. Pipa is a simple and practical extension to the Java Modeling Language (JML), a BISL for Java. Pipa uses the same basic approach as JML to specify AspectJ classes and interfaces, and extends JML, with just a few new constructs, to specify aspects. Pipa specifies both the syntactic interface and the behavior of aspects. The syntactic interface of an aspect consists of the signatures of its advice, intertype methods, normal methods, and the names and types of its fields. The behavior is specified in assertions, given as pre- and postconditions and aspect invariants. Pipa allows assertions (pre- and postconditions and aspect invariants) to be specified for aspects. The predicates in Pipa are written using regular AspectJ expressions extended with logical operators and universal and existential quantifiers. Pipa specifications are expressed as special comments in AspectJ interface definitions, following "`//@`" or enclosed between "`/**@`" and "`*/`".

Pipa can be used as a tool for describing test oracles during testing of aspects. Our approach uses specifications as test oracles and Pipa is used to write such specifications. The specification of an aspect describes what the aspect does but not how it does. Each aspect under test is assumed to be annotated with Pipa assertions, such as pre- and postconditions, and aspect invariants, which describe the aspectual behavior of the aspect. We propose to develop a runtime assertion checker for Pipa, which can be used to detect assertions violations at runtime and to interpret them as either test successes or failures during the testing process.

## 6. CONCLUSION

Automated software testing helps reduce manual testing effort in aspect-oriented software development. We have proposed our perspectives on automated testing of aspect-oriented programs. We classify automated testing techniques based on three dimensions: testing aspectual behavior or aspectual composition, unit tests or integration tests, and test-input generation or test oracles. We have illustrated various techniques including test-input generation, test selection based on coverage information and mutation testing, and runtime behavior checking with specifications.

Many techniques and tools that we have developed are based on leveraging existing Java tools to test aspect-oriented programs such as AspectJ programs because Java bytecode produced by an AspectJ compiler [1, 9] can often be analyzed or tested by existing Java testing tools. However, due to the specific features in aspect-oriented programs, sometimes we cannot directly apply these Java testing techniques or tools to test aspect-oriented programs. In such cases, we create some new wrappers or other mechanisms to enable us to leverage existing Java testing tools.

On the other hand, although much work has been done for testing aspect-oriented programs, there are still many open problems in this area. For example, how to develop a systematic approach for integration testing of aspect-oriented programs? How to empirically evaluate the existing testing techniques for aspect-oriented programs? How to develop an efficient benchmark suite that is suitable for evaluating existing testing techniques and tools for aspect-oriented programs? Moreover, the role of static analysis techniques (such as program slicing [23]) should also be investigated in supporting automated testing of aspect-oriented programs.

## 7. REFERENCES

[1] AspectJ compiler 1.2, May 2004. `http://eclipse.org/aspectj/`.

[2] P. Anbalagan and T. Xie. APTE: Automated pointcut testing for AspectJ programs. In *Proc. 2nd Workshop on Testing Aspect-Oriented Programs*, pages 27–32, July 2006.

[3] P. Anbalagan and T. Xie. Efficient mutant generation for mutation testing of pointcuts in aspect-oriented programs. In *Proc. 2nd Workshop on Mutation Analysis*, pages 51–56, November 2006.

[4] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[5] L. Baresi and M. Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001.

[6] B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[7] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34:1025–1050, 2004.

[8] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *Proc. 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 154–163, 1994.

[9] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proc. 3rd International Conference on Aspect-Oriented Software Development*, pages 26–35, 2004.

[10] JUnit, 2003. `http://www.junit.org`.

[11] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. 11th European Conference on Object-Oriented Programming*, pages 220–242. 1997.

[12] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Trans. Software Eng.*, 9(3):347–354, 1983.

[13] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998.

[14] C. V. Lopes and T. Ngo. Unit testing aspectual behavior. In *Proc. 1st Workshop on Testing Aspect-Oriented Programs*, March 2005.

[15] Y.-S. Ma, J. Offutt, and Y. R. Kwon. MuJava: an automated class mutation system. *Softw. Test. Verif. Reliab.*, 15(2):97–133, 2005.

[16] D. Marinov and S. Khurshid. Testera: A novel framework for automated testing of java programs. In *Proc. 16th IEEE International Conference on Automated Software Engineering*, pages 22–31, 2001.

[17] Mock objects, 2004. `http://www.mockobjects.com`.

[18] P. Netisopakul, L. J. White, J. Morris, and D. Hoffman. Data coverage testing of programs for container classes. In *Proc. 13th International Symposium on Software Reliability Engineering*, pages 183–194, November 2002.

[19] Parasoft. Jtest manuals version 4.5. Online manual, April 2003. `http://www.parasoft.com/`.

[20] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367–375, 1985.

[21] D. Saff and M. D. Ernst. Automatic mock object creation for test factoring. In *Proc. Workshop on Program Analysis for Software Tools and Engineering*, pages 49–51, June 2004.

[22] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proc. 21st International Conference on Software Engineering*, pages 107–119, 1999.

[23] M. Weiser. Program slicing. In *Proc. 5th International Conference on Software Engineering*, pages 439–449, 1981.

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

[25] T. Xie, D. Marinov, W. Schulte, and D. Noktin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, April 2005.

[26] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of AspectJ programs. In *Proc. 5th International Conference on Aspect-Oriented Software Development*, pages 190–201, March 2006.

[27] T. Xie, J. Zhao, D. Marinov, and D. Notkin. Automated test generation for AspectJ program. In *Proc. 1st Workshop on Testing Aspect-Oriented Programs*, March 2005.

[28] T. Xie, J. Zhao, D. Marinov, and D. Notkin. Detecting redundant unit tests for AspectJ programs. In *Proc. 17th IEEE International Conference on Software Reliability Engineering*, pages 179–188, November 2006.

[29] Y. Yamazaki, K. Sakurai, S. Matsuura, H. Masuhara, H. Hashiura, and S. Komiya. A unit testing framework for aspects without weaving. In *Proc. 1st Workshop on Testing Aspect-Oriented Programs*, March 2005.

[30] J. Zhao. Data-flow-based unit testing of aspect-oriented programs. In *Proc. 27th IEEE International Computer Software and Applications Conference*, pages 188–197, Nov. 2003.

[31] J. Zhao and M. Rinard. Pipa: A behavioral interface specification language for AspectJ. In *Proc. 6th International Conference on Fundamental Approaches to Software Engineering*, pages 150–165, April 2003.

[32] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.