Automated Test Generation for AspectJ Programs

Tao Xie[1] Jianjun Zhao[2] Darko Marinov[3] David Notkin[1]
[1] Department of Computer Science & Engineering, University of Washington, USA
[2] Department of Computer Science & Engineering, Fukuoka Institute of Technology, Japan
[3] Department of Computer Science, University of Illinois, Urbana-Champaign, IL 61801, USA
{taoxie,notkin}@cs.washington.edu, zhao@cs.fit.ac.jp, marinov@cs.uiuc.edu

ABSTRACT

Aspect-oriented software development is a new paradigm that improves separation of concerns in software development, and has gained popularity with the adoption of languages such as AspectJ. Automated test generation for AspectJ programs is important for reducing manual effort during testing AspectJ programs. This position paper proposes Wrasp, a framework for automatically generating tests for AspectJ programs. To generate integration tests for bases classes and aspects, Wrasp creates a wrapper class for each base class under test and feeds the wrapper class to the existing test generation tools adapted for generating integration tests towards three levels of units in AspectJ programs: advised methods, advice, and intertype methods. To generate unit tests for advice, Wrasp treats an aspect class as the class under test and generates unit tests for advice in the aspect class.

1. INTRODUCTION

Aspect-oriented software development (AOSD) is a new technique that improves separation of concerns in software development [6, 15, 16, 21]. AOSD makes it possible to modularize crosscutting concerns of a software system, and therefore make the system easy to maintain and evolve.

Current research in AOSD has focused mainly on the activities of software system design, problem analysis, and language implementation. Although it is well known that testing is a labor-intensive process that can account for even half the total cost of software development [5], little research on (especially automated) testing of AOSD has been carried out. AOSD can lead to better-quality software, but it does not provide the correctness by itself. An aspect-oriented design can lead to a better system architecture, and an aspect-oriented programming language enforces a disciplined coding style, but they do not protect against mistakes made by programmers during the system development. As a result, software testing remains an inevitable task in AOSD.

Aspect-oriented programming languages, such as AspectJ [15], introduce some new language constructs (such as join points, advice, intertype declarations, and aspects) to the common object-oriented programming languages, such as Java. These specific constructs require adapting the common testing concepts.

Our research focuses on both unit testing and integration testing for aspect-oriented programs. Unit testing is the process of testing each basic component (a unit) of a program to validate that it correctly implements its detailed design [32], whereas integration testing is the process of testing a partially integrated application to expose defects involving the interaction of collaborating components. For an AspectJ program, we can preform unit testing on aspects in isolation and perform integration testing on aspects in the context with some affected classes since the intended use of an aspect is to affect the behavior of one or more classes through join points and advice. Integration testing allows for testing the complex interactions between the aspect and the affected classes. We can use the existing tools that automate test generation for Java to automate test generation for the aspects and their affected classes in an AspectJ program. Test-generation tools for Java are available commercially (e.g., Jtest [17]) or as research prototypes (e.g., Java Pathfinder [14,23], JCrasher [9], Rostra [24,25], and Symstra [26]). Given the bytecode of a Java class, these tools test the class by generating and executing various method sequences on the objects of the class.

In this paper, we propose Wrasp, a novel framework for automatically generating both unit and integration tests for AspectJ programs. The core component of Wrasp is its wrapper mechanism to support the generation of integration tests. To generate integration tests for a base class and aspects, Wrasp first creates a wrapper class for the base class and then feeds the wrapper class instead of the woven base class to the existing test generation tools. The wrapper mechanism provides a nice interface between the program under test and the test generation tools. The wrapper mechanism also allows the test generation tools to take advice related to `call` join points into account during test generation. In addition, the mechanism prevents the methods in generated test classes from being advised by unwanted advice. To support the generation of integration tests, the other component of Wrasp adapts the existing test generation techniques to generate integration tests towards three types of units in AspectJ programs: advised methods, advice, and intertype methods. To generate unit tests for advice, Wrasp treats an aspect class as the class under test and generates unit tests for advice in the aspect class.

The remainder of this paper is structured as follows: Section 2 starts with the background information on AspectJ and automated test generation for object-oriented programs. Next, Section 3 presents the example that is used to illustrate the Wrasp framework. Section 4 introduces the Wrasp framework for generating both integration tests and unit tests. Section 5 discusses related work and Section 6 concludes.

2. BACKGROUND

Wrasp generates tests for AspectJ programs based on the existing automated test generation tools for object-oriented programs. We next introduces background information on AspectJ and automated test generation for object-oriented programs.

2.1 AspectJ

The Wrasp framework is proposed to automate test generation for programs written in AspectJ [1], a widely used aspect-oriented language, and compiled with the AspectJ compiler [1, 12], but the

underlying ideas of Wrasp apply to the general class of aspect-oriented languages such as Hyper/J [21].

AspectJ adds to Java some new concepts and associated constructs including join points, pointcuts, advice, intertype declarations, and aspects. The *join point* in AspectJ is an essential concept in the composition of an aspect with other classes. It is a well-defined point in the execution of a program, such as a call to a method, an access to an attribute, an object initialization, or an exception handler. A *pointcut* is a set of joint points that optionally expose some of the values in the execution of these joint points. AspectJ defines several primitive *pointcut designators* that can identify all types of join points. Pointcuts in AspectJ can be composed and new pointcut designators can be defined according to these combinations.

Advice is a method-like mechanism used to define certain code that executes *before*, *after*, or *around* a pointcut. The `around` advice executes *in place* of the indicated pointcut, which allows the aspect to replace a method. An aspect can also use an *intertype declaration* to add a public or private method, field, or interface implementation declaration into a class.

Aspects are modular units of crosscutting implementation. Aspects are defined by aspect declarations, which have similar forms of class declarations. Aspect declarations may include pointcut, advice, and intertype declarations, as well as method declarations that are permitted in class declarations.

The AspectJ compiler [1,12] uses *aspect weaving* to compose the code of the base classes and the aspects to ensure that applicable advice runs at the appropriate join points. After aspect weaving, these base classes are then called *woven classes* and the methods in these classes are called *advised methods*.

During the weaving process, each aspect in an AspectJ program is compiled into a standard Java class (called *aspect class*) and each piece of advice declared in the aspect is compiled into a public non-static method in the aspect class. The parameters of this public method are the same as the parameters of the advice, possibly in addition to some `thisJoinPoint` parameters. The body of this public method is usually the same as the body of the advice. Some calls to the advice are inserted at appropriate locations of the base class. At each site of these inserted calls, a singleton object of an aspect class is first obtained by calling the static method `aspectOf` that is defined in the aspect class. Then a piece of advice is invoked on the aspect object.

Each intertype method declaration in the aspect is compiled into a public static method (called *intertype method*) in the aspect class and each intertype field declaration is compiled into a field in the base class. The parameters of this public method are the same as the parameters of the declared method in the aspect except that the declared method's receiver object is inserted as the first parameter of the intertype method. A wrapper method is inserted in the base class which invokes the actual method implementation in the aspect class. Moreover, all accesses to the fields inserted in the base class are through two public static wrapper methods in the aspect class for getting and setting field respectively. For more information about AspectJ weaving, refer to [12].

2.2 Automated Test Generation

There are two main types of automated test generation for object-oriented programs: specification-based (black-box) test generation and program-based (white-box) test generation. Specification-based test generation takes advantage of specifications during test generation. For example, Korat [7] monitors field accesses within the execution of a Java predicate (an implementation for checking class invariants) and uses this information to prune the search for valid object states. AsmlT [10, 11] produces finite state machines by executing abstract state machines and generates tests based on the extracted finite state machines. Given a Java predicate, Java Pathfinder [14, 23] generates valid object states by using symbolic execution implemented upon its explicit-state model checker [22].

Program-based test generation takes advantage of implementations during test generation. For example, both Parasoft Jtest [17] and JCrasher [9] generate random method sequences for the class under test. Buy *et al.* [8] use dataflow analysis, symbolic execution, and automated deduction to produce method sequences for the class under test. Both Java Pathfinder [23] and Rostra [24, 25] (developed in our previous work) generate method sequences by exploring the concrete-object-state space. Symstra [26] (developed in our previous work) uses symbolic executions to produce symbolic states instead of concrete states produced by concrete executions in Rostra. Then Symstra checks the subsumption relationship among symbolic states and prunes the state space based on the state subsumption. Symstra can effectively generate tests for achieving higher structural coverage faster than Rostra.

3. EXAMPLE

We next illustrate Wrasp by using a simple integer stack example adapted from Rinard *et al.* [18]. Figure 1 shows the implementation of the class. This class provides standard stack operations as public non-constructor methods: `push` and `pop`. The class also has one package-private method: `iterator` returns an iterator that can be used to traverse the items in the stack. The implementation of the iterator class is shown in Figure 2.

The stack implementation accommodates integers as stack items. Figure 3 shows three aspects that enhance the stack implementation. The `NonNegativeArg` aspect checks whether a method arguments are nonnegative integers. The aspect contains a piece of advice that goes through all arguments of an about to be executed method to check whether they are nonnegative integers. The advice is executed before a call of any method. The `NonNegative` aspect checks the property of nonnegative items: the aspect contains a piece of advice that iterates through all items to check whether they are nonnegative integers. The advice is executed before an execution of a `Stack` method.

The `PushCount` aspect counts the number of times a `Stack`'s `push` method is invoked on an object since its creation. The aspect declares an intertype field `count` for the `Stack` class. The field keeps the number of times a `Stack`'s `Push` method is invoked. The aspect declares a public intertype method `increaseCount` for the `Stack` class. The method increases the `count` intertype field of `Stack`. Note that we declare this intertype method as public for illustration purpose. Then a client can invoke the `increaseCount` method to increase `count` without invoking `push`. The aspect also contains a piece of `around` advice that invokes the `Stack`'s intertype method `increaseCount` declared in the aspect. The advice is executed around any execution of `Stack`'s `push` method.

4. FRAMEWORK

We propose the Wrasp framework for generating both integration and unit tests for AspectJ programs. Integration tests are test inputs to the base classes woven with aspect classes[1], which can be generated towards advised methods, advice, or intertype methods. Unit tests are test inputs to advice in aspect classes, which are generated to test advice in isolation.

[1]Integration tests for the base classes woven with aspect classes can also be seen as unit tests for the base classes when our focus is not the interaction between the base classes and aspect classes.

```
class Cell {
    int data;  Cell next;
    Cell(Cell n, int i) {
       next = n;
       data = i;
    }
}

public class Stack {
   Cell head;
   public Stack() {
        head = null;
   }
   public boolean push(int i) {
      head = new Cell(head, i);
      return true;
   }
   public int pop() {
      if (head == null)
        throw new RuntimeException("empty");
      int result = head.data;
      head = head.next;
      return result;
   }
   Iterator iterator() {
      return new StackItr(head);
   }
 }
```

Figure 1: An integer stack implementation

```
interface Iterator {
   public boolean hasNext();
   public int next();
}

public class StackItr implements Iterator {
   private Cell cell;
   public StackItr(Cell head) {
      this.cell = head;
   }
   public boolean hasNext() {
      return cell != null;
   }
   public int next() {
      int result = cell.data;
      cell = cell.next;
      return result;
   }
}
```

Figure 2: Stack Iterator

4.1 Generation of Integration Tests

The Wrasp framework for integration testing consists of two components. The first component generates a wrapper class for a base class (Section 4.1.1). The second component adapts the existing test generation techniques by treating the wrapper class as the class under test (Section 4.1.2).

4.1.1 Wrapper Generation

Several automated test generation tools generate tests based on Java bytecode instead of source code. For example, both Parasoft Jtest [17] and JCrasher [9] generates random method sequences for the class under test based on its bytecode. Based on Java bytecode, our previous work developed Rostra [24, 25] and Symstra [26] for generating only method sequences that produce different inputs for methods under test. To generate tests for AspectJ programs, we can simply feed their woven bytecode to these existing test generation tools and use these tools to generate tests for the woven bytecode.

However, there are several issues to be addressed when we reuse these existing test generation tools to generate tests for AspectJ programs. First, 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

```
aspect NonNegativeArg {
   before() : call(* *.*(..)) {
      Object args[] = thisJoinPoint.getArgs();
      for(int i=0; i<args.length; i++) {
         if ((args[i] instanceof Integer) &&
          (((Integer)args[i]).intValue() < 0))
          throw new RuntimeException("negative arg of " +
            thisJoinPoint.getSignature().toShortString());
      }
    }
}

aspect NonNegative {
   before(Stack stack) : execution(* Stack.*(..)) &&
           && target(stack) {
   Iterator it = stack.iterator();
      while (it.hasNext()) {
          int i = it.next();
          if (i < 0) throw new RuntimeException("negative");
     }
  }
}

aspect PushCount {
  int Stack.count = 0;
     public void Stack.increaseCount() {
         count++;
    }
     boolean around(Stack stack):
            execution(* Stack.push(int)) && target(stack) {
        boolean ret = proceed(stack);
         stack.incrementCount();
        return ret;
     }
}
```

Figure 3: `NonNegative`, `NonNegativeArg`, and `PushCount` aspects

call sites, which are not available before test generation. Second, although we can use the AspectJ compiler [1, 12] to weave the generated test 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 test code invoke these intertype methods. In addition, weaving the generated test classes with the aspect classes could introduce unwanted advice into the test classes.

To address these issues, we generate a wrapper class for each base class under test. There are six steps for generating integration tests based on the wrapper class:

1. Compile and weave the base class and aspects into class bytecode using the AspectJ compiler.
2. Generate a wrapper class for the base class based on the woven class bytecode.
3. Compile and weave the base class, wrapper class, and aspects into class bytecode using the AspectJ compiler.
4. Clean up unwanted woven code in the woven wrapper class.
5. Generate tests for the woven wrapper class using the existing test generation tools based on class bytecode.
6. Compile the generated test class into class bytecode using the Java compiler [3].

In the second step, we generate a wrapper class for the base class under test. In this wrapper class, we generate 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, we also generate a wrapper method for each public intertype method woven into the base class. This wrapper method uses Java reflection [3] to

invoke the intertype method; otherwise, the compilation in the third step can fail because intertype methods are not recognized by the AspectJ compiler before compilation. Figure 4 shows the wrapper class generated for the `Stack` class (Figure 1) woven with the three aspects (Figure 3).

In the third step, we use the AspectJ compiler to weave the wrapper class with the base class and aspects. This step ensures that the advice related to `call` join points is executed during 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.

In the fourth step, we need to clean up unwanted woven code in the woven wrapper class. For example, suppose that we change the `call` join point of the `NonNegativeArg` aspect to an `execution` join point, the wrapper method for the `push` method of `Stack` is also advised by the `execution` advice. The advice is unwanted by the wrapper method; otherwise, `push`'s arguments are checked twice during test generation, one time in the wrapper method and the other time in the advised method. We scan the bytecode of the woven wrapper class and remove the woven code that are for advice related to `execution` join points. Note that we need to keep the woven code that are for advice related to `call` join points.

In the fifth step, we feed the woven wrapper class to the existing test generation tools based on bytecode, such as Parasoft Jtest [17], JCrasher [9], Rostra [24, 25], and Symstra [26]. These tools export generated tests to test code, usually as a JUnit test class [13]. The next section discusses how we adapt the existing test generation techniques for testing AspectJ programs.

In the final step, we use the Java compiler [3] to compile the exported test class. We do not use the AspectJ compiler to weave the exported test class with the wrapper class, base class, or aspects, because the weaving process can introduce unwanted woven code into the test class.

4.1.2 Test Generation

In our previous work [27], we proposed Aspectra, a framework for detecting redundant tests for AspectJ programs. Aspectra detects a subset of automatically generated tests towards three levels of units in AspectJ programs: advised methods, advice, and intertype methods. In this work, Wrasp further explores how to generate integration tests effectively towards these three levels of units in AspectJ programs.

When generating tests for advised methods, Wrasp enables the existing test generation techniques to take into account the effect of advice on the advised methods because the code under test (wrapper class) eventually invokes woven advice, even advice related to `call` join points. The existing test generation tools based on concrete-state exploration, such as Java Pathfinder [22,23] and Rostra [24, 25], can directly operate on the wrapper class produced by Wrasp. For example, the `NonNegativeArg` advice has been woven in the wrapper class `StackWrapper`, in particular in the call sites of the three public `Stack` methods. Although the `NonNegativeArg` advice does not directly alter the states of a `Stack` object, when the argument of `push` is a negative integer, the advice alters the control flow of the method execution by throwing an exception, thus changing the method execution's effect on the `Stack` object state. Without using Wrasp, tools such as Java Pathfinder and Rostra cannot take into account the effect of the `NonNegativeArg` advice during state exploration.

When generating tests for advice, Wrasp proposes techniques to adapt the existing test generation tools such as Rostra [24, 25] and Symstra [26]. Because a piece of advice often reads only a subset of the base class' fields, only these fields are relevant for affecting

```
public class StackWrapper {
   Stack s;
   public StackWrapper() {
      s = new Stack();
   }
   public boolean push(int i) {
      return s.push(i);
   }
   public int pop() {
      return s.pop();
   }

   public void increaseCount() {
      Class cls = Class.forName("Stack");
      Method meth = cls.getMethod("increaseCount", null);
      meth.invoke(s, null);
   }
}
```

Figure 4: The wrapper class for `Stack`

the behavior of the advice. We can use Rinard *et al.*'s static analysis [18] to determine which fields of the base class can be read by the advice. Then during the concrete-state exploration of Rostra [24, 25], we can project the whole concrete state of the base class on these fields. With this state projection technique, Rostra can be better guided to explore different inputs for the advice. For example, the `PushCount` advice reads only the intertype field `count`. During the concrete-state exploration of Rostra, we can consider two object states to be equivalent for the `PushCount` advice if these two states have the same value of `count` (even if the stack elements in the state are different). When focusing on testing the `PushCount` advice, Rostra can effectively generate `push`'s inputs (receiver states and arguments) whose executions eventually produce different inputs for the `PushCount` advice.

In order to achieve structural coverage for advice, Wrasp extends Symstra to also apply symbolic execution on the advice code. For example, the `NonNegativeArg` advice contains a conditional `((args[i] instanceof Integer) && (((Integer)args[i]).intValue() < 0))`. We can rewrite the conditional to apply generalized symbolic execution [14]. After the symbolic execution of the `NonNegativeArg` advice, Symstra can use a constraint solver to generate both positive and negative integers for the `push` method's argument.

When generating tests for a public intertype method of a base class, Wrasp treats the intertype method as a regular public method of the base class; in particular, Wrasp creates a public wrapper method in the wrapper class for the intertype method and then generates tests to exercise the wrapper method. In addition, we can also use Rinard *et al.*'s static analysis [18] to determine which intertype fields or fields of the base class can be read by the intertype method, and then focus on these read fields during state matching and exploration. For example, the `increaseCount` intertype methods defined in the `PushCount` aspect reads only the `count` intertype field; therefore, we can explore and generate only different values of `count` in the receiver object state for `increaseCount`.

4.2 Generation of Unit Tests

The Wrasp framework for unit-test generation produces test inputs to directly exercise advice in an aspect class in isolation. Wrasp treats an aspect class as the class under test and advice defined in the aspect class as the methods under test. For example, after we use the AspectJ compiler [1, 12] to compile the aspect classes in Figure 3, we can get the interfaces of the compiled aspect classes shown in Figure 5. We do not list those helper methods such as `aspectOf` and `hasAspect`; these helper methods are the same for all aspect classes and it is not necessary to invoke them when testing

```
class NonNegativeArg {
 public NonNegativeArg();
 public void
     ajc$before$NonNegativeArg$1$8d7380d7(JoinPoint j);
   ...
}

class NonNegative {
 public NonNegative();
 public void
     ajc$before$NonNegative$1$d9be608f(Stack s);
   ...
}

aspect PushCount {
 public PushCount();
 public static int
   ajc$interMethod$PushCount$Stack$incrementCount(Stack s);
 public int
   ajc$around$PushCount$1$69ee3a63(Stack s, AroundClosure c);
   ...
}
```

Figure 5: The interfaces of the compiled aspect classes of `NonNegative`, `NonNegativeArg`, and `PushCount`

advice in isolation. In addition, we do not list a dispatch method for the intertype method `increaseCount` and several dispatch methods for initializing, getting, setting the intertype field `count`; it is not necessary to invoke these dispatch methods when testing advice in isolation.

We feed the complied aspect classes to the existing test generation tools such as Parasoft Jtest [17]. Jtest then generates unit tests for the public methods in the compiled aspect classes. For example, one of the unit tests generated by Jtest for the `NonNegative` aspect class is as follows:

```
public void testNonNegative1() {
 Stack t0 = new Stack();
 NonNegative THIS = new NonNegative();
 THIS.ajc$before$NonNegative$1$d9be608f(t0);
}
```

One of the unit tests generated by Jtest for the `PushCount` aspect class is as follows:

```
public void testPushCount1() {
 Stack t0 = new Stack();
 PushCount.ajc$interMethod$PushCount$Stack$incrementCount(t0);
}
```

However, Jtest is not able to generate meaningful tests for the `before` advice method in the `NonNegativeArg` aspect or the `around` advice method in the `PushCount` aspect because the types of their method arguments include `JoinPoint` or `AroundClosure`. These two classes belong to the AspectJ runtime environment and Jtest cannot create appropriate objects for them. In future work, we plan to use automatically created mock objects [19] to simulate the method calls on these objects based on the execution history during integration testing.

5. RELATED WORK

The Wrasp framework is complementary to Aspectra (proposed in our previous work [27]), a framework for detecting redundant tests for AspectJ programs. Aspectra operates on any set of generated tests for AspectJ programs and reduces the size of the generated tests for inspection when specifications are not written for AspectJ programs. Wrasp automatically generates tests for AspectJ programs, which can be minimized by Aspectra for inspection.

Souter *et al.* [20] developed a test selection technique based on concerns. A concern is the code associated with a particular maintenance task. An aspect in AspectJ programs can be seen as a concern. To reduce the space and time cost of running tests on instrumented code, they proposed to instrument only the concerns for collecting runtime information. They also proposed to select or prioritize tests for the selected concerns. In particular, they selects a test if the test covers a concern that has not been been exercised by previously selected tests. Zhou *et al.* [31] also used the same technique for selecting tests for an aspect. These two test selection approaches assume that there already exist a set of tests for an AspectJ program (or just for the base classes in the AspectJ program), whereas Wrasp focuses on automatically generating tests for an AspectJ program during the unit and integration testing of the aspects and base classes in the AspectJ program.

Xu *et al.* [28] presented a specification-based testing approach for aspect-oriented programs. The approach creates aspectual state models by extending the existing FREE (Flattened Regular Expression) state model, which was originally proposed for testing object-oriented programs. Based on the model, they developed two techniques for testing aspect-oriented programs. The first technique transforms an aspectual state model to a transition tree and generates tests based on the tree. The second techniques constructs and searches an aspect flow graph for achieving statement coverage and branch coverage. Their work focuses on testing aspect-oriented programs based on abstract state models, whereas Wrasp focuses mainly on automatically generating integration and unit tests based on implementations.

Alexander *et al.* [2] proposed a fault model for aspect-oriented programming, including six types of faults that may occur in aspect-oriented systems. Their fault model provides useful guidance in developing testing coverage tools for aspect-oriented programs, whereas Wrasp proposes an automated approach for generating tests to achieve structural coverage and object-state coverage.

Zhao [29, 30] proposed a data-flow-based unit testing approach for aspect-oriented programs. For each aspect or class, the approach performs three levels of testing: intra-module, inter-module, and intra-aspect or intra-class testing. His work focused on unit testing of aspect-oriented programs based on data flow, whereas Wrasp focuses on automatically generating both unit and integration tests for AspectJ programs based on primarily object states.

Rinard *et al.* [18] proposed a classification system for aspect-oriented programs and developed a static analysis to support automatic classification. Their system characterizes the interactions between advice and advised methods based on field accesses. Developers can use the classification system and analysis to structure their understanding of the aspect-oriented programs. Wrasp uses Rinard *et al.*'s static analysis to determine whether a piece of advice reads one field in the aspect or base class. Based on this information, Wrasp can focus on these read fields during state exploration for test generation and thus effectively explore different inputs for the advice.

6. CONCLUSION

We have proposed Wrasp, a novel framework for automatically generating tests for AspectJ programs. For integration testing, Wrasp creates a wrapper class for each base class under test. The wrapper mechanism allows test generation tools to indirectly exercise advice related to `call` join points during test generation. At the same time, the mechanism prevents the methods in generated test classes from being advised by unwanted advice. Wrasp also adapts the existing test generation techniques for generating integration tests towards three types of units in AspectJ programs: pieces of advice, advised methods, and intertype methods. For unit testing, Wrasp feeds compiled aspect classes to the existing test generation tools and generates unit tests to directly exercise advice.

Acknowledgments

This work was supported in part by the National Science Foundation under grants ITR 0086003 and CCR00-86154 and the Japan Society for Promotion of Science (JSPS) under Grand-in-Aid for Scientific Research (C) (No.15500027). We acknowledge support through the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

7. REFERENCES

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

[2] 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, Fort Collins, Colorado, 2004.

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

[4] K. Beck. *Extreme programming explained*. Addison-Wesley, 2000.

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

[6] L. Bergmans and M. Aksits. Composing crosscutting concerns using composition filters. *Commun. ACM*, 44(10):51–57, 2001.

[7] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis*, pages 123–133, 2002.

[8] U. Buy, A. Orso, and M. Pezze. Automated testing of classes. In *Proc. the International Symposium on Software Testing and Analysis*, pages 39–48. ACM Press, 2000.

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

[10] Foundations of Software Engineering, Microsoft Research. The AsmL test generator tool. `http://research.microsoft.com/fse/asml/doc/AsmLTester.html`.

[11] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *Proc. International Symposium on Software Testing and Analysis*, pages 112–122, 2002.

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

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

[14] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, April 2003.

[15] 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.

[16] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Commun. ACM*, 44(10):39–41, 2001.

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

[18] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Proc. 12th International Symposium on the Foundations of Software Engineering*, pages 147–158, 2004.

[19] D. Saff and M. D. Ernst. Automatic mock object creation for test factoring. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'04)*, pages 49–51, Washington, DC, USA, June 7–8, 2004.

[20] A. L. Souter, D. Shepherd, and L. L. Pollock. Testing with respect to concerns. In *Proc. International Conference on Software Maintenance*, page 54, 2003.

[21] 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.

[22] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE)*, pages 3–12, 2000.

[23] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, 2004.

[24] T. Xie, D. Marinov, and D. Notkin. Improving generation of object-oriented test suites by avoiding redundant tests. Technical Report UW-CSE-04-01-05, University of Washington Department of Computer Science and Engineering, Seattle, WA, Jan. 2004.

[25] 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.

[26] 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 (TACAS 2005)*, April 2005.

[27] T. Xie, J. Zhao, D. Marinov, and D. Notkin. Detecting redundant unit tests for AspectJ programs. Technical Report UW-CSE-04-10-03, University of Washington Department of Computer Science and Engineering, Seattle, WA, Oct. 2004.

[28] D. Xu, W. Xu, and K. Nygard. A state-based approach to testing aspect-oriented programs. Technical Report NDSU-CS-TR04-XU03, North Dakota State University Computer Science Department, September 2004.

[29] J. Zhao. Tool support for unit testing of aspect-oriented software. In *Proc. OOPSLA'2002 Workshop on Tools for Aspect-Oriented Software Development*, Nov. 2002.

[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] Y. Zhou, D. Richardson, and H. Ziv. Towards a practical approach to test aspect-oriented software. In *Proc. 2004 Workshop on Testing Component-based Systems (TECOS 2004), Net.ObjectiveDays*, Sept. 2004.

[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.