APTE: Automated Pointcut Testing for AspectJ Programs

Prasanth Anbalagan[1] Tao Xie[2]
Department of Computer Science, North Carolina State University, Raleigh, NC 27606, USA
[1]panbala@ncsu.edu [2]xie@csc.ncsu.edu

Abstract

Aspect-Oriented Programming (AOP) has been proposed as a methodology that provides new modularization of software systems by allowing encapsulation of crosscutting concerns. AspectJ, an aspect-oriented programming language, provides two major constructs: advice and pointcuts. The scope of pointcuts spans across various objects instantiated from the classes. With the increase in the number of objects, classes, and integration of source code, it is likely that a developer writes a pointcut that does not serve its intended purpose. Therefore, there is a need to test pointcuts for validating the correctness of their expressions.

In this paper, we propose APTE, an automated framework that tests pointcuts in AspectJ programs with the help of AJTE, an existing unit-testing framework without weaving. Our new APTE framework identifies joinpoints that satisfy a pointcut expression and a set of boundary joinpoints, which are events that do not satisfy a pointcut expression but are close to the matched joinpoints. The boundary joinpoints are identified as those unmatched joinpoint candidates whose distance from the matched joinpoints are less than a predefined threshold value. A developer could inspect these matched joinpoints and boundary joinpoints for correctness of the pointcuts.

1 Introduction

Aspect-Oriented Programming(AOP) [5, 4] attempts to aid programmers in the separation of concerns: breaking down of a program into distinct parts that overlap in functionality. In particular, AOP focuses on the modularization of concerns as appropriate for the host language and provides a mechanism for describing concerns that crosscut each other. AspectJ [5, 13, 8] is an implementation of AOP for the Java [12] programming language, being built as an extension to the language. The major components of this AOP language are joinpoints, pointcuts, advice, and aspects. *Joinpoints* are well-defined locations within the primary code where a concern will crosscut the application. Joinpoints can be method calls, constructor invocations, or some other points in the execution of a program. *Pointcuts* are constructs that match the joinpoints, which perform a specific action called *advice* when triggered. The encapsulation of joinpoint, pointcut and advice is provided by an *aspect*.

The crosscutting behavior of AspectJ can be divided into two major components: what the behavior does (advice) and where the behavior applies (pointcut). Unit testing [16, 7, 15, 14] in AOP can be carried out by weaving or without weaving the aspect code with the encapsulating components (called the target classes). Unit testing without weaving involves tests in isolation that verify individual components of the AOP language independent of other components.

Pointcuts are predicates that match an event (Joinpoints) in the execution of a program. Pointcuts are modelled using expressions that identify the type, scope, or context of the events. AspectJ pointcuts provide the features of abstraction and composition, which include various designators, wildcards, and their combination with logical operators. Developers often lack high confidence on assuring that these pointcuts are specified as intended. In addition, during program evolution, specified pointcuts may not be robust enough to stand the maximum chance of continuing to match the intended joinpoints, and only the intended join points.

In this paper, we propose a framework that automatically checks the correctness of the pointcut expressions in aspect code. In the implementation of the framework, we use AJTE [15], an existing framework that performs unit testing without weaving. Our framework receives a threshold value and a list of source files, including the source of aspects and target classes. The framework outputs a list of matched joinpoints in the target classes as well as a list of boundary joinpoints, which are events that do not satisfy a pointcut expression but are close to the matched joinpoints. The framework also outputs the distances of these boundary joinpoints from the matched joinpoints, being measured to quantify their deviation from the matched ones. The boundary joinpoints are identified as those unmatched joinpoint

candidates whose distance from the matched joinpoints are less than a predefined threshold value. This threshold value is the maximum distance against which the distances of unmatched joinpoint candidates are compared and is supplied by the user to the framework. A developer could inspect the matched joinpoints and boundary joinpoints for correctness of the pointcuts.

The rest of the paper is organized as follows. Section 2 presents an overview of pointcuts in AspectJ. Section 3 illustrates our APTE framework. Section 4 describes the implementation of the framework. Section 5 provides preliminary results of applying the framework on selected subjects. Section 6 discusses issues of the framework and Section 8 concludes the paper.

2 Pointcuts in AspectJ

This section presents an example to explain pointcuts in AspectJ and discusses potential problems of pointcuts during program evolution. Section 2.1 presents an example program. Section 2.2 explains sample situations where problems could occur.

2.1 Example

We illustrate the concept of pointcuts by using a simple RSA encryption example that uses the FastDivision algorithm for initial primality test. Figure 2 shows the implementation of the classes. The `RSA` class provides the operation of RSA encryption and the `FastDivision` class performs the primality test.

The `NonNegativeArg` aspect checks whether a method argument is a nonnegative integer. The aspect contains a piece of advice that goes through the arguments of a method and checks whether these arguments are nonnegative integers. A method whose arguments need to be checked is identified by the pointcut expression "`checkarg`". In this example, the pointcut expression has been designed to match any public method that ends with "`Prime`", has an integer argument, and is invoked by an instance of any class with `boolean` as the return value.

2.2 Problems with pointcuts

The pointcut `checkarg` appears to be robust as it meets its requirements and matches only the joinpoint "`isPrime(int num)`" with a single integer argument. But in practice, the choice of primality test algorithms in encryption varies with implementation or as the program evolves. For example, consider an implementation of RSA encryption with another primality test algorithm called MillerRabin. The example provided in Figure 2 is improved by performing a second round of primality test on numbers

```
aspect NonNegativeArg {

 pointcut checkarg() :
            execution(public boolean *.*Prime(int));

 before() : checkarg() {
   Object args = thisJoinPoint.getArgs();
      if ((args instanceof Integer) &&
       (((Integer)args).intValue() < 0))
         throw new RuntimeException("negative arg of " +
            thisJoinPoint.getSignature().toShortString());
 }
}
```

Figure 1. `NonNegativeArg` **aspect**

```
class FastDivision {
     public boolean isPrime(int num){
         //Fast division primality test algorithm
         ...
     }
}

class RSA {
     public void rsa() {
          //Algorithm to perform RSA encryption
          boolean result;
          int n = genRandom();
          FastDivision obj = new FastDivision();
          result = obj.isPrime(n);
     }
}
```

Figure 2. An RSA encryption with the FastDivision algorithm

```
class MillerRabin {
  public boolean isPrime(int base, int exponent){
        //Miller Rabin primality test algorithm
        ...
  }
}

class RSA {
     public void rsa() {
        //Algorithm to perform RSA encryption
        boolean result;
        int n = genRandom();
        FastDivision obj = new FastDivision();
        result = obj.isPrime(n);
        if(result) {
          int exp = genRandom();
          MillerRabin obj = new MillerRabin();
          result = obj.isPrime(n, exp);
        }
     }
}
```

Figure 3. An RSA encryption with the MillerRabin algorithm

detected as prime by the FastDivision algorithm. Figure 3 shows the implementation of the class.

Now the pointcut matches only the "`isPrime`" method of the `FastDivision` class. Because only a single argument has been matched in the pointcut expression , the

"isPrime" method of the MillerRabin class is ruled out. Here we need to modify the pointcut expression to be more generic on the number of arguments in order to check for non-negative arguments on both the primality test functions. Similarly there could be instances where the designed pointcut matches unwanted joinpoints and has to be narrowed down to avoid the unwanted matches. Identifying such errors manually is tedious when the number of objects, classes and source codes is large.

Even when developers design pointcuts for the current program version, they could make mistakes in defining the right pointcuts: designed pointcuts may be too narrow leaving out some necessary joinpoints out of matched scope or including more than necessary joinpoints in the matched scope.

3 The APTE Framework

Figure 4 provides a high level overview of the framework. Our framework receives source files of the aspects and target classes under test. In particular, the source files are given as input to three components in the framework: candidate generator, pointcut generator, and AJTE. AJTE outputs a test bench that consists of methods for testing the joinpoint candidates generated from the candidate generator. Then the framework feeds this result and pointcuts obtained from the pointcut generator to the distance measure component to identify the boundary joinpoints, which are events that do not satisfy a pointcut expression but are close to the matched joinpoints. The framework also outputs the distances of these boundary joinpoints from the matched joinpoints, being measured to quantify their deviation from the matched ones. A developer could inspect these selected joinpoints and joinpoint candidates for correctness of the pointcuts.

Our APTE framework has been designed to verify the correctness of the pointcut expressions in aspect code. The framework is based on the approach of automatically generating the test inputs (joinpoint candidates and pointcut expressions) to test the pointcuts and identify the boundary joinpoints with the measure calculated using the Levenshtein algorithm (also called Edit-Distance)[1].

Our framework uses AJTE, a unit-testing framework for aspects without weaving, which provides methods for testing pointcuts. This framework also provides methods for creating joinpoint objects, which are in turn used as arguments to pointcut-testing methods. The TestJoinPoint class of AJTE is the class for processing a joinpoint as an object. The TestPointcut class is the class for processing a pointcut expression as an object. Once these objects are created, the testPointcut method is used to check if the joinpoint object matches the pointcut object.

[1] http://www.levenshtein.net/

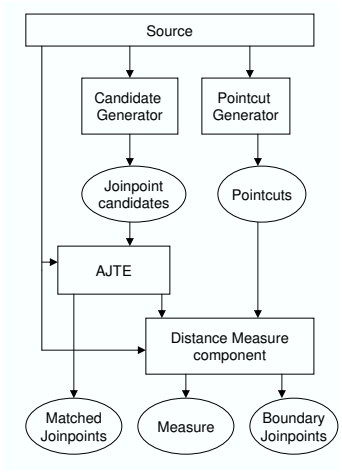

Figure 4. Overview of the APTE framework

Our framework automatically generates likely join points from the Java class file. This list forms the joinpoint candidates used to test the pointcuts. The framework generates all possible joinpoints from the Java class. Figure 5 shows the joinpoint candidates generated for the RSA encryption example. The framework identifies two execution joinpoints and two call joinpoints. The execution joinpoints identify the execution of the public method "isPrime". The call joinpoints identify the call sites of the same method. These functions are invoked by the objects of the classes MillerRabin and FastDivision. Once the joinpoint candidates are identified, they are fed as input to the TestJoinPoint factory class of AJTE to produce joinpoint objects. These joinpoint objects are then fed as arguments to the TestJoinPoint factory class of AJTE to verify if the joinpoints match the pointcut expression. This step is performed on all joinpoint candidates.

The second set of inputs include the pointcut expressions from the AspectJ source file. We parse the AspectJ source code to identify the pointcut expressions. Figure 6 shows the list of pointcut expressions identified for the NonNegativeArg aspect. This list of pointcut expressions is used to identify two categories of joinpoints: joinpoint candidates that satisfy a pointcut expression and those that do not. For each pointcut expression, the distance between the preceding two categories are measured using the Levenshtein algorithm. The result of this algorithm is an integer value that signifies the number of transformations (i.e., insertions, deletions) that should be performed on a unmatched joinpoint candidate to transform it into a joinpoint that successfully matches a pointcut expression. Figure 7 shows a sample measure between a joinpoint (which matches a pointcut expression) and a joinpoint candidate (which does not match a pointcut expression). The measure of 4 indicates that the joinpoint candidate re-

```
execution(public boolean FastDivision.isPrime(int))

execution(public boolean MillerRabin.isPrime(int,int))

call(public boolean FastDivision.isPrime(int))

call(public boolean MillerRabin.isPrime(int,int))
```

Figure 5. JoinPoint candidates

```
execution(public boolean *.*Prime(int))
```

Figure 6. Pointcut expression

```
Pointcut expression:
  execution(public boolean *.*Prime(int))

Joinpoint:
  execution(public boolean FastDivision.isPrime(int))

Joinpoint candidate:
  execution(public boolean MillerRabin.isPrime(int,int))

Distance Measure:   4
```

Figure 7. Distance Measure

quires a transformation of 4 letters to become a successful joinpoint. Then the framework identifies boundary joinpoints by comparing joinpoint candidates' distance against the user-defined threshold value and outputs these boundary joinpoints along with the list of matched joinpoints.

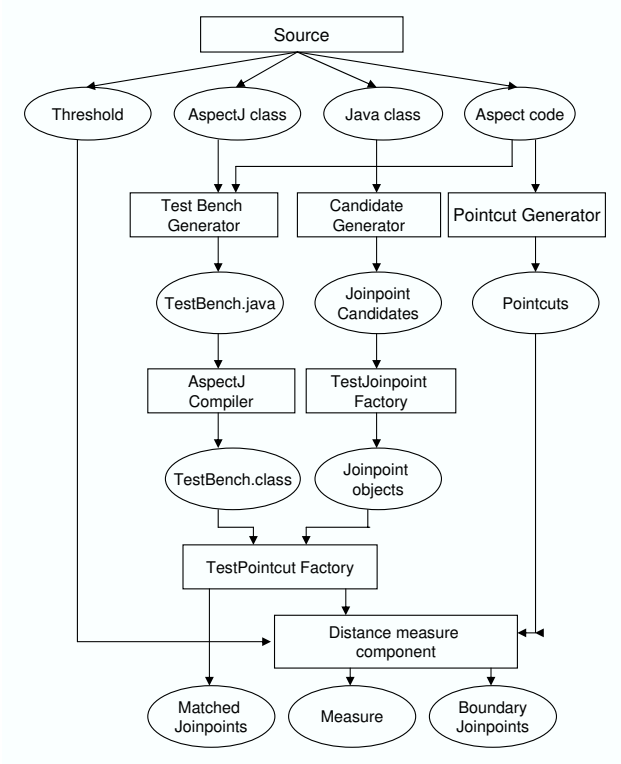

Figure 8. Implementation of the APTE framework

4 Implementation

We have implemented the framework for AspectJ and Java code using the Byte Code Engineering Library (BCEL) [2], Java reflection API [11], and AJTE. The current implementation of the framework supports an AspectJ compiler called ajc [3] Version 1.5 and Java 5 [1]. The main components of the framework includes the test bench generator, pointcut generator, candidate generator, and distance measure component, as shown in Figure 8.

4.1 Test Bench Generator

The Test Bench generator is a part of AJTE that translates the data from the aspect class (generated by the ajc AspectJ compiler) and provides two wrapper classes: `TestJoinPoint` and `TestPointcut`.

The `TestJoinPoint` class is a class for processing a join point as an object. The `TestPointcut` class is a class for processing a pointcut expression as an object. The `testPointcut` method has both a pointcut expression object and a join point object as the parameters. If the former matches the latter, it returns true; otherwise, it returns false. The framework feeds the aspect class and aspect source code as input to the test bench generator, which generates the Test Bench Java file. Then the framework automatically compiles the Test Bench Java source using the ajc AspectJ compiler to generate the Test Bench class file.

4.2 Pointcut Generator

The pointcut generator parses the AspectJ source code to identify pointcut expressions. The generated pointcuts are passed as parameters to the distance measure component to identify the group of joinpoints that are matched against a particular pointcut expression and then use them to measure the distance.

4.3 Candidate Generator

The framework feeds the Java class file as input to the candidate generator. This component uses the Java reflection API to generate the methods [6] in the class file, parse the class file and identify the call sites of the methods in the code. The component produces joinpoint candidates using the identified call sites and execution points. Then the framework feeds these joinpoint candidates to the

`TestJoinPoint` factory class to generate joinpoint objects.

4.4 Distance Measure Component

The framework feeds the list of joinpoints that satisfy a pointcut expression, candidate joinpoints that fail to satisfy any pointcut expression, and the pointcut expressions to the distance measure component. This component uses the Levenshtein algorithm to compute the distance. The distance signifies the number of characters that need to be inserted, deleted, or modified in the candidate joinpoints to transform them into joinpoints that match a particular pointcut expression. Finally the framework identifies boundary joinpoints by comparing joinpoint candidates' distance against the user-defined threshold value and outputs boundary jionpoints along with the list of matched joinpoints.

5 Preliminary Results

We describe selected outputs of applying our framework on a few pointcut expressions and joinpoints identified from the AspectJ Benchmark Suite[2] for our sample code,as shown in Table 1. Columns 1-4 show the pointcut expressions, joinpoints (which successfully match the pointcut expressions), joinpoint candidates, and the distance measures using the Levenshtein algorithm, respectively. The listed joinpoint candidates are the boundary joinpoints identified with the threshold of 20.

The first pointcut expression matches the execution of any public method that ends with "`Blocks`", accepts an integer as an argument, and returns an integer value. The executions of methods `combineBlocks` and `deleteBlocks` match these pointcut expressions whereas the executions of methods `turnblock` and `getBlock` fail. These two failed joinpoints are considered as joinpoint candidates and are measured against the joinpoints that matched the pointcut expression. The measures are shown in the last column. For example, the joinpoint candidate "`execution(public int Apte.turnBlock(int))`" differs from the joinpoint "`execution(Public int Apte.deleteBlocks(int))`" by a measure of 8. This measure indicates that the joinpoint candidate requires a transformation of 8 characters to be a successful joinpoint. Similarly, the measures for other joinpoint candidates could be found in the table.

6 Discussion

The current implementation of the framework provides support for automatic generation of joinpoint candidates for the following types of designators in a pointcut: execution (method or constructor call execution), call (method or call), initialization, and args. We plan to extend the framework to support other designators in a pointcut expression in future work.

The current implementation of the Levenshtein algorithm computes measure between a joinpoint and a joinpoint candidate. We plan to extend the distance measure component to compute distance measure against pointcut expressions: we can automatically analyze probable transformation of the pointcut expression in order to being matched by an unmatched joinpoint candidate.

7 Related Work

Lopes and Ngo [7] classify behavior of aspect-oriented programs into aspectual behavior or aspectual composition. Xie and Zhao [14] developed the Aspectra framework that automatically generates tests for testing aspectual behavior by leveraging existing Java test generation tools.

Our APTE framework performs unit testing of pointcuts for AspectJ programs by using the AJTE framework [15]. The AJTE framework allows to test whether the pointcut expression associated with a piece of advice matches (manually) provided joinpoints. Manually identifying joinpoints for testing pointcuts is tedious especially when the number of objects and classes is large. Our APTE framework reduces human effort in the testing process because APTE automates test-input generation as well test execution.

Störzer and Graf [10] developed the pointcut delta analysis was developed to support evolution of aspect-oriented software by analyzing different versions of an AspectJ program. Their analysis detects semantic differences in the program behavior because of changed pointcut semantics. Their approach does not verify the correctness of a pointcut expression in a single version of the AspectJ program. Their approach assumes that the pointcut expressions are logically correct. It compares the set of matched joinpoints for both versions. It does not attempt to search for probable joinpoints that may have been missed out due to an error in pointcut construction. Our approach helps detect these probable joinpoints. Our approach verifies the pointcut expressions against all possible joinpoints and verifies if the constructed pointcut expression is logically correct.

Mortensen and Alexander [9] proposed adequate testing of AspectJ programs, which provides a set of mutation operators to find incorrect strengths in pointcut patterns and thereby evaluate the effectiveness of a test suite. Their approach does not verify the correctness of the expression or measure unmatched joinpoints. Our approach computes the measure of the boundary joinpoints and indicates the amount of transformation required to change them into successful joinpoints.

[2] `http://www.sable.mcgill.ca/benchmarks/`

Table 1. Selected outputs of the APTE framework

Pointcuts	Joinpoints	Joinpoint Candidates	Distance
execution(public int Apte.*Blocks(int))	execution(public int Apte.combineBlocks(int))	execution(public int Apte.turnBlock(int))	8
		execution(public int Apte.getBlock(int))	9
	execution(Public int Apte.deleteBlocks(int))	execution(public int Apte.turnBlock(int))	8
		execution(public int Apte.getBlock(int))	6
execution(public String Apte.*To*(int))	execution(public String Apte.typeToString(int))	execution(public Color Apte.typeToColor(Color))	17
		execution(public Image Apte.typeToImage(Image))	17
	execution(public String Apte.typeToName(int))	execution(public Image Apte.typeToImage(Image))	17
call(private boolean Apte.readSpec*(short,boolean))	call(private boolean Apte.readSpecFile(short,boolean))	call(private boolean Apte.readSpecArray(short ,boolean, long))	12

8 Conclusion

In aspect-oriented programs, with the increase in the number of objects, classes, and integration of source code, a developer may likely write a pointcut that fails to serve its intended purpose. In this paper, we have proposed APTE, an automated framework that tests pointcuts in AspectJ programs. The framework receives a list of source files, including the source of aspects and target classes. The framework outputs a list of matched joinpoints in the target classes as well as a list of boundary joinpoints, which are events that do not satisfy a pointcut expression but are close to the matched joinpoints. The framework also outputs the distances of these boundary joinpoints from the matched joinpoints, being measured to quantify their deviation from the matched ones. Developers could inspect these selected joinpoints and joinpoints candidates for correctness of the pointcuts. Our preliminary results show that the both matched joinpoints and identified boundary joinpoints deserve developers' attention.

Acknowledgments

We would like to thank the authors of the AJTE tool for their valuable support in providing the latest version of the tool and examples for our use.

References

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

[2] M. Dahm and J. van Zyl. Byte Code Engineering Library, April 2003. `http://jakarta.apache.org/bcel/`.

[3] Eclipse. AspectJ compiler 1.5, May 2005. `http://eclipse.org/aspectj/`.

[4] R. E. Filman and T. Elrad. *Aspect Oriented Software Development*. Addison-Wesley Publishing Co., Inc.,, 2005.

[5] 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.

[6] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Online manual, 2001.

[7] C. V. Lopes and T. Ngo. Unit testing aspectual behavior. In *Proc. AOSD 05 Workshop on Testing Aspect-Oriented Programs*, March 2005.

[8] R. Miles. *AspectJ Cookbook*. O'Reilly, 2004.

[9] M. Mortensen and R. T. Alexander. An approach for adequate testing of aspectj programs. In *Proc. AOSD 05 Workshop on Testing Aspect-Oriented Programs*, March 2005.

[10] M. Störzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *Proc. International Conference on Software Maintenance*, pages 653–656, 2005.

[11] Sun Microsystems. *Java Reflection API*. Online manual, 2001.

[12] Sun Microsystems. Java 2 platform standard edition v1.4.2 API specification, 2003. `http://java.sun.com/j2se/1.4.2/docs/api/`.

[13] The AspectJ Team. The AspectJ programming guide. Online manual, 2003.

[14] 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 (AOSD 2006)*, pages 190–201, March 2006.

[15] 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 (WTAOP)*, March 2005.

[16] 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.