# Efficient Mutant Generation for Mutation Testing of Pointcuts in Aspect-Oriented Programs

Prasanth Anbalagan[1] Tao Xie[2]
Department of Computer Science , North Carolina State University, Raleigh, NC 27695, USA
[1]panbala@ncsu.edu [2]xie@csc.ncsu.edu

## Abstract

*Fault-based testing is an approach where the designed test data is used to demonstrate the absence of a set of pre-specified faults, typically being frequently occurring faults. Mutation testing is a fault-based testing technique used to inject faults into an existing program, i.e., a variation of the original program and see if the test suite is sensitive enough to detect common faults. Aspect-Oriented Programming (AOP) provides new modularization of software systems by encapsulating crosscutting concerns. AspectJ, a language designed to support AOP uses abstractions like pointcuts, advice, and aspects to achieve AOP's primary functionality.*

*Developers tend to write pointcut expressions with incorrect strength, thereby selecting additional events than intended to or leaving out necessary events. This incorrect strength causes aspects, the set of crosscutting concerns, to fail. Hence there is a need to test the pointcuts for their strength. Mutation testing of pointcuts includes two steps: creating effective mutants (variations) of a pointcut expression and testing these mutants using the designed test data. The number of mutants for a pointcut expression is usually large due to the usage of wildcards. It is tedious to manually identify effective mutants that are of appropriate strength and resemble closely the original pointcut expression. Our framework automatically generates mutants for a pointcut expression and identifies mutants that resemble closely the original expression. Then the developers could use the test data for the woven classes against these mutants to perform mutation testing.*

## 1 Introduction

Aspect-Oriented Programming (AOP) [8] attempts to aid programmers in the separation of crosscutting concerns. AspectJ [7, 10], an AOP language uses abstractions like join points, pointcuts, advice, and aspects to aid AOP. Join points are well-defined locations like method calls and constructor invocations within the primary code. Pointcuts are predicates that match a join point. Pointcuts select a join point using signatures that include various designators, wildcards, and their combinations with logical operators. The strength of the pattern in the signature determines which join points are selected. If the pattern is too strong, some necessary join points are not selected. If the pattern is too weak, additional join points that should be ignored are selected. Hence there is a need to test the strength of a pointcut.

The fundamental premise of mutation testing [4] as stated by Geist et al. [9] is that, in practice, if the software contains a fault, there will usually be a set of mutants that can only be killed by a test that also detects that fault. In other words, the ability to detect small, minor faults such as mutants implies the ability to detect complex faults. Mutation testing measures how good our tests are by inserting faults into the program under test. Each fault results in a new program (called a mutant) that is slightly different from the original program. The idea is that the tests are adequate if they detect all mutants.

Mutation testing has many costs, including the possible generation of vast numbers of mutants. Another cost of mutation testing is the detection of equivalent mutants [12]. Equivalent mutants, by definition, are unkillable because the mutants are semantically equivalent to the original program. Detecting such mutants in software is generally intractable [5] and historically has been done by hand [12].

Performing mutation testing to test the strength of a pointcut requires generation of effective mutants, i.e., variations of the pointcut expression that resemble closely the original pointcut expression. In this paper, we propose a framework 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. Finally the framework reduces the total number of mutants from the initial large number of generated mutants.

Our framework 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 and their matched join points are then compared with those of the original pointcut and classified as different types of mutants for selection. When more than one mutant has the same set of join points, it is necessary to select the best mutant for that particular set of join points. The best one is selected using a simple heuristic (explained in Section 3). The classified mutants are ranked using a string similarity measure to help the developer choose a mutant that resembles closely original one. 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.

The rest of the paper is organized as follows. Section 2 presents an overview of the pointcuts and potential problems with generation of mutants for pointcuts. Section 3 illustrates our 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. Section 7 discusses related work and Section 8 concludes the paper.

## 2 Pointcuts in AspectJ

This section presents an overview of pointcuts in AspectJ and discusses potential problems of generating mutants for mutation testing of pointcuts. Pointcuts are predicates that match join points in the execution of a program. Pointcuts are modeled using expressions that identify the type, scope, or context of the events.

Figure 1 provides an example of pointcuts with the join point candidates, which are the probable join points being identified. `Pointcut 1` is used to match read accesses of of class `Blocks`' fields whose names start with "`current`" and whose type can be any. If the intention of the pointcut is to track current "X" and "Y" positions, then this expression would match additional join points like `get(int Blocks.currentBlock)`. `Pointcut2` matches calls of class `Blocks`' methods whose names start with "`typeTo`", whose argument is `int`, and whose return can be of any type. If the intention of the pointcut is to match all type conversion methods, it would leave out the method `typeToString(String)`. Hence it is necessary to test pointcuts for their strength.

Mutation testing of pointcuts requires generation of effective mutants for a pointcut. For example, consider the mutant `get(* Blocks.*)`. This mutant is too weak since it matches all variables belonging to class `Blocks`. Due to the usage of wildcards in pointcut expressions, the number of mutants that can be formed is very large. But it is necessary to generate mutants that are of appropriate strengths

```
Pointcut 1
  pointcut xyposition() : get(* Blocks.current*);
Matched join points
  get(static int Blocks.currentXPos);
  get(static int Blocks.currentYPos);
  get(static int Blocks.currentBlock);

Pointcut 2
  pointcut Types() : call(* Blocks.typeTo*(int));
Matched join points
  call(Color Blocks.typeToColor(int));
  call(Image Blocks.typeToImage(int));
Unmatched join point
  call(String Blocks.typeToString(String));
```

**Figure 1. Sample pointcuts and join point candidates**

and resemble closely the original expression in order to perform effective mutation testing on pointcuts.

Our preliminary application of the framework on Tetris (an AspectJ Benchmark [2]) produced 1455 mutants for a pointcut expression. Performing mutation testing on such a large number of mutants is practically not efficient. Developers would prefer to select those expressions that resemble closely the original one and use them for the mutation testing. But it is tedious to manually identify such mutants.

## 3 Approach

To reduce human efforts in mutation testing of pointcuts, we develop a framework to generate only relevant mutants rather than arbitrary strings. The relevant mutants are selected from the initial large set of mutants by classifying them and also by detecting equivalent mutants. These relevant mutants resemble closely the original pointcut expression.Our framework is based on a candidate fault model for pointcuts proposed by Alexander et al. [11] as well as AJTE [18], an existing unit-testing framework without weaving. Figure 2 provides an overview of our framework. In particular, the source files (aspects and target classes under test) are given as input to three components in the framework: the candidate generator, pointcut parser, and AJTE.

AJTE outputs a test class that consists of methods for testing the join point candidates generated from the candidate generator. In particular, AJTE provides APIs for creating join point objects, which are in turn used as arguments to pointcut-checking methods. The `TestJoinPoint` class of AJTE is the class for processing a join point as an object. The `TestPointcut` class is the class for processing a pointcut expression as an object.

Our framework automatically identifies likely join points from the Java class file. These likely join points form the join point candidates used to classify the pointcuts. Once the join point candidates are identified, they are fed as input to the `TestJoinPoint` factory class of AJTE to produce

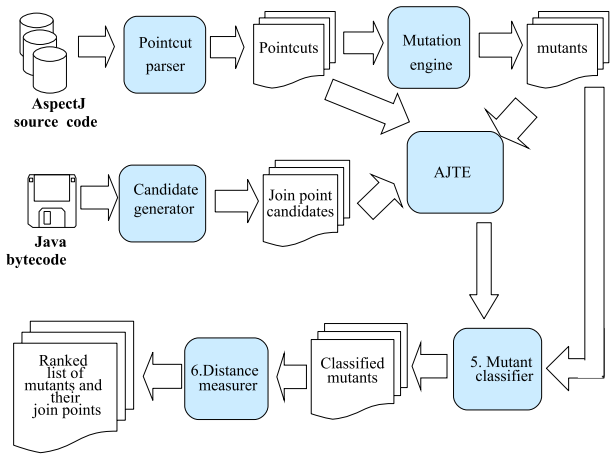


**Figure 2. Overview of the framework**

join point objects. These join point objects are then fed as arguments to the `TestPointcut` factory class of AJTE to verify whether the join point objects match the pointcut expression. This step is performed on all join point candidates.

The second set of input includes the pointcut expressions from the AspectJ source file. We parse the AspectJ source code to identify the pointcut expressions based on the keyword `pointcut` used to define pointcuts. For primitive and anonymous pointcuts where keywords are not used, the pointcuts are identified by the presence of designator types.

The mutation engine forms pointcut mutants based on two mutation operators (*pointcut strengthening* and *pointcut weakening*) in a candidate fault model [11] for pointcuts. The idea behind the two mutation operators is to reduce or increase the number of join points that a pointcut matches, by creating mutants of the original pointcut. Initially the mutation engine forms mutants for different naming parts of the original pointcut, and the matched and unmatched join point candidates. The pointcut mutants are formed by performing all possible combinations of all mutants of the naming parts.

We develop two approaches to form mutants. First, the naming parts are considered as a whole and positions of wildcards within the naming parts are modified to form mutants. Second, the naming parts are split into portions and the wildcards are substituted for each portion. In this latter approach, positions of wildcards are varied to form different combinations of the split portions and wildcards.

In the first approach, there are three techniques. First, a wildcard is inserted at the end of the naming part. Then the wildcard is moved from the right end to the left end. As the wildcard is moved towards left, it replaces each character in the naming part. For example, consider the join point `get(static int Blocks.curretnXPos)`. The mutants formed for the naming part `Blocks` are `Blocks*`, `Block*`, `Bloc*`, `Blo*`, `Bl*`, and `B*`. Second, the wildcard is inserted at the beginning of the naming part. Then the wildcard is moved from the left end to the right end. As the wildcard is moved right, it replaces each character in the naming part. The mutants formed for the naming part `Blocks` by this technique are `*Blocks`, `*locks`, `*ocks`, `*cks`, `*ks`, and `*s`. Third, the mutants are formed by placing the wildcard in between the beginning and end of the naming part and varying the position of the wildcard. Although this technique does not enumerate all possible mutants with a wildcard in the middle (doing so would cause combinatorial explosion), mutants generated by this technique favors putting the wildcard around or in the middle of the naming part, complementing the first two techniques. The mutants formed for `Blocks` in this fashion are `B*s`, `Bl*s`, and `Bl*ks`.

The second approach is to split the naming part into portions so that only the first character in each portion can be in uppercase (except for the first portion where all characters could be in lowercase)[1]. The approach then substitutes each portion with a wildcard. The portion that the wildcard replaces is varied each time to generate a different mutant; multiple wildcards can appear in the resulting mutants. For example, consider the naming part `typeToString` of the join point `get(String Blocks.typeToString(String))`. The `typeToString` method is split into three portions: `type`, `To`, and `String`. The mutants formed for `typeToString` using the second approach are `typeTo*`, `type*String`, `*ToString`, `*To*`, `type*`, and `*String`.

In the case of naming parts like modifiers and return types, mutants are formed only by just replacing an entire modifier or return type with the wildcard. In the case of arguments, AspectJ allows a special type of wildcard "..". The wildcard ".." denotes any number of any type of arguments. Similar to forming mutants for other naming parts with wildcards, we form mutants for arguments with ".." . For example, consider the arguments "`(int,float,String)`". The mutants formed for this argument will be `(int,float,..)`, `(int,..,String)`, `(..,float,String)`, `(..,float,..)`, `(int,..)`, `(..,String)`, and `(..)`.

Pointcut mutants are also generated from the original pointcut in a similar way as from the join point candidates. The two approaches for generating pointcut mutants from join point candidates are applied to the naming parts of the original pointcut. Unlike join point candidates, the original pointcut might include wildcards. All naming parts of the original pointcut except for wildcards are changed to form mutants. It is possible that the original pointcut includes

[1]Based on common Java naming conventions, each portion would represent a meaningful word and pointcuts written by developers are usually formed by replacing these words with wildcards.

only wildcards. In such a case, pointcut mutants are formed only from the join point candidates.

The mutants and the original pointcut expression are verified using the test class from AJTE. Then the mutants, original pointcut, and their respective join points are fed as input to the mutant classifier in order to classify the mutants. Mutants with the same set of join points as that of the original expression are classified as *neutral*. The neutral mutants here are equivalent mutants. If the set of join points of the original expression is a subset of the set of join points of the mutant, then the mutant is classified as *weak*; otherwise, it is classified as *strong*. A weak mutant matches more events compared to the original pointcut expression and a strong mutant does not match all events matched by the original pointcut expression.

Under each category, there maybe more than a mutant with the same set of join points, then the mutant with the longest expression (in string length) is chosen as the best one. The longest expression would consist of a larger number of characters and more characters indicate that the pointcut is closer to the join point. A smaller number of characters indicate that the pointcut is generalized and is more susceptible to fragile pointcut problems [14]. Hence we select the longest pointcut to avoid potential fragile pointcut problems. Our framework excludes wildcards like "*" and "..", supported by AspectJ, when calculating the length of each expression for finding out the longest expression, because including these wildcards could produce longer expressions, which however are more general.

The distance measurer ranks the mutants in each category based on the Monge Elkan distance [13]. The distance measure is an integer value that indicates the number of transformations (i.e., insertions and deletions) that should be performed on the original expression to transform it into the mutant. Selecting a better match among a set of mutants with the same join points and ranking these mutants finally yield mutants that resemble closely the original pointcut expression. This mechanism also reduces the total number of mutants that finally form the output. The developers could use designed test data (for the woven classes) along with these mutants to perform mutation testing of pointcuts.

## 4 Implementation

We have implemented the framework for AspectJ and Java code using the Byte Code Engineering Library (BCEL) [3], Java reflection APIs [15], and AJTE. The current implementation of the framework supports an AspectJ compiler called ajc [6] Version 1.5 and Java 5 [16]. The main components of the framework include the AJTE, pointcut generator, candidate generator, mutation engine, mutant classifier, and distance measurer, as shown in Figure 2.

AJTE provides two classes: `TestJoinPoint` class for processing a join point as an object and `TestPointcut` 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 pointcuts, the mutants, and the join points to the two classes.

The pointcut parser parses the given AspectJ source code to identify pointcut expressions. The generated pointcuts are passed as input to the mutation engine to generate mutants of this pointcut expression. The pointcut expression also serves as input to the distance measurer to measure the distance of the pointcut expression from its mutants.

The candidate generator receives the given Java class files as input. This component uses the Java reflection APIs [15] to generate join point candidates from the input files. Then the framework feeds these joinpoint candidates to the `TestJoinPoint` factory class to generate joinpoint objects.

The mutation engine produces mutants only from meaningful data obtained from within the Java code. For example, to form mutants of an expression that involves a method name, other method names from the code are used rather than arbitrary strings. This technique allows filtering out expressions with arbitrary strings that match no join point.

Each pointcut expression is checked against the join point objects using the `TestPointcut` factory of AJTE to identify the set of joinpoints (say Set A) that are matched by the pointcut expression. Similarly each mutant of the pointcut is checked to identify their set of join points (say Set B). The two sets are then compared as follows: If Set A is equal to Set B, then the mutant is classified as *neutral*. If Set A is a subset of Set B, then the mutant is classified as *weak* since the mutant matches more join points than the original pointcut; otherwise, the mutant is classified as *strong* since the mutant does not match all join points matched by the original pointcut. If under any category, there are more than one expression with the same set of join points, the longest expression is chosen as the best match for that set of join points. The classified mutants are then fed as input to the distance measurer.

The distance measurer uses the Monge Elkan distance measure as the measure to find the deviation of the mutant from the pointcut expression. The distance denotes the number of characters that need to be inserted, deleted, or modified in the pointcut expression to transform it into the mutant. The mutants under each category are then ranked based on the calculated distance. The classification of the mutants helps the developers to identify mutants with different strengths and the ranking of the mutants helps to identify mutants of the pointcut expression that resemble closely the pointcut expression. If the developers wish to select only

**Table 1. Preliminary Results**

| Pointcut : call(String Blocks.typeToString(int)); | | | |
|---|---|---|---|
| Mutant Type | Mutant | Joinpoints | Distance |
| Strong | None | None | None |
| Weak | call(* Blocks.typeTo*(int,..)) | call(public static String Blocks.typeToString(int)) | 12 |
| | | call(public static Color Blocks.typeToColor(int)) | |
| Weak | call(* Blocks.t*(int,..)) | call(public int[][] Blocks.turnBlock(int[][])) | 15 |
| | | call(public static String Blocks.typeToString(int)) | |
| | | call(public static Color Blocks.typeToColor(int)) | |
| Weak | call(* Blocks.*(..)) | call(public static String Blocks.typeToString(int)) | 19 |
| | | call(public int[][] Blocks.turnBlock(int[][])) | |
| | | call(public void Blocks.deleteLine(int,int[][])) | |
| | | call(public static int[][] Blocks.getBlock(int)) | |
| | | call(public static Color Blocks.typeToColor(int)) | |
| Neutral | call(public static String Blocks.*typeToString(int,..)) | call(public static String Blocks.typeToString(int)) | 4 |

a few mutants from our output, the ranking would help in choosing the best mutants with the shortest distances from the original expression.

## 5 Preliminary Results

We have implemented the approach and performed preliminary experiments on a few sample sets from an AspectJ benchmark called Tetris [2]. Table 1 shows an example of an original pointcut and its mutants. The mutants have been classified as strong, neutral, and weak, and ranked using the Monge Elkan similarity measure. For a pointcut expression from the Tetris benchmark, our approach identified 1455 mutants. Classifying these mutants and selecting the best match for mutants with the same set of join points reduced the number to 9 weak mutants and 6 neutral mutants. Manually identifying these mutants is tedious and the ranked list along with their join point sets provided by our framework helps the developers to identify quickly the mutants that resemble closely the original one.

## 6 Discussion

In our current implementation, the join points are formed based on static analysis of the code and currently does not support forming mutants in the presence of dynamic context like `cflow`. The distance measure provided by the distance measurer is the syntactical difference between the pointcut expression and its mutant. But it is likely that pointcuts that differ a lot syntactically may have similar sets of join points. To address such cases, we displayed the mutants along with their join point sets, which help the developers select pointcuts based on the set of join points they had actually intended to. This technique helps select appropriate mutants when the pointcut and mutants differ by a large measure syntactically but still resemble closely in their sets of join points. The current implementation supports generating mutants for performing mutation testing on pointcuts in aspect-oriented programs. In future work, we plan to extend the framework to generate test data (for woven classes) that could be used to kill these mutants.

## 7 Related Work

An approach developed by Mortensen and Alexander [11] 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 account for selection of effective mutants. Our approach provides an automated framework to generate mutants and select mutants that resemble closely the original expression; the selected mutants could be used with a test suite to perform mutation testing.

Yamazaki et al. [18] present the AJTE framework for unit testing aspects without weaving. This framework generates testing methods from an aspect definition so that test cases can directly verify properties of aspects such as the advice behavior and pointcut matching. This framework provides only APIs to be used for testing pointcuts but it does not help to generate mutants of a pointcut. Our framework automatically generates mutants for a pointcut expression that can be used in mutation testing.

Our previous work [1] presents an automated framework that tests pointcuts in AspectJ programs with the help of AJTE. This framework identifies joinpoints that are matched by a pointcut expression and a set of boundary joinpoints, which are events that are not matched by a pointcut expression but are close to the matched joinpoints in terms of their string names. The developers can inspect this set of join points and verify the correctness of a pointcut.

This framework helps only to verify the correctness of a pointcut and does not generate mutants of a pointcut. Our framework helps generate efficient mutants that resemble closely the original pointcut expression.

Xie and Zhao [17] present an automated tool that leverages existing test-generation tools to generate test inputs for the woven classes; these test inputs indirectly exercise the aspects. Their tool can be combined with our framework to form a complete mutation testing system for testing pointcuts in aspect-oriented programs.

## 8 Conclusion

We have developed a framework that automatically generates mutants for a pointcut expression and identifies mutants that resemble closely the original expression. This framework generates only relevant mutants rather than arbitrary strings. It also classifies the mutants and detects equivalent mutants. This framework eases developers' efforts of generating effective mutants and identifying equivalent mutants. Our preliminary results show that the framework is promising since manually identifying such mutants is tedious. Automatically generating the mutants and ranking them help the developers save manual efforts in identifying equivalent mutants and generating efficient mutants rather than a potentially large number of mutants for a pointcut expression.

## References


[1] 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.

[2] AspectJ benchmark, 2004. `http://www.sable.mcgill.ca/benchmarks/`.

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

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[5] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, 1991.

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

[7] Eclipse. *The AspectJT M 5 Development Kit Developer's Notebook*. Online manual, April 2005.

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

[9] R. Geist, A. J. Offutt, and F. Harris. Estimation and enhancement of real-time software reliability through mutation analysis. *IEEE Transactions on Computers*, 41(5):55–558, 1992.

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

[11] M. Mortensen and R. T. Alexander. An approach for adequate testing of AspectJ programs. In *Proc. 1st Workshop on Testing Aspect-Oriented Programs*, March 2005.

[12] J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 45–55, October 2000.

[13] Simmetrics, 2004. `http://www.dcs.shef.ac.uk/~sam/simmetrics.html`.

[14] 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.

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

[16] Sun Microsystems. Java 2 platform standard edition v1.5.0 API specification, 2004. `http://java.sun.com/j2se/1.5.0/docs/api/`.

[17] 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.

[18] 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.