Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs

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

Abstract

Aspect-Oriented Programming (AOP) provides new modularization of software systems by encapsulating crosscutting concerns. AspectJ, an AOP language, uses abstractions such as pointcuts, advice, and aspects to achieve AOP's primary functionality. Faults in pointcuts can cause aspects to fail to satisfy their requirements. Hence, testing pointcuts is necessary in order to ensure correctness of aspects. In mutation testing of pointcuts (a type of fault-based pointcut testing), the number of mutants (i.e., variations) for pointcuts 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, reflecting the kind of mistakes that developers may make. To reduce developers' effort in this process, we have developed a new framework that automatically identifies the strength of each pointcut and generates pointcut mutants with different strengths. Developers can inspect the pointcut mutants and their join points for pointcut correctness or choose the mutants for conducting mutation testing. We conducted an empirical study on applying our framework on pointcuts from existing AspectJ programs. The results show that our framework can provide valuable assistance in generating effective mutants that are close to the original pointcuts and are of appropriate strength.

1 Introduction

Aspect-Oriented Programming (AOP) [8] provides modular implementation for crosscutting behavior of a software system. AspectJ [9] is a widely used AOP language and provides special constructs such as aspects, advice, join points, and pointcuts. AspectJ uses *join points* to represent locations where a concern crosscuts a software system. *Pointcuts* are constructs modeled using pointcut expressions to capture join points. *Advice* contains the set of actions that has to be applied at the identified join points. The type of advice determines when the actions are to be applied when a join point is triggered.

Pointcuts identify the type, scope, or context of a join point where the crosscutting behavior applies. AspectJ allows pointcuts to be designed with various designators, wildcards, and their combinations with logical operators. A pointcut consists of expressions that define the pattern of the join points to be matched. Based on the pattern written by a developer, the pointcut selects an appropriate set of join points. It is likely that the pattern written by the developer selects unintended join points or leaves out intended join points. When unintended join points are selected, the crosscutting behavior is applied at these unintended join points. Application of the crosscutting behavior at unintended join points would lead to erroneous behavior of the system. When intended join points are left out, the crosscutting behavior is not applied at these left-out intended join points. In this case, the system fails to perform its complete functionality. In either case, the system fails to perform its intended functionality. Hence, there is a strong need to test the strength of a pointcut.

The fundamental premise of mutation testing [4, 7] (a type of fault-based testing) is that, in practice, if a program contains a fault, there usually will be a set of mutants that can be killed by only a test that also detects that fault. Mutation testing measures how good our tests are by seeding 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 faults seeded in all or most mutants.

Mutation testing has many costs, including the generation of a possibly vast number of mutants and the detection of equivalent mutants [12]. Equivalent mutants, by definition, are unkillable because these mutants are semantically equivalent to the original program. Detecting such mutants for a program is generally intractable [5] and has historically been done by hand [12].

Performing mutation testing to test the strength of a pointcut requires generation of effective mutants, i.e., the pointcut's variations that resemble closely the original

pointcut, reflecting the kind of mistakes that developers may make. In this paper, we propose a framework[1] 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 large number of initial generated mutants. Our framework also classifies the mutants and ranks them using a string similarity measure to help the developer choose a mutant that resembles closely the original one. Developers can inspect the output of the framework for pointcut correctness or use these mutants along with test cases (for the woven code produced by aspect weaving) to conduct mutation testing. In the mutation testing, the test cases are executed on the woven code with a pointcut mutant, and if at least one test case fails, the mutant is killed (i.e., the seeded fault is detected).

We conducted an empirical study on applying our framework on pointcuts from existing AspectJ programs. The results show that our framework can provide valuable assistance in generating effective mutants that are close to the original pointcuts and are of appropriate strength.

The rest of the paper is organized as follows. Section 2 presents our illustrative example. Section 3 illustrates our framework. Section 4 provides the results of applying the framework on selected subjects. Section 5 discusses issues of our approach. Section 6 presents related work, and Section 7 concludes with future work.

2 Example

In this section, we use an example to illustrate pointcuts and potential issues with the strength of a pointcut. Figure 1 provides a partial implementation of an `Account` class. The `Account` class consists of two crosscutting concerns: `AccessController.checkPermission(new BankingPermission("accountOperation"))` and `Account.updateAccount(amount)`. Figure 2 shows an `aspectAccount` aspect for the `Account` class. The `aspectAccount` aspect consists of the pointcuts `checkPermission()` and `updateAccount()`. Figure 2 shows the pointcuts of the `aspectAccount` aspect along with their matched join points[2], which are the join points identified from the `Account` class where the crosscutting concerns occur.

[1] An earlier version of this work is described in a workshop position paper presented at MUTATION 2006 [2].

[2] For simplicity of illustration, we omit the class name in each join point; the class name in each join point is "`Account`".

```
public class Account {
   ...
   public int getAccountNumber() {
        AccessController.checkPermission(
            new BankingPermission("accountOperation"));
   }
   public void accessCreditAccount(float amount) {
        Account.updateAccount(amount);
   }
   public void accessDebitAccount(float amount) {
        Account.updateAccount(amount);
   }
   public void createNewAccount(float amount) {
        Account.updateAccount(amount);
   }
   public float getAccountBalance() {
        AccessController.checkPermission(
             new BankingPermission("accountOperation"));
   }
   public float getGeneralInfo() { ... }
}
```

Figure 1. Partial implementation of an `Account` class

`Pointcut1` is used to match join points for the crosscutting concern `AccessController.checkPermission(new BankingPermission("accountOperation"))`. The pointcut has been designed to match the execution of all methods (belonging to the `Account` class) whose names start with "`get`" and have any number of arguments. The intention of the pointcut is to match all methods that have the crosscutting concern `AccessController.checkPermission(new BankingPermission("accountOperation"))`. But from the list of the join point candidates, we find that the pointcut matches an additional method `public void getGeneralInfo()`. In such a case, the crosscutting behavior `AccessController.checkPermission(new BankingPermission("accountOperation"))` is applied before the execution of this method. The actual purpose of the method `getGeneralInfo()` is to obtain general information about operations on a bank account. This functionality does not require any permission check because it is accessible to all users. Applying the crosscutting behavior before the execution of this method makes it unaccessible to general users, causing erroneous behavior. In this case, we call the strength of `Pointcut1` to be too *weak* (i.e., too general), and it matches more join points than expected.

`Pointcut2` is used to match join points for the crosscutting concern `Account.updateAccount()`. The pointcut matches the execution of methods (in the `Account` class) whose names start with "`access`", end with "`Account`", and have "`float`" as the argument. The intention of the pointcut is to match all methods that have the crosscutting concern `Account.updateAccount()`. But

```
public aspect accountAspect {

  pointcut checkPermission(): execution(* Account.get*(..))
     before checkPermission() {
        AccessController.checkPermission(
          new BankingPermission("accountOperation"));
  }

  pointcut updateAccount():
    execution(public void Account.access*Account(float))
      after updateAccount() {
        Account.updateAccount(amount);
  }
}

Pointcut1
  pointcut checkPermission(): execution(* Account.get*(..))
Join points
  execution(public int getAccountNumber());
  execution(public float getAccountBalance());
  execution(public void getGeneralInfo());
Pointcut2
  pointcut updateAccount():
    execution(public void Account.access*Account(float))
Join points
  execution(public void accessCreditAccount(float amount));
  execution(public void accessDebitAccount(float amount));
```

Figure 2. `Account` **aspect with pointcuts and their matched join points**

from the list of the matched join points, we can find that the pointcut leaves out the method `public void createNewAccount(float amount)`. In such a case, the crosscutting behavior `Account.updateAccount()` is not applied after the execution of this method. The actual purpose of the method `createNewAccount` is to create a new account. When a new account is created, the account is updated with an initial amount deposited by the user. If the crosscutting behavior is not applied after the execution of this method, then the account would not be updated with the initial amount deposited by the user and the functionality of the method would be incomplete. In this case, we call the strength of `Pointcut2` to be too *strong* (i.e., too specific), and it matches fewer join points than expected.

In either of the preceding cases, the incorrect strength of pointcuts causes incorrect behavior of the woven target code. Hence it is necessary for developers to verify the strength of the original pointcuts and identify their mutants with different strengths. One straightforward tool support is to automatically identify a list of join points matched by a pointcut and then developers can inspect and verify this list. However, for a large system, it is tedious to inspect and verify the identified list. Moreover, inspecting and verifying only the matched list of join points may not easily help discover intended join points not matched by the pointcut.

Given a software system and a pointcut defined for this system, our framework automatically generates a small number of pointcut mutants, which are similar to the given pointcut in terms of its syntactic form or in terms of the set of matched join points. For each pointcut mutant, the framework also automatically produces the differences between the set of join points matched by the pointcut mutant and the given pointcut; these differentiating join points can be viewed as test inputs (for the pointcut) that kill the pointcut mutant. Our framework proposes to use these pointcut mutants and their differentiating join points to help detect pointcut faults; such an application of mutation testing is different from traditional applications of mutation testing such as assessment of a test suite's fault-detection capability or quality in general. In particular, developers can inspect and verify pointcut mutants and the differences of their matched join points from the ones matched by the given pointcut (rather than inspecting the join points matched by the given pointcut).

For example, for `Pointcut1`, our framework automatically generates one pointcut mutant and the join point differences as below:

```
Pointcut mutant:
  pointcut checkPermission():
                execution(*Account.getAccount*(..))
Join point differences (-1)
  -execution(public void getGeneralInfo());
```

The "(-1)" after "Join point differences" indicates that one join point in the set of join points matched by `Pointcut1` is not matched by the pointcut mutant. This number "1" is the value for *downsize measure*, which indicates the number of join points matched by the original pointcut but not matched by the pointcut mutant. The "–" symbol before the listed join point indicates that the join point is originally matched by the given pointcut but not by the pointcut mutant.

By inspecting this pointcut mutant and the join point differences, developers would think about whether this `getGeneralInfo()` method should be matched. They could find out that this method should not be matched and then replace the pointcut expression in `Pointcut1` with the pointcut mutant, which successfully excludes the `getGeneralInfo()` method.

For `Pointcut2`, our framework automatically generates one pointcut mutant and the join point differences as below:

```
Pointcut mutant:
  pointcut updateAccount():
        execution(public void Account.*Account(float))
Join point differences (+1)
  +execution(public void createNewAccount(float amount));
```

The "(+1)" after "Join point differences" indicates that one join point in the set of join points matched by the pointcut mutant is not matched by `Pointcut1`. This number "1" is the value for *upsize measure*, which indicates the number of join points matched by the pointcut mutant but not matched by the original pointcut. The "+" symbol before the listed join point indicates that the join point is matched by the pointcut mutant but originally not matched by the given pointcut. Note that when the pointcut mutant matches additional join points (not matched by the original pointcut) and does not match some join points matched by the original pointcut, there will be two numbers after "Join point

differences" (one prefixed by "+" and the other prefixed by "-").

By inspecting this pointcut mutant and the join point differences, developers would think about whether this `createNewAccount(float amount)` method should be matched. They could find out that this method should be matched and then replace the pointcut expression in `Pointcut2` with the pointcut mutant, which successfully matches the `createNewAccount(float amount)` method.

The preceding two example pointcut mutants show that pointcut mutants with different strengths (from their original pointcut) can help developers detect pointcut faults by using the mutants. We next illustrate on the concept of equivalent mutants.

For illustration purposes, let us now pretend that the `getGeneralInfo()` method does not exist when developers verify the pointcuts in the system. Then for `Pointcut1`, our framework automatically generates one pointcut mutant and the join point differences as below:

```
Pointcut mutant:
  pointcut checkPermission():
                execution(*Account.getAccount*(..))
Join point differences (0)
```

Note that either the upsize or downsize measure for this pointcut mutant is 0, which indicates that the set of join points matched by the pointcut mutant is the same as the one matched by the original pointcut. Pointcut mutants with measure 0 are thus identified as equivalent mutants.

3 Framework

To reduce manual effort in identifying the strength of a pointcut and generating pointcut mutants that resemble closely the original pointcut, we develop an automated framework to identify the strength of a pointcut, and generate pointcut mutants that match more join points, fewer join points, or the same set of join points matched by the original pointcut. The input to our framework is AspectJ source code and Java bytecode of the base program (the software system that the aspects are woven into). The output from our framework is a ranked list of pointcut mutants for each original pointcut in the AspectJ source code and the differences of the join points matched by the original pointcut and the pointcut mutants.

In our framework, we identify pointcuts from AspectJ source code and join point candidates from the Java bytecode of the base program; join point candidates are program locations that can potentially be join points. The join point candidates are identified based on static analysis of the bytecode. We verify join point candidates against the pointcuts and identify the join points matched by each pointcut. Then we generate pointcut mutants for the original pointcut based on a candidate fault model [3] for pointcuts. The

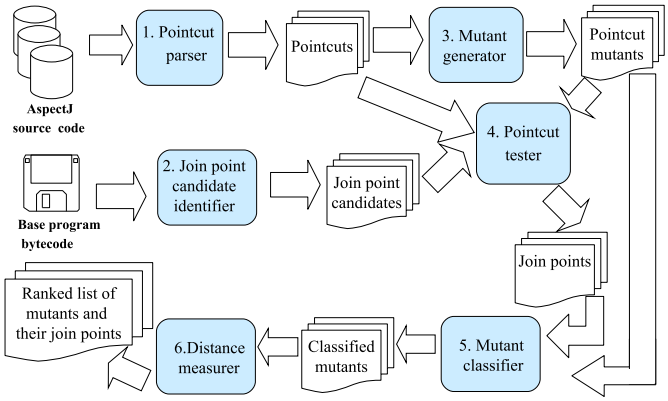

Figure 3. Framework overview

join points for the mutants are identified in a similar way as for the original pointcut. To facilitate developer inspection, we classify and rank pointcut mutants based on a similarity measure with the original pointcut. This ranking mechanism helps developers to quickly identify pointcut mutants that resemble closely the original pointcut.

Figure 3 provides an overview of our framework. The framework consists of six components: pointcut parser, join point candidate identifier, mutant generator, pointcut tester, mutant classifier, and distance measurer. We next illustrate the details of each of these six components.

3.1 Pointcut Parser

The pointcut parser identifies pointcuts in the given AspectJ source code. Pointcuts are specified by a structure that consists of access type, name, designator, identifier, and type. The access type and name specify the scope of the pointcut in the aspect class and name of the pointcut. The designator and type pattern define the pattern of the join points to be matched. Each pointcut consists of one or more pointcut expressions separated by logical operators. The pointcut parser performs several functionalities: identifying the pointcuts in the source code, pointcut expressions in each pointcut, and the different structural components of each pointcut expression in a pointcut.

Figure 2 shows the `aspectAccount` aspect with the pointcuts `checkPermission` and `updateAccount`. In this example, the pointcuts have been defined using the keyword `pointcut`. Both the pointcuts `checkPermission` and `updateAccount` consist of a single pointcut expression, the designator `execution`, and the methods to be matched. The methods to be matched are specified with the patterns "`* Account.get*(..)`" and "`public void Account.access*Account(float)`". The different structural parts of the two patterns are the modifiers ("`*, public`"), the return type ("`*, void`"), the class to which the methods belong to ("`Account`"), the pattern for the method names ("`get*, access*Account`"), and the arguments ("`(..), (float)`"). The identified pointcuts

```
execution(public int Account.getAccountNumber())
execution(public void Account.accessCreditAccount(
                                    float amount))
execution(public void Account.accessDebitAccount(
                                    float amount))
execution(public void Account.createNewAccount(
                                    float amount))
execution(public float Account.getAccountBalance())
execution(public float Account.getGeneralInfo())
```

Figure 4. Join point candidates for the `Account` class

and the structural parts of each pointcut expression are fed as input to the mutant generator.

3.2 Joint Point Candidate Identifier

The joint point candidate identifier identifies the join point candidates from the the given Java bytecode for the base program. The functionality of the candidate identifier is to identify all well-defined execution points (in the Java source code) by examining the Java bytecode. Our framework currently does not support identifying join point candidates related to dynamic contexts like `cflow`; therefore, our framework currently does not provide support to pointcuts related to dynamic contexts. In our framework, we use the pointcut tester (Section 3.4) to verify the join point candidates against the pointcuts. The pointcut tester requires certain test inputs to perform testing without weaving the aspects to the base program. In order to test pointcuts, the test inputs would be join point candidates and the pointcut tester would identify the join points for a pointcut from the list of join point candidates.

The candidate identifier scans through the bytecode to locate all identifiable points in the execution of a program. Identifiable points could be a method call or execution, constructor call or execution, exception handler execution, read or write access to a variable, and class or object initialization. Then we collect details that belong to these execution points. The details include the different naming parts of the candidate. For example, in the case of a method call or execution, the details include the different naming parts of the method, i.e., the method name, modifiers, return type, arguments, the class to which the method belongs, and the class where the method is invoked (if the candidate is a method call).

Figure 1 shows the `Account` class. The `Account` class consists of several methods. The candidate identifier scans the bytecode of the `Account` class, identifies the methods, and collects the details of each method. The naming parts of each method are combined along with the designator to form a join point candidate. Figure 4 shows the join point candidates generated from the `Account` class.

3.3 Mutant Generator

The mutant generator forms mutants for the pointcuts identified by the pointcut parser. Pointcut mutants are formed based on two mutation operators (*pointcut strengthening* and *pointcut weakening*). The idea behind the mutation operators is to reduce or increase the number of join points that a pointcut matches, by creating variations of the original pointcut. The mutant generator generates pointcut mutants by performing lexical variations of the original pointcut, the join point candidates that the original pointcut matches, and the unmatched join point candidates.

We next describe how we generate mutants for the original pointcut and matched or unmatched join point candidates (Section 3.3.1). The number of join point candidates could be large for large systems, and varying all these join point candidates is infeasible. Considering that only a small subset of mutants generated from join point candidates may be relevant, with short-distance measures from the original pointcut, we select a subset of join point candidates for variations based on heuristics (Section 3.3.2).

3.3.1 Mutation of Pointcut and Join Point Candidates

Initially the mutant generator forms mutants for different naming parts of the original pointcut and join point candidates such as join points matched by the original pointcut and the join point candidates unmatched by the original pointcut. The mutant generator then forms pointcut mutants by combining mutants of different naming parts of the pointcut.

Pointcuts allow the usage of wildcards within different naming parts of an expression. Based on this characteristic of pointcuts, we develop two techniques to form mutants. In the first technique, there are three ways of inserting wildcards. 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 starting with placing the wildcard in the middle of the naming part while keeping only the leftmost and rightmost characters, and then keeping adding characters to either end incrementally one at a time. For example, the mutants formed for `Blocks` in this fashion are `B*s`, `Bl*s`, and `Bl*ks`. Although this way does not enumerate all possible mutants with a wildcard in the middle (doing so would cause combinatorial explosion), mutants generated by this way favors putting the wildcard around

the middle of the naming part, complementing the first two ways.

The second technique 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)[3]. The technique then substitutes each portion with a wildcard. During each time of generating a different mutant, one or more different portions are selected to be replaced with the wildcard; 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 technique are `typeTo*`, `type*String`, `*ToString`, `*To*`, `type*`, and `*String`.

In the case of naming parts such as 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 `(..)`.

The pointcut mutants are also generated from the original pointcut in a similar way as from the join point candidates. The two techniques 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 only wildcards. In such a case, pointcut mutants are formed only from the join point candidates.

3.3.2 Selection of Join Point Candidates for Mutation

Apart from generating pointcut mutants from the original pointcut with the insertion of wildcards, our framework generates mutants from matched and unmatched join points of the original pointcut. Mutants generated from matched join points have a higher probability of resembling the original pointcut because their original join points are already matched by the original pointcut. But mutants that are generated from unmatched join points are less likely to resemble the original pointcut because their original join points do not match the original pointcut. Because the usage of wildcards in forming pointcut mutants can generate a large number of mutants, it is desirable to reduce the space of unmatched join points used for forming mutants. In our framework, unmatched join points are compared to the original pointcut based on a string similarity measure. Only those unmatched join points that fall within a specific threshold are selected for forming mutants. The threshold is defined by the developers and is a numerical value that indicates the number of characters that differ between an unmatched join point and the original pointcut.

[3]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.

3.4 Pointcut Tester

The pointcut tester verifies the join point candidates identified by the candidate identifier against a pointcut identified by the pointcut parser. In general, the pointcut tester, developed based on an AspectJ unit testing framework [16], can be used to verify pointcuts of an aspect class without weaving the aspect code to the base program. Our pointcut tester automatically feeds the unit testing framework with input including the join point candidates generated by the candidate identifier and the original pointcut identified by the pointcut parser. Our pointcut tester verifies each join point candidate against a pointcut (the original pointcut or pointcut mutant). Based on the result, the matched and unmatched join points are collected separately for the pointcut. The matched and unmatched join points are fed to the mutant classifier for classifying pointcut mutants.

3.5 Mutant Classifier

The functionality of the mutant classifier is to identify the type of a pointcut mutant in comparison with the original pointcut. *Pointcut strengthening* is to increase the strength of a pointcut by reducing the set of join points that it matches. Here the resulting pointcut mutant can be called as a *strong* pointcut mutant. *Pointcut weakening* is to decrease the strength of a pointcut by expanding the set of join points that it matches. Here the resulting pointcut can be called as a *weak* pointcut mutant. We also introduce another type of pointcut mutant called *neutral* pointcut mutants, whose matched join points are the same as those matched by the original pointcut. Neutral pointcut mutants are especially useful for refactoring pointcuts, whereas strong or weak pointcut mutants are especially useful for detecting and correcting pointcuts with incorrect strengths.

Note that a pointcut mutant may match only a subset of the join points matched by the original pointcut and match additional join points not matched by the original pointcut. This type of pointcut mutant is still considered by the mutant classifier as strong pointcut mutants, because this type of pointcut mutants does not cover all the join points of the

original one, i.e., fewer join points from the perspective of the original pointcut. For simplicity, we do not create a new type specifically to characterize this type of pointcut mutants but treat them as strong pointcut mutants, without affecting the overall effectiveness of pointcut testing.

The original pointcut, its join points, the pointcut mutants, and their join points are fed as input to the mutant classifier. Let Set O be the set of join points for the original pointcut and Set M be the set of join points for a pointcut mutant. The two sets are then compared as follows.

- If Set O is a subset of Set M, which means that the pointcut mutant matches more join points than the original pointcut, the mutant is termed as *weak* pointcut mutant.
- If set O and set M are equal, which means the original pointcut and the mutant match the same set of join points, the pointcut mutant is termed as *neutral* mutant.
- Otherwise, the mutant is termed as *strong* pointcut mutant.

Note that more than one pointcut mutant may share the same set of matched join points. In such cases, developers can configure the framework to select the pointcut mutant with the longest expression (in string length) as the representative pointcut mutant for that set of matched join points. Pointcuts with the same set of join points indicate that all these pointcuts are semantically equivalent. The longest expression often indicates that this expression is more specific in terms of the number of characters used in the expression. More characters of a pointcut indicate that the pointcut is designed to be closer (in naming) to the join points in the base program. Fewer characters indicate that the pointcut is generalized and has higher probability of being affected by fragile pointcut problems [14]. Hence we select the longest pointcut mutant to avoid potential fragile pointcut problems. Note that when we calculate the length of each expression for finding out the longest expression, we exclude characters of wildcards "*" (as well as ".." inside the argument list, being used as wildcards by AspectJ for arguments), because including these wildcards could produce longer expressions, which however are more general.

Weak or strong pointcut mutants can help developers discover a pointcut with incorrect strength and fix the pointcut by replacing it with a weak or strong pointcut mutant with the correct strength. We call this type of activities as *pointcut correction*.

Neutral pointcut mutants can help developers discover refactoring opportunities for a pointcut and refactor the pointcut by replacing it with a neutral pointcut mutant that better reflects the developers' expectation in matching future new added join points. We call this type of activities as *pointcut refactoring*.

3.6 Distance Measurer

Because the set of pointcut mutants in each of the three types (strong, weak, and neutral ones) may be large, it is often infeasible for developers to inspect each of them. Instead of inspecting all mutants, developers would be often more interested in inspecting pointcut mutants that resemble closely the original pointcut. To address this issue, the distance measurer helps developers identify pointcut mutants that resemble closely the original pointcut and focus their inspection efforts on these pointcut mutants.

In particular, the original pointcut and the pointcut mutants are fed as input to the distance measurer. For each pointcut mutant, the distance measurer uses a syntactic similarity measure to compute *pointcut distance*, which is the lexical distance between the strings of the original pointcut and the pointcut mutant. The pointcut distance reflects the deviation extent of the pointcut mutant from the original pointcut. In particular, the pointcut distance is defined as the number of characters that need to be inserted, deleted, or modified in the original pointcut to transform it into the mutant. The mutants under each category are then ranked based on their calculated pointcut distances: the smaller pointcut distance a pointcut mutant has, the higher rank the pointcut mutant is given. The classification of the mutants helps developers identify similar pointcuts with different strengths and the ranking of the mutants helps identify mutants of the pointcuts that resemble closely the original pointcut.

It is possible that the join points of the original pointcut and the pointcut mutant could be similar in functionality (reflected by their matched join points) but still have quite different expressions. In such a case, the lexical distance would be large, making such potentially useful pointcut mutants ranked later in the ranked list. To address this issue, we also intentionally promote the ranking of those pointcut mutants with small differences of their matched join points and the join points matched by the original pointcut.

In particular, we define a *semantic downsize measure*, which indicates the number of join points matched by the original pointcut but not matched by the pointcut mutant. We also define *semantic upsize measure*, which indicates the number of join points matched by the pointcut mutant but not matched by the original pointcut. Then we define the measure of *join point distance*, which is the sum of semantic downsize measure and semantic upsize measure for the pointcut mutant in contrast to the original pointcut. If the join point distance for a pointcut mutant is smaller, we adjust its original ranking (based on the syntactic similarity measure) to be higher.

In addition, to facilitate developer inspection, we display the differences of join points matched by pointcut mutants and the original pointcut along with pointcut mutants in the

result. Developers could inspect the ranked list of pointcut mutants together with the join point differences and select a correct or better pointcut that fits the developers' intended purpose.

4 Empirical Study

This section presents an empirical study of our framework. We obtained the empirical results by applying the implementation of our framework on selected pointcuts from the aspects `Bean`, `NullCheck`, `Tetris`, and `Cona-sim` in AspectJ benchmark suites (available at `http://www.sable.mcgill.ca/benchmarks/`).

Table 1 shows the empirical results obtained by applying our framework on the subjects. Columns 1 and 2 show the AspectJ benchmarks and their individual pointcuts on which our implementation has been applied, respectively. Column 1 additionally shows the lines of code for these benchmarks after their names. Column 3 shows the number of join point candidates identified by our join point candidate identifier. Column 4 shows the number of join points matched by each pointcut. Column 5 shows the number of pointcut mutants generated for each pointcut. Columns 6-8 show the number of strong, weak, and neutral mutants identified from the pointcut mutants, respectively. Note that in our results, from a set of pointcut mutants that share the same set of matched join points, we configure our framework to select the pointcut mutant with the longest expression (in string length) as the representative pointcut mutant (as described in Section 3.5). Column 9 shows the total number of pointcut mutants for developers to inspect. This count indicates the number of meaningful pointcut mutants identified from the large number of pointcut mutants initially generated (Column 5). Basically, Columns 5 and 9 show the initial number of generated pointcut mutants and the number of pointcut mutants after our effective reduction technique, which selects one representative pointcut mutant from the set of pointcut mutants with the same set of matched join points (Section 3.5). We observe that our reduction technique is effective in reducing a large number of mutants to only a few representative mutants for inspection.

As is described in Section 3.6, two measures of pointcut distance and join point distance have been provided to help developers identify pointcut mutants that resemble closely the original pointcut. Column 10 shows the average of all pointcut distances. Column 11 shows the average of all join point distances.

Table 2 shows the sample result for the `Bean` AspectJ benchmark. Row 1 shows the original pointcut and its matched join points. Columns 1 and 2 show the types of pointcut mutants as well as the pointcut mutants. Column 3 shows the differences in the set of join points matched by the pointcut mutant and the set of join points matched by the original pointcut. Columns 4 and 5 show the calculated pointcut and join point distances, respectively.

To illustrate the details in Table 2, let us consider a strong pointcut mutant `call(public void Point.setY(..))`. The pointcut mutant matches only one join point, `call(public void Point.setY(int))` and therefore does not match two join points matched by the original pointcut. The calculated pointcut distance of 3 indicates that the lexical distance between the original pointcut and the pointcut mutant is three characters, i.e., the original pointcut requires transformation, insertion, or deletion of three characters to change itself into the pointcut mutant. The original pointcut matches three join points. The calculated join point distance of two indicates that the matched join points of the mutant differs from the matched join point of the original pointcut by two join points.

From Table 2, we observe that pointcut mutants generated by our framework are close to the original pointcut. These mutants can be inspected for pointcut correctness or used to verify if a test suite for the woven code kills these mutants when they are used in place of the original pointcut.

In summary, from Table 1, we observe that the number of pointcut mutants generated for each pointcut is large. It is tedious to manually generate such a large number of pointcut mutants, identify join points for each pointcut mutant, compare its strength with the strength of the original pointcut, and choose an alternative one when the original pointcut does not serve the intended purpose. Our framework has been shown useful here. The preceding steps including identifying the strength of a pointcut and generating effective pointcut mutants have been completely automated in our framework, reducing the manual effort of developers. We observe that our framework generates meaningful pointcut mutants. Our carefully designed mechanisms in mutant generators allow meaningful and useful pointcut mutants to be generated.

5 Discussion

Analysis cost. Our framework basically generates mutants by lexical variation of a pointcut, its matched join points, and its unmatched join point candidates, thereby reducing or increasing the number of join points that the pointcut matches. Hence the analysis cost of the framework is dependent on the complexity (e.g., string length) of the original pointcut and the size of the join point candidates (which is proportional to the size of the program). More join points for a pointcut indicate that there will be an increase in the analysis cost.

Pointcut-fault detection. Although our framework is originated from mutation testing, our framework can be used for fault detection. From our empirical results, developers can inspect the pointcut mutants and their join points

Table 1. AspectJ benchmarks and results

AspectJ bench	pointcut	#join point cands	#join points	#pointcut mutants	#weak pointcut mutants	#neutral pointcut mutants	#strong pointcut mutants	#final pointcuts mutants	#avg pointcut distance	#avg join point distance
Bean (239)	call(public void Point.set*(*))	5	3	862	6	4	4	14	7	2
Tetris (1475)	call(String Logic. Blocks.typeToString(int))	32	1	1454	9	6	0	15	4	4
	get(static int Logic. Blocks.NUMBEROFTYPES)	34	1	782	3	2	0	5	6	1
	call(* Logic.Blocks.deleteLine(..))	32	1	2246	5	2	0	7	10	3
	call(* TetrisImages.loadImage())	32	1	2078	5	11	0	16	9	4
	call(* AspectTetris.restartGame())	32	1	2494	5	2	0	7	7	4
NullCheck (2926)	call(Object+ *.*(..))	50	1	446	3	2	0	5	18	1
Cona-sim (2651)	execution(public * Entity.getEntity*(..))	72	1	1568	2	1	0	3	8	1

Table 2. Sample results for the `Bean` benchmark

Pointcut : call(public void Point.set*(*))
Join points : call(public void Point.setX(int)), call(public void Point.setY(int)), call(public void Point.setRectangular(int,int))

Mutant type	Pointcut mutant	Join point differences	Pointcut distance	Join point distance
Strong	call(public void Point.set*(int))	- call(public void Point.setRectangular(int,int))	3	1
	call(public void Point.setY(..))	- call(public void Point.setX(int)) -call(public void Point.setRectangular(int,int))	3	2
	call(public void Point.setX(..))	- call(public void Point.setY(int)) -call(public void Point.setRectangular(int,int))	3	2
	call(public void Point.setRectangular(..))	- call(public void Point.setY(int)) - call(public void Point.setX(int))	13	2
Neutral	call(public void Point.set*(..,int))	None	6	0
Neutral	call(public void Point.set*(int,..))	None	6	0
Weak	call(public void Point*.*set*(int,..))	+ call(public void Point.offset(int,int))	10	1

to choose a pointcut mutant with the intended strength to replace and fix a faulty pointcut.

Pointcut evolution. During the evolution of a base program, new code may be added to the program, introducing potential join points that an existing pointcut can match. Although sometimes an existing pointcut correctly matches a right set of join points in a base program's current version, the pointcut may be too specific to match intended join points added later in a future program version or too general to leave out unintended join points added later. In such cases, developers can use the identified equivalent pointcut mutants for use in evolved programs. Note that in the presence of evolution of pointcuts or base programs, our current framework needs to be reapplied on the new version even when the framework had been applied on the old version. In future work, we plan to improve our framework to perform incremental analysis in the presence of software evolution.

6 Related Work

Lopes and Ngo [11] presented a unit testing framework for testing aspectual behavior. The framework tests the aspect behavioral implemented in advice. Xie and Zhao [15] presented Aspectra, an automatic test generation tool to test aspectual behavior by leveraging existing Java test generation tools. Our new framework can be seen as a testing approach for pointcuts, complementing these existing approaches for testing aspectual behavior.

Our previous work [1] presented an automated framework that tests pointcuts in AspectJ programs with the help of AJTE [16]. This framework identifies join points that can be matched by a pointcut expression and a set of boundary join points, which are join points that are not matched by a pointcut but resemble closely the matched join points. This framework does not generate variations of a pointcut, whereas our new framework helps generate mutants that resemble closely the original pointcut. These two frameworks are complementary in helping assure the correctness of pointcuts.

Baekken and Alexander [3] developed a candidate fault model for pointcuts. The model provides a set of mutation operators to find incorrect strengths in pointcut patterns and thereby evaluate the effectiveness of a test suite. Fabiano et al. [6] propose mutation operators based on generalization of faults for AO programs and cost analysis based on evaluation of the mutation operators on real-world applications. The idea behind the mutation operators has been leveraged by our framework in generating pointcut mutants.

Lemos et al. [10] proposed a fault classification system to help developers identify unintended join points and the in-

tended join points that were left out due to incorrect strength in pointcuts. Listing the neglected join points or unintended join points only informs developers the presence of faults. Correcting the faults would be tedious since the developers have to manually write a new pointcut. This issue in turn leads back to the problem of fragile pointcut problems where developers are not confident about the strength of a written pointcut. Our framework helps identify the strength of pointcuts as well as generating pointcut mutants with different strengths.

Kouhei et al. [13] proposed an approach of test-based pointcuts. They use automated unit test cases to identify execution points in a program and validate them against the program. Although this approach does not solve the fragile pointcut problem, it can be seen as an effective pointcut design approach. Our new framework complements their approach by generating effective pointcut mutants for mutation testing of pointcuts.

Ye and Volder [17] developed a tool for pointcut evaluation in identifying nearly matched join points for a pointcut as well as the reasons why a join point does not match a pointcut. Our framework complements their tool in providing mutation testing support for pointcuts.

7 Conclusion

We have developed an automated framework to generate pointcut mutants for a pointcut as well as the strength differences of the pointcut and pointcut mutants. The framework also classifies the generated pointcut mutants and ranks them based on a similarity measure, reflecting how similar they are to the original pointcut. Developers can inspect the ranked list of pointcut mutants along with their strength differences for pointcut correctness or choose the mutants for conducting mutation testing. The empirical study conducted using our framework shows that the framework can provide valuable assistance in generating effective mutants that are close to the original pointcut and are of appropriate strength.

In future work, we plan to conduct more case studies on applying our framework on various existing pointcuts. We plan to expand the mutant generator to generate pointcut mutants with more complex patterns of wildcards. We plan to investigate tool support for generating pointcut mutants related to dynamic contexts such as `cflow`. We plan to evaluate the effectiveness of the generated mutants by conducting user case studies.

Acknowledgments

This work is supported in part by NSF grant CCF-0725190.

References

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

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

[3] J. S. Baekken and R. T. Alexander. A candidate fault model for AspectJ pointcuts. In *Proc. International Symposium on Software Reliability Engineering*, pages 169–178, 2006.

[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] F. C. Ferrari, J. C. Maldonado, and A. Rashid. Mutation testing for aspect-oriented programs. In *Proc. International Conference on Software Testing, Verification, and Validation*, pages 52–61, 2008.

[7] 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.

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

[9] R. Laddad. *AspectJ in Action*. Manning, 2003.

[10] O. A. L. Lemos, F. C. Ferrari, P. C. Masiero, and C. V. Lopes. Testing aspect-oriented programming pointcut descriptors. In *Proc. Workshop on Testing Aspect-Oriented Programs*, pages 33–38, 2006.

[11] C. V. Lopes and T. Ngo. Unit testing aspectual behavior. In *Proc. Workshop on Testing Aspect-Oriented Programs*, 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, 2000.

[13] K. Sakurai and H. Masuhara. Test-based pointcuts for robust and fine-grained join point specification. In *Proc. International Conference on Aspect-Oriented Software Development*, pages 96–107, 2008.

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

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

[17] L. Ye and K. D. Volder. Tool support for understanding and diagnosing pointcut expressions. In *Proc. International Conference on Aspect-Oriented Software Development*, pages 144–155, 2008.