A Test-Driven Approach to Developing Pointcut Descriptors in AspectJ

Romain Delamare, Benoit Baudry, Sudipto Ghosh, Yves Le Traon

▶ **To cite this version:**

Romain Delamare, Benoit Baudry, Sudipto Ghosh, Yves Le Traon. A Test-Driven Approach to Developing Pointcut Descriptors in AspectJ. ICST '09: Proceedings of the $2^n d$ International Conference on Software Testing, Verification, and Validation, 2009, Denver, Colorado, USA, United States. inria-00477530

HAL Id: inria-00477530

https://inria.hal.science/inria-00477530

Submitted on 29 Apr 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Test-Driven Approach to Developing Pointcut Descriptors in AspectJ

Romain Delamare, Benoit Baudry
IRISA / INRIA Rennes
{*romain.delamare,benoit.baudry*}*@irisa.fr*

Sudipto Ghosh
Colorado State University
ghosh@cs.colostate.edu

Yves Le Traon
IT-Telecom Bretagne
yves.letraon@telecom-bretagne.eu

Abstract

Aspect-oriented programming (AOP) languages introduce new constructs that can lead to new types of faults, which must be targeted by testing techniques. In particular, AOP languages such as AspectJ use a pointcut descriptor (PCD) that provides a convenient way to declaratively specify a set of joinpoints in the program where the aspect should be woven. However, a major difficulty when testing that the PCD matches the intended set of joinpoints is the lack of precise specification for this set other than the PCD itself.

In this paper, we propose a test-driven approach for the development and validation of the PCD. We developed a tool, AdviceTracer, which enriches the JUnit API with new types of assertions that can be used to specify the expected joinpoints. In order to validate our approach, we also developed a mutation tool that systematically injects faults into PCDs. Using these two tools, we perform experiments to validate that our approach can be applied for specifying expected joinpoints and for detecting faults in the PCD.

Keywords: *Aspect-oriented programming, joinpoints, pointcut descriptors, mutation analysis, test-driven development, testing tool*

1. Introduction

With the emergence of the aspect-oriented programming (AOP) paradigm, the testing community must adapt existing testing techniques or develop new ones for systems developed using this paradigm. When implementing a program with aspects, the core concerns are implemented in a base program and the cross-cutting concerns are implemented in aspects. An aspect is composed of two main parts: (1) an advice that implements the behavior of the cross-cutting concern, and (2) the pointcut descriptor (PCD) that designates a set of joinpoints in the base program where the advice should be woven.

A number of researchers have studied the new types of faults that can be introduced by AOP and should, thus, be targeted by testing techniques. These works identify new faults that can occur in the interactions between the base program and the aspect, in the advice, or in the PCD. This last category of faults is specific to aspect-oriented languages since they introduce new constructs to define the PCD. As observed by Ferrari et al. [6], the PCD is the place that is the most fault-prone in an aspect.

The consequence of an incorrect PCD is that the advice is woven in unexpected places or is not woven where it is expected. This is known as the fragile pointcut problem in AOP [8, 13]. In turn, this introduces faults in the program resulting from aspect weaving. Moreover, when aspect-oriented programs evolve, there is a well-known risk that the PCD may match unintended joinpoints. This is known as the evolution paradox issue in AOP [14].

A major challenge to detecting faults in the PCD is that a PCD is an abstract declaration of a set of joinpoints and there is no other specification of the set of joinpoints that it should match. To cope with this lack of specification, we propose a test-driven approach to developing the PCDs, where the tests can be used to validate that the joinpoints matched by the PCD are the intended ones. The approach requires the creation of test cases that specify the intended and unintended join-

points. These test cases should not pass before aspect weaving and should pass after weaving if the correct set of joinpoints is matched by the PCD.

In order to write these test cases, we need to build an oracle that checks that an advice is woven at a particular point in the program. Currently, using AspectJ and JUnit, we can only build oracles that check whether an advice has executed correctly at a specific place in the base program. This is not satisfactory for us because, if such a test case fails, it is not possible to determine if this happened because the advice was not woven (the fault we are looking for) or because the advice does not behave as expected. In order to have more precise oracles for our test-driven approach, we implemented a tool called AdviceTracer that can be used with JUnit. AdviceTracer can determine at runtime which advice (defined in a particular aspect) is executed and at which place in the base program. This information can then be used to build oracles that specifically target the presence or absence of an advice, and do not just check if the advice executes correctly.

We also developed a mutation tool, called AjMutator, that systematically injects faults into the PCD and checks if a set of test cases is able to detect these faults. The faults are based on the model proposed by Ferrari et al. [6]. We performed experiments to evaluate the effectiveness of the test cases written using AdviceTracer in terms of their ability to detect faults introduced by AjMutator in the PCD. We used Tetris and an auction application as examples. Tetris has three aspects and the auction system has two. For each of these systems, we developed test cases that specify the intended joinpoints and generated mutants for the PCDs in the aspects. We observed that the test cases are able to detect different types of faults. We also measured the effort required to create the test cases that detected these faults.

Section 2 discusses the main issues encountered when testing a PCD. Section 3 presents the AdviceTracer tool and Section 4 presents the AjMutator tool. Section 5 describes our case studies. Section 6 discusses related research. We present our conclusions in Section 7.

2. Testing Pointcut Descriptors

When testing a PCD, we look for four kinds of faults, as defined by Lemos *et al.* [9]. Figure 1 illustrates them using a set abstraction. The lined set corresponds to the intended joinpoints and the grey set corresponds to the joinpoints actually matched by the PCD. An fault in the PCD can produce (1) both unintended and neglected joinpoints, (2) only neglected joinpoints, or (3) only unintended joinpoints. A PCD with a type

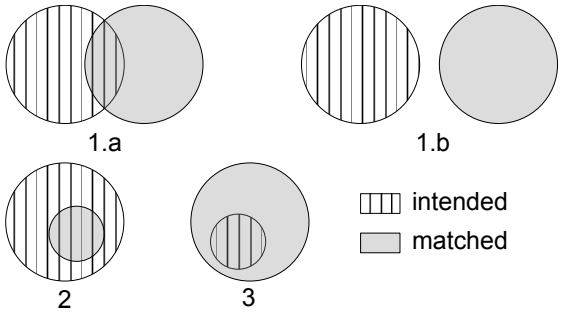

Figure 1. The four types of PCD faults

(1) fault can match intended joinpoints (1.a) or not (1.b).

In order to detect these faults we need an alternate way to specify the set of intended joinpoints. In this paper, we propose a test-driven approach for PCD development. In test-driven approaches, the test cases are the specification of the program [3]. Thus, in our approach the test cases specify the set of joinpoints that are expected to be matched by the PCD.

Our solution provides a specification of the PCD. Now, it is possible to check for inconsistencies between the PCD and this specification. A difference between the sets of intended and matched joinpoints reveals a fault. Another advantage of a test-driven approach is that it provides test suites for regression testing in the early stages of development. Thus, it becomes possible to check that no unexpected changes were introduced both during the development and the evolution of the program.

A key issue for such a test-driven approach is the lack of support for oracles that precisely specify the presence or absence of a joinpoint. In the current state of AspectJ, to test that a joinpoint was matched by the PCD at a specific place of the system, a test case needs to check that the behavior implemented by the joinpoint executes correctly at this place.

For example, let us consider the aspect in Listing 1. The expression

```
pointcut deleteLines(): execution
    (* AspectTetris.deleteLines())
```

is a PCD that will match all the executions of all `deleteLines()` methods, irrespective of their return type. The advice declared in the body of the expression,

```
after() : deleteLines() {...}
```

is woven after each joinpoint matched by the PCD.

Suppose that we need to write a test case that specifies that the advice is expected to be woven each time `deleteLines()` of the `AspectTetris` class is executed. Using JUnit, one would write a test case such as

```
1  public aspect Counter {
2     pointcut deleteLines() :
3     execution(* AspectTetris.deleteLines())
          ;
4
5     protected int currentLines;
6     protected int totalLines;
7
8     @AdviceName("deleteLines")
9     after() : deleteLines() {
10       totalLines += currentLines;
11       if(currentLines != 0)
12          System.out.println("Deleted␣"
13            + currentLines
14            + "␣lines␣(Total:␣"
15            + totalLines + ").");
16       }
17 }
```

Listing 1. An aspect example extracted from the Tetris system

`testdL()` shown in Listing 2. However, `testdL()` is not precise enough to specify a joinpoint. The first problem is that if this test case fails, it is not possible to conclude that the failure is caused by a fault in the PCD:

- If += is replaced by = in the advice of aspect `Counter`, `testdL()` will fail, but not because of a fault in the PCD.
- If $*$ is replaced by `int` in the PCD, `testdL()` will fail again, this time because of a fault in the PCD.

The second problem is that `testdL()` will not detect all the faults in the PCD. For example, in Listing 1, if the PCD is replaced by `execution(* AspectTetris.*())`, `testdL()` will pass even though there is a fault in the PCD. This fault is of type 3 as defined in Figure 1, which means that the expected joinpoints are matched, but the PCD also matches unintended joinpoints. To detect that fault, we need to write test cases that specify that no advice should be woven in some places of the code. However, this is not possible to do with test cases such as `testdL()`. These test cases do not explicitly mention the presence or absence of an advice; they just implicitly assume the presence of an advice. Not allowing explicit mention of an advice is a limitation of JUnit-based unit testing.

In order to experiment with a test-driven approach for PCD validation and overcome the limitations of JUnit, we developed a tool called AdviceTracer. This tool enables us to define an oracle for test cases that explicitly specify the presence or absence of an advice at a specific place in the program.

```
1  public class TestCounter {
2     @Test
3     public void testdL() {
4        AspectTetris tetris = new
              AspectTetris();
5        tetris.startTetris();
6        tetris.deleteLines();
7        assertEquals(tetris.totalLines,1);
8     }
9  }
```

Listing 2. A JUnit test class for the Counter aspect

```
1  public class C {
2    public void m1() { ... }
3    public void m2() {
4      ...
5      m1();
6    }
7  }
```

Listing 3. A Java class example

3. AdviceTracer

The AdviceTracer tool [4] allows a programmer to write test cases that focus on checking whether or not a joinpoint has been matched by the PCD. More precisely, AdviceTracer is used to specify an oracle that expects the presence or absence of an advice at a particular point in the base program. Test cases can specify the PCD without executing the behavior of the advice.

3.1. Illustrative example

Listings 3, 4 and 5 illustrate the use of AdviceTracer. Listing 3 shows a Java class `C` with two methods `m1` and `m2`; `m2` calls `m1`. Listing 4 shows an Aspect `A`, in which an annotation has been added in order to name the advice, `A1`. The advice is woven before the executions of `m1` which are in the control flow of `m2`.

Listing 5 shows two test cases that specify the set of expected joinpoints. In a textual form the specification of the expected joinpoints can be expressed as "the advice `A1` should be executed only before the executions of `m1` in the control flow of `m2`". This means that we must specify that the advice `A1` is expected when `m2` calls `m1`, and also that it should not be executed when `m1` is executed outside the control flow of `m2`. In a test-driven approach, we need two test cases to specify this.

- `test1()` specifies that no advices should be executed (line 8) when only `m1` is called (line 7). Thus, the test case will pass only if weaving an aspect (here aspect `A`) does not introduce the exe-

```
1  public Aspect A {
2    @AdviceName("A1")
3    before(): execution(void C.m1())
4       && cflow(execution(void C.m2())) {
5      ...
6    }
7  }
```

Listing 4. An AspectJ aspect example

```
1  public class Test {
2    @Test
3    public void test1() {
4      C c = new C();
5      addTracedAdvice("A1");
6      setAdviceTracerOn();
7      c.m1();
8      setAdviceTracerOff();
9      assertAdviceExecutionsEquals(0);
10   }
11
12   @Test
13   public void test2() {
14     C c = new C();
15     addTracedAdvice("A1");
16     setAdviceTracerOn();
17     c.m2();
18     setAdviceTracerOff();
19     assertAdviceExecutionsEquals(1);
20     assertExecutedAdviceAtJoinpoint("A1","C
           .m1:2");
21 }
```

Listing 5. A JUnit test class illustrating how to use AdviceTracer

cution of an advice when executing `m1`. The test case will fail if a joinpoint of `m1` is matched by the advice (unintended joinpoint).

- `test2()` calls `m2` and then specifies (at lines 19-20) that advice `A1` should be executed from a joinpoint at line 2 of `C`, within `m1`. So if the test case passes, we know that the advice was executed within the context of execution of `m1`.

If the PCD of Listing 4 is replaced by **`execution`** `(`**`void`** `C.m1())`, then `test1()` fails because the advice is executed in this test scenario. If the pointcut is replaced by `execution(void C.m2())`, then `test2()` fails although the advice is executed because it is woven within `m2` instead of `m1`.

This example also illustrates how AdviceTracer can handle dynamic PCDs. The PCD of Listing 4 is dynamic: it matches the execution of `m1` in the control flow of `m2`. This kind of PCD can only be resolved at runtime. To specify this PCD we first execute `m1` outside the control flow of `m2` and check that the advice was not executed (`test1` of Listing 5), and then we execute `m1` in the control flow of `m2` and check that the advice was executed at the correct joinpoint (`test2` of Listing 5).

AdviceTracer makes it possible to specify the expected joinpoints in order to check that there are no neglected joinpoints. In such an approach, there must be test cases that specify every place in the base program that should be matched by each PCD.

As shown in `test1`, it is also possible to write test cases that specify unintended joinpoints. In Section 5.3, we present a strategy to reduce the effort required to write test cases.

3.2. Primitive methods of AdviceTracer

The above two test cases illustrate how the primitive methods of AdviceTracer are used. These primitives are of three distinct types: those that start or stop AdviceTracer, those that configure the traced advices, and those that define assertions to specify the oracle.

3.2.1. Starting AdviceTracer. To start tracing, AdviceTracer must be set on by calling the static method `setAdviceTracerOn()`. It should be called before calling a method where an advice is expected (or not expected) to be woven. The static method `setAdviceTracerOff()` stops tracing. Between `setAdviceTracerOn()` and `setAdviceTracerOff()`, AdviceTracer stores information about which advices (identified by their name) were executed and where they were executed.

3.2.2. Restricting the traced advices. Using AdviceTracer each test can specify the advices to be traced. If a test case does not specify any advice, then all the advices are traced. On lines 5 and 15 of Listing 5, the method `addTracedAdvice` is called. It adds the advice as a parameter to the collection of traced advices. Another method, `setTracedAdvices`, specifies a collection of advices to be traced. In listing 5, `test1` specifies the absence of the advice `A1` and `test2` specifies the presence of the advice `A1`.

The test cases `test1` and `test2` are said to be *modular* because they specify a set of joinpoints where a specific advice must or must not be woven. A test case is not modular when it is not specific to a particular advice; it passes as long as an advice is woven (or no advice is woven) instead of passing when a specific advice is woven (or when a specific advice is not woven). Restricting the traced advices improves the modularity of the test cases.

The benefit of modular test cases is that they are less affected by removal or addition of advices in the aspects. They are only affected by changes made in

the PCDs of the advices they trace. For instance, if a new advice is woven within `C.m1`, then the test cases of Listing 5 will still pass because they only consider advice `A1`.

If the PCD changes, the test case needs to be updated. If the base program changes, we may also need to add more test cases.

3.2.3. Assertions provided by AdviceTracer. AdviceTracer provides three new assertions that are extensions of JUnit assertions.

assertAdviceExecutionEquals(int n) : Passes if *n* executions of some advices occur, fails otherwise.

assertExecutedAdvice(String advice) : Passes if the advice, whose name is passed as the parameter, was executed, fails otherwise.

assertExecutedAdviceAtJoinpoint(String advice, String joinpoint) : Passes if the specified advice was executed at the specified joinpoint, fails otherwise. The format of the joinpoint parameter is: `className.methodName:lineNumber` where `lineNumber` refers to the line where the joinpoint is expected.

3.3. Implementation of AdviceTracer

AdviceTracer provides an API consisting of the primitive methods described above and contains an aspect, implemented using AspectJ. The advice in this aspect retrieves the name of the advice that is being traced and the location of the joinpoint that triggered the advice execution. This information is stored in a `TraceElement` object, which is a pair of strings, one for the advice (e.g., `A1`) and one for the joinpoint (e.g., `C.m1:2`). The string for the advice is its name, specified with the `@AdviceName` annotation. The string for the joinpoint is built with the qualified name of the method and the line number where it is located (separated by the '`:`' character).

The advice is woven before each joinpoint matched by all the aspects under test (i.e. before each execution of an advice). The PCD in the AdviceTracer aspect is "**`adviceexecution`**`()&& !` **`within`**`(AdviceTracer)`". This PCD matches the execution of all the tested advices and not AdviceTracer's own advice.

All the `TraceElement` objects are stored in a list that can be retrieved in each test case by calling a static method (`getExecutedAdvices`). This list is reset each time AdviceTracer is set on, and thus it only contains the `TraceElement` objects corresponding to the advice execution that were traced between the last calls to `setAdviceTracerOn` and `setAdviceTracerOff`.

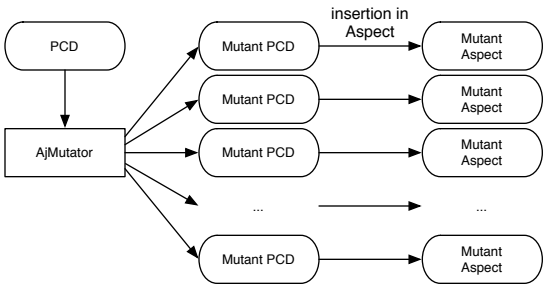

Figure 2. The PCD mutation process

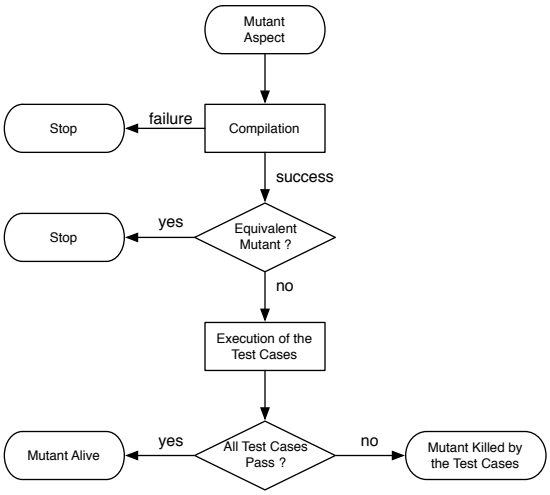

Figure 3. The mutant evaluation process

4. Mutation Tool

To validate AdviceTracer, we implemented a mutation tool for PCDs in AspectJ, called AjMutator. This tool is able to automatically insert faults in PCDs to change the set of matched joinpoints. AjMutator inserts seven different types of faults based on seven mutation operators, all defined by Ferrari *et al.* [6].

PCCE : Pointcut changing by switching `execution`/`call` PCDs. The PCCE operator replaces a `call` PCD by an `execution` PCD or vice versa. It changes the context where the advice is woven (with a `call` PCD, the advice is woven where the method is called, with an `execution` PCD, the advice is woven in the method). It produces both neglected and unintended joinpoints – type (1) fault.

PCGS : Pointcut changing by switching `get`/ `set` PCDs: The PCGS operator replaces a `set` PCD by a `get` PCD or vice versa. It produces both neglected and unintended joinpoints.

PCLO : Pointcut changing by changing logical operators. This operator replaces a conjunction ('&&') by a disjunction ('||') or the contrary. It produces type (3) and type (2) faults, respectively.

PCTT : Pointcut changing by switching `this`/ `target` PCDs. The PCTT operator replaces a `this` PCD by a `target` PCD. It produces type (1) faults.

POEC : Pointcut weakening or strengthening by changing exception throwing clauses. The POEC operator adds, removes or changes the throwing clauses in the PCDs that specify a method. It produces neglected and/or unintended joinpoints.

POPL : Pointcut weakening or strengthening by changing parameter lists. The POPL operator adds, removes or changes the parameters in the PCDs that specify a method or a constructor. It produces neglected and/or unintended joinpoints.

PWIW : Pointcut weakening by inserting wildcards into the PCD. The PWIW operator replaces an identifier in the PCD by a wildcard ('*'). It weakens the PCD (i.e., it becomes more general) and only produces unintended joinpoints – type (3) fault of Figure 1.

Figure 2 shows the mutation process using AjMutator. AjMutator takes a PCD as input and successively applies different operators to the PCD. Each operator can produce several mutants from a single PCD. In order to run the mutation analysis, these mutant PCDs are then inserted into the aspect to replace the original PCD. One mutant aspect is generated for each mutant PCD.

Figure 3 shows the mutation analysis process. First, the mutant is compiled. If the compilation fails, then the mutant is discarded. For example, the PCLO operator can generate mutants that do not compile. For instance, in the `NewBlocks` aspect of the *tetris* example, the following PCD is used:

```
pointcut getBlock(int type) : call
     (int[][] Blocks.getBlock(int))
      && args(type);
```

The expression `args(type)` specifies that the argument of the method is bound to the parameter of the pointcut (`type`). Replacing the conjunction by a disjunction results in an inconsistent binding and is reported as an error by the AspectJ compiler.

Table 1. Metrics for Auction and Tetris

System	# Class	# Method	# Test Cases (Intended/ Unintended)
Auction	41	177	16 (5/11)
Tetris	11	28	41 (14/27)

The second step of the process checks for *equivalent mutants* and removes them from the set of mutants to kill. A mutant PCD, `p'`, is equivalent to the initial PCD, `p`, with respect to the base program, `BP`, if `p'` matches exactly the same set of joinpoints as `p` in `BP`. The AspectJ compiler, AJDT, provides the API to obtain information on the set of joinpoints matched by `p` and `p'`. Then, by comparing these two sets, it is possible to determine if the PCDs are equivalent. Thus, in our case, the detection of equivalent mutants can be automated.

After the set of valid mutants is selected, the test cases are executed with the mutant aspect in the third step of the process. If at least one test case fails, then the mutant is considered "killed", otherwise it is "alive". Thus, unlike classical mutation analysis, a mutant is killed not based on the difference between the behavior of the initial and the mutant program. Instead, we use the oracle of the test cases to kill the mutants. This difference is because of the different goals for mutation analysis. Mutation analysis is generally used to validate the quality of test data, for which the oracle might not be available. In our case, mutation analysis serves a different purpose: it aims at validating the quality of the oracle in test cases that specify intended joinpoints.

5. Evaluation

We evaluated our approach on two AspectJ systems. For each one we wrote test cases for specifying the different PCDs defined in the aspects. Then we used our mutation tool to insert faults in the PCDs and we checked whether the test cases written with AdviceTracer can detect the introduced faults.

The *auction* example is an implementation of an online auction system where users can buy or sell items. This system was developed by the Triskell research group at IRISA. The *Reserve* aspect allows the user to add an optional and secret reserve price. If the reserve price is not reached at the end of the auction, then the sale is canceled. The *AltBid* aspect modifies the way that the price is calculated using the second highest bid.

The *tetris* example is an implementation of the classic video game that was developed by Gustav Ev-

Table 2. Mutation scores for PCDs in Tetris and Auction

Mutants	Non-Eq	Killed	%
Auction			
Reserve			
31	10	10	100%
AltBid			
64	13	13	100%
Tetris			
NewBlocks			
72	7	7	100%
Counters			
51	15	15	100%
Levels			
66	12	12	100%

Table 3. Results for Auction with only the test cases for the intended joinpoints

Mutants	Non-Eq	Killed	%
Reserve			
64	13	12	92.3%
AltBid			
31	10	8	80%

ertsson [5]. The *NewBlocks* aspect adds new kinds of blocks to the game. The *Counters* aspect counts the deleted lines and prints them on the game layout. The *Levels* aspect adds a level system to the game: each time a certain number of lines are deleted, the level is increased and the blocks fall faster.

Table 1 shows relevant metrics of the two systems. The *auction* example has 41 classes and 177 methods, and 16 test cases were written to specify the PCDs (5 test cases to specify the intended joinpoints and 11 to specify the unintended joinpoints). The *tetris* example has 11 classes and 28 methods, and 41 test cases were written to specify the PCDs (14 test cases to specify the intended joinpoints and 27 to specify the unintended joinpoints).

5.1. Results for Mutation Analysis

The operators described in 4 were used to generate mutants. 359 number of mutants were produced, out of which 75 were non-compilable. Out of the remaining 284 mutants, 57 were non-equivalents mutants.

The PWIW operator produced the most mutants (224), but 214 of them are equivalent (only 10 non-equivalent mutants). In most cases, the wildcard introduced by the operator does not allow the selection of more joinpoints. For instance, if `call( void AspectTetris.deleteLines())` is replaced by `call(void *.deleteLines())`, the mutant is equivalent as there is no other class than `AspectTetris` which has a method called `deleteLines` that takes no parameter.

Table 2 shows the results of our evaluation. For each aspect, it shows the number of mutants, the number of non-equivalent mutants, and the number of mutants killed.

In the *Auction* system, 31 mutants were produced for the *Reserve* aspect, and 10 of them are non-equivalent. 64 mutants were produced for the *AltBid* aspect, with 13 non-equivalent mutants. All the non-equivalent mutants were killed by the test cases.

Int *Tetris*, 72 mutants were produced for the *NewBlocks* aspect, with 7 non-equivalent mutants. 51 mutants were produced for the *Counters* aspect, with 15 non-equivalent mutants. For the *Levels* aspect, 66 mutants were produced, and 12 of them were non-equivalents. All the non-equivalent mutants were killed by the test cases.

These results show that test cases written using AdviceTracer are actually able to detect faults in the PCD, and thus such test cases can specify PCDs in a test-driven approach.

Table 3 shows the results of the mutation analysis with only the test cases for the intended joinpoints. The results for *Tetris* are the same as in Table 2, which means that the test cases for the intended joinpoints can kill all the mutants of *Tetris*. For the *Reserve* aspect, 92.3% of the mutants are killed, and for the *AltBid*, 80% of the mutants are killed.

The 3 mutants that are not killed were generated by the PWIW operator. This operator only produces mutants with unintended joinpoints, so usually mutants generated by this operator cannot be detected by test cases for intended joinpoints.

These results allow one to find a tradeoff between the testing effort and the level of confidence in the PCD. Writing only the test cases for intended joinpoints is usually a limited effort and this can guarantee that the PCD matches at least these joinpoints. Of course, to validate that the PCD matches only those joinpoints, it is necessary to write more test cases. In section 5.3 we discuss a strategy to reduce the cost of generation of these test cases.

5.2. Evaluation of the AdviceTracer Framework

In section 2 we explained why JUnit was not well adapted for a test-driven development of PCDs. Here,

we discuss how AdviceTracer is actually better suited for the development of test cases that specify expected joinpoints.

Consider the aspect `Counter` shown in Listing 1. With AdviceTracer, we can specify the expected set of joinpoints with `testdL1()` and `testdL2()` shown in Listing 6. These test cases capture the intent of the specification more precisely than `testdL()` of Listing 2.

AdviceTracer can make precise oracles that specifically target the PCDs. There are three points that make test cases made with AdviceTracer more precise than regular JUnit test cases:

- Test cases written with AdviceTracer do not fail because of a fault in the advice. If, on line 10 of Listing 1, += is replaced by =, `testdL1` and `testdL2` pass but `testdL` does not.

- Test cases written with AdviceTracer make fault localization easier. If `testdL1` fails, we know there is a fault in the PCD that produced a neglected joinpoint. If `testdL2` fails, we know there is a fault in the PCD that produced an unintended joinpoint. If `testdL` fails, we only know that there is a fault, most likely localized in the PCD or in the advice.

- Test cases written with AdviceTracer are more modular. The success of `testdL1` and `testdL2` only depends on one PCD, changes made on other aspects do not affect them. New aspect, or changes within the advice `A1` could change the result of `testdL`, even if the PCD is unchanged.

5.3. Strategy for specifying the unintended joinpoints

Unit test cases are well suited for the specification of intended joinpoints since they target single methods and intended joinpoints are defined at the level of a single method. However, unit test cases might not be as well adapted for the specification of unintended joinpoints. In that case, it is necessary to write unit test cases for each method where no joinpoint is expected. This can require a large number of unit test cases since for most PCDs, the number of intended joinpoints is small.

We need a strategy for specifying the unintended joinpoints that can save time and drastically reduce the number of required test cases. Our strategy is to use system test cases for specifying unintended joinpoints. These test cases cover a greater part of the program, thereby reaching a larger number of unintended joinpoints with less effort. However, system test cases might also cover intended joinpoints. In that case, the oracle needs to check that every executed advice corresponds to an intended joinpoint, otherwise the test case fails. This kind of test cases actually only specify unintended joinpoints. If there are neglected joinpoints and no unintended joinpoints, these test cases will pass. However, if advices are executed at unintended joinpoints by these test cases, then they fail.

This strategy reduce the cost of writing test cases for the unintended joinpoints. It allowed us to write only a few test cases specifying unintended joinpoints, 11 for *Auction* and 24 for *Tetris*.

```
1  public class TestCounter2 {
2    @Test
3    public void testdL1() {
4      AspectTetris tetris =
5          new AspectTetris();
6      tetris.startTetris();
7      addTracedAdvice("deleteLines");
8      setAdviceTracerOn();
9      tetris.deleteLines();
10     setAdviceTracerOff();
11     assertAdviceExecutionsEquals(1);
12     assertExecutedAdviceAtJoinpoint(
13           "deleteLines",
14           "AspectTetris.deleteLines:23");
15   }
16
17   @Test
18   public void testdL2() {
19     AspectTetris tetris =
20         new AspectTetris();
21     tetris.startTetris();
22     addTracedAdvice("deleteLines");
23     setAdviceTracerOn();
24     tetris.gameOver();
25     setAdviceTracerOff();
26     assertAdviceExecutionsEquals(0);
27   }
28 }
```

Listing 6. A JUnit test class for the Counter aspect

6. Related work

Ye *et al.* [19] tackle the issue of correctness of a PCD. They propose tools to assist developers in diagnosing faults and fixing PCDs. First, according to a PCD, they compute a set of joinpoints that are almost matched by this PCD. This means they compute the set of joinpoints that would be matched if the PCD was slightly different. That way, a developer who analyzes these joinpoints can check whether he/she expected one of these joinpoints to be matched. If this is the case,

then the PCD must be fixed. For this step, Ye *et al.* also propose a tool that can explain why a joinpoint is not matched by the PCD.

The solution proposed by Ye *et al.* to tackle the problem of faulty PCDs is thus different from our approach. Instead of specifying the set of expected joinpoints a priori, they provide assistance for manual inspection by the developer. The benefit of their approach is that it does not require additional work from the developer. In case the aspects evolve, with this approach the developer has to manually check all the PCDs to be sure that there are no regressions. With our approach the developer can run all the test cases again. If a new advice is added, new test cases must be added to check its PCD. If existing PCDs are modified, the test cases for that PCD must be changed. All the other test cases remain unchanged.

There exist several works related to mutation analysis in the context of AOP. McEachen *et al.* [11] and Baekken *et al.* [2] propose several fault models. These works analyze aspect-oriented languages and identify AOP-specific faults that can occur. In particular, Baekken *et al.* focus on different categories of faults that can occur in PCDs. Anbalagan *et al.* [1] also propose mutation analysis of the PCD. Like Ye *et al.*, they also associate a notion of similarity with the initial PCD to select a subset of the mutants and limit the cost of mutation analysis. Ferrari *et al.* [6] analyze the different faults models previously proposed for AOP and study how they map on to different languages for aspect-oriented programming.

Another work related to our approach is proposed by Sakurai *et al.* [12], who propose to describe pointcuts in unit test cases. Although this approach at first glance appears to be similar to our work, it is actually different. They propose a new language for pointcut description based on unit test cases. Here the test cases are not meant to validate a pointcut described as a regular expression. Instead, they are meant to replace the regular expression. This should allow PCDs to be more robust with respect to evolution. However, these PCDs can still be erroneous and the authors do not tackle the issue of the correctness of their PCDs.

Other work on testing aspect-oriented programs focus on other aspects of the testing activity. In particular, some works have focused on unit testing in AOP and regression testing when the base program or aspects evolve.

Xu *et al.* [18] tackle the issue of regression test selection for AOP. They extended a technique for Java introduced by Harrold *et al.* [7] to take aspects into account. Based on the comparison of the control-flow graphs for test cases on the program before and after evolution, they identify the subset of test cases that must be executed for regression testing.

Xie *et al.* [15] focus on the automatic generation of test data for aspect-oriented programs. Their framework, called Aspectra, focuses on the validation of the behavior implemented in the advice. This tool leverages existing tools for automatic generation of test data for Java programs. A major issue of automatic test data generation is that the number of generated data can be large. To tackle this issue, Xie *et al.* [16] introduced a framework for detecting redundant unit tests in AspectJ programs. The proposed framework removes the test cases that do not exercise a new behavior.

Xu *et al.* [17] propose a model-based testing approach for AOP. The behaviors of the base program and the advices are modeled with statecharts. The authors merge these statecharts and generate test data to cover the paths in the composed statechart that correspond to interactions between the base program and the advice.

Lopes *et al.* [10] focus on unit-testing the advices. Their approach relies on JAML (Java Aspect Markup Language), an aspect language for Java where the advices are implemented in regular Java classes and the PCDs are described in XML. This allows advices to be called as regular methods, and thus to be unit-tested. They also provide JamlUnit, a JUnit extension to test JAML advices.

7. Conclusions and Future Work

Pointcut descriptors in aspects are critical because they specify the locations where a cross-cutting concern should be woven in a program. However, as pointed out by several papers on AOP testing, these pointcut descriptors are fault-prone and should, thus, be tested. In this paper we identified two major issues related to testing PCDs: the lack of specification for intended joinpoints and the limitations of JUnit for explicitly asserting the presence or absence of an advice at a specific point in the program.

We proposed a test-driven approach for the development of PCDs. This approach allows us to tackle the issue of lack of specification; in our approach the test cases become the specification of the intended joinpoints. We also developed a tool called AdviceTracer that provides an API to testers so that they can define oracles that explicitly specify an expected joinpoint or the expected absence of a joinpoint.

AdviceTracer collects trace information that can also be used to debug aspect-oriented programs. Another possible way to use AdviceTracer is to assess change impact through dynamic analysis during the evolution of base program and the aspects.

A second contribution of this work is the development of mutation tool for PCDs. Based on mutation operators defined in other works, AjMutator automatically generates mutants for a PCD. We used this tool to validate that the test cases developed using AdviceTracer can actually detect faults in PCDs. While experimenting with our test-driven approach, we observed that unit test cases are indeed able to specify the intended joinpoints and that these test cases can detect faults in the PCDs. Moreover, we observed that unit test cases that specify unintended joinpoints also detect faults in PCDs, but this requires a large number of unit test cases. The experiments also allowed us to validate that AdviceTracer is better suited to a test-driven development of PCDs than plain JUnit.

In the future, we will investigate the ability of AdviceTracer to detect other types of faults that could not be injected in the systems studied here. We also want to study how such a test-driven approach for PCDs can help detect faults when the base code evolves. Another point that could be explored is a way to adapt existing test cases, in particular system tests, in order to reduce the effort required to specify unintended joinpoints.

References

[1] P. Anbalagan and T. Xie. Efficient mutant generation for mutation testing of pointcuts in aspect-oriented programs. In *Mutation'2006*, pages 51–56, Raleigh, NC, USA, November 2006.

[2] J. Baekken and R. T. Alexander. A candidate fault model for AspectJ pointcuts. In *ISSRE'06 (Int. Symposium on Software Reliability Engineering)*, pages 69 –178, Raleigh, NC, USA, November 2006 2006.

[3] K. Beck. *Test Driven Development: By Example*. Addison-Wesley, Boston, MA, USA, 2003.

[4] R. Delamare. Advicetracer. `http://www.irisa.fr/triskell/Softwares/protos/advicetracer`.

[5] G. Evertsson. Tetris in AspectJ. `http://www.guzzzt.com/coding/aspecttetris.shtml`.

[6] F. C. Ferrari, J. C. Maldonado, and A. Rashid. Mutation testing for aspect-oriented programs. In *ICST '08: Proceedings of the 1st International Conference on Software Testing, Verification and Validation*, 2008.

[7] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *OOPSLA'01: Proceedings of the 16th conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 312–326, 2001.

[8] C. Koppen and M. Storzer. Pcdiff: Attacking the fragile pointcut problem. In *European Interactive Workshop on Aspects in Software (EIWAS)*, September 2004.

[9] O. A. L. Lemos, F. C. Ferrari, P. C. Masiero, and C. V. Lopes. Testing aspect-oriented programming pointcut descriptors. In *WTAOP '06: Proceedings of the 2nd Workshop on Testing Aspect-Oriented Programs*, pages 33–38, New York, NY, USA, 2006. ACM.

[10] C. V. Lopes and T. Chi Ngo. Unit testing aspectual behavior. In *Proceedigns of the 1st Workshop on Testing Aspect-Oriented Programs*, 2005.

[11] N. McEachen and R. T. Alexander. Distributing classes with woven concerns: an exploration of potential fault scenarios. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 192–200, New York, NY, USA, 2005. ACM.

[12] K. Sakurai and H. Masuhara. Test-based pointcuts for robust and fine-grained join point specification. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 96–107, New York, NY, USA, 2008. ACM.

[13] M. Störzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM'05*, pages 653– 656, Budapest, Hungary, September 2005.

[14] T. Tourwe, J. Brichau, and K. Gybels. On the existence of the aosd-evolution paradox. In *AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies*, Boston, USA, 2003.

[15] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of AspectJ programs. In *AOSD'06: Proceedings of the 5th international conference on Aspect-Oriented Software Development*, pages 190–201, 2006.

[16] T. Xie, J. Zhao, D. Marinov, and D. Notkin. Detecting redundant unit tests for AspectJ programs. In *ISSRE'06: Proceedings of the 17th International Symposium on Software Reliability and Engineering*, pages 179–190, 2006.

[17] D. Xu and W. Xu. State-based incremental testing of aspect-oriented programs. In *AOSD'06: Proceedings of the 5th international conference on Aspect-Oriented Software Development*, pages 180–189, 2006.

[18] G. Xu and A. Rountev. Regression test selection for AspectJ software. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 65–74, 2007.

[19] L. Ye and K. D. Volder. Tool support for understanding and diagnosing pointcut expressions. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 144–155, New York, NY, USA, 2008. ACM.