HAL
open science

An Approach for Testing Pointcut Descriptors in AspectJ

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

To cite this version:

Romain Delamare, Benoit Baudry, Sudipto Ghosh, Shashank Gupta, Yves Le Traon. An Approach for Testing Pointcut Descriptors in AspectJ. Journal of Software Testing, Verification and Reliability, 2011. hal-00641263

HAL Id: hal-00641263

https://inria.hal.science/hal-00641263v1

Submitted on 15 Nov 2011

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.

An Approach for Testing Pointcut Descriptors in AspectJ

Romain Delamare[1], Benoit Baudry[1], Sudipto Ghosh[2], Shashank Gupta[3], Yves Le Traon[4]

[1] *INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France*

[2] *Colorado State University, Fort Collins CO 80523, USA*

[3] *Progressive, Denver, CO USA*

[4] *University of Luxembourg, Faculté des Sciences de la Technologie et de la Communication, L-1359 Luxembourg, Luxembourg*

---

*Benoit Baudry

SUMMARY

Aspect-oriented programming (AOP) promises better software quality through enhanced modularity. Crosscutting concerns are encapsulated in separate units called aspects and are introduced at specific points in the base program at compile-time or runtime. However, aspect-oriented mechanisms also introduce new risks for reliability that must be tackled by specific testing techniques in order to fully benefit from the use of AOP. This paper focuses on the pointcut descriptor (PCD) that declares the set of points in the base program's execution where the crosscutting concern must be woven. A fault in the PCD can have a ripple effect and result in many different faults. New behavior may be added in unexpected places, or places where new behavior should be added may be missed. When implementing aspect-oriented programs with AspectJ, JUnit is most commonly used to test the program. However, JUnit does not offer any mechanism to look for faults specifically located in the PCD. As a consequence, these faults can be detected only through complex test scenarios and side effects that are difficult to trigger and observe. This paper proposes to

*Correspondence to: INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France

monitor the execution of advices in an aspect-oriented program and use this information to build test cases that target faults in PCDs. The AdviceTracer tool has been developed to automatically monitor and store all information related to advice executions. It also offers a set of operations that can be used to check the presence or absence of advices at specific points in the execution. These operations improve the definition of an oracle for PCD test cases. An empirical study is performed to compare JUnit and AdviceTracer for testing PCDs in terms of the complexity of test cases and their ability to detect faults. The study is performed on a Healthwatcher system that has 93 classes and 19 PCDs. It reveals that test cases that use AdviceTracer to test PCDs are easier to write (shorter test cases and written in less time than with JUnit) and detect more faults.

Copyright © 0 John Wiley & Sons, Ltd.

1. Introduction

Aspect-oriented programming (AOP) [15] promises better modularity by isolating crosscutting concerns in separate units called aspects. Aspects contain an advice that implements the crosscutting behavior and a pointcut descriptor (PCD) that declares a set of join points when the advice must execute. Aspects can also declare intertype definitions to modify the static structure of the base program. Aspects are woven with the core concerns in order to build a complete system. Despite claims regarding increased software quality, AOP has not been massively adopted [22]. This can be partly explained by the lack of trust of developers in AOP because of the lack of effective tools for understanding the interactions between aspects and the base program [4].

It is necessary to consider features unique to aspect oriented languages in order to develop testing techniques that are well suited for AOP. A number of researchers have studied the new types of faults that can be introduced by AOP [1, 9]. They identify new faults that can occur in the interactions between the core concerns and the aspects, in the advice, or in the PCD. The last category of faults is specific to aspect-oriented languages that introduce new constructs to define the PCD. As observed by Ferrari et al. [9], the PCD is the place that is the most error-prone in an aspect.

This work focuses on the development of a tool and an associated method to assist in the definition and validation of pointcut descriptors. An incorrect PCD can generate numerous faults in the program: it can miss expected join points and / or match unintended join points. As a consequence, the advice can introduce unexpected behavior at numerous places. Alternatively, expected behavior can be missing at several places in the program. This is known as the fragile pointcut problem in AOP [16, 25]. Unforeseen problems can also occur when evolving aspect-oriented programs [21]. There is a risk that the PCD may match unintended join points. This is known as the evolution paradox issue in AOP [26]. Pointcut descriptors thus represent an important issue for the correctness of aspect-oriented programs.

A major challenge to detecting faults in the PCD is that it declares a set of join points and there is no other specification of the set of join points that it should match. For example, when developing an aspect-oriented program with AspectJ, there exists no testing approach that allows a tester to specify the expected locations of join points. One way to address this problem is to write test cases, such as with JUnit, that check whether or not the behavior introduced by the advice executes correctly at the expected place in the program. The drawback of this solution is that these test cases do not target the correct type of fault: they target faults in the advice's behavior, and, only as a side effect, may capture faults in the PCD, such as if the advice does not execute correctly because it has not been woven at the expected join point.

This paper proposes to monitor the execution of advices and define a set of query operations to retrieve information about this execution. For example, one operation can check the presence of an advice at specific place in the execution of the program. Such operations are meant to build automatic oracles for PCD test cases. A tool called AdviceTracer has been developed, which handles monitoring and storage of information regarding what advices defined in a particular aspect are executed and at which place in the base program. AdviceTracer offers a set of operations to retrieve this information. These operations can be used to define test cases that specifically target the presence or absence of an advice. AdviceTracer can be integrated with existing testing frameworks such as JUnit in order to

Copyright © 0 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

develop test cases for PCDs. This paper proposes an approach for testing PCDs based on AdviceTracer to capture faults that miss join points or that match unintended join points.

AdviceTracer is evaluated in terms of its effectiveness and utility for writing test cases that target faults in AspectJ PCDs. The study is performed as a comparison between test cases that use JUnit only and test cases that use AdviceTracer with JUnit. Previous work [6] reported an initial evaluation of AdviceTracer. However, it only considered the ability of test cases that use AdviceTracer to detect faults. Moreover, the subjects used in the study had a small number of PCDs and AdviceTracer was not compared with another technique. This paper reports a comparison using the HealthWatcher system that has 93 classes and 19 PCDs. HealthWatcher is a popular benchmark and has been used in several research studies on aspect-oriented software development. For example, Soares et al. [24] evaluate the impact of AspectJ to implement persistence aspects in HealthWatcher and Hoffman et al. [14] evaluate their proposal for explicit join points.

The empirical study considers two questions. First, the study evaluates the ability of AdviceTracer to decrease the effort for writing the test cases. This effort is evaluated through the number of test cases, the complexity of the test cases, and the precision of the oracle. The second question evaluates the ability of AdviceTracer to improve the fault revealing abilities of test cases. A mutation analysis is performed and the number of mutants killed, the time taken to kill them, and the number of equivalent mutants detected is compared for the two approaches. This study reveals that test cases written with AdviceTracer are simpler than pure JUnit test cases (shorter test scenario, more compact oracles, less lines of code) and detect more faults.

This paper is an extension of the authors' previous work published at ICST'09 [6] and the MUTATION'09 workshop [7]. The major changes are in sections 2.3 and 5. Section 2.3 provides detailed illustrations of the motivations and challenges for testing PCDs in AspectJ with currently available techniques. Section 5 is a completely new empirical study on a mid-size system that compares JUnit and AdviceTracer for testing PCDs in terms of testing time, complexity of the test cases and fault detection abilities.

Copyright © 0 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

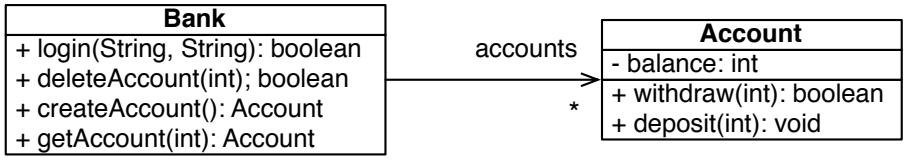

Figure 1. The UML class diagram of the Bank example.

Section 2 presents aspect-oriented programming (AOP) and the challenges to test PCDs. Section 3 details the approach and AdviceTracer, the tool that supports it. Section 4 presents AjMutator, a tool for the mutation analysis of AspectJ PCDs that was used to validate the approach. Section 5 describes an empirical study that was conducted to compare the approach with traditional JUnit testing of PCDs. Section 6 presents the related work. Finally Section 7 concludes this paper.

2. Testing Pointcut Descriptors in AspectJ

Aspect-Oriented Programming (AOP) is a programming paradigm that encapsulates crosscutting concerns into modules called *aspects*. This work uses AspectJ, a popular and mature implementation of AOP for Java. This section first presents the core concepts of AspectJ through a running example. Then, it illustrates the consequence of a fault in the pointcut descriptors (PCDs) and the difficulties in using JUnit to test these PCDs.

2.1. Bank Example

AOP is illustrated through a simple example. Figure 1 shows the UML class diagram of a simple bank system. The `Bank` class has methods to log in, create or delete an account, and a method to get an account with its number. The class Account has two methods to withdraw or deposit money.

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

```
1  public aspect AccessControl {
2     pointcut controlledAccess():
3        execution(* Account.*(int));
4
5     @AdviceName("AccessControl")
6     before(): controlledAccess() {
7        if(!checkAccess(thisJoinPoint.getTarget()))
8           throw new DeniedAccessException();
9     }
10 }
```

Listing 1. An example of an AspectJ aspect for the Bank example.

Listing 1 shows the code of an AspectJ aspect that controls the access to an account to ensure that only authorized accesses are made. Whenever a method of the `Account` class is executed, the aspect checks if the access is authorized; otherwise it throws an exception.

2.2. Aspect Oriented Programming with AspectJ

AOP encapsulates crosscutting behaviors into units called *aspects*. An aspect is composed of *advices* and *pointcut descriptors* (PCDs). A PCD selects a set of points in the execution flow (called *join points*) where an advice is woven. An advice is a piece of code that implements the crosscutting behavior.

In AspectJ, a PCD is defined using primitive PCDs. The primitive PCDs can be combined using conjunction ("`&&`") or disjunction ("`||`"); a PCD can be negated ("`!`"). A primitive PCD selects a set of join points, depending on its parameters.

Some PCDs require a pattern as a parameter. This pattern describes an element in the code, a class, a method, a constructor, or an attribute. Wildcards can be used in a pattern to make it select different elements. The "`*`" wildcard matches any sequence of characters; the "`..`" wildcard matches any sequence of parameters. For instance, the PCD on line 2 of Listing 1 has a pattern that describes a method ("`* Account.*(int)`"). This pattern selects all the methods of the `Account` class that takes a single integer parameter, irrespective of their return type (the first "`*`" wildcard) and their

Copyright © 0 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

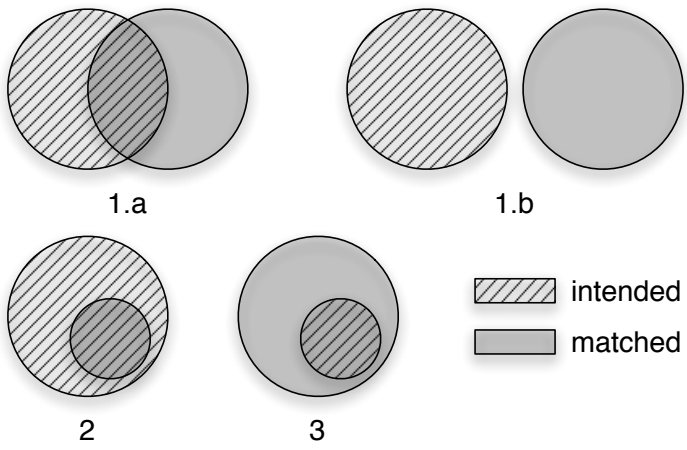

Figure 2. The four types of PCD faults

name (second wildcard). The PCD is an `execution` primitive PCD, which means that it matches the executions of the methods selected by its pattern.

PCDs such as **this**, **target**, **args**, **cflow**, **cflowbelow**, and **if** are dynamic because the exact set of join points can only be computed at runtime. A dynamic PCD always has a static part and a dynamic part. The dynamic part only constrains at runtime the set of join points selected by the static part.

Advices are woven, *before*, *after*, or *around* the join points selected by the PCD. Around means that the join point is not executed and the advice is executed instead; the advice can still execute the join point by calling a special method (`proceed`).

2.3. Challenges for Testing PCDs

When testing a PCD, four kinds of faults, defined by Lemos *et al.* [17] are targeted. Figure 2 illustrates them using a set abstraction. The lined set corresponds to the intended join points and the gray set corresponds to the join points actually matched by the PCD. A fault in the PCD can produce (1) both

Copyright © 0 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

```
public aspect AccessControl {
   pointcut controlledAccess():
      execution(* Bank.*(int));
}
```

Listing 2. Example 1 of a PCD.

```
public aspect AccessControl {
   pointcut controlledAccess():
      execution(boolean Bank.*(..));
}
```

Listing 3. Example 2 of a PCD.

```
public aspect AccessControl {
   pointcut controlledAccess():
      execution(boolean Account.*(int));
}
```

Listing 4. Example 3 of a PCD.

unintended and neglected join points, (2) only neglected join points, or (3) only unintended join points. A PCD with a type 1 fault can match intended join points (1.a) or not (1.b).

Listings 2 and 3 show an example of the type 1.a fault. The PCD in Listing 2 matches `deleteAccount` and `getAccount`, whereas in Listing 3 the PCD matches `deleteAccount` and `login`.

A type 1.b fault occurs if the intended PCD is the one in Listing 2 and the actual PCD is as in Listing 1 because they match completely different sets of methods: Listing 2 matches all methods that have an 'int' type parameter in `Bank`, whereas Listing 1 matches all methods that have an 'int' type parameter in `Account`.

If the intended PCD was as shown in Listing 1 but the actual PCD used was the one shown in Listing 4, then this would be a type 2 fault because the actual PCD matches only return types 'boolean' whereas the intended PCD matches all return types. On the other hand, an example of type 3 fault is where the intended PCD is as shown in Listing 4 and the actual PCD is in Listing 1.

Copyright © 0 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

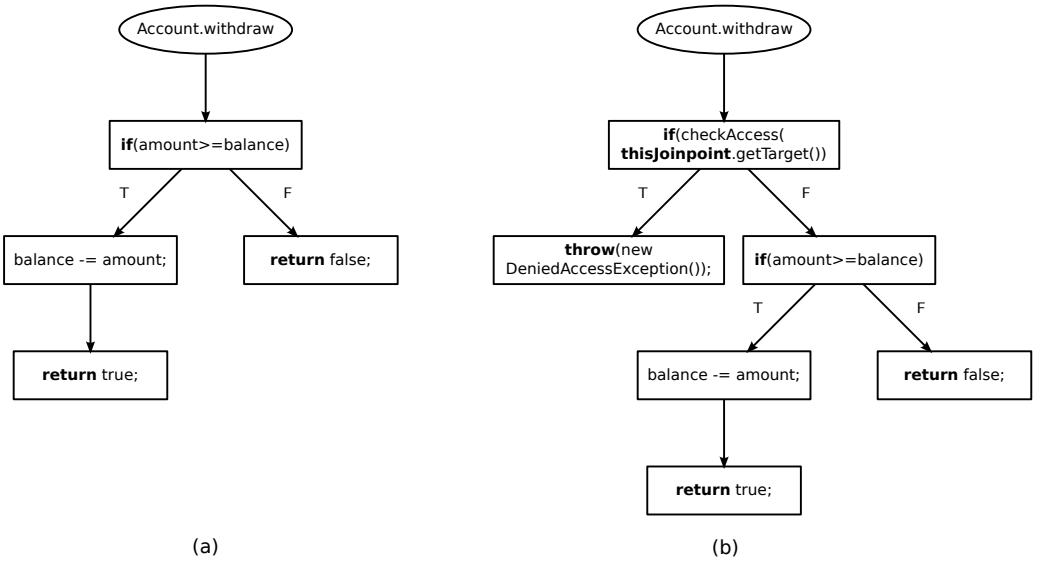

Figure 3. The control flow graphs of the `Account.withdraw` method before (a) and after (b) weaving.

Testing a PCD is both important and challenging. It is important because a faulty PCD can introduce numerous types of faults in the program, some harmful. It is challenging because it is difficult to write oracles that check if a PCD is correct.

A faulty PCD can have dangerous effects. The control flow graph of Figure 3 shows the control flow graph of the `Account.withdraw` method before weaving (Figure 3(a)), and after the `AccessControl` aspect shown in Listing 1 is woven (Figure 3(b)). These graphs represent all potential execution paths in the method. Rectangular nodes in the graph represent statements, oval nodes represent the entry points to the methods, and edges represent potential transitions between statements.

The `AccessControl` aspect modifies the control flow. An *if* statement has been added at the beginning of the control flow. If the condition is fulfilled an exception is thrown, otherwise the original control flow is executed. This means that in some cases the original behavior may not be executed

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

```
1 public void testAccessControl() {
2    bank.login(nonAuthorizedUser,password);
3    try {
4       account.withdraw(30);
5       fail("Access␣should␣not␣be␣authorized");
6    } catch(DeniedAccessException() {}
7 }
```

Listing 5. A test case for the `AccessControl` aspect.

anymore. If the PCD is not correct it can introduce critical problems. If the PCD matches more join points than it should, then access to functionalities of the system will be reduced by the advice. Even worse, if the PCD does not match the intended join points, it introduces a critical security hole, as anyone can withdraw money from any account.

The test case in Listing 5 illustrates the challenges of testing a PCD using JUnit only. Testing that the PCD is correct requires checking that the advice has been woven at the correct places. One way to do it is to check that the behavior of the advice executes correctly. The test case logs a user who is not authorized to withdraw money from a particular account, and attempts to withdraw money on that account. If an exception is thrown, the test case ends and succeeds, but if no exception is thrown, then the test case fails (the assertion of line 5 always fails).

This test case is simple and seems to be a good way to test the PCD, but there are several problems with this test case. The first problem is that if the test case fails, the oracle is not precise enough to conclude if there is a fault in the PCD or in a different part of the program. This can hinder fault localization.

- If the first "`*`" wildcard of line 3 of Listing 1 is replaced by "`void`", the `withdraw` method is not selected, and `testAccessControl` will fail because there is a fault in the PCD.
- If on line 7 of Listing 1 the negation ("`!`") is removed, the behavior of the advice is inverted, and `testAccessControl` will fail. However, the test failure will reveal a fault in the advice, but not in the PCD, which is correct.

Copyright © 0 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

The second problem is that `testAccessControl` will not detect all the faults in the PCD. For example, suppose that in the method pattern of Listing 1, line 3, the name `Account` is replaced by a wildcard. Then the advice will be woven before the executions of `Bank.deleteAccount`, and `testAccessControl` will pass even if there is a fault in the PCD. This fault is of type 3 as defined in Figure 2, which means that the expected join points are matched, but the PCD also matches unintended join points. To detect that fault, test cases must be written that specify that no advice should be woven in some places of the code. However, test cases such as `testAccessControl` cannot check this, they do not explicitly mention the presence or absence of an advice by testing its behavior; they just implicitly assume the presence of an advice. Not allowing explicit mention of an advice is a limitation of JUnit-based unit testing when it is used to precisely capture faults in the PCD.

Finally, a fault in a PCD can lead to the same observable behavior as the original PCD. In this case, it is not possible to write a JUnit test case that can detect different functional behavior between the original and the faulty PCD. One could argue that if functionalities are the same there is no fault. However, the set of matched join points is actually different and it can lead to problems with non-functional requirements, such as higher execution time or space. Moreover, as we show in Section 5, with the proposed approach, it is possible to exhibit the fault with a test case.

To overcome these problems, a new approach is needed to define oracles that directly target faults in the PCDs, and that does not rely on the behavior of the advice. The next section proposes a solution for the definition of oracles dedicated to PCDs. This solution is based on monitoring the execution of advices when running the test cases and on the definition of a set of dedicated assertions to query these executions. This solution is implemented as a tool called AdviceTracer and the assertions can be used to define an automatic oracle for PCD test cases.

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

3. AdviceTracer

The AdviceTracer[†] tool allows a programmer to write test cases that focus on checking whether or not a join point 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 join points expected to be matched by a PCD without requiring the execution of advice behavior.

3.1. Approach

AdviceTracer implements an approach for testing PCDs. The key idea is that executions of the advice during test scenarios are traced and assertions are made on these execution traces instead of on the behavior of the advices.

The general approach is separated in two steps, one for checking that all intended join points have been matched by the PCDs, and one for checking that no unintended join points have been matched.

The following paragraphs describe (1) how to write the test cases for the intended and unintended join points, and (2) how to interpret the results of these test cases to locate the faults. Each test case consists of a test scenario and an oracle (using assertions).

3.1.1. Test cases for the intended join points

In the first step, test cases are written to check that all intended join points are matched by the PCDs. It is necessary to write one test case for each join point. While it may seem demanding, a previous study [22] showed that most of the advices are woven only at few places. Thus, there are usually only a few test cases for the set of join points that are intended to be matched by one PCD.

[†] `http://www.irisa.fr/triskell/Softwares/protos/advicetracer`

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

The test scenario of a test case that targets an intended join point should be as simple as possible. The sole purpose of the test scenario is to trigger the intended join point.

The oracle should verify that the advice associated with the PCD under test has been executed, and that it has been executed at the correct place. Thus, the assertions need to be specific to make sure the intended join point has been matched by the PCD.

Enumerating the intended join points contradicts one of the goal of using declarative PCDs (such as AspectJ PCDs), which is to avoid the enumeration of numerous join points. However, there exists no other specification for a PCD against which we can check conformance. Thus, enumerating all the join points is the only way to thoroughly verify the PCDs. One advantage of using AdviceTracer is that it allows for a coarser verification by just counting the number of matched join points, which greatly reduce the cost.

3.1.2. Test cases for the unintended join points

In the second step, test cases are written to check that no unintended join points have been matched by the PCDs. The unintended join points can be located anywhere in the program so it can be difficult to write such test cases. The proposed approach can reduce the cost of writing test cases for the unintended join points.

In order to limit the number of test cases that validate the absence of unintended join points, it is necessary to build test cases that cover large portions of the program. Since AdviceTracer is based on execution traces, the larger the execution trace, the more can one test case check. AdviceTracer offers an assertion that checks the number of occurrences of a specific advice in an execution trace. It can be used to validate that the advice has not been woven more than expected in large execution traces.

Determining the expected number of executions of an advice in a specific test scenario is not an easy task but not harder than determining what assertions need to be specified in JUnit test cases. The tester needs to determine this number of executions by inspecting the code or by using the specifications of the system and the aspects (e.g., design models and requirement documents).

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

It might seem difficult to interpret a result based on a simple count of occurrences of advices. Indeed, if the number of executed advices is as expected by the test case, it may still not be clear if they are actually executed at the correct join points. That is why it is important to test the intended join points before the unintended ones. If it is known that the intended join points were correctly matched, then the number in a large execution trace must correspond to the intended join points (and not to any arbitrary join point). This way, the oracle remains simple even if the test scenario is complex.

Testing that a PCD does not match any unintended join point is more difficult than testing that the PCD matches every intended join point. In general, there are far more unintended join points than intended join points. Indeed, each instruction contains at least a possible join point, and can contain dozens of possible joinpoints. Thus if the first step fails to guarantee that intended join points have been matched, testing that no unintended have been matched is almost impossible. This is why we propose a two-step approach. the first step is required for the second step to succeed.

3.1.3. Fault localization

If a fault is detected, it is possible to know which type of fault (as presented in figure 2) and where it is located by interpreting the test results.

First, the tester inspects the results of the test cases for the intended join points. If some test cases for the intended join points fail, it means that the intended join points they target have not been selected by the PCDs they test. The fault is easily located as each test case targets one PCD and one intended join point.

Then the tester inspects the results of the test cases for the unintended join points. If some test cases for the unintended join points fail, the PCDs they target match unintended join points. Each assertion in the test case specifies how many times an advice must be executed. If an assertion fails, then the PCD associated with the advice checked by the assertion matches unintended join points.

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

```
public class C {
  public void m1() { ... }
  public void m2() {
    ...
    m1();
  }
}
```

Listing 6. A Java class example

```
public Aspect A {
  @AdviceName("A1")
  before(): execution(void C.m1())
     && cflow(execution(void C.m2())) {
    ...
  }
}
```

Listing 7. An AspectJ aspect example

For the unintended join points, the tester must find a compromise between large and small test scenarios. A large test scenario means fewer test cases, but a small test scenario means better fault localization.

To summarize, if some test cases for the intended join points fail, then there is a fault of type 1 or type 2, and if some test cases for the unintended join points fail, then there is a fault of type 1 or type 3.

3.2. Illustrative Example

Listings 6, 7, and 8 illustrate the use of AdviceTracer. Listing 6 shows a Java class `C` with two methods `m1` and `m2`; `m2` calls `m1`. Listing 7 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`.

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

```
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("A1",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("A1",1);
20     assertExecutedAdviceAtJoinpoint("A1","C.m1:2");
21 }
```

Listing 8. A JUnit test class illustrating how to use AdviceTracer

To facilitate the identification and designation of advices it is required to use the `@AdviceName` annotation. This annotation, provided by AspectJ, takes a string parameter to specify the name of the advice it is associated with.

Listing 8 shows two test cases that specify the set of expected join points. In textual form, the specification of the expected join points 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 the test 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`.

- `test1` specifies that no advices should be executed (line 9) when only `m1` is called (line 7). Thus, `test1` will pass only if weaving an aspect (here aspect `A`) does not lead to the execution

Copyright © 0 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

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

- `test2` calls `m2` and then specifies (at lines 19-20) that advice `A1` should be executed from a join point at line 2 of `C`, within `m1`. So if `test2` passes, it is clear that the advice was executed within the context of execution of `m1`.

If the PCD of Listing 7 is replaced by "`execution(void C.m1())`", `test1` fails because the advice is executed in this test scenario. If the pointcut is replaced by "`execution(void C.m2())`", the advice is executed but `test2` fails because it is woven within `m2` instead of `m1`. Finally, if the pointcut is replaced by "`call(void C.m1()) && cflow(execution(void C.m2()))`", both test cases will fail as the weaving context changes: with `call` the join point is located within the calling method whereas with `execution` the join point is the called method.

This example also illustrates how AdviceTracer can handle dynamic PCDs. The PCD of Listing 7 is dynamic: it statically matches all the executions of `m1`, but the advice is executed only if `m1` is executed in the control flow of `m2`. To specify this PCD, the test first executes `m1` outside the control flow of `m2` and checks that the advice is not executed (`test1` of Listing 8). Then the test executes `m1` in the control flow of `m2` and checks that the advice is executed at the correct join point (`test2` of Listing 8).

AdviceTracer makes it possible to specify the expected join points in order to check that there are no neglected join points. 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 join points.

3.3. 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.

Copyright © 0 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

3.3.1. Starting AdviceTracer

By default, AdviceTracer does not trace anything. To make assertions on the number of executions of the advices, AdviceTracer needs to know when the test scenario begins and when it ends. Thus, in each test case, AdviceTracer must be set on before the test scenario, and off thereafter.

AdviceTracer is set on by calling the static method, `setAdviceTracerOn`. The static method, `setAdviceTracerOff`, stops tracing. Between `setAdviceTracerOn` and `setAdviceTracerOff`, AdviceTracer stores information about which advices were executed and where they were executed.

3.3.2. Restricting the traced advices

Using AdviceTracer each test can specify the advices to be traced. If a test case does not specify a particular advice, then all the advices are traced. On lines 5 and 15 of Listing 8, 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 8, `test1` specifies the absence of advice `A1`, and `test2` specifies the presence of advice `A1`.

The test cases `test1` and `test2` are said to be *modular* because they specify a set of join points 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 some 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 during the course of software evolution, a new advice is woven within `C.m1`, then the test cases of Listing 8 will still pass because they only consider advice `A1`.

Copyright © 0 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

3.3.3. *Assertions provided by AdviceTracer*

AdviceTracer provides three new assertions that are extensions of JUnit assertions.

assertAdviceExecutionsEquals(String advice, int n) : Passes if the number of executions of the specified advice is equal to n, fails otherwise. This assertion is needed to detect unintended join points, as explained in Section 3.1.

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 join point, fails otherwise. The format of the join point parameter is: `className.methodName:lineNumber` where `lineNumber` refers to the line number in the source where the join point is expected. The class name and the line number alone are sufficient to identify a join point, but the method name makes it easier to understand the assertion.

3.4. Implementation of AdviceTracer

The monitoring part of AdviceTracer is implemented with an aspect, which can be woven at compile time or at runtime. Compile time weaving offers better performance, but runtime weaving allows running the test case without modifying the byte code.

Information that is stored by AdviceTracer can be accessed through an API that offers a set of static methods used to define test cases. Figure 4 shows the class diagram of AdviceTracer. The advice in this aspect retrieves the name of the advice that is being traced and the location of the join point 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 join point (e.g., `C.m1:2`). The string for the advice is its name, specified with the `@AdviceName` annotation. The string for the join point is built

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

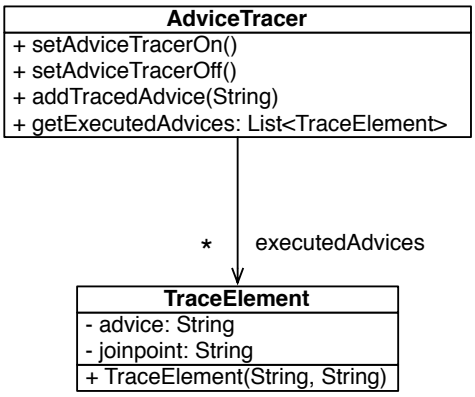

Figure 4. The class diagram of AdviceTracer.

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 join point 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. Figure 5 shows the control flow graph of the `Account.withdraw` method after AdviceTracer has been woven. AdviceTracer is woven before the execution of other advices. As there are no conditional statements in the added control flow, AdviceTracer do not affect the behavior of the program with its own aspect. Here, when `Account.withdraw` is executed, AdviceTracer is executed first, then the `AccessControl` advice. The original control flow may or may not be executed, depending on the `AccessControl` 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

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

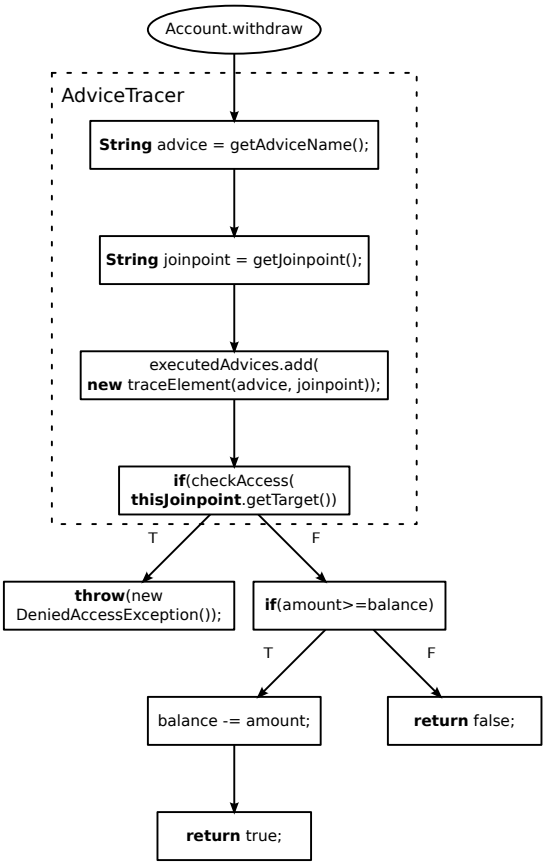

Figure 5. The control flow graph of the `Account.withdraw` method after AdviceTracer has been woven.

it only contains the `TraceElement` objects corresponding to the advices that were traced between the last calls to `setAdviceTracerOn` and `setAdviceTracerOff`.

4. AjMutator: A Tool for Mutation Analysis of PCDs

Mutation analysis is useful for evaluating testing techniques targeting faults in the PCD. Mutation analysis is a means to quantify the ability of a test suite to detect faults. In the empirical study reported

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table I. Operators implemented in AjMutator.

Operator	Description
PCCC	Replaces a `cflow` PCD with a `cflowbelow` PCD, or the contrary
PCCE	Replaces a `call` PCD with an `execution` PCD, or the contrary
PCGS	Replaces a `get` PCD with a `set` PCD, or the contrary
PCLO	Changes the logical operators in a composition of PCDs
PCTT	Replaces a `this` PCD with a `target` PCD, or the contrary
POEC	Adds, removes or changes exception throwing clauses
POPL	Changes the parameter list of primary PCDs
PSWR	Removes wildcards
PWAR	Removes annotation from type, field method and constructor patterns
PWIW	Adds wildcards

in Section 5, mutation analysis was used. Thus, a tool for the mutation analysis of AspectJ PCDs was developed.

In mutation analysis [8, 11], mutation operators produce mutants of the original program. A mutant is a modified version of the program where one and only one simple fault has been introduced by the operator. Each operator introduces one type of fault.

Mutation analysis relies on the assumption that if a set of test cases can kill all the mutants, then it is adequate for testing the original program. A mutant is considered killed if the result of the execution of the test cases on the mutant is different from that of the execution of the test cases on the original program. The operators introduce faults corresponding to identified fault types wherever possible in the original program. If all the mutants are killed, the set of test cases can detect all the introduced faults.

Mutation analysis results in a mutation score which indicates the adequacy of a test suite. The mutation score is the number of killed mutants divided by the number of non-equivalent mutants. Equivalent mutants are mutants that are semantically equivalent to the original program. The higher the mutation score, the better the adequacy of the test suite.

A mutant PCD is a PCD that matches a set of join points different from the set of join points matched by the original PCD. Ferrari *et al.* [9] have presented several operators for the mutation analysis of

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

AspectJ programs. These operators are based on fault types that have been identified in previous work. The authors presented three kinds of operators: operators for PCDs, operators for AspectJ declarations, and operators for advice definitions and implementations. This work only considered operators for PCDs. Table I shows the operators that were implemented in AjMutator.

4.1. Classification of the mutants

The mutant PCDs are automatically classified by comparing the set of join points they match with the set of join points matched by the original PCD. The classification of the mutants is similar to the classification of the faults of Figure 2, in Section 2.3.

A mutant that matches unintended join points and does not match some intended join points, and thus introduces a type 1 fault, is a mutant of class 1. A mutant that does not match some intended join points, and thus introduces a type 2 fault, is a mutant of class 2. A mutant that matches unintended join points, and thus introduces a type 3 fault, is a mutant of class 3. Finally a mutant that matches the same set of join points as the original is an equivalent mutant, as it introduces no fault.

It is important to note that the classification of a mutant is system-dependent, i.e., the same mutant can have two different classifications in two different systems, even if the original PCD is the same. Adding or removing lines of code can add or remove join points, and can thus change the set of join points matched by a PCD.

4.2. AjMutator

AjMutator is a tool for the mutation analysis of AspectJ pointcut descriptors. It produces mutants of AspectJ aspects by inserting faults in the PCDs.

AjMutator consists of three distinct parts, corresponding to the following three phases:

1. the generation of mutant source files from the AspectJ source file
2. the compilation of the mutant source files
3. the execution of test cases on the mutants to calculate the mutation score.

Copyright © 0 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

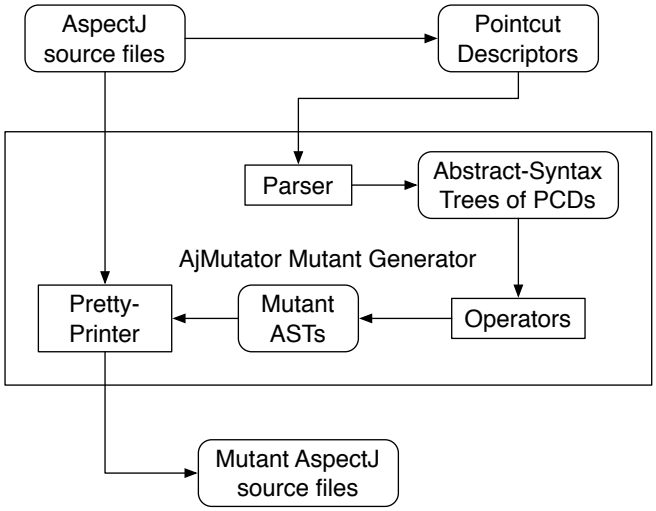

Figure 6. The AjMutator process of generation of the mutants.

4.2.1. Generation of the mutants

Figure 6 shows how AjMutator generates the mutants. A parser builds an abstract-syntax tree (AST) for each PCD in the AspectJ source files. The operators insert faults in copies of the AST, so there is an AST for the original PCD and an AST for each mutant PCD. A pretty-printer then produces a mutant AspectJ source file for each mutant AST.

The parser has been developed using SableCC [5], an open-source compiler generator. The parser directly produces an AST from a PCD. Only the PCD is parsed in the AspectJ source file, as the PCD has a different grammar from the rest of the AspectJ syntax, which is close to the Java syntax.

The mutation operators are implemented using the visitor pattern [10]. Each operator extends the abstract class Operator, which is a visitor for the AST. When an operator encounters a place in the AST where a fault can be injected, a mutant of the AST is generated. The mutant AST is then inserted in a copy of the original AspectJ source file using a pretty printer. To reduce the memory consumption the

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

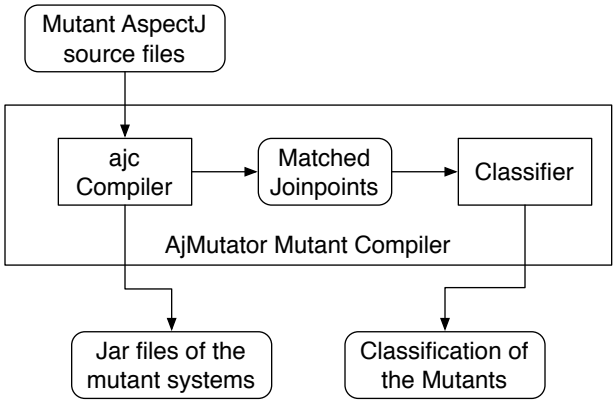

Figure 7. The AjMutator process of compilation and classification of the mutants.

mutant AST is immediately saved into a file and the reference is not kept so that the garbage collector can free its memory space.

4.2.2. *Compilation of the mutants*

After the mutants have been generated, they need to be compiled. The compilation is required to run the test cases on the mutant systems, but also to classify the mutants automatically.

Figure 7 shows the process of compiling and classifying the mutants. It relies on the ajc compiler [‡], the compiler of AspectJ. The ajc compiler produces a jar file of the system for each mutant. It also provides information on the join points matched by the PCDs. The information is then used by AjMutator to classify the mutants.

As a single modification of an aspect can affect various classes throughout the whole system, it is necessary to keep a complete version of the system for each mutant, not only the class file of the mutant aspect.

[‡] `http://www.eclipse.org/aspectj`

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

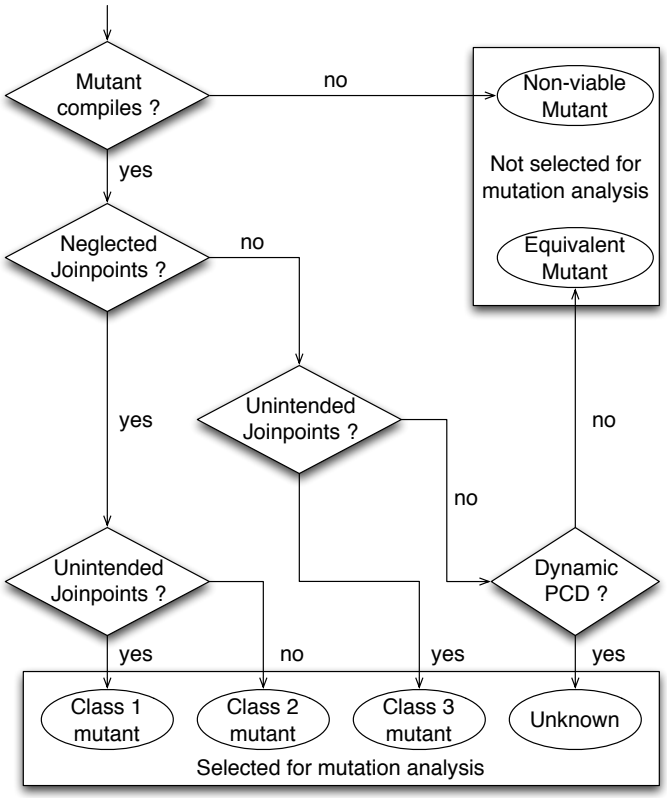

Figure 8. Process for classifying the mutants.

Figure 8 shows the process for the classification of the mutants. Mutation analysis only considers compilable mutants. Thus, if the mutant is non-compilable, it is not selected as for mutation analysis. If the mutant system has both neglected and unintended join points, the mutant is in class 1. If the mutant system has neglected join points but no unintended join points, the mutant is in class 2. If the mutant system has unintended join points but no neglected join points, the mutant is in class 3. Finally, if the mutant system has no unintended join points and no neglected join points, the mutant is equivalent and not selected for mutation analysis. The accuracy of the classification process depends on whether the original PCD of the mutant is static or dynamic.

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Static PCD: the set of matched join points can be computed statically, so the classification is exact.

Dynamic PCD: the set of matched join points can only be over-approximated, so the classification is not perfectly accurate. Two cases may arise:

Non-equivalent: if the mutant is classified as non-equivalent, it means that the fault was inserted in its static part, thus it is actually non-equivalent.

Equivalent: if the mutant is classified as equivalent it just means that the static part is semantically equivalent, but the dynamic part might be different. In this case, the mutant must be selected for mutation analysis because one cannot be sure that it is actually equivalent.

4.2.3. Execution of the test cases

A mutant is considered killed if a test suite can exhibit a difference between the original system and the mutant system. So if the mutant is killed, the test suite can detect the inserted fault, whereas if the mutant is still alive, the test suite is not able to detect this fault and it needs to be improved, either with new test data or better oracles.

AjMutator assumes that all JUnit test cases succeed on the original version. Thus, a mutant is killed if at least one assertion in a JUnit test case is violated when the test case is executed against the mutant.

5. Empirical study

The goal of the empirical study is to determine if AdviceTracer actually improves the testing of PCDs. This is done by comparing two test suites, one written only with regular JUnit assertions, and the other written using AdviceTracer and JUnit. The two test suites specify the pointcut descriptors of the same system under test. In the following, the JUnit-only test suite is called T_{JU}, and the AdviceTracer test suite is called T_{AT}.

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table II. Number of occurrences of each primitive PCD in the HealthWatcher system, as well as the locations of the advices.

Primitive PCD	**Number of Occurrences**
adviceexecution	0
args	12
call	2
cflow	0
cflowbelow	0
execution	10
get	0
handler	0
if	0
initialization	8
preinitialization	0
set	0
staticinitialization	0
target	12
this	1
within	3
withincode	0
Advice location	**Number of Occurrences**
before	3
after	13
around	4

5.1. The HealthWatcher system

The system under test is the HealthWatcher system[§]. This system has been developed at the University of Lancaster in the context of the AOSD European Network of Excellence. It was initially intended to serve as a benchmark in the Network to compare different AOSD techniques at different levels in the development process. It has then been reused in various experiments outside the Network [14, 24].

It is a web-based application that collects and manages health complaints. It has been implemented in different languages, including AspectJ. There are different versions of the HeatlthWatcher system,

[§] `http://www.comp.lancs.ac.uk/~greenwop/tao/implementation.htm`

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

which vary according to various design decisions and functionalities. This study used a version which incorporates the state and command patterns [10].

The HealthWatcher software was slightly modified for this study. In the HealthWatcher system, there is a persistence mechanism which can be turned on or off. The persistence mechanism relies on a distributed data base that is only compatible with Microsoft Windows. As the developers do not give any information on the structure of the data base (tables, fields, etc.), it is not possible to create or reset the database. The only way to use the persistence is to use Windows and the given data base file, which means that the state of the database at the beginning of a test case is the state of the database at the end of the previous test case. This can introduce a lot of side effects between the test cases and reduce the testability, so it was turned off. Four aspects were also removed because they have no effect when the persistence is turned off.

Two design patterns are used in the HealthWatcher system. The first pattern is the state pattern [10]. This pattern encapsulates the state of an object in another object. It is used to manage the different states of a complaint. As there are different types of complaints, different state classes coexist in the system. The second pattern [10] is the command design pattern, which encapsulates a method call into an object. In the HealthWatcher system, commands are used to represent different requests that can be received by the server. For each request there is a corresponding class and an instance of each class is stored in a table and called depending on the received HTTP requests.

The HealthWatcher system contains nine aspects, which fulfill four different goals. Two aspects are in charge of synchronization to avoid concurrent access on some classes. Two aspects manage exceptions in the HealthWatcher by catching thrown exceptions. One aspect implements the command design pattern by registering the set of available commands. Finally, four aspects implement the state design pattern to add a state object to the different types of complaints.

The system test has 93 classes, with 530 methods, 9 aspects, and 19 advices (and thus 19 PCDs). It has 6435 total lines of code, of which 438 lines of code are for the aspects. Table II shows the number

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

of occurrences for each primitive PCD in the HeatlthWatcher system. It shows that some primitives are not represented, while others are used often.

5.2. Research Questions

This empirical study attempts to answer the following two research questions:

Question 1 *Does AdviceTracer reduce the testing effort for testing PCDs compared to JUnit ?* Testing PCDs using only JUnit assertions can be difficult, AdviceTracer aims to reduce the effort required to write the test cases. Thus, the study verifies that AdviceTracer actually achieves this goal.

The testing effort can be evaluated by comparing the time spent on writing the test suites. When testing a system, time is often critical. Also, time is often a good indicator of the required effort.

The testing effort can also be evaluated by comparing the size of the two test suites, in terms of number of test cases and in terms of the number of lines of code. If a test suite detects the same faults with fewer test cases, it means that its test cases are more effective as they detect more faults. On the other hand, if a test suite detects the same faults with smaller test cases, it is likely that its test cases are simpler to write. Thus, the study compares the number of test cases, the total number of lines of code, and the average number of lines per test case.

Question 2 *Does AdviceTracer increase the ability of the test cases to detect faults in the PCDs compared to JUnit ?* Some faults may be hard or impossible to detect using only JUnit assertions. As AdviceTracer targets testing often PCDs, it is necessary to verify that the AdviceTracer test suite can detect at least as many faults as the JUnit test suite. Since mutation analysis is used here, the study compares JUnit and AdviceTracer with respect to the difficulty at detecting equivalent mutants.

Copyright © 0 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

5.3. Experimental Protocol

Delamare and Gupta, two graduate students and co-authors of this paper, built the two test suites in parallel. They have comparable advanced programming skills in Java, AspectJ, and JUnit. None of them were familiar with the HealthWatcher system.

The developers were given a set of mutants of the HealthWatcher system. One was asked to detect all the faults with JUnit and the other with AdviceTracer. Both of them had to record the time taken to detect all the faults. One developer recorded the time to test each class, whereas the other recorded the number of mutants killed after each hour. They were given no instruction on the order in which the classes were to be tested.

The oracle the developers had to write was related to the PCD, so the oracle had to capture how well the advices were woven. In order to write the oracle, the developers had to understand the interactions between the aspects and the base code, not so much the global functionality of HW.

The generation of the two test suites had to be driven by a certain test adequacy criterion. The mutation score was used as a criterion for the test suites. The goal was to achieve a 100% mutation score if possible, or the highest possible mutation score otherwise. Before stating that a mutant was not possible to kill, a developer had to found the reason why. This mitigates the risk that a team gives up the experiment early on simply because they cannot kill the mutant. Mutants of the pointcut descriptor were generated using AjMutator as described in Section 4.

AjMutator generated 343 mutants. AjMutator is able to automatically detect some of the equivalent mutants, and some mutants are not compilable. After compiling the mutants, 243 were left as compilable and not detected as equivalent by AjMutator. There were, however, equivalent mutants remaining that were not detected by AjMutator.

The two developers had to find the equivalent mutants on their own, without communicating the results one to another. Because finding the equivalent mutants can take a lot of time, it would influence the results if the developers shared the results. At the end of the study, 76 mutants were identified as non-equivalent.

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table III. Metrics of the two test suites.

	JUnit only	**AdviceTracer**
Test Cases	26	24
Lines of Code	1298	324
Lines per Test Case Mean	49.92	13.50
Lines per Test Case Standard Deviation	27.66	7.76
Time (minutes)	480	420

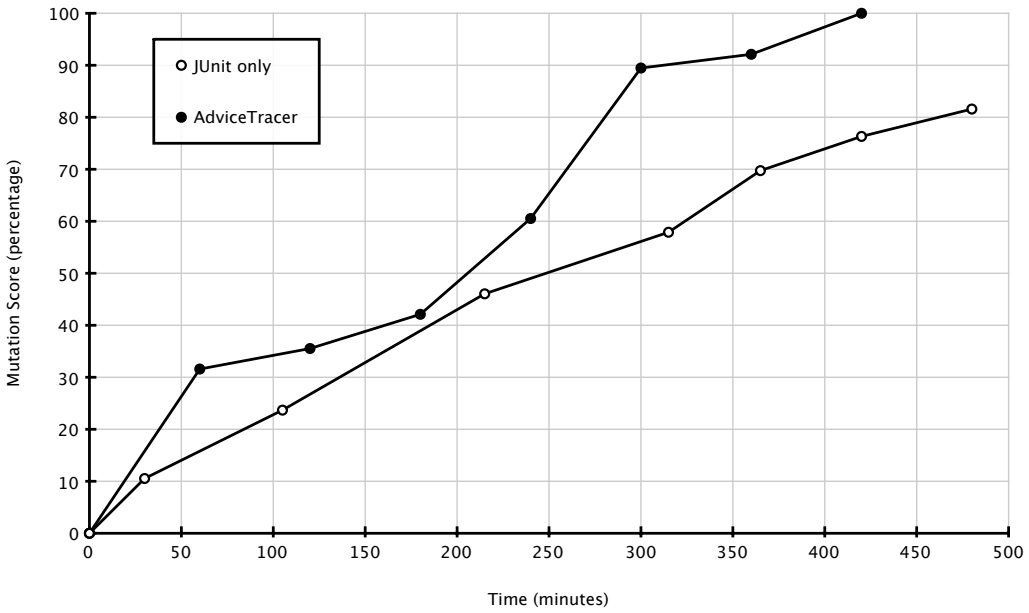

Figure 9. Evolution of the mutation score for each test suite.

5.4. Results

5.4.1. Question 1

The T_{JU} test suite took longer to write. The T_{AT} test suite was written in approximately 420 minutes (seven hours), whereas T_{JU} was written in approximately 480 minutes (eight hours). Figure 9 shows

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

the evolution of the mutation score for the two test suites. The mutation score of the T_{AT} is always higher than the mutation score of T_{JU}, which means that it is faster to find faults with AdviceTracer.

Table III shows metrics of the two test suites. Although they have a similar number of test cases (26 for T_{JU}, 24 for T_{AT}), T_{JU} has significantly more lines of code than T_{AT} (respectively 1298 and 324 lines of code).

The test cases written with AdviceTracer are smaller than the test cases written only with JUnit. JUnit only test cases have a mean of 49.92 lines per test cases whereas AdviceTracer test cases have a mean of 13.50 lines per test cases. The standard deviation of the lines per test cases is also smaller with AdviceTracer, with 7.76 compared to 27.66 with JUnit only. This means that AdviceTracer test cases have very similar sizes (from 7 to 30 lines of code) whereas JUnit only test cases can have very different sizes (from 19 to 137 lines of code). There is no obvious relation between a PCD and the size of the test cases that test it.

In order to illustrate why JUnit test cases can become much larger than AdviceTracer test cases, the development of test cases for one PCD in HealthWatcher is analyzed in detail in the following paragraphs.

Listings 9 and 10 show two examples of test case that test the PCD of the advice `AnimalComplaintStateAspect3` (Listing 11), in `AnimalComplaintStateAspect`. This advice is responsible for changing the state of an `AnimalComplaint` object when `AnimalComplaintState.setStatus` is called. The HealthWatcher system manages different kinds of complaints. An `AnimalComplaint` is a type of complaint relative to an animal. It contains information, such as the type of animal, date of event, date of the complaint. The status can be tracked, i.e., if the complaint is still being investigated or if it has been dealt with.

When using JUnit only (Listing 9), the test case must check that the state is correctly changed when `setStatus` is called. The problem is that the state is a private attribute. The only way to check that the state has changed is to trigger a method whose behavior changes according to the state. This is the case of the method `setAnimal` which actually changes the string attribute `animal` if

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

```
1  public class JUnitTest extends TestCase {
2     public void testAnimalComplaintStateAspect3() {
3        Employee employee = new Employee("login",
4                                "password", "name");
5        Date date1 = null;
6        Date date2 = null;
7        Date date3 = null;
8        try {
9           date1 = new Date(21, 7, 2009);
10          date2 = new Date(22, 7, 2009);
11          date3 = new Date(23, 7, 2009);
12       }
13       catch(InvalidDateException e) {
14          fail("Invalid␣date");
15       }
16       Address address = new Address();
17
18       AnimalComplaint complaint = new AnimalComplaint(
19          "complainant", "description", "observation",
20          "e-mail", employee, open, date1, date2, address,
21          date3, "elephant", address);
22
23       AnimalComplaintState state = new
24          AnimalComplaintStateOpen();
25       state.setAnimal(complaint.getAnimal());
26
27       state.setStatus(closed, complaint);
28       complaint.setAnimal("tiger");
29
30       assertEquals("elephant", complaint.getAnimal());
31    }
32 }
```

Listing 9. Example of JUnit only test case for the `AnimalComplaintStateAspect` aspect.

the complaint is opened, but does nothing if the complaint is closed. So the first thing to do is to create an `AnimalComplaint` object whose state is open (line 18). This constructor requires a list of parameters, which are initialized from line 3 to line 16. The complaint is initialized with “elephant” as the value for the `animal` attribute. On line 27 the complaint is closed. On line 28 the test case tries to

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

```
public class AdviceTracerTest extends TestCase {
   public void testAnimalComplaintStateAspect3() {
      AnimalComplaint complaint = new AnimalComplaint();
      addTracedAdvice("AnimalComplaintStateAspect3");
      setAdviceTracerOn();
      complaint.setStatus(closed);
      setAdviceTracerOff();
      assertAdviceExecutionsEquals("AnimalComplaintStateAspect3",1);
      assertExecutedAdviceAtJoinpoint(
            "AnimalComplaintStateAspect3",
            "healthwatcher.model.complaint.AnimalComplaint.setStatus
                :54");
   }
}
```

Listing 10. Example of AdviceTracer test case for the `AnimalComplaintStateAspect` aspect.

```
@AdviceName("AnimalComplaintStateAspect3")
after(int status, AnimalComplaint animalComplaint,
      AnimalComplaintState state):
      execution(void AnimalComplaintState+.setStatus(int,
                                    AnimalComplaint)) &&
      args(status, animalComplaint) && target(state) {
   if(status == open){
      animalComplaint.setComplaintState(
         new AnimalComplaintStateOpen(state)
      );
   } else if(status == closed){
      animalComplaint.setComplaintState(
         new AnimalComplaintStateClosed(state)
      );
   }
}
```

Listing 11. The `AnimalComplaintStateAspect3` advice.

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

change the value of `animal` to “tiger”. If the state has been correctly changed by the advice, the value of `animal` should remain “elephant”, as verified by the assertion on line 30.

AdviceTracer offers a set of assertions to check the presence of an advice, which allows the tester to specifically target faults in the PCD and reduce the complexity of the test cases. Since it is not required to test the behavior of the object under test, the empty constructor can be used to instantiate a complaint, as on line 3. On line 4, the test case specifies that only the `AnimalComplaintStateAspect3` advice should be traced. Then on line 5 AdviceTracer is set on. The method `AnimalComplaint.setStatus` is called on line 6; this method calls `AnimalComplaintState.setStatus` where the advice should be woven. Once the test scenario is over, the AdviceTracer is set to off in line 7. Finally, there are two assertions on lines 8 and 9. The first assertion checks that the advice has been executed once, and the second assertion checks that it has been executed from the correct location.

This example shows several benefits of AdviceTracer. The test scenario is simpler with AdviceTracer. As it is not necessary to check the behavior of the advice, test cases only have to trigger behaviors where the advice is expected. On the other hand, using JUnit alone, it is necessary to set up a complex test scenario to be able to observe the behavior of the advice. The oracle written with AdviceTracer is more precise. The AdviceTracer test case can pass only if the PCD captures the correct set of join points, and thus if the advice has been woven at the correct place. The JUnit-only test cases pass if the value of `animal` has not changed, which can occur both because the advice is woven at the correct place but its behavior is faulty, or because the advice is not woven at the correct place.

In summary, AdviceTracer reduces the required effort to test the PCDs. The test cases written with AdviceTracer were written faster than the test cases written only with JUnit. AdviceTracer test cases are also simpler, as they have smaller size than JUnit test cases.

Copyright © 0 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

5.4.2. *Question 2*

The mutation score of the AdviceTracer test suite is greater than the mutation score of T_{JU} (Figure 9). T_{AT} achieved a 100% mutation score, which means it was able to kill all the mutants, and thus the test suite detects all the inserted faults. T_{JU} only achieved a 81.59% mutation score. This means that T_{JU} was not able to detect all the faults. The following paragraphs discuss the faults that were not detected.

There are 14 mutants that were not killed by T_{JU}, although they were killed by T_{AT}. These mutants are impossible or very hard to kill without AdviceTracer.

Four of the mutants not killed by T_{JU} are mutants of the aspects that implement the state design pattern. After the initialization of an object, its state is initialized by the advice. With the mutants the advice is executed several times, which can be detected by AdviceTracer, with the assertion on line 8 of Listing 10. For T_{JU} it is not possible as the result of the initialization is the same with the mutants as with the original.

Two of the mutants not killed by T_{JU} are mutants of the aspect that implement the command design pattern. The commands are registered in the table at the initialization of the system. One mutant weaves the advice in several unintended join points so the table is reinitialized several times, but as the command table is private and not accessible with public methods, it is not possible to kill this mutant without AdviceTracer. The other mutant weaves the advice that is responsible for executing the command after the removal of a command. So with this mutant, the advice tries to execute a command just after it is removed and fails, but the failure is not observable (no events, e.g., exception, error message). As the command is actually removed, there is no observable behavior and the mutant cannot be killed without AdviceTracer.

The last eight mutants are related to an aspect that synchronizes several methods to prevent concurrent executions. While it is possible to kill these mutants with JUnit alone, creating a deterministic test for concurrent behavior is difficult. With AdviceTracer these mutants are easily killed, as there is no need to check the behavior of the advice in the oracle.

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

In summary, AdviceTracer increases the ability of the test cases to detect faults in PCDs. With JUnit some faults in PCDs cannot be detected as the observable behavior of the associated advice cannot exhibit a fault. AdviceTracer on the other hand does not rely on the behavior of the advices and is able to detect all the kinds of faults inserted by the operators of AjMutator.

It has to be noted that faults that do not modify the observable behavior do not necessarily reveal an equivalent mutant. Since mutation analysis is performed on PCDs, an equivalent mutant is a PCD that matches exactly the same set of join points as the original one. Thus, if a mutant matches a different set of join points, the mutant is not equivalent, but can still lead to the same global behavior as the original PCD.

5.5. Threats to validity

This section discusses the internal and external threats to validity.

The threats to internal validity are those that can bias the measurements made during this study. Mutation analysis was used as a test adequacy criterion to write the test cases. It is possible that the test suites would have been different if different criteria were selected. However, as AjMutator introduces different types of faults identified by Ferrari *et al.* [9], the test suites need to detect different types of faults and therefore they have various test cases. Thus, it is unlikely that other test adequacy criteria would produce different results. Another concern is that there were only 2 subjects. Other subjects could have written shorter or fewer test cases.

Another threat to validity is the fact that HealthWatcher was modified to improve its testability. However, this modification actually favors JUnit. Since AdviceTracer does not require tests to verify the behavior of the advices, it is less sensitive to testability issues than JUnit.

The threats to external validity are biases that affect the generalization of the results to other systems. HealthWatcher does not include all the types of primitive PCDs as shown in Table II. Previous work reported a study on 38 AspectJ projects [22]. This study shows that the primitive PCDs used in

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

HealthWatcher are typical of what is used in most AspectJ projects. The primitive PCDs that are not used by HealthWatcher are also not used by most of the other projects.

The interval validity is also threatens by the subjects of this study. Having two subjects is a threat to internal validity, as such a small number may not be representative of the developers in the industry. Also, both testers wrote the test suite alone, which means that the results of this study may not generalize to bigger group of testers. Finally, both developers were students familiar with software testing techniques, which may not be the case of all the developers in the industry.

6. Related Work

Ye *et al.* [32] tackle the issue of correctness of a PCD. They propose tools to assist developers in diagnosing faults and fixing PCDs. First, they compute a set of join points from a PCD that are almost matched by this PCD. This means they compute the set of join points that would be matched if the PCD was slightly different. That way, a developer who analyzes these join points can check whether he/she expected one of these join points 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 join point is not matched by the PCD.

The solution proposed by Ye *et al.* to tackle the problem of faulty PCDs is thus different from the proposed approach. Instead of specifying the set of expected join points 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 their approach the developer has to manually check all the PCDs to be sure that there are no regressions. The approach proposed in this paper is automatic, so 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.

Another work related to the proposed approach is Sakurai *et al.* [23], who specify pointcuts in the form of unit test cases. Although their approach at first glance appears to be similar to the proposed

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

approach, it is actually different. They propose a new language for pointcut description based on unit test cases. 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.

Lemos *et al.* [18] have proposed a test adequacy criterion for testing advices. This criterion, called *all-crosscutting-node*, requires to execute each advice at each join point where it is woven. Wedyan and Ghosh [27] call this criterion *join point coverage* and have proposed a tool for measuring it. This criterion is intended for testing advices, but test suites targeting the PCDs should satisfy this test adequacy criterion, as the test case have to specify each intended join point.

There exist several studies related to mutation analysis in the context of AOP. Anbalagan *et al.* [2] present a tool for the generation of mutant PCDs. This tool produces mutants that are rated following their resemblance with the original PCD. This rating process also automatically detects the equivalent mutants.

Although the work of Anbalagan *et al.* is related to the mutation analysis approach used in AjMutator, they focused on a particular type of mutation analysis. One of their goals is to rank and select mutants so that the developer can choose a mutant, which is close to the PCD written in the aspect, to help develop the PCD.

To rank the mutant, their tools computes the difference between the mutant and the original PCD as an integer. This is possible because they have only two operators, one that only generates mutants of class 2 (only neglected join points) and one that only generates mutants of class 3 (only unintended join points). AjMutator is more general and extensible. It has more operators and operators can easily be added by implementing a visitor of the PCD AST.

Ferrari *et al.* [9] identified fault types for aspect-oriented programs and provided a set of mutation operators. This operators can insert faults in various parts of an AspectJ system, such as the advice or the PCD.

Copyright © 0 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

AjMutator implements a set of the operators presented by Ferrari *et al.*. Table I shows the operators that have been implemented. These operators injects faults in the PCD that can produce mutants of class 1, 2 or 3 (or equivalent mutants).

McEachen *et al.* [20] and Baekken *et al.* [3] propose several fault models. These studies 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.

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

Xu *et al.* [31] tackle the issue of regression test selection for AOP. They extended a technique for Java introduced by Harrold *et al.* [13] 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.* [28] 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 amount of generated data can be large. To tackle this issue, Xie *et al.* [29] 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.

Harman *et al.* [12] have extended the work Xie *et al.* by using search-based techniques. They automatically generate test data for the base program that indirectly exercise the aspects. The goal is to achieve structural coverage of the aspects. This approach has been implemented by a prototype called EvolutionaryAspectTester. The authors have also conducted an empirical study that demonstrates that search-based testing for AOP is effective.

Xu *et al.* [30] 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

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

data to cover the paths in the composed statechart that correspond to interactions between the base program and the advice.

Lopes *et al.* [19] 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. Conclusion

Pointcut descriptors (PCD) in AspectJ offer a flexible way of introducing crosscutting behavior in Java programs. However, they are also a major source of faults; a fault in one PCD can result in the introduction of the crosscutting behavior at several unintended places or miss certain desired places. PCDs must be thoroughly verified in order to ensure the correctness of aspect-oriented programs. With the current state of testing techniques, JUnit is the only possibility to automatically check the validity of a PCD. However, it has been shown that this is not sufficient, mainly because this does not allow a tester to precisely target faults in the PCD. As a consequence, a JUnit test case that is meant to capture faults in a PCD can miss faults. It can also capture faults that are not in the PCD, but in the advice, leading to an interpretation issue for diagnosis.

This work proposed the AdviceTracer tool. It monitors the execution of advices in AspectJ programs and offers the possibility to retrieve information about this execution through a small set of assertions. These assertions can in turn be used to build test cases that target faults in PCDs. For evaluation purposes, an empirical analysis was performed that compares AdviceTracer with JUnit for writing effective test cases for AspectJ test cases. Two dimensions of effectiveness were investigated: The complexity of test cases and their ability at detecting faults. The empirical study was performed on a HealthWatcher system that is frequently used as benchmark for aspect-oriented software development work. It was observed that test cases that use AdviceTracer to test PCDs are smaller, require less time to build, have oracles more focused on faults in PCDs, and detect more faults than test cases that use

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

JUnit only. Also, oracles that use AdviceTracer capture only faults in PCDs and cannot capture faults in the advice. This reduces the interpretation issue when a test case delivers a verdict. However, this oracle is based on the line number where the advice should occur and this might make the oracle fragile when the base program evolves.

In future work, the evolution of test cases needs to be explored. PCDs are particularly vulnerable to evolution. If new join points are added because the base program evolved, join points could be improperly matched by the existing PCDs, or on the contrary, join points could be improperly ignored by the existing PCDs. It would be useful to conduct an empirical study to see how test suites written with different techniques can detect such faults and can evolve with the program.

REFERENCES

1. R. Alexander, J. Bieman, and A. Andrews. Towards the systematic testing of aspect-oriented programs. Technical Report CS 04-110, Department of Computer Science, Colorado State University, Colorado, December 2004.
2. P. Anbalagan and T. Xie. Automated generation of pointcut mutants for testing pointcuts in aspectj programs. In *ISSRE '08: Proceedings of the 19th International Symposium on Software Reliability Engineering*, pages 239–248, Nov. 2008.
3. J. S. Bækken and R. T. Alexander. A candidate fault model for AspectJ pointcuts. In *ISSRE '06: Proceedings of the 17th International Symposium on Software Reliability Engineering*, volume 0, pages 169–178, Los Alamitos, CA, USA, 2006. IEEE Computer Society.
4. D. Beuche and C. Beust. Aop has yet to prove its value. *IEEE Software*, 23(1):73 – 74, January - February 2006.
5. E. M. Cagnon and L. J. Hendren. Sablecc, an object-oriented compiler framework. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 140–154. IEEE Computer Society, August 1998.
6. R. Delamare, B. Baudry, S. Ghosh, and Y. Le Traon. A test-driven approach to developing pointcut descriptors in AspectJ. In *ICST '09: Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation*, April 2009.
7. R. Delamare, B. Baudry, and Y. Le Traon. Ajmutator: A tool for the mutation analysis of aspectj pointcut descriptors. In *Proceedings of the 4th International Workshop on Mutation Analysis (Mutation'09)*, 2009.
8. R. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.
9. 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*, pages 52–61, April 2008.

Copyright © 0 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.
11. R. Geist, A. J. Offutt, and J. Harris, Frederick .C. Estimation and enhancement of real-time software reliability through mutation analysis. *Computers, IEEE Transactions on*, 41(5):550–558, May 1992.
12. M. Harman, F. Islam, T. Xie, and S. Wappler. Automated test data generation for aspect-oriented programs. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 185–196, New York, NY, USA, 2009. ACM.
13. 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.
14. K. Hoffman and P. Eugster. Towards reusable components with aspects: an empirical study on modularity and obliviousness. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 91–100, New York, NY, USA, 2008. ACM.
15. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, 1997.
16. C. Koppen and M. Storzer. Pcdiff: Attacking the fragile pointcut problem. In *European Interactive Workshop on Aspects in Software (EIWAS)*, September 2004.
17. 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.
18. O. A. L. Lemos, A. M. R. Vincenzi, J. C. Maldonado, and P. C. Masiero. Control and data flow structural testing criteria for aspect-oriented programs. *J. Syst. Softw.*, 80(6):862–882, 2007.
19. C. V. Lopes and T. Chi Ngo. Unit testing aspectual behavior. In *Proceedings of the 1st Workshop on Testing Aspect-Oriented Programs*, 2005.
20. 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.
21. F. Munoz, B. Baudry, and O. Barais. Improving maintenance in aop through an interaction specification framework. In *Proceedings of the 24th International Conference on Software Maintenance (ICSM'08)*, pages 77 – 86, October 2008.
22. F. Munoz, B. Baudry, R. Delamare, and Y. Le Traon. Inquiring the usage of aspect-oriented programming: an empirical study. In *Proceedings of the 25th International Conference on Software Maintenance (ICSM'09)*, September 2009.
23. 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.

Copyright © 0 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

24. S. Soares, P. Borba, and E. Laureano. Distribution and persistence as aspects. *Software Practice and Experience*, 36:711–759, June 2006.
25. 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.
26. T. Tourwé, 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*, 2003.
27. F. Wedyan and S. Ghosh. A joinpoint coverage measurement tool for evaluating the effectiveness of test inputs for aspectj programs. In *ISSRE '08: Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, pages 207–212, Washington, DC, USA, 2008. IEEE Computer Society.
28. 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.
29. T. Xie, J. Zhao, D. Marinov, and D. Notkin. Detecting redundant unit tests for AspectJ programs. In *Proceedings of the 17th International Symposium on Software Reliability and Engineering*, pages 179–190, 2006.
30. D. Xu and W. Xu. State-based incremental testing of aspect-oriented programs. In *Proceedings of the 5th International conference on Aspect-Oriented Software Development*, pages 180–189, 2006.
31. 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.
32. L. Ye and K. De 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.

Copyright © 0 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr