AjMutator: A Tool For The Mutation Analysis Of AspectJ Pointcut Descriptors

Romain Delamare, Benoit Baudry, Yves Le Traon

AjMutator: A Tool For The Mutation Analysis Of AspectJ Pointcut Descriptors

Romain Delamare, Benoit Baudry
IRISA / INRIA Rennes
Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 Rennes Cedex – France
{*rdelamar,bbaudry*}*@irisa.fr*

Yves Le Traon
IT-Telecom Bretagne
Campus Universitaire de Beaulieu
2, rue de la Châtaigneraie
35576 Cesson Sévigné Cedex – France
yves.letraon@telecom-bretagne.eu

Abstract

Aspect-oriented programming introduces new challenges for software testing. In particular the pointcut descriptor (PCD) requires particular attention from testers. The PCD describes the set of joinpoints where the advices are woven. In this paper we present a tool, AjMutator, for the mutation analysis of PCDs. AjMutator implements several mutation operators that introduce faults in the PCDs to generate a set of mutants. AjMutator classifies the mutants according to the set of joinpoints they match compared to the set of joinpoints matched by the initial PCD. An interesting result is that this automatic classification can identify equivalent mutants for a particular class of PCDs. AjMutator can also run a set of test cases on the mutants to give a mutation score. We have applied AjMutator on two systems to show that this tool is suitable for the mutation analysis of PCDs on large AspectJ systems.

1. Introduction

Aspect-oriented programming (AOP) is a paradigm that separates the core concern from the cross-cutting concerns. The cross-cutting concerns are encapsulated in aspects composed of two parts: (1) an advice that implements the cross-cutting concern, and (2) a pointcut descriptor (PCD) that designates a set of joinpoints in the base program where the advice should be woven.

AOP introduces new kinds of fault types that should be addressed by testing techniques. Faults can be located in the advice, in the PCD or can arise from the composition of the aspects. The PCD is the place that is the most fault-prone in an aspect, as observed by Ferrari *et al.* [7].

Pointcut descriptors are composed of primary PCDs, using conjunctions, disjunctions, and negations. Most of the primary PCDs are static, i.e. the set of matched joinpoint can be computed statically. Dynamic primary PCDs, restrain the set of joinpoints at compile time, so the set of joinpoints statically computed is an over-approximation of the actual set of matched joinpoints. PCDs with dynamic primary PCDs are dynamic.

The mutation analysis of the PCD is useful for testing AOP systems and for evaluating testing techniques targeting faults in the PCD. A mutation analysis [6, 9] introduce faults in a program and runs test cases on the mutants to check if they can detect the introduced faults. The mutation analysis can be used to improve the set of test cases or to compare different sets of test cases.

In this paper we present a tool, AjMutator, for the mutation analysis of AspectJ pointcut descriptors. AjMutator can generate mutants by introducing faults in the PCDs, and execute a set of test cases on the generated mutants.

We also introduce a classification of the mutant PCDs. AjMutator automatically classifies the mutants by comparing the sets of of joinpoints matched by the mutant and the initial PCD. We automate this classification at compile time by leveraging the static analysis performed by the compiler that computes the set of joinpoints matched by the PCDs. We show that a benefit of this classification is that, for a class of PCDs, we can conclude that if the mutant matches the same set of joinpoints as the initial PCD, the mutant is equivalent.

Section 2 introduces the specificities of mutation analysis of PCDs. Section 3 details the implementation of AjMutator. Section 4 presents two cases studies that shows that AjMutator can perform mutation analysis on AspectJ systems. Section 5 discuss works related

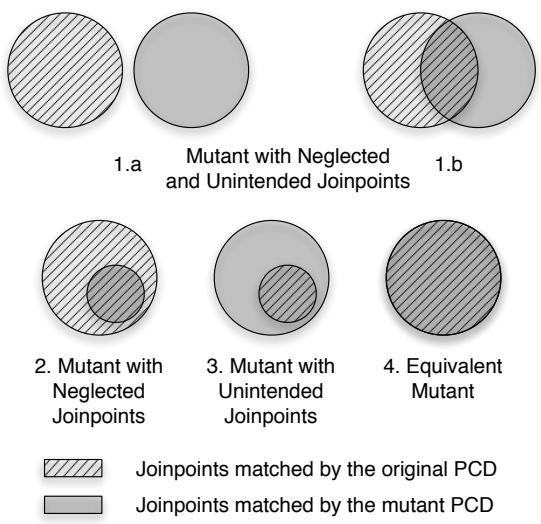

Figure 1. Classification of the Mutants

to mutation analysis and AOP. Finally we present our conclusion in Section 6.

2. Mutation analysis of AspectJ Poincut Descriptors

2.1. Mutation analysis

A mutant PCD is a PCD that matches a set of joinpoint different from the set of joinpoints matched by the original PCD. If the set of joinpoint is different, the advice is not correctly woven, and it can cause huge side effects.

Ferrari *et al.* [7] have presented several operators for the mutation analysis of AspectJ programs. These operators are based on fault types that have been identified in several works. The authors presented three kinds of operator: operators for PCDs, operators for AspectJ declarations, and operators for advice definitions and implementations. In this work we only considered operators for PCDs. Table 1 shows the operators that have been selected for AjMutator.

2.2. Classification of the mutants

The mutant PCDs are classified by comparing the set of joinpoints they match with the set of joinpoints matched by the original PCD. If a joinpoint is matched by the original PCD but not by a mutant PCD, it is called a neglected joinpoint. If a joinpoint is matched by the mutant PCD but not by the original PCD, it is called an unintended joinpoint.

Figure 1 shows the classification of the mutants. Mutants with both neglected and unintended joinpoints are in class 1. Mutants in class 1.a match a completely different set of joinpoints whereas mutants in class 1.b match some joinpoints matched by the original mutant. Mutants with only neglected joinpoints are in class 2. Mutants with only unintended joinpoints are in class 3. Mutants that match exactly the same set of joinpoints as the original PCD are equivalent: they are semantically equivalent so it is not possible for a set of test cases to distinguish them from the original PCD.

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 joinpoints, and thus can change the set of joinpoints matched by a PCD.

3. 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 is separated in three distinct parts:

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

3.1. Generation of the mutants

Figure 2 shows the AjMutator process of generation of 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 [4], 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 as a very different grammar from the rest of the AspectJ syntax – which is very close to the Java syntax.

The mutation operators are implemented using the visitor pattern [8]. 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

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

Table 1. Selected operators

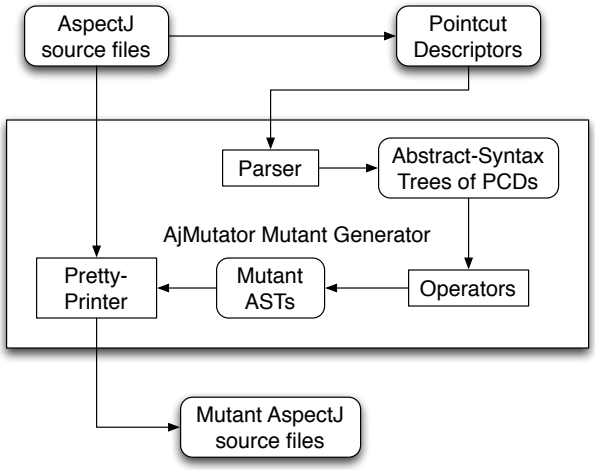

Figure 2. The AjMutator process of generation of the mutants.

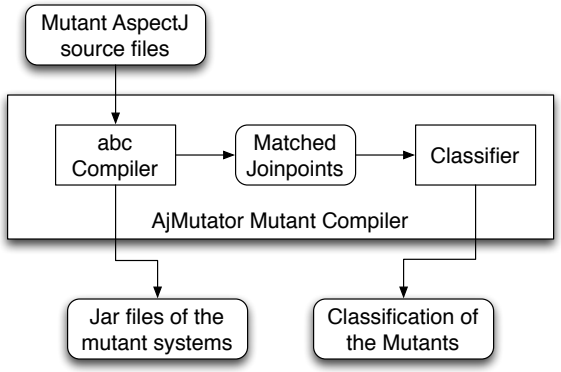

Figure 3. The AjMutator process of compilation and classification of the mutants.

AspectJ source file using a pretty printer. To reduce the memory consumption the mutant AST is immediately printed and the reference is not kept so that the garbage collector can free its memory space.

3.2. 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 3 shows the process of compiling and classifying the mutants. It relies on the abc compiler [3], which is an alternative compiler for AspectJ. The abc compiler produces a jar file of the system for each mutant. It also provides information on the joinpoints matched by the PCDs. The information is then used by AjMutator to classify the mutants.

As a single modification of an aspect can affect var-

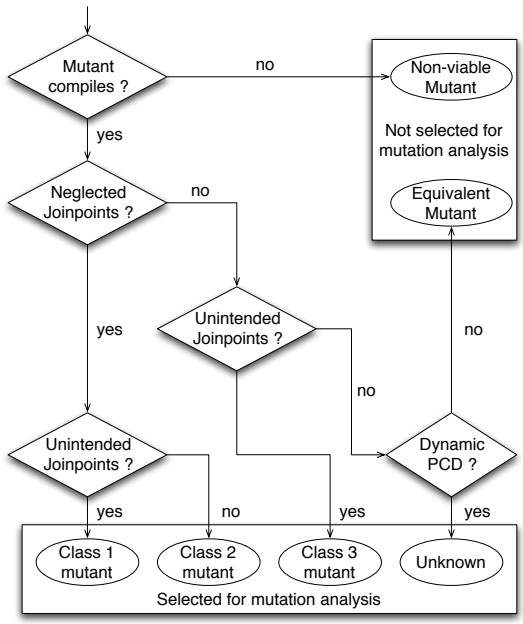

Figure 4. The process for the classification of the mutants.

ious 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.

Figure 4 shows the process for the classification of the mutants. First if the mutant is non compilable it is not selected as a mutation analysis considers only compilable mutants. If the mutant system has both neglected and unintended joinpoints, the mutant is in class 1. If the mutant system has neglected joinpoints but no unintended joinpoints, the mutant is in class 2. If the mutant system has unintended joinpoints but no neglected joinpoints, the mutant is in class 3. Finally if the mutant system has no unintended joinpoints and no neglected joinpoints, the mutant is equivalent and not selected for the mutation analysis.

The accuracy of the classification process depends on whether the original PCD of the mutant is static or dynamic. This can be summarized as follows:

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

Dynamic PCD the set of matched joinpoints can only be over-approximated, so the classification is not perfectly accurate. Two cases are distinguish:

- **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 we must select the mutant for the mutation analysis as we cannot be sure that it is actually equivalent.

As shown on Figure 4, before classifying a mutant as equivalent we first check if its original PCD is dynamic or not. If it is dynamic, the mutant is not classified and it is selected for the mutation analysis.

3.3. Execution of the test cases

The goal of a mutation analysis is to evaluate a test suite with a mutation score. The mutation score is the ratio of the number of killed mutant to the total number of mutant; it evaluates the adequacy of the test suite for detecting faults in the mutated system.

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.

Class	Number of Mutants	Mutation Score
1.a	3	100%
1.b	1	100%
2	4	75%
3	10	60%
unknown	5	60%
Total Selected	23	69.6%
Equivalent	72	–
Non-compilable	15	–
Total	110	–

Table 2. Classification and mutation score for the Auction system

Class	Number of Mutants
1.a	38
1.b	17
2	50
3	129
unknown	65
Total Selected	299
Equivalent	296
Non-compilable	90
Total	685

Table 3. Classification of the mutants produced for the HealthWatcher

AjMutator relies on JUnit for the test cases. A mutant is killed if at least one test cases has a different result on the mutant system. So if all the test cases pass on the original system, a mutant is killed if at least one test cases fails.

4. Experiments

To evaluate AjMutator we have used it on two different systems. The first system is an Auction system we developped, and the second is the HealthWatcher [1]. The HealthWatcher system has been implemented in different languages such as Java or AspectJ, and for each language there is ten different versions. For these experiments we used the tenth version in AspectJ, which is the larger one.

The Auction system has 41 classes and two aspects. There is also 209 test cases that satisfy the statement coverage criterion. The results of the mutation analysis are shown in Table 2. Out of the 110 produced mutants, 72 were detected as equivalent. The mutation score of the test suite is 69.6%.

The HealthWatcher system is larger, with 114 classes and 23 aspects. No test cases were available for the HealthWatcher so we did not perform a muta-

tion analysis. Table 3 shows the classification of the produced mutants.

These experiments show that AjMutator is able to generate and compile a large number of mutants on large systems. It can also perform a mutation analysis to obtain a mutation score.

The automatic classification offers great benefits. A manual selection of the mutants would have been long and difficult. If all the compilable mutants were selected, the mutation score for the Auction system would have been 16.8% which is very far from the actual mutation score.

5. Related Work

Anbalagan *et al.* [2] have presented 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 very related to our work, they focused on a very particular type of mutation analysis. One of their goal is to rank and select mutants so that the developer can choose a mutant that is close to the PCD he wrote, to help him 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 joinpoints) and one that only generates mutants of class 3 (only unintended joinpoints). AjMutator is more general and extensible. It has more operators and operators can easily be added by implementing a visitor of the PCD AST (see Section 3.1).

Ferrari *et al.* [7] have identified fault types for aspect-oriented programs and they have identified a set of mutation operators. This operators can insert faults in various parts of an AspectJ system, such as the advice or the PCD.

AjMutator implements a set of the operators presented by Ferrari *et al.*. Table 1 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).

6. Conclusion

We have presented AjMutator, a tool for the mutation analysis of AspectJ pointcut descriptors. It is able to generate mutant PCDs, to compile the mutants, and to execute a set of test cases on the mutants to obtain a mutation score.

AjMutator leverages the static analysis performed by the compiler to automatically detect equivalent mutants for a class of PCDs. Non-equivalent mutants are also classified depending on the set of joinpoints they match.

Experiments have shown that AjMutator works on large scaled system. The automatic classification is helpful when a large number of mutants are generated. The automatic classification of the equivalent mutants also avoids the unnecessary execution of these mutants and allows a more accurate mutation score.

AjMutator has also already been used to evaluate the adequacy of a new testing technique for detecting faulty PCDs [5].

References

[1] Healthwatcher. `http://www.comp.lancs.ac.uk/ greenwop/tao`

[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] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible aspectj compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[9] 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.