On generating mutants for AspectJ programs

Fadi Wedyan\textsuperscript{a}, Sudipto Ghosh\textsuperscript{b,\textasteriskcentered}

\textsuperscript{a}Department of Software Engineering, Hashemite University, Zarqa, Jordan
\textsuperscript{b}Department of Computer Science, Colorado State University, Fort Collins, CO, USA

\textbf{ARTICLE INFO}

\textbf{Article history:}
Available online 9 December 2011

\textbf{Keywords:}
AspectJ
Aspect-oriented programming
Mutation testing
Fault models
Test generation
High order mutation

\textbf{ABSTRACT}

\textbf{Context:} Mutation analysis has been widely used in research studies to evaluate the effectiveness of test suites and testing techniques. Faulty versions (i.e., mutants) of a program are generated such that each mutant contains one seeded fault. The mutation score provides a measure of effectiveness.

\textbf{Objective:} We study three problems with the use of mutation analysis for testing AspectJ programs:

- The manual identification and removal of equivalent mutants is difficult and time consuming. We calculate the percentage of equivalent mutants generated for benchmark AspectJ programs using available mutation tools.
- The generated mutants need to cover the various fault types described in the literature on fault models for AspectJ programs. We measure the distribution of the mutants generated using available mutation tools with respect to the AspectJ fault types.
- We measure the difficulty of killing the generated mutants.

We propose the use of simple analysis of the subject programs to prevent the generation of some equivalent mutants.

\textbf{Method:} We revised existing AspectJ fault models and presented a fault model that removes the problems in existing fault models, such as overlapping between fault types and missing fault types. We also defined three new fault types that occur due to incorrect data-flow interactions occurring in AspectJ programs. We used three mutation tools: \textit{AjMutator}, Proteum/AJ, and \textit{MuJava} on three AspectJ programs. To measure the difficulty of killing the mutants created using a mutation operator, we compared the average number of the mutants killed by 10 test suites that satisfy block coverage criterion.

\textbf{Results:} A high percentage of the mutants are equivalent. The mutation tools do not cover all the fault types. Only 4 out of 27 operators generated mutants that were easy to kill.

\textbf{Conclusions:} Our analysis approach removed about 80\% of the equivalent mutants. Higher order mutation is needed to cover all the fault types.

\textcopyright 2011 Elsevier B.V. All rights reserved.

\section{1. Introduction}

Experimental studies in software testing require the evaluation of a testing approach in terms of its ability to detect faults in subject programs containing known faults. However, researchers often have a problem in finding subject programs that meet their research requirements and also contain real faults. Even when such programs are available, the number of real faults is often not large enough to allow achieving statistically significant results [3]. Therefore, researchers typically seed faults in the subject programs, either manually or with mutation operators. The latter approach has several advantages. A large number of faulty versions can be generated, in a systematic and easily replicable manner [26]. Moreover, Andrews et al. [3,4] provided evidence that faults generated with mutation operators are similar to real faults in evaluating test effectiveness, while hand-seeded faults are harder to detect than real faults.

One major problem with mutation testing is that mutation operators generate a large number of equivalent mutants (i.e., faults produced by the mutants do not propagate to produce a different output). These mutants need to be identified and eliminated from the analysis. In a recent study, Schuler and Zeller [30] showed that, for Java programs, about 45\% of the undetected mutants turned out to be equivalent. They reported that it took about 15 min to assess one single mutant. Budd and Angluin [8] proved that detecting equivalent mutants is an undecidable problem. Therefore, many researchers have proposed techniques for partially detecting equivalent mutants. These techniques include using compiler optimization [6], constraint-based testing [28], program slicing [18,19,33], and co-evolutionary approaches [1].

For AspectJ programs, few studies have been performed that involve seeding faults using mutation operators. We can view aspect-oriented mutation with AspectJ as a superset of
object-oriented mutation with Java because every Java program is an AspectJ program. Thus, all the mutation operators that mutate Java programs can be used to mutate AspectJ programs. However, since AspectJ programs contain constructs that do not exist in Java programs, more mutation operators are needed to mutate these constructs. Thus, we need operators for mutating the pointcut descriptor, aspect declarations, and new constructs used in the advices (e.g., proceed statement).

Delamare et al. [11] used their tool, AjMutator, to generate mutants from the pointcut descriptors of two AspectJ programs. Their results show that about 13% of the generated mutants did not compile and about 54% of the compiled mutants were equivalent. Similarly, Ferrari et al.’s [16] tool, called Proteum/AJ, generated approximately 11% non-compilable mutants and 46% equivalent mutants. In our previous work [34], we used AjMutator and MuJava [23] to seed faults in an AspectJ program. We found that for both tools, about 50% of mutants did not compile and about 50% of the compiled mutants were equivalent. All these results show that current AspectJ mutation tools generate a high percentage of equivalent mutants.

In this paper, we present the results of a study on the use of available mutation tools for seeding faults in AspectJ programs. We performed the study using three mutation tools AjMutator, Proteum/AJ, and MuJava [23], to seed faults in three AspectJ programs. We first measured the percentage of non-compilable and equivalent mutants produced by the tools. Our results show that AjMutator produced a high number (29%) of non-compilable mutants. We also noted a wide variation in the percentage of equivalent mutants. Depending on the mutation operator, the percentage ranged from 0% to 100%. AjMutator generated the highest percentage of equivalent mutants while MuJava generated the lowest.

Adequate coverage of fault types is needed when testing approaches are evaluated. We investigated the types of faults that can be generated by using AspectJ mutation operators. First, we solved problems with existing fault models of AspectJ including overlapping fault types and missed fault types. We revised the fault models and classified the generated mutants according to the types of faults in the revised fault model that they represented. The available operators were able to generate faults in all the types except one that required performing more than one change to the pointcut. Therefore, we propose that higher order mutants (e.g., see [20]) be used to cover the remaining fault types.

We measured the difficulty of killing AspectJ mutants. For each operator, we created mutants and measured the average percentage killed using 10 test suites that satisfy the block coverage criterion.

Our results show a wide variation in the results for each operator. These results can help in choosing what mutation operators to use for evaluating the quality of test suites.

We present an approach to reduce the number of equivalent mutants. We inspected the equivalent mutants generated by each operator to determine why a tool generated the mutant and what mechanism can be added to the tool to prevent the generation of such a mutant. We found that many equivalent mutants can be eliminated if the mutation tools were provided with information about the static structure of the subject programs. Therefore, we propose performing a simple analysis of the subject programs before generating mutants. The analysis can be performed on the source code or on the bytecode of the AspectJ programs. The computational cost of running the analysis is negligible compared with the time needed to inspect equivalent mutants. Our results show that about 92% of MuJava’s equivalent mutants, 67% of AjMutator’s and 76% of Proteum/AJ’s mutants can be reduced by analyzing the static structure of the programs that we investigated.

In Section 2, we present our revised fault model. We describe the study method in Section 3. Sections 4–7 present the results for the study of equivalent mutants and coverage of fault types for the four types of mutation operators we used (1) pointcut descriptors, (2) aspect declarations, (3) aspect implementations, and (4) Java classes. We discuss how to reduce the number of equivalent mutants for pointcut descriptors, aspect implementations, and class implementations. The results of measuring the difficulty of killing the mutants generated by each operator are described in Section 8. Section 9 describes the threats to validity. Section 10 summarizes our results.

2. Revised fault model

Several fault models for aspect-oriented programs have been proposed in the literature [2,5,9,13,16]. Ferrari et al. [16] presented a summary that combines all the fault types from previous studies as well as faults that the authors identified. However, the resulting fault model has several shortcomings. First, as the authors themselves stated, inclusion relations between these faults were not studied (i.e., some of the fault types can overlap). Second, we noted that the summary missed fault types corresponding to incorrect data-flow interactions in a program.

We define fault types as patterns of faults. With each fault type, we specify the constructs that can contain the faults and state the effect of these faults on the program.

We classify the fault types into four categories in a similar way as Ferrari et al. [16] did, but with some modifications: (1) pointcut descriptor faults, (2) aspect declaration faults, (3) advice, aspect method, and intertype method implementation faults, and (4) class implementation faults. We made three modifications. First, we moved the advice declaration faults that were present in category 3 to category 2 because we wanted to keep the declaration faults and implementation faults in two separate categories. Different mutation tools are used to generate the faults in these two categories. Second, we included faults that occur in intertype methods and aspect methods in category 3, which already included faults in the advice implementation. Third, we added new fault types resulting from incorrect data-flow interactions to categories 3 and 4 in the fault model.

2.1. Pointcut descriptor faults (F1)

Faults in this category occur in the descriptor of a pointcut. The fault types are as follows:

- (F1-1) The pointcut matches a set of join points that contains only unintended join points [22].
- (F1-2) The pointcut matches a set of join points that contains unintended join points and some intended join points [22].
- (F1-3) The pointcut matches all intended join points and some unintended join points [22].
- (F1-4) The pointcut matches a subset of intended join points and no unintended join points [22].
- (F1-5) The pointcut does not match any join point.

Lemos et al. [22] identified four types of faults that can occur in a pointcut descriptor. These fault types were also adopted by Ferrari et al. [16] and Delamare et al. [11]. The fault types are defined according to the set of join points matched by the pointcut, and these are the first four types in our revised fault model. A special case of F1-1 and F1-4 occurs when the pointcut does not match any join point. In our study we treat this case separately and refer to it as F1-5.

van Deursen et al. [13] defined three faults types which Ferrari et al. [16] adopted. However, their fault types overlap with the fault types we kept in the revised model. For example, the fault
type “incorrect use of primitive pointcut designator” or the fault type “incorrect pointcut composition rules”, can generate a different set of join points than the intended one, and thus, overlap with any of the fault types, F1-1 through F1-5. In order to prevent overlapping faults, we discarded these types. Baekken and Alexander [5] described a fault model for pointcut descriptors based on the constructs that cause the fault. That is, for every construct, the authors enumerated all possible faults that might occur. These fault types are subsumed by our revised fault types.

The above faults can be generated by any pointcut designator, logical operator, or parameters used in a pointcut description. Baekken and Alexander [5] described these constructs in detail. The pointcut fault model can be used to define pointcut mutation operators that generate the faults types in the revised model.

The effects of this category of fault types are summarized in Table 1.

### 2.2. Aspect declaration faults (F2)

In this category, we combine faults that occur in the different declaration statements used in an aspect. These include intertype declarations, aspect precedence, and advice declarations. The revised fault types are:

- (F2-1) Incorrect method name in introduction, leading to a missing or unanticipated method override [13].
- (F2-2) Incorrect class name in a member-introduction [13].
- (F2-3) Incorrect declaration of parent class or interface.
- (F2-4) Incorrect declaration of error and warning statements [16].
- (F2-5) Incorrect aspect precedence [2,13].
- (F2-6) Incorrect aspect instantiation rules and deployment [16].
- (F2-7) Incorrect advice type specification [13,16].
- (F2-8) Advice bound to incorrect pointcut [16].

Fault type F2-1 occurs when an intertype method unintentionally overrides a method in the class. The fault is limited to inherited methods because AspectJ does not allow overriding methods defined in the class by intertype methods. Ceccato et al. [9] defined a fault type which they described as “incorrect changes in polymorphic calls”, that occurs when an aspect incorrectly overrides a method inherited from a superclass. We discard this fault type since it is actually an effect of a fault type, such as F2-1. Fault type F2-2 occurs when the intertype method is defined in a wrong class. Fault type F2-3 occurs when the aspect declares incorrect parent classes or interfaces to the classes. van Deursen et al. [13] defined the fault types “inconsistent parent declaration”, and “inconsistent overridden method introduction”. We discard these two fault type since they are effects of fault type F2-3 as shown in Table 2.

Fault type F2-4 occurs in the declare warning or declare error statements, using which, a developer can instruct the compiler to issue a warning or error at specified locations in the program. Fault type F2-5 occurs in the statement declare precedence, which specifies the order in which advices that match the same join point must be executed.

Fault types F2-6 through F2-8 occur in the advice declaration statement. Type F2-6 occurs when the developer incorrectly sets the aspect instantiation rule. By default, an aspect has exactly one instance that cuts across the entire program. However, AspectJ allows using different aspect instantiation rules using the aspect declaration statement (e.g., per executing object using per-this(Pointcut)) [16]. Fault type F2-7 occurs when developers incorrectly specify the advice type, such as by using before instead of after. Fault type F2-8 occurs when the developer associates the advice to the incorrect pointcut. For example, instead of associating the advice to the pointcut that matches the class constructors, the advice is associated with the pointcut that matches the class method.

The effects of this category of fault types are summarized in Table 2.

### 2.3. Advice, aspect method, and intertype method implementation faults (F3)

Several prior fault models contained faults types in this category, and were also adopted by Ferrari et al. [16]. However, three of the fault types were defined based on the effects of the faults. The first type was defined by Zhang and Zhao [37] as “infinite loop resulting from interactions among advices”, which they described as resulting from “the accidental matching of unexpected join point”. This fault type is an effect of matching unintended join points. The other two types, which were defined by Alexander et al. [2], are “inconsistent control or data flow changes”, and “violating state invariant and failing to establish expected post-conditions”. These are also the effects of faults that can occur in a pointcut descriptor, an aspect declaration, or an advice implementation.

The revised fault types are as follows:

- (F3-1) Incorrect guarding statement or missing proceed in around advice [13]. This fault type occurs when the proceed statement in an around advice is missed or the guarding condition to call the statement is incorrect.
- (F3-2) Incorrect altering of base class object state variables: An advice has access to the state variables of the base class instances using the designators this and target. We added this

### Table 1

<table>
<thead>
<tr>
<th>Type</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1-1, F1-2, F1-3</td>
<td>Incorrect changes in polymorphic calls</td>
</tr>
<tr>
<td>F1-4, F1-5</td>
<td>Failure to obey the post-condition of the method where the join point is matched</td>
</tr>
</tbody>
</table>

### Table 2

<table>
<thead>
<tr>
<th>Type</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2-1</td>
<td>Incorrect changes in polymorphic calls</td>
</tr>
<tr>
<td>F2-2</td>
<td>Incorrect changes in polymorphic calls</td>
</tr>
<tr>
<td>F2-3</td>
<td>Failure to preserve advised method post-conditions</td>
</tr>
<tr>
<td>F2-4</td>
<td>Failure to preserve advised method post-condition</td>
</tr>
<tr>
<td>F2-5</td>
<td>Failure to preserve advised method post-conditions</td>
</tr>
<tr>
<td>F2-6</td>
<td>Failure to preserve the advised method post-condition</td>
</tr>
<tr>
<td>F2-7</td>
<td>Failure in preserving the advised method post-condition</td>
</tr>
<tr>
<td>F2-8</td>
<td>Failure in preserving the advised method post-condition</td>
</tr>
</tbody>
</table>

### Table 3

<table>
<thead>
<tr>
<th>Type</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3-1</td>
<td>Incorrect guarding statement or missing proceed in around advice</td>
</tr>
<tr>
<td>F3-2</td>
<td>Incorrect altering of base class object state variables</td>
</tr>
</tbody>
</table>

## References

fault type to include faults that can occur due to incorrect data-flow interactions in the aspect-oriented program.

- **(F3-3)** Intra-advice level faults: These faults occur when the functionality of an advice is implemented incorrectly. They are similar to the method level faults described by Ma et al. [23] for Java methods.
- **(F3-4)** Incorrect access to join point static information [16]: This fault type occurs when the construct, `this JOIN POINT`, is incorrectly used.

The effects of this category of fault types are summarized in Table 3.

### 2.4. Class implementation related faults (F4)

We classify faults in this category as follows. First, we define new implementation faults that are specific to aspect-oriented programs. These are fault types F4-1 and F4-2, which result from incorrect data flow interactions. Second, we add object-oriented fault types that can occur in Java programs. Finally, we include implementation faults that can occur in procedural programming (i.e., intra-method faults).

- **(F4-1)** Passing an object in an unexpected state to an advice: An advice can expect the objects of the base class at a join point to be in a certain state. Failure to pass an object with the expected state causes an incorrect behavior during advice execution.
- **(F4-2)** Arguments passed to the advice have incorrect values: An advice might require the passed arguments to obey certain pre-conditions. A fault can be caused by an advised method or another advice that alters these values and passes the incorrect values to an advice, which causes the advice to behave incorrectly.
- **(F4-3)** Object-oriented faults: These are faults that occur in object-oriented implementations as described by Ma et al. [23] and include: access control, inheritance, polymorphism, overloading, and Java-specific features. The last type refers to

Java language features that do not occur in other object-oriented languages. In this paper we report all object-oriented related faults as one type.

- **(F4-4)** Intra-method level faults: These faults occur when the functionality of a method is implemented incorrectly [23]. We used the faults described by Ma et al. [23].

Table 4 shows the effects that may result from class implementation faults.

### 3. Study method

The following paragraphs describe the research questions for the study, the metrics used, the subject programs, the mutation tools, and the experiments settings.

#### 3.1. Research questions and evaluation metrics

We use the Goal-Question-Metric (GQM) paradigm [7,31] and state the study goals, research questions, and metrics below.

- **Goal G1:** Measure the percentage of non-compilable mutants.
  - **Question Q1:** For each mutation operator, what percentage of the generated mutants can compile?
  - **Metrics:** number of compilable mutants generated by the tool for an operator, number of compiled mutants that are equivalent.

- **Goal G2:** Reduce the cost of removing equivalent mutants that are generated using each operator.
  - **Question Q2a:** For each mutation operator, what percentage of compiled mutants is equivalent?
  - **Metrics:** number of compilable mutants generated by the tool for an operator, number of compiled mutants that are equivalent.

We propose that a mutation tool can reduce the percentage of equivalent mutants by performing an analysis of the subject programs. In order to keep the analysis simple, we only require providing mutation tools with information about the static structure of the classes.

- **Question Q2b:** How effective was the analysis process in reducing the number of equivalent mutants?
  - **Metrics:** number of equivalent mutants generated by a tool or operator, number of equivalent mutants that can be eliminated using our approach.

- **Goal G3:** Ensure that mutation operators can cover all the fault types in the revised aspect-oriented fault model presented in Section 2.
  - **Question Q3a:** Can the mutation operators seed faults of the types described in the revised aspect-oriented fault model? Metrics: number of faults of each type that are seeded in the subject programs.
  - **Question Q3b:** How effective were the mutation operators in generating faults of each type? Metrics: number of generated faults of each type that can be mutated in order to generate faults of each type.

<table>
<thead>
<tr>
<th>Table 3</th>
<th>Effects</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure to obey advised method post-condition</td>
<td>F3-1</td>
<td></td>
</tr>
<tr>
<td>Incorrect implementation of the core concern</td>
<td>F3-2</td>
<td></td>
</tr>
<tr>
<td>Incorrect object state used in advice and method</td>
<td>F3-3</td>
<td></td>
</tr>
<tr>
<td>Violate state invariant</td>
<td>F3-4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4</th>
<th>Effects</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure to obey advised method post-condition</td>
<td>F4-1</td>
<td></td>
</tr>
<tr>
<td>Incorrect object state used in advice and method</td>
<td>F4-2</td>
<td></td>
</tr>
<tr>
<td>Violate state invariant</td>
<td>F4-3</td>
<td></td>
</tr>
<tr>
<td>Incorrect implementation of the core concern</td>
<td>F4-4</td>
<td></td>
</tr>
</tbody>
</table>
program does not contain the constructs that are required for a mutation operator, then that fault cannot be seeded.

- Question Q3b: What improvement in fault type coverage is observed when new mutation operators are added?

Metrics: number of faults of each type that are seeded in the subject programs using the suggested operators, number of constructs in the program that can be mutated in order to generate faults of each type.

- Goal G4: Measure the difficulty of killing the mutants using the mutation operators.

Metrics: number of faults of each type that are seeded in the subject programs using the suggested operators, number of constructs in the program that can be mutated in order to generate faults of each type.

We used 10 test suites that satisfy the block coverage criterion and measured their average mutation score with respect to the mutants that were generated using a particular mutation operator. Other studies [14,35] have used such a measure to reduce the number of mutants used in mutation testing by eliminating mutants that are easier to kill.

3.2. Subject programs

We used 3 subject programs in our study. Table 5 shows the main characteristics of these programs. For each program shown in column 1, column 2 shows the lines of code (LOC). The third and fourth columns show the number of classes and the number of aspects in each program, respectively. The number of tested classes in each program is shown in column 5. Columns 6, 7, and 8 show the number of before, after, and around advices, respectively. The last column in the table shows the number of intertype methods.

### Table 5

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>#Classes</th>
<th>#Aspects</th>
<th>#Tested classes</th>
<th>#Before advices</th>
<th>#After advices</th>
<th>#Around advices</th>
<th>#ITM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kettle</td>
<td>125</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Telecom</td>
<td>928</td>
<td>10</td>
<td>3</td>
<td>5</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>CruiseControl</td>
<td>1008</td>
<td>9</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>All</td>
<td>2061</td>
<td>20</td>
<td>8</td>
<td>9</td>
<td>4</td>
<td>14</td>
<td>4</td>
<td>23</td>
</tr>
</tbody>
</table>

1. **Telecom** is standard benchmark that is shipped with the qic AspectJ compiler [32]. The program is a simulation of a telephony system that allows customers to make, accept, merge, and hangup both local and long distance calls. **Telecom** contains three aspects:
   - **Timing** aspect, which measures the connection duration for the customers by initializing and stopping a timer associated with each connection.
   - **Billing** aspect, which specifies the payer of each call and ensures that local and long distance calls are charged appropriately.
   - **TimerLog** aspect, which implements a log that prints the times whenever a connection is established or dropped.

2. **Kettle** is an incremental modification aspect that generates faults of each type.
   - **CarSimulatorFix** aspect is an incremental modification aspect that enforces the precondition that the car engine is on for accelerate and brake events.
   - **SpeedControlIntegrator** aspect adds the implementation of a speed controller to a car.

3.3. Tools

We used three tools to mutate the programs, **AjMutator**, **Proteum/AJ** [17], and **MuJava**. Both **AjMutator** and **Proteum/AJ** implement operators that mutate AspectJ constructs (e.g., operators that mutate pointcut descriptors, advice declarations). **AjMutator** was developed by Delamare et al. [11]. It implements a subset of the operators for pointcut descriptors proposed by Ferrari et al. [16]. **AjMutator** parses a pointcut descriptor from the aspect source code and performs the mutations. The modified pointcut is then used to generate a mutant. **AjMutator** classifies the mutants according to the set of join points they match compared to the set of join points matched by the original pointcut. The tool also runs JUnit test cases and identifies non-compilable mutants.

**Proteum/AJ** [17] implements 3 more pointcut mutation operators than **AjMutator** from the same set of operators proposed by Ferrari et al. [16]. The tool also implements 2 advice declaration operators, 4 advice implementation operators, and 5 intertype declaration operators.

**Proteum/AJ** allows the tester to selectively apply the operators. The tool uses the same concept of equivalent pointcut mutants as that used in **AjMutator**. The tool also runs JUnit test cases and computes the mutation score for a given test suite. **Proteum/AJ** is not yet available for public use. Therefore, we sent our subject
programs to the developers of the tool, and they generated the mutants for us.

Both AjMutator and Proteum/AJ can identify equivalent mutants when the set of join points matched by the mutants is equal to the set of join points matched by the original pointcut.

Since every Java program is an AspectJ program, tools that mutate OO programs can also be used to mutate AspectJ programs. An overview of Java mutation tools can be found in [24]. MuJava [23] is a widely used tool for mutating Java programs. MuJava uses two types of mutation operators, class level and method level. Class level mutation operators generate faults related to OO-specific features while method level operators generate intra-method faults. MuJava relies on the tester to manually identify equivalent mutants.

The current version of MuJava does not support AspectJ. That is, the tool cannot mutate code in the aspects, and to the best of our knowledge, except for the 4 operators implemented in Proteum/AJ, there is no tool that can directly mutate code in the advice, aspect methods, and introduced methods. Therefore, we used an indirect approach to perform mutation in the aspects using MuJava. We generate a class from the aspect bytecode with the help of a decompilation tool (e.g., Jad)\(^1\). We mutate the decompiled class with MuJava. We choose the mutated line that resides in the advice, intertype method, or aspect method and copy it back to the aspect. We repeat this step to generate all the mutants, where each mutant contains one fault. Note that since the mutated class is produced from the bytecode, it contains some extra methods that are not present in the source code of the aspect. These methods include those that the aspect inherits from the AspectJ base class (e.g., methods aspectOf, hasAspect), and methods that correspond to some AspectJ constructs. For example, the declare precedence statement is compiled into a method in the bytecode. We discard mutants that reside in these extra methods.

For base classes we use MuJava directly to generate mutants. MuJava only produces mutants that compile. However, MuJava compiles mutants with a Java compiler, not an AspectJ compiler. Therefore, we compile all the mutants generated by MuJava with the AspectJ ajc compiler.

For test case generation, we used RANDOOP [29], which generates JUnit test cases for Java programs. RANDOOP generates new test cases by randomly selecting a method to call and finding arguments from among previously found inputs. Since RANDOOP is not designed for AspectJ, we performed the following steps to use RANDOOP with AspectJ programs:

- We rewrote the aspects in the subject programs using the annotation style, which is a feature of AspectJ 5, also known as @Aspect annotation. Using this feature, we can write aspects with regular Java syntax and then annotate the aspect declarations so that they can be interpreted by the AspectJ weaver. In the annotation style, intertype methods are declared in an interface that the class implements. This feature allows RANDOOP to recognize these methods since they are now part of the class declaration and RANDOOP can generate calls for them.
- We generate test cases only for the classes in the subject programs (i.e., not for the aspects). RANDOOP allows the tester to specify which classes to test and which methods can be called by the test cases. Using this feature, we can avoid having calls to aspect methods and advice from the test cases.

For each subject program, we set RANDOOP to generate 5000 test cases. We provided RANDOOP with a set of input values suitable for the subject programs. For example, for the telecom program, we provided RANDOOP with a set of customer names to chose from, and values for call durations that must be used. We also used an option of RANDOOP called the observers option, which is not yet available for public use. This allows creating custom assertions. In the public version of RANDOOP, the tool generates assertions that check for null values, reactivity, and symmetry of equality of the variables [25]. However, using the observers option, we can specify observer methods that helps generate stronger assertions that are application-specific.

### 3.4. Experiments

We performed two experiments. The first experiment is performed to answer research questions 1 through 3. In the first experiment, for each mutation operator, we generated mutants from the subject programs, identified non-compileable and equivalent mutants, classified the mutant type according to the revised AO fault model, and provided suggestions on how to reduce the percentage of non-compileable and equivalent mutants. We report the results of the first experiment based on the fault category of the mutants. These results are shown in Sections 4–7. We used three mutation tools in the experiment, however, mutants of the fault types in every category cannot be generated by each of the three tools because of the limitations inherent in the tool or the approach or both. Mutants of the fault types in category F1 can be generated using AjMutator and Proteum/AJ. Only Proteum/AJ can generate mutants of the fault types in category F2. Mutants of the fault types in category F3 can be generated using Proteum/AJ and MuJava. For the mutants of the fault types in category F4, only MuJava can be used to generate them.

The second experiment is performed in order to answer research question 4. For each of the subject programs, we used RANDOOP to generate 10 test suites that satisfy block coverage test criterion in each of the advised class. In our study, we generate test suites by repeatedly adding test cases to a suite until full coverage is obtained. In each iteration, we produce a new test suite by adding a test case to the existing test suite. If coverage is increased, we save the new test suite and repeat the process of adding test cases until full coverage level is reached. We computed the mutation score obtained by each test suite when used to kill mutants generated by each mutation operator. We also computed the mutation score obtained by each test suite on the mutants generated by each tool.

### 4. Mutant generation from pointcut descriptors

Table 6 lists the operators that actually generate faults in the three subject programs. Operators that apply to constructs not used in the programs are not shown (e.g., operators that mutate annotations). The first and second columns of the table show the operator name and description, respectively. The third and fourth columns specify whether the operator is implemented in Proteum/AJ [16] and AjMutator [11], respectively. Note that some operators are implemented in both the tools but in different ways. For example, POPL in Proteum/AJ is implemented by either adding or removing the parameter list, which is how the operator is defined by Ferrari et al. [16]. However, AjMutator only removes the parameter list from the descriptor if the list is specified. There are other implementation differences between the two tools that we will highlight throughout this section. For a complete description of the mutation operators, please refer to Ferrari et al. [16].

#### 4.1. Compiling the mutants

Tables 7 and 8 summarize the results of generating and compiling the pointcut mutants for the subject programs using AjMutator.

and Proteum/AJ, respectively. For each operator in Tables 7 and 8, we show the number of generated mutants and the number of mutants that compiled for each of the subject programs, as well as the total number of generated mutants and the total number of compiled mutants for all programs. The last column shows the percentage of compiled mutants from all programs. Operators that mutate constructs not used in the subject programs did not generate any mutant and are not shown in the tables.

As the tables show, all the non-compilable mutants were generated by two operators: PCLO and PWIW. In the Kettle program, none of the PCLO mutants from either tool compiled. The reason is that the pointcuts in the programs use the designators this(type), target(type), or args(type) to pass arguments to the pointcut parameters. These parameters need to match the number and order of the passed arguments. Otherwise, the compiler reports parameter unbounded or inconsistent error. All the non-compilable mutants generated by PCLO had the problem described above. These non-compilable mutants cannot be determined statically since they occur only in dynamic pointcuts.

With AjMutator, the operator PWIW, which inserts a wildcard in several locations in the pointcut, generated the non-compilable mutants in telecom (36%), Kettle (10%), and CruiseControl (23%). These mutants resulted from replacing the constructor designator new with a wildcard, or from inserting the wildcard before or after new. Therefore, the pointcut becomes incorrect because it matches methods without having a field for the return type. The tools can prevent the generation of these mutants by avoiding the mutation action described above.

### 4.2. Equivalent mutants

We manually analyzed the mutants to determine whether they were equivalent or not. In our experiments, two mutants are equivalent if they produce the same output. Pointcut mutants that match the same set of join points are equivalent because the same advice will be woven at the same join points, and thus the resulting mutant must have the same behavior. This rule applies for mutants generated by operators PCCE, POEC, PCTT, POPL, PSWR, and PWIW. For operators PCCE and PCTT, there are some mutants that match different sets of join points but are still equivalent (i.e., produce the same output as the original program). In our test cases, we used assertions on the return values of the methods and method post-conditions.

Tables 9 and 10 show the results of analyzing equivalent mutants generated using AjMutator and Proteum/AJ, respectively. For each program, the tables show the number of equivalent mutants and the percentage of equivalent mutants out of the compiled mutants generated by each operator. For example, the first row in Table 9 shows that operator PCCE had 3 equivalent mutants in the telecom program. These contribute about 37.5% of the 8 compiled mutants the operator generated. The last column in Tables 9 and 10 shows the percentage of equivalent mutants
generated by each operator out of the total number of equivalent mutants. For example, in the first row of Table 10, operator PCCE generated 17 equivalent mutants out of the 198 equivalent mutants that Proteum/AJ generated in all programs. Therefore, it contributes about 8.6% of the total number of equivalent mutants.

The last column in both tables shows that most of the equivalent mutants were generated by the PWIW operator. We inspected how these mutants were generated and presented the results in Table 11. About 71% of the equivalent PWIW mutants generated by AjMutator and 54% by Proteum/AJ resulted from inserting a wildcard before or after a string in the class name field or method name field. In the subject programs, such a change in the pointcut did not generate a different set of matched join points since neither the class name, nor the method names contain a common string. For example, the two mutants shown in Fig. 1, which add a wildcard before and after the string Connection are equivalent, since there is no other class in the telecom program that starts or ends with the string Connection.

Proteum/AJ also creates mutants by inserting a wildcard in the declare precedence statement. An example of such a mutant is shown in Fig. 2. The example shows how the PWIW operator mutates the declare precedence statement in the SafetyControl aspect of the Kettle program. Since both programs have exactly two aspects, adding the wildcard did not change the original precedence rule. Moreover, the declare precedence statement is not part of the pointcuts and it decides the order of executing advices that match the same join point but not the selected join points. Note also that this use of the PWIW operator should really belong to the F-2 category (aspect declaration fault), but this is how the PWIW operator works.

We expected that replacing a class name with a wildcard will produce non-equivalent mutants. However, since all the pointcuts in the programs specify which methods to match and there are no two methods with the same name in the different classes, this mutating approach always created equivalent mutants. The tools also altered the return type by inserting a wildcard (AjMutator) or a subtype wildcard (Proteum/AJ). Fig. 3 shows an example of an equivalent mutant generated by inserting a wildcard after the return type.

### Table 9
<table>
<thead>
<tr>
<th>Opr</th>
<th>Eqv telecom</th>
<th>Eqv kettle</th>
<th>Eqv CruiseControl</th>
<th>Eqv all</th>
<th>Contr (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCCE</td>
<td>3</td>
<td>37.5</td>
<td>2</td>
<td>66.7</td>
<td>8.8</td>
</tr>
<tr>
<td>PCLO</td>
<td>0</td>
<td>0</td>
<td>NA</td>
<td>NA</td>
<td>5.7</td>
</tr>
<tr>
<td>PCTT</td>
<td>4</td>
<td>50.0</td>
<td>3</td>
<td>100</td>
<td>10.4</td>
</tr>
<tr>
<td>POEC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7.8</td>
</tr>
<tr>
<td>POPL</td>
<td>0</td>
<td>0</td>
<td>NA</td>
<td>NA</td>
<td>0.5</td>
</tr>
<tr>
<td>PWIW</td>
<td>37</td>
<td>88.1</td>
<td>12</td>
<td>85.7</td>
<td>66.8</td>
</tr>
<tr>
<td>All</td>
<td>44</td>
<td>57.9</td>
<td>17</td>
<td>81.0</td>
<td>100</td>
</tr>
</tbody>
</table>

### Table 10
<table>
<thead>
<tr>
<th>Opr</th>
<th>Eqv telecom</th>
<th>Eqv kettle</th>
<th>Eqv CruiseControl</th>
<th>Eqv all</th>
<th>Contr (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCCE</td>
<td>4</td>
<td>25.0</td>
<td>3</td>
<td>60.0</td>
<td>8.6</td>
</tr>
<tr>
<td>PCLO</td>
<td>0</td>
<td>0</td>
<td>NA</td>
<td>NA</td>
<td>0.5</td>
</tr>
<tr>
<td>PCTT</td>
<td>4</td>
<td>50.0</td>
<td>3</td>
<td>100</td>
<td>10.1</td>
</tr>
<tr>
<td>POAC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>POPL</td>
<td>4</td>
<td>50.0</td>
<td>0</td>
<td>0</td>
<td>9.6</td>
</tr>
<tr>
<td>PSWR</td>
<td>1</td>
<td>100</td>
<td>NA</td>
<td>NA</td>
<td>0.5</td>
</tr>
<tr>
<td>PWIW</td>
<td>50</td>
<td>90.9</td>
<td>20</td>
<td>90.9</td>
<td>71.2</td>
</tr>
<tr>
<td>All</td>
<td>63</td>
<td>59.4</td>
<td>26</td>
<td>70.3</td>
<td>100</td>
</tr>
</tbody>
</table>

### Table 11
<table>
<thead>
<tr>
<th>Mutating approaches</th>
<th>AjMutator</th>
<th>Proteum/AJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert + before or after method name</td>
<td>55</td>
<td>41</td>
</tr>
<tr>
<td>Insert + before or after class name</td>
<td>37</td>
<td>36</td>
</tr>
<tr>
<td>Replace class name with +</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Replace return type with +</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Insert + before or after return type</td>
<td>15</td>
<td>NA</td>
</tr>
<tr>
<td>Insert + after class name</td>
<td>NA</td>
<td>17</td>
</tr>
<tr>
<td>Insert + after return type</td>
<td>NA</td>
<td>9</td>
</tr>
<tr>
<td>Insert + in declare precedence</td>
<td>NA</td>
<td>16</td>
</tr>
</tbody>
</table>
return type \texttt{void}. The example is taken from the set of mutants generated for the \textit{Timing} aspect of the \textit{telecom} program. Fig. 4 shows an example of inserting a subtype operator after the return type. The example is taken from the set of mutants generated by \textit{Proteum/AJ} for the \textit{TimerLog} aspect in the \textit{telecom} program. Using a subtype wildcard with primitive data types will not generate a different set of join points since primitive types cannot be subtyped. For the wildcard, unless there is a type that contains a primitive type name, the change can only generate equivalent mutants.

Table 11 summarizes the approaches used by \textit{AjMutator} and \textit{Proteum/AJ} to implement the operator PWIW for cases that produced the equivalent mutants. The numbers show how many mutants were generated by each mutation approach. The letters “NA” denote that none of the mutants compiled, so there was no need to identify equivalent mutants. The non-equivalent mutants of PWIW were not generated by any of the mutating approaches shown in Table 11. Instead, all the non-equivalent mutants were generated by replacing a method name with a wildcard.

There were several cases where the mutants generated by operators PCCE and PCCT were equivalent even though the mutant pointcut matched a different set of join points than the original pointcut. In many Aspectj programs, swapping \texttt{call} with \texttt{execution}, or \texttt{this} with \texttt{target} does not change the behavior of the program. We encountered such cases in both the subject programs. Such equivalent mutants can only be identified by inspecting the mutants. The tools cannot avoid producing them.

\textit{Proteum/AJ} implements operator POPL by adding or removing the parameter list in the method or constructor field of the pointcut. In the subject programs, adding a parameter list did not change the set of matched join points since the matched methods do not have parameters. Finally, operator PSWR generated only one mutant which was equivalent. The mutant removed a subtype wildcard for a class that is not inherited by any class.

\subsection*{4.3. Reduction of equivalent mutants}

Upon examining the mutation approaches that led to the production of the equivalent mutants, we discovered that many of these mutants can be eliminated by performing a simple analysis of the subject programs. We can obtain information about the class hierarchy, method signatures, state variable definitions and scope, static join points, and aspect precedence rules from the bytecode of the subject programs (e.g., using BCEL [10]).

Using such information, we suggest the following guidelines for implementing operators PWIW, POPL, and PSWR:

\begin{itemize}
  \item For operator PWIW, insert a wildcard or a subtype wildcard in the return type field only if the return type is not a primitive type. Primitive types cannot be subtyped and thus, there is no need to insert a subtype wildcard. One exception is to allow inserting a wildcard if the subject program defines classes with names that contains a primitive type name (e.g., a class name that contains the substring \textit{int}, or substring \textit{void}). Such information can be obtained from the class structure.
  \item For operator PWIW, insert a wildcard in the method name field of the pointcut only if other methods exist in the program that contain the same name. The scope of the search for method names depends on how restricted is the original pointcut. For example, if the pointcut matches a specified class, then the search should be within that class (e.g., the pointcuts in the two subject programs). However, if the pointcut uses a wildcard instead of a class name, the search can be expanded to the package. The scope can be expanded in the same way for packages.
  \item For operator PWIW, insert a wildcard in the class name field of the pointcut only if there is another class name in the package that contains the class name used in the pointcut.
  \item Use a wildcard in the precedence declaration statement only if there is at least one join point that is matched by more than one advice. The insertion of a wildcard in \textit{declare precedence} statement should also be implemented as part of operator DAPC.
  \item For operator POPL, adding or removing the parameter list should only be implemented if the matched methods are overloaded.
  \item For operator PSWR, apply the removal of the subtype wildcard only if the matched classes are inherited.
\end{itemize}

By performing the analysis and following the implementation guidelines described above, we calculated that 81.3\% of the equivalent mutants generated by \textit{Proteum/AJ}, and 67.4\% by \textit{AjMutator} can be eliminated.

\subsection*{4.4. Classification of mutants}

Tables 12 and 13 show the types of mutants generated from the subject programs using \textit{AjMutator} and \textit{Proteum/AJ}, respectively. The tables show that most of the mutants are of type F1-5, while fault types F1-1, F1-3, and F1-4, were generated by one operator for each tool, and no type F1-2 mutants were generated.

We inspected the mutants to understand why the operators generated such fault types. Operator PCLO generated mutants of type F1-1 only. The reason is that the non-equivalent mutants generated by PCLO negate the logical condition in the pointcut, causing only unintended join points to be matched. We do not have mutants for PCLO using \textit{Proteum/AJ}.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
Operator & F1-1 & F1-2 & F1-3 & F1-4 & F1-5 \\
\hline
PCCE & 0 & 0 & 0 & 0 & 0 \\
PCLO & 6 & 0 & 0 & 6 & 1 \\
PCCT & 0 & 0 & 0 & 0 & 4 \\
POEC & 0 & 0 & 0 & 0 & 10 \\
POPL & 0 & 0 & 0 & 0 & 5 \\
PWIW & 0 & 0 & 14 & 0 & 0 \\
All & 6 & 0 & 14 & 6 & 30 \\
\hline
\end{tabular}
\caption{Classification of pointcut mutants generated using \textit{AjMutator}.}
\end{table}
The results for operators PCCE, PCCT, POPL, and PWIW, are similar for both tools. Operators PCCE, PCCT, and POPL only generated mutants that caused the pointcut to match an empty set of join points. For operator PCCE, the reason was that the pointcuts in both programs are bound with either this or target designators, and thus, swapping call with execution for the non-equivalent mutants causes the pointcut to match different objects than the ones specified by this or target.

For operator POEC, the mutants did not match any join point since the pointcuts in neither program used the throw exception clause. This is the reason why the developers of Proteum/AJ did not generate mutants using POEC.

POPL mutants did not match any join point because there are no overloaded methods in the programs (i.e., same name with different parameter list). Otherwise, the operator can produce faults of type F1-2.

Finally, PCCT mutants are of type F1-5 because none of the objects referenced by this matched the required type of the pointcut. Operator PWIW generated mutants of type F1-3 because the operator expands the set of matched join points, and thus match a superset of the intended join points. Operator POAC, which is implemented only in Proteum/AJ, generated mutants of type F1-4 since the operator replaces after advice clauses with after returning, causing the pointcut to select a subset of the intended join points.

None of the mutation operators produced mutants of type F1-2. Generating mutants of this type requires expanding the set of matched join points to include extra join points and at the same time, narrowing the set to miss some of the intended join points. The operators implemented by AjMutator and Proteum/AJ (like all traditional first-order mutation operators) perform one change in the code, which does not guarantee that F1-2 mutants can be generated, especially when the pointcuts are bound to objects of specified types, like the pointcuts in the subject programs. This observation leads us to consider generating mutants by applying two operators. The resulting mutant is called a higher order mutant (HOM) [12,20]. Mutants can be classified into first order mutants (FOMs), which are created by applying a mutation operator once, and higher order mutants (HOMs) of degree k, which are generated by applying mutation operators more k times [12,20]. A HOM of degree two (or a second order mutant) is constructed by applying two mutation operators.

Our goal is to generate faults of the types that the FOMs missed. Given two mutants m1 and m2, where m1 produces a fault of type f1 and m2 produces a fault of type f2, the following four conditions must hold before we generate the HOMs:

1. Condition 1: Sm1 ≠ Sm2, where Sm1 and Sm2 are the sets of join points matched by mutants m1 and m2, respectively.
2. Condition 2: S_{empty} ≠ ∅, where S_{empty} is the set of join points matched by the original pointcut.
3. Condition 3: S_{m1} ≠ S_{m2} and S_{m1} ≠ S_{empty}, i.e., the mutants m1 and m2 are not equivalent to the original program.
4. Condition 4: m1 and m2 mutate the same pointcut descriptor.

Given conditions 1 through 4, we take a mutant corresponding to fault type F1-3 and one mutant of type F1-4 and combine them to get a mutant of type F1-2. We used the mutants from both the tools. If the mutants were identical, we kept one of them.

In the Kettle program, this approach gave us 2 mutants of type F1-3 and 4 mutants of type F1-4. We could generate 4 HOMs of type F1-2 because every mutant of type F1-3 matches the same pointcut in two of F1-4 mutants. In the telecom program, we have 5 mutants of type F1-3, and 18 mutants of type F1-4. We were able to generate 6 HOMs of type F1-2 because 3 of the 5 F1-3 mutants mutate the same pointcut of 2 mutants of type F1-4.

Fig. 5 shows a high order mutant generated for the telecom program. Mutant 1 generated by operator PWIW is of type F1-3 because the mutant matches a superset of the intended join points. Mutant 2 generated by operator POAC is of type F1-4 because it matches a subset of the intended join points (i.e., misses the join point when the method Timer.start throws an exception). The generated HOM is of type F1-2 because it matches a subset of the intended join points (i.e., misses the join point after an exception is thrown), and some unintended join points (i.e., after returning from other methods of class Timer).

5. Mutant generation from aspect declarations

We seeded faults in the aspect declaration statements using Proteum/AJ, which implements six operators for seeding such faults. The results of applying these operators are shown in Table 14. The table shows the number of generated mutants in column 3, the number of compilable mutants in column 4, and the number of equivalent mutants in column 5. Two aspect declaration operators mutate constructs not used in the subject programs, and are, therefore, not shown.

As the table shows, about 76.4% of the mutants compiled. The operator ABPR produces the non-compilable mutants because binding the advice with a different pointcut might leave some of the variables in the pointcut unbounded or might lead to type mismatches. For example, in the telecom program, binding the advice that should be executed after the constructor of class Call with

<table>
<thead>
<tr>
<th>Opr</th>
<th>Description</th>
<th>Gen</th>
<th>Comp</th>
<th>Eqv</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABAR</td>
<td>Change advice kind</td>
<td>30</td>
<td>30</td>
<td>14</td>
</tr>
<tr>
<td>ABPR</td>
<td>Change pointcut advice binding</td>
<td>89</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>DAPC</td>
<td>Swap aspect precedence</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>DAPO</td>
<td>Remove declare precedence</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>123</td>
<td>94</td>
<td>14</td>
</tr>
</tbody>
</table>

Fig. 5. HOM for telecom.

Table 13
Classification of pointcut mutants generated using Proteum/AJ.

<table>
<thead>
<tr>
<th>Operator</th>
<th>F1-1</th>
<th>F1-2</th>
<th>F1-3</th>
<th>F1-4</th>
<th>F1-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCCE</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>PCCT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PCLO</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>POPL</td>
<td>0</td>
<td>0</td>
<td>36</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>PWIW</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>All</td>
<td>3</td>
<td>0</td>
<td>16</td>
<td>43</td>
<td>35</td>
</tr>
</tbody>
</table>
the pointcut that matches the join points after classConnection
constructor is executed, leads to type mismatches.

Only 14 of the mutants were equivalent, which is about 15% of
the compiled mutants. These mutants were generated by replacing
after with before in the advice declarations that match the con-
structors of classes in the telecom and CruiseControl programs.
The advises perform initialization of the intertype state variables
and can run either before or after the constructor. These equivalent
mutants cannot be detected by performing our simple analysis be-
cause deciding whether they are equivalent or not requires infor-
amation about the semantics of the program that our analysis
cannot provide. We identified these equivalent mutants by manu-
ally inspecting the code.

The declaration operators implemented by Proteum/AJ map di-
rectly to their expected fault types in category F2. Operator ABAR
generates mutants of type F2-7, operator ABPR generates mutants
of F2-8, and operators DAPC and DAPO generate mutants of type
F2-5. We could not seed faults of type F2-1 and F2-2 because Pro-
teum/AJ does not have operators that can seed such types of faults.
Other fault types (i.e., F2-4 and F2-6), do not occur in the subject
programs.

6. Mutant generation from aspect implementations

We used Proteum/AJ and MuJava to mutate the body of the ad-
cives, aspect methods, and intertype methods. Proteum/AJ has five
operators for seeding faults in the advice body, three of which gen-
erated mutants in the subject programs. To generate the rest of
fault types, we applied MuJava on the aspects using the procedure
described in Section 3.3.

6.1. Compiling mutants and identifying equivalent mutants

The results of applying Proteum/AJ on the subject pro-
grams are shown in Table 15. Applying these operators did not pro-
duce any non-compilable or equivalent mutants. Operators ABHA
and APSR did not generate any equivalent mutant because the mu-
tant with a removed advice or method call is not equivalent to the
original program, unless the advice or the method does not effect
the program behavior. Operator APER, which changes the guarding
condition that surrounds the proceed statement in around advice,
did not generate equivalent mutants because the mutants either
call the advised method when it should not be called, or do not call
the advised method when it should. Both cases are not equivalent
to the original program. Note that since the telecom program does
not have any around advice, operators APER and APSR only gener-
ated mutants in the kettle and CruiseControl programs.

The results of running MuJava for the subject programs are shown in Table 16. We do not show the number of generated mu-
tants since all of them compiled. Operator AOIS, which inserts
arithmetic operators in arithmetic expressions, generated about
47.3% of the equivalent mutants. These mutants resulted from
mutating local variables and method parameters that are not used
after the statement where they are mutated. Operator ROR, which
replaces relational operators, produced about 37.8 of the equiva-
 lent mutants. The rest of equivalent mutants were generated by
the operator LOI, which inserts logical operators, and the operator
COI, which inserts conditional operators. Our results for aspect
implementation are close to the results obtained for object-orien-
ted programs (e.g., [30]). This is because the code in the advice
and intertype implementation is a Java code and applying the Mu-
Java operators produce similar types of faults as in Java programs.

6.2. Reduction of equivalent mutants

The equivalent mutants that the operators ROR, LOI, and COI
generated resulted from mutating relational and logical expres-
sions that evaluate to the same value as the original expressions.
Identifying these mutants requires testing the values of logical
expressions dynamically. Static analysis alone cannot evaluate
the values and identify whether these mutants are equivalent.

Researchers have often used data-flow analysis to identify cer-
tain types of equivalent mutants [27]. Data-flow analysis can help
in identifying the equivalent mutants generated by AOIS by per-
forming intra-method data-flow analysis for the methods in the
subject programs. That is, we find, the set of uses (i.e., statements
where the variable is read), for each local variable and parameter.
When mutating, the tool can check whether the mutated def (i.e., a
variable is defined), has further uses in the method. In other words,
there is a def-clear path between the def and at least one use. If not,
the mutant can be discarded.

An example of such a mutant is shown in Fig. 6, which shows a
mutated around advice from the Kettle program. Operator AOIS
mutated the advice by changing the statement proceed(t, amt) to
proceed(t, amt++). However, since the parameter amt does not have
any use after the def added by the mutation operator, the mutant is
equivalent. Data-flow analysis can be performed for the def in the
mutated statement to decide whether or not the def can reach any
use of the parameter. We manually performed data-flow analysis
for the aspect mutants generated by the AOIS operator in the sub-
ject programs. Our results show that 47.3% of the equivalent mu-
tants in the aspects can identified (i.e., all the equivalent mutant of
the AOIS operator).

6.3. Classification of mutants

Table 17 shows the results of the classification of the mutants
generated from the advice, intertype methods, and aspect meth-
ods. Each number shows how many mutants of each type were
void around(Kettle t, int amt): pchAddWater(t)

64 args {amt} {

if (t.waterAmount+amt>t.size) {

t.waterAmount=t.size;

} else proceed(t, amt++); //mutated statement

Fig. 6. Equivalent AOIS mutant that can be identified using data-flow analysis.
generated by each operator. Proteum/AJ operators map directly to the fault types they target. Operator APER faults are of type F3-3 because the advice body removal operator causes the same effect as the advice not being implemented at all. About 40% of the mutants generated by MuJava operators are of type F3-2. This is expected because these operators perform many incorrect changes to the state variables. MuJava operators seeded faults of type F3-1 by incorrectly changing the guarding condition of the proceed statement. Thus, our results show that all types of faults can be seeded using Proteum/AJ and MuJava.

7. Mutant generation from Java classes

We used MuJava to mutate the Java classes of the subject programs.

7.1. Compiling mutants and identifying equivalent mutants

Table 18 shows the data we obtained. We combined all class mutants into one type called “Class” for space limitations. MuJava produced 32 non-compilable mutants with the ajc AspectJ compiler. These constitute about 3.3% of the generated mutants. All non-compilable mutants resulted from the change of a static variable to a state variable, while these static variables have uses in the aspects and are referenced by the class name.

Operator AOIS produced about 52.3% of the equivalent mutants. As in the aspects, these mutants resulted from applying arithmetic operators on local variables or parameters such that the variables were not used after the mutated statement. Class equivalent mutants resulted from quantifying state variable names with this pointer in methods that do not have local variables or parameters with the same name. Finally, the two equivalent mutants generated by LOI resulted from negating variables that compared with the same name. Finally, the two equivalent mutants generated by the class operators can be eliminated by providing the tool with information about the class state variables, and local variables and parameters of methods. Therefore, 73.5% of the equivalent mutants can be eliminated.

7.2. Reduction of equivalent mutants

As in the advices, equivalent mutants generated by the AOIS operator can be eliminated by performing data-flow analysis for the methods of the classes. Moreover, the equivalent mutants generated by the class operators can be eliminated by providing the tool with information about the class state variables, and local variables and parameters of methods. Therefore, 73.5% of the equivalent mutants can be eliminated.

7.3. Classification of mutants

Table 19 shows the classification of the mutants generated from the classes in the subject programs. About 46.7% of the faults were of type F4-1. The reason for having such a high percentage of faults of this type is that operator AOIS, which generates about 40.1% of the non-equivalent mutants, intensively applies mathematical operators on the state variables, which puts the objects in a state that is unexpected by the advices. The low percentage for the F4-2 faults resulted from having only two advices in the subject programs that receive exactly one argument from the methods they advice. As the results show, the operators implemented by MuJava can generate mutants of all the types in the revised fault model.

8. Evaluation of the difficulty of killing the mutants

In order to measure how hard it is to kill the generated mutants, we generated 10 test suites for each program and calculated the mutation score obtained by the test suites for the mutants of each operator. The test suites satisfy the block coverage criterion. We used CodeCover\(^2\) to measure the coverage for the block coverage obtained by the test suites. CodeCover is an open source tool that measures statement and branch coverage in Java programs. However, the tool does not support AspectJ programs. In order to measure the coverage in the aspects, we applied the tool on the @AspectJ annotation version of each program. We manually identified unreachable code by inspecting the subject programs and dropped unreachable blocks from the coverage computation.

We generated the test suites by randomly selecting test cases from the pool of test cases as described in Section 3.3. We did not set a limit on the size of any test case. Tables 20–22 show the results obtained for AjMutator, Proteum/AJ, and MuJava, respectively. Column 2 of the tables show the number of mutants created for each type. Column 3 of the tables show the number of mutants that were not killed by any of the 10 test suites. Column 4 shows the average mutation score obtained by the 10 test suites for each of the operators.

The results show that there is a large variation in the mutation score, ranging from 25% for mutants generated by PCTT in both

---

\( ^2 \) http://codecover.org.
AjMutator and Proteum/AJ, to 100% for operators APER, APSR, DAPC of Proteum/AJ, and ASRS of MuJava. The mutants generated by the last four operators were easy to kill because the changes they introduce in the program affect all the control-flow paths. Therefore, the test suites kill all these mutants.

We manually inspected the mutants that were not killed by any test suites. Our goal was to know whether the mutants were not killed because they are hard to kill, or whether the test suites were too weak and could be improved to kill these mutants. We observe the following about the “always alive” mutants:

- The “always alive” mutants of operator POAC resulted from changing after to after returning. After advices are executed after a method returns or throws an exception, while after returning only applies to normal returning from a method. These mutants are hard to kill since they require test cases that cause an exception to be thrown in the advised methods (i.e., test cases that expect the after throwing part to be executed).
- Operator DAPO generated a mutant that removes the declare precedence for each subject program. The mutant was killed by all the test suites for the telecom program but was never killed for the Kettle program. The reason is that in the absence of a declare precedence directive, AspectJ chooses to execute aspects in an arbitrary order. In the Kettle program, the arbitrary order happened to be the same order that was specified in the original program.

The class operator that changes state variables to static generated 6 out of the 10 “always alive” mutants. We consider these mutants as hard to kill because killing requires verifying the values of the state variables of more than one object of the same class. In other words, killing them requires test suites that cover intra-class data-flow interactions.

The remaining “always alive” may be killed by test suites that satisfy data-flow criteria. In our previous work [34], we show how data-flow test criteria can be effective in detecting faults in AO programs.

For the HOMs, the average percentage of killed mutants was 57%, with 4 mutants that always remained alive. The HOMs were as easy to kill as the easiest FOM that was used to generate them. All the HOMs were generated using operators PWIW and POAC. If there were test suites available that killed the FOMs, then the test suites also killed the HOM. For example, the POAC operator generated two types of mutants, one by replacing after with after throwing, which was an easy-to-kill mutant, and the other by replacing after with after returning, which is a hard-to-kill mutant. When the hard-to-kill POAC mutant is combined with a PWIW mutant to generate a HOM, the resulting mutant becomes easier to kill than the POAC mutant that generated it. This type of HOM is shown in the example in Fig. 7 where a HOM is generated using the easy-to-kill PWIW mutant, which causes the advice to run after every method of the class, and the hard-to-kill mutant of POAC, which causes the advice to run only when the method has a normal return. A test suite that killed a PWIW mutant was also able to kill the HOM, even though the test suite did not kill the POAC mutant.

9. Threats to validity

All empirical studies have limitations. We identify four types of threats to the validity of our empirical study: internal validity, external validity, construct validity, and statistical conclusion validity.
Internal validity is concerned with cause and effect relationships, the extent to which we can state that the changes in dependent variables are caused by changes in independent variables. We recognize one internal threat to validity, which is related to the test cases that we used to evaluate how hard it is to kill the mutants. A more effective test suite may kill more mutants than a less effective test suite and might indicate that the mutants are easy to kill. Thus, we performed our experiment by creating test suites that obtain complete block coverage. Nevertheless, there are issues related to the length of the test cases generated by RANDOOP. Even though all the test suites achieve the same block coverage, because the test cases may contain method call sequences of different lengths, there may be differences in branch or data flow coverage, which may lead to some test cases performing better than others. We set RANDOOP to produce high quality test cases (e.g., no limit on the size of the test cases and the use of the observer option). We also minimized the threat by the use of 10 different test suites, each of which contains different test cases. The threat can be further minimized by producing test cases using another tool.

External validity refers to how well the results can be generalized outside the scope of the study [15]. The external validity of this study is limited mainly by the subject programs. We studied three programs and there is no evidence that the results can be extended or generalized to other aspect-oriented programs. Depending on the constructs that are present in the programs, some operators may or may not be applicable. However, as mentioned earlier, the programs contain many characteristics of aspect-oriented software. The sizes of the studied programs are relatively small (less than or around 1,000 lines of code). The larger the program, the more difficult it is to analyze and manually identify equivalent mutants.

Construct validity refers to the meaningfulness of measurements [21]. We have four dependent variables in this study: the percentage of non- compilable mutants generated by an operator, the percentage of equivalent mutants generated by an operator, the percentage of equivalent mutants that can be reduced by analyzing the program, and the percentage of mutants corresponding to an operator killed by the test suites. The percentage of the non-compilable mutants reflects the percentage of computational time required to generate and identify the non-compilable mutants. For the percentage of equivalent mutants generated by an operator, it is possible that the time and difficulty to identify every type of equivalent mutant is not equal. Some equivalent mutants can be hard to identify and therefore require more time and effort. However, a large number of equivalent mutants should minimize the impact of the variability. The percentage of equivalent mutants that can be reduced by performing our analysis reflects the reduction in time and effort that can be achieved; the cost of our analysis is negligible compared to the cost of identifying equivalent mutants.

Statistical conclusion validity is related to the question whether the presumed causal variable and its effect are statistically related. While we used 10 different test suites, we did not perform statistical analysis of the data. More programs and test suites will be necessary for statistically conclusion validity.

10. Conclusions and future work

We presented an empirical study that aims to evaluate the generation of mutants for AspectJ programs in terms of the percentage of the equivalent mutants generated by the mutation operators, the distribution of the mutants into aspect-oriented fault types, and the difficulty of killing the mutants. We also revised existing fault models and presented a fault model that solves the problems with previous fault models. We also defined three fault types that occur due to incorrect data-flow interactions in aspect-oriented programs.

We classified the generated mutants according to the revised fault model. Our results show that MutJava and Proteum/AJ can seed faults of all types that occur in the implementation classes, and the advices and methods in the aspects. However, both AjMutator and Proteum/AJ did not generate one type of faults in the pointcut descriptor. The missed fault type requires performing two changes in the pointcut descriptor. We propose using HOMs for generating mutants of this type. We presented a set of rules that can be used to produce pointcut HOMs from FOMs. HOMs can be a promising approach for generating mutants of fault types that FOMs cannot produce, or to produce mutants that are harder to kill.

Our main contribution is the proposed approach for eliminating equivalent mutants. As the empirical results show, the mutation operators generate a high percentage of equivalent mutants. Detecting these mutants is costly in terms of the time and effort of manually inspecting them. Therefore, we proposed an approach for eliminating equivalent mutants by performing a static analysis of the subject programs. Below we summarize the results of the reduction of equivalent mutants in Table 23 for each tool. In column 2 of the table, we show the number of compiled mutants generated by each tool, column 3 shows the operators that can be targeted by our approach, and columns 4 through 6 show the percentage of equivalent mutants before and after applying our analysis approach, and the percentage of the reduction achieved, respectively. As these results show, the reduction achieved for AjMutator and Proteum/AJ is higher than the one achieved for MutJava. However, for all these tools, the average reduction rate is about 80%, which shows that our approach is promising.

For the operators implemented by AjMutator and Proteum/AJ, and the class operators of MutJava, static analysis needs to provide the tools with information about the static structure of the classes. For the AOIS operator of MutJava, static analysis is available. Second, we will perform a large scale empirical study on several larger subject programs in order to evaluate the effectiveness of the proposed equivalent mutant reduction approach. Finally, we will investigate the use of HOMs for aspect-oriented programs. In particular, we will investigate combinations of first order mutants in different constructs of the aspect, not just the pointcut (e.g., one mutant in the pointcut combined with another mutant in the advice). We will also investigate orders higher than 2, and subsuming HOMs. We will also evaluate the use of HOM to...
avoid equivalent mutants and compare HOMs with our proposed approach.

Acknowledgments

We would like to thank Fabiano C. Ferrari, Elisa Y. Nakagawa, Awaï Rashid, and José C. Maldonado, the developers of Protemu/Aj, for generating the mutants for us and for answering questions regarding the tool. We also thank Carlos Pacheco for providing us with a new version of RANDOOP.

References