An approach and tool for measurement of state variable based data-flow test coverage for aspect-oriented programs

Fadi Wedyan a, Sudipto Ghosh b,*, Leo R. Vijayasarathy c

a Department of Software Engineering, Hashemite University, Zarka, Jordan
b Department of Computer Science, Colorado State University, USA
c Department of Computer Information Systems, Colorado State University, USA

Article info
Article history:
Received 31 July 2014
Received in revised form 7 November 2014
Accepted 25 November 2014
Available online 3 December 2014

Keywords:
Aspect-oriented programming
Control and data flow test criteria
Empirical study
Higher order mutation
Mutation testing
Testing tools

Abstract
Context: Data-flow testing approaches have been used for procedural and object-oriented programs, and shown to be effective in detecting faults. However, few such approaches have been evaluated for aspect-oriented programs. In such programs, data-flow interactions can occur between base classes and aspects, which can affect the behavior of both. Faults resulting from such interactions are hard to detect unless the interactions are specifically targeted during testing.

Objective: This paper presents an approach and tool implementation for measuring data-flow coverage based on state variables defined in base classes or aspects in AspectJ programs. The paper also reports on an empirical study that compares the cost and effectiveness of data-flow test criteria that are based on state variables with two control-flow criteria.

Method: Effectiveness of the criteria was evaluated for various fault types. Cost-effectiveness of test suites that cover all state variable definition-use associations (DUAs) was evaluated for three coverage levels: 100%, 90%, and 80%.

Results: The effort needed to obtain a test case that achieves data-flow coverage is higher than the effort needed to obtain a test case that covers a block or a branch in an advised class. Covering certain data flow associations requires more effort than for other types of data flow associations. The data-flow test criteria based on state variables of a base-class are in general more effective than control-flow criteria.

Conclusions: Overall, it is cost-effective to obtain test suites at the 90% coverage level of data-flow criteria.

1. Introduction

A software system can contain two types of concerns: core concerns, which refer to the main behaviors that are needed by the system, and crosscutting concerns, which refer to the behaviors that are common to multiple system core modules (i.e., modules that implement core concerns). While object-oriented programming (OOP) provides a methodology for modeling the system core concerns through the use of objects, the implementations of the crosscutting concerns are scattered throughout several core modules [1]. Aspect-oriented software development (AOSD) is a programming paradigm that supports the modularization of crosscutting concerns.

AspectJ [1] is an aspect-oriented (AO) extension to Java and is considered to be the de facto standard for aspect-oriented programming (AOP). In AspectJ, a crosscutting concern is modeled using a construct called an aspect. The weaving mechanism of AspectJ integrates aspects with core concerns (i.e., the base classes) to produce an AO program. An aspect contains three main components: (1) a pointcut, which specifies where an aspect can intercept the execution of the base class methods at locations called join points, (2) an introduction, which adds attributes and methods to classes, and (3) an advice, which provides implementations of the crosscutting concern [1].

Several approaches have been proposed for testing AO programs. Based on the artifact used to derive the test cases, these approaches can be classified as model-based [2–4], structural [5–9], or mutation [10–13]. A major concern about model-based testing for AO programs is the lack of AO constructs in existing specification languages or modeling languages. Therefore, it is hard to automate the process of test generation and coverage measurement without having modeling tools for AO programs. Moreover, a problem of using model-based testing is that most software systems are not formally or even informally specified. Furthermore, even when a program is specified, model-based testing may not...
detect errors caused by implementation details not addressed in the specifications [14].

Structural testing has been widely used for procedural and object-oriented programs. In the context of AOP, existing work in structural testing concentrates on defining control and data-flow test criteria, and addressing challenges specific to testing AO programs [6–8]. Wedyan and Ghosh [15] presented a fault-model that describes the types of faults in AO programs. New types of faults might result from the data-flow interactions in an AO program. In particular, aspects can alter the values of the base class state variables. Aspects can also introduce new methods and state variables to the base class, and can change the class hierarchy. Aspects can be affected by the base class methods either through the parameters passed by a method that has a matching join point (called advised method) to the advices, by the base class state variables used in the aspects, or by base class method calls made from the advice. Faults resulting from incorrect data-flow interactions might be difficult to detect unless such interactions are considered by the testing approach. While approaches to data-flow testing exist for procedural and object-oriented programs, there is a need to extend these approaches to better support the new concepts and constructs of AOP languages.

Existing data-flow test criteria do not consider all types of data-flow interactions in an AO program. For example, Lemos et al. [6] proposed a test criterion called all-crosscutting-uses, which requires covering data-flow interactions based on parameter passing from the advised methods in the base class to the advice. However, data-flow interactions that are based on state variables are not considered, whether these are interactions between the aspects and the base classes or between advices in the same or different aspects. Further, there are few empirical studies that evaluate the cost and effectiveness of the test criteria.

Wedyan and Ghosh [16] proposed a set of test criteria that require covering the data-flow interactions that are based on state variables. State variable def-use associations (DUA}s) were classified into five types and proposed data-flow test criteria called aspect-oriented state variable (AOSV) test criteria, which require covering these interactions.

A tool called Data-flow Coverage Tool for AspectJ (DCT-AJ) was implemented to measure coverage for the AOSV test criteria. DCT-AJ is built on top of an existing framework called AJANA [17], which creates the interprocedural control flow graph (ICFG) of a given AO program. In order to obtain the DUAs between public methods, the ICFG was extended with frame edges that connect base class public methods. DCT-AJ works in three phases: (1) DUA identification, in which it obtains the DUAs for the state variables, (2) instrumentation, in which the program is instrumented using an AO approach with code that can monitor the execution of the DUAs and measure their coverage, and (3) test execution of test suites that satisfy the AOSV test criteria and generation of coverage reports.

A cost-effectiveness study was conducted to compare the AOSV criteria with two control-flow criteria. These are: (1) AO blocks criterion, which requires exercising all the blocks in the methods of the advised class, and (2) AO branches criterion, which requires exercising all the branches in the methods of the advised class. The aim is to answer the following questions:

1. What are the relative costs of using the different test criteria?
2. What is the relative effectiveness of each criterion in terms of their ability to detect faults?
3. What types of faults can be detected by using test suites that satisfy the AOSV test criteria?
4. What is the cost-effectiveness of achieving various coverage levels (80%, 90%, and 100%) for the AOSV test criteria?

The rest of the paper is organized as follows. AspectJ concepts and terminology are summarized in Section 2. Section 3 describes the approach using the AOSV test criteria. Section 4 describes the prototype implementation of the approach. Section 5 describes how the empirical study was set up for evaluating the cost and effectiveness of the AOSV test criteria. Section 6 presents the results of the empirical study. Section 7 describes the threats to validity. Section 8 summarizes related work in testing AO programs. Finally, conclusions and future work are discussed in Section 9.

2. AspectJ concepts

AspectJ [18] is a general purpose AO extension to Java. In AspectJ, a crosscutting concern is modeled using a construct called aspect. Aspects can crosscut the system base classes (i.e., classes that implement core concerns) to define the behavior of concerns that they implement. An AO program refers to a base class and all aspects that affect it. The process of integrating aspects with the base classes is called weaving. AspectJ performs weaving by inserting well-defined points in the execution of a program called join points. During program execution, when a join point is reached, a piece of code from the aspect gets executed. Join points can be method or constructor calls and executions, the handling of exceptions, or field assignments and accesses. In AspectJ, crosscutting is called dynamic if join points refer to events during the flow of execution of the program; otherwise, crosscutting is static [18].

An aspect encapsulates three main components: pointcuts, advices, and introductions. Pointcuts are program elements that select join points using pointcut expressions. A pointcut expression is a predicate that matches join points. AspectJ provides a set of primitive pointcut expressions, called designators, that can be used to target the desired join points. A pointcut expression consists of one or more pointcut expressions combined using logical operators. Pointcuts are used by advices that contain code to be executed when execution of the base class reaches a join point. An advice can be executed before, after, or in place of a join point (called before, after, or around advice, respectively). Advices are method-like components that can have parameters and local variables. Parameters allow developers to pass (also called publish) data from base classes to advices. AspectJ has three designators that can be used to publish join point context data for the advice’s arguments, this, target and args. This returns the currently executing object (i.e., the object referenced by this in Java), target returns the target object of a join point, and args passes the arguments of an advised method to advices. Finally, introductions, also called inter-type declarations, are declarations that allow changing a program’s static structure. Using these declarations, an aspect can: (1) add methods, constructors, or state variables to classes, (2) add concrete implementation to an interface, (3) declare that a class extends a new class or implements a new interface, (4) declare aspect precedence, and (5) declare new compilation error and warning messages.

Aspects can also contain methods, data fields, and default constructors (i.e., constructors without parameters). Aspect components can be named. Naming components like pointcuts allows developers to use the component in more than one place. In AspectJ, Java rules for inheritance and polymorphism apply to aspects. For example, in an abstract aspect, a named pointcut or an aspect method can be defined as abstract to allow sub-aspects to provide their own implementations.

The concepts are illustrated using a Kettle program taken from Wedyan and Ghosh [16]. This code will also be used in Section 3 to illustrate the data-flow criteria. Fig. 1 shows the Java class Kettle, which simulates the functionality of an electric kettle for heating.
water. The class has methods for adding water and pouring water from the kettle. Kettle objects can be in four states indicated by the state variable, status: (1) OFF, where the device is off power and cannot heat water, (2) ON, where the device is turned on and ready to work, (3) HEATING, where the device is heating the water it contains, and (4) HOT, where the device heated the water it contains to the boiling temperature. Class Kettle is the base class. The aspects shown in Figs. 2 and 3 affect it. A class is affected by an aspect if: (1) an aspect advises one or more methods in the class, (2) an aspect introduces one or more methods or state variables to the class, or (3) the aspect changes the class inheritance hierarchy.

Fig. 2 shows the HeatControl aspect, which optimizes the power consumption of the kettle. The aspect introduces to the Kettle class a state variable called temperature, which holds the value of the water temperature in the kettle. The aspect also introduces methods for reading and setting the temperature value. The aspect defines an after advice that sets the kettle status to HOT when the temperature of the water reaches 100 degrees Celsius. The advice is executed after each method or constructor of class Kettle. The HeatControl aspect also defines around advices for the Kettle class methods, pourWater and addWater, to ensure that the amount of water in the kettle does not go below zero or exceed the kettle size.

The SafetyControl aspect shown in Fig. 3 defines an advice that executes after each Kettle method or constructor, and turns the kettle off when it becomes empty. The declare precedence statement in the SafetyControl aspect specifies that if a join point has advices from the two aspects, then the precedence of the advice will be the order stated in the list. The after advices from the HeatControl and SafetyControl aspects match the same join points (i.e., after each class constructor or method). Using the declare precedence statement ensures that the kettle status is set to OFF rather than HEATING when it becomes empty.

**3. Data-flow test criteria for aspect oriented programs**

Wedyan and Ghosh [16] proposed a group of data-flow test criteria, called AO state variable (AOSV) test criteria, that cover all data-flow interactions in an AO program that are based on state variables. This section previews some data-flow testing concepts and summarizes the definitions of the AOSV test criteria. For a complete description of the AOSV test criteria, please refer to Wedyan [19].

Data flow testing ensures that a definition of a variable and its subsequent uses are exercised. A definition, def, of a variable $v$
occurs in a node of a control flow graph (CFG) where \( v \) is given a value; a use of \( v \) occurs in a node where \( v \) is accessed. For a variable \( v \), a definition-use association (DUA) is a triple \( \langle v, d, u \rangle \) where node \( d \) contains a def of \( v \), node \( u \) contains a use of \( v \), and there is a def-clear path from node \( d \) to node \( u \). A def-clear path from node \( d \) to node \( u \) for variable \( v \) is a path \((d, n_1, n_2, \ldots, n_k, u)\), \( k \geq 0 \), containing no defs of \( v \) in nodes \((n_1, n_2, \ldots, n_k) \) [20].

Data flow test criteria are used to select particular DUAs as the test requirements for a program. The all-defs criterion requires exercising at least one use for every definition of a variable. Depending on the scope of the DUAs, the DUAs can be defined using methods CFGs (i.e., intra-method testing), Inter-procedural Control Flow Graph (ICFG) for inter-method level testing, or framed CFG, for inter-class level testing.

### 3.1. Approach for constructing ICFG

The data-flow test criteria described below are defined using a framed CFG for AO programs. The framed ICFG is obtained by adding a call frame to the ICFG produced using AJANA [17]. We begin by describing how AJANA produces the ICFG for AO programs. The description is demonstrated using the Kettle program shown in Figs. 1–3.

#### 3.1.1. How AJANA works

AJANA constructs the CFGs of the methods and advices in the AO program. The CFGs of advised methods are then merged with the CFGs of the corresponding advices using interaction graphs (IG). The IGs model the interaction between methods and advices at join points. An IG is built for each join point. The role of the IG is similar to that of the call graph in OO programs. A call graph shows methods calling other methods.

The steps performed at each join point that matches a before or an after advice are: (1) call-site and return-site nodes are inserted in the CFG of the advised method, (2) the call-site node is connected with entry node in the CFG of the matched advice, and (3) the exit node in the CFG of the matched advice is then connected to the return-site node. Fig. 4 shows the steps performed to obtain the CFG of the advised single-argument constructor in class Kettle.

For around advices, the CFG of the around advice replaces the CFG of the advised method. If the around advice contains a proceed statement, the CFG of the advice is connected to the CFG of the advised method using a call-site and return-site nodes. Fig. 5 shows the steps performed to obtain the CFG of the advised method, addWater. Starting from the CFG of the aroundAdd advice, we first add call-site and return-site nodes for advice afterHeat, advice afterSafety, and method addWater. After that, we connect the call and return sites with the CFGs of the corresponding advices and method.

#### 3.1.2. Extending AJANA

The ICFG shows what methods and advices are invoked from a single call to each method and constructor of the class. Fig. 6 shows the ICFG of the Kettle class. Using an ICFG, inter-procedural and intra-method DUAs can be found. For an AO program, this includes DUAs that are defined within the scope of an advised method (e.g., DUA < waterAmount.H25,S5 > in Fig. 6). However, obtaining intra-class DUAs requires having paths between the CFGs of the class public methods (whether advised or not). For example, consider the DUA < waterAmount.H25,H17 > in Fig. 6. This DUA cannot be defined unless we have a path that connects the CFGs of the method addWater and method pourWater. For OO programs, Harrold and Rothermel [14] proposed the use of a frame that provides paths between public methods. For AO programs, Zhao [9] proposed the use of a frame for the ICFG created for the class and the aspect. The frame provides possible subsequent calls to the class public methods.

Since AJANA does not provide such a frame, we constructed a frame by adding the following nodes and edges to the ICFG:

- **Frame entry** node, which represents the entry to the frame and has frame edges to the entry nodes of the CFGs of the public constructors of the base class.
- **Frame exit** node, which represents exiting from the frame. Each exit node in the CFGs of the base class public methods and constructors have frame edges connected to the exit frame node.
- **Frame edges**, which connect the exit node of the CFG of each public method and constructor to the entry node of the CFG of every public method or constructor.

Fig. 7 shows the framed ICFG for the Kettle class. The graph is not meant to be a visual display for users, but an internal model

![Fig. 4. Obtaining the CFG of the single argument Kettle class constructor.](image-url)
for DUA computation. In the figure, a regular CFG edge is shown as a solid line while a frame edge is shown as a dashed line. With the frame, the DUA \(<\text{waterAmount}; \text{H}_25; \text{H}_17>\) can be defined because of the frame edge that connects the CFGs of the method, \text{addWater}, and the method, \text{pourWater}. Methods introduced by aspects (e.g., method \text{readTemperature}) are treated as any other method of the base class. Due to space limitations, the figure shows only some of the \text{defs} and \text{uses} of the kettle state variables.

### 3.2. AOSV criteria

The AOSV criteria require executing the following types of state variable DUAs that we defined in [16]:

**Observation DUA** (\text{aDUA}): Occurs when an advice uses a state variable that a method defines. Advices may use state variables that the methods define. In Fig. 7, the \text{def} of state variables size and \text{waterAmount} at statements K11 and K12 in the \text{Kettle} class...
reach their uses in statement H25 in the HeatControl aspect. Therefore, \(<\text{waterAmount}, K12, H24>\), and \(<\text{size}, K11, H24>\) are both aDUAs.

Formally, an oDUA is a triple \(<d, u, v>\), where \(d\) is a node in the CFG of a method that contains a def of a state variable \(v\), \(u\) is a node in the CFG of an advice or aspect method that contains a use of \(v\); and there is a def-clear path between \(d\) and \(u\) for \(v\) in the framed ICFG.

**Activation DUA (aDUA):** Occurs when a method uses a state variable that an advice defines. For example, in Fig. 7, the def of the state variable waterAmount in H18 has a use in K17. Therefore, \(<\text{waterAmount}, H18, K17>\) is a aDUA.

Formally, an aDUA is a triple \(<d, u, v>\), where \(d\) is a node in the CFG of an advice or aspect method that contains a def of state variable \(v\), \(u\) is a node in the CFG of a method that contains a use of \(v\), and there is a def-clear path between \(d\) and \(u\) for variable \(v\) in the framed ICFG.

**Class DUA (cDUA):** Occurs when a method uses a state variable that a method defines (possibly the def and use are in the same method). In Fig. 7, the def of waterAmount in K12 reaches its use in K17. \(<\text{waterAmount}, K12, K17>\) is a cDUA.

Formally, a cDUA is a triple \(<d, u, v>\), where \(d\) and \(u\) are nodes in the CFGs of the base class methods that contain a def or a use of state variable \(v\), respectively, and there is a def-clear path between \(d\) and \(u\) for \(v\) in the framed ICFG.

**Aspect DUA (asDUA):** Occurs when an advice uses a state variable that is defined by an advice or method of the same aspect (possibly the def and use are in the same advice). In Fig. 7, the def of waterAmount in H25, reaches its use in H24 in aspect HeatControl. \(<\text{waterAmount}, H25, H24>\) is an asDUA. Formally, an asDUA is a triple \(<d, u, v>\), where \(d\) and \(u\) are nodes in the CFGs of the advices or methods of aspect \(s\) that contain a def or a use of state variable \(v\), respectively, and there is a def-clear path between \(d\) and \(u\) in the framed ICFG.

**Multiple Aspects DUA (maDUA):** Occurs when an advice uses a state variable that is defined in an advice or aspect method that belongs to another aspect. In Fig. 7, the def of waterAmount in H18 of aspect HeatControl reaches the use in statement S5 of aspect SafetyControl.

Formally, an maDUA is a triple \(<d, u, v>\), where \(d\) is a node in the CFG of an advice or method of aspect \(s_1\) that contains a def of state variable \(v\), \(u\) is a node in the CFG of an advice or method of aspect \(s_2\) that contains a use of \(v\), and there is a def-clear path between \(d\) and \(u\) for \(v\) in the framed ICFG of the AO program.

Given the above types of DUAs, the AOSV test criteria are defined as follows:

1. **All-uses-observation (all-usese):** Requires exercising all oDUAs in the AO program at least once.
2. **All-uses-activation (all-usesa):** Requires exercising all aDUAs in the AO program at least once.
3. **All-uses-class (all-usesc):** Requires exercising all cDUAs in the AO program at least once.
4. **All-uses-aspect (all-usesa):** Requires exercising all asDUAs in the AO program at least once.
5. **All-uses-multiple-aspects (all-usenes):** Requires exercising all maDUAs in the AO program at least once.
6. **All-uses-state (all-usess):** Requires satisfying the all-usese, all-usesa, all-usesc, all-usesa, and all-usenes criteria.

The nature of the DUAs (i.e., location of the def and use) makes subsumption relationships not possible except for all-usesa, which subsumes all the other criteria.
4. Tool implementation

In this section, we describe the tool that we developed to measure the coverage of the AOSV test criteria. Our tool, called Dataflow Coverage Measurement Tool for AspectJ Programs (DCT-AJ), works in three phases: DUA identification, program instrumentation, and test execution. Fig. 8 shows the steps involved in using DCT-AJ and how the components inside the tool interact. In Sections 4.1, 4.2 and 4.3, we describe each of the three phases of DCT-AJ. We describe the limitations of DCT-AJ in Section 4.4. Appendix A provides instructions on how to use DCT-AJ.

4.1. Phase 1: DUA identification

This phase identifies the DUAs for the state variables in each class of the subject program. DCT-AJ depends on AJANA to produce the ICFG for each class. We modified and extended AJANA as follows.

1. **Extend the ICFG of the class by including calls to non-advised methods.** AJANA produces the ICFG using the interaction graphs of the advised methods (see Section 3). We extend the ICFG by adding calls to the CFGs of the non-advised methods.

2. **Add a frame to the ICFG.** We add the frame described in Section 3 to the ICFG.

3. **Process defs and uses.** The ICFG produced by AJANA identifies defs and uses of variables in each node of the ICFG. DCT-AJ parses this ICFG and builds a list of defs and uses for each state variable, where a def or a use is defined by a triple that consists of (1) the class or aspect in which the def or use resides, (2) the method or advice name that uses or defines the state variable, and (3) the statement number which contains the def or use. We changed the way AJANA deals with variables of array types; it considers every access to an array element (whether def or use) as a use of the variable. Accessing an array element is handled by two Jimple statements. Jimple is an intermediate representation suitable for optimization produced by Soot, a framework that the abc compiler is built on. AJANA uses the abc AspectJ compiler and uses the jimple representation produced by the static weaving component of the abc compiler.

4. **List the DUAs.** We implemented the iterative data-flow algorithm proposed by Pande et al. [21] to identify the DUAs of the state variables. Our implementation does not deal with aliasing.

5. **Map jimple method names to bytecode method names.** The Jimple representation produces method and advice names different from their corresponding names in the program bytecode. Therefore, DCT-AJ parses the program bytecode, using the Apache Bytecode Engineering Library (BCEL), and maps Jimple methods and advice names to their corresponding bytecode names.

6. **Classifying and saving DUAs.** DUAs are classified according to the types described in Section 3. Finally, DUA information is saved in a file in an XML format. We save the type of the DUA, and for each def (or use) of a state variable, we save the class name, method name, source code line number in which the def (or use) occurred, and whether or not it occurred in an intertype method.

4.2. Phase 2: Instrumentation

The goal of the instrumentation phase is to produce bytecode instrumented with code that can monitor the execution of the targeted DUAs and measure their coverage. We used an aspect-oriented approach for performing the instrumentation because monitoring the execution of the DUAs is a crosscutting concern that can be implemented with AOP. Moreover, we could use the AspectJ weaver to perform the instrumentation of bytecode instead of having to write an instrumenter ourselves. DCT-AJ parses the bytecode of the classes and aspects, and the previously generated XML files to generate two tracing aspects for each class.

4.2.1. Method call tracing aspect

This aspect traces the currently executing method or advice during program execution. The aspect name is a concatenation of the word CallTrace, followed by the package name and the class

---

Fig. 8. DCT-AJ: data-flow coverage measurement tool.

name. Therefore, identical aspect names will never be generated for two different classes. The aspect contains two pointcuts:

1. `traceMethods`, which is matched whenever a method or a constructor of the class being traced is executed.
2. `traceAdvices`, which is matched whenever an advice in an aspect in the program under test is executed.

Fig. 9 shows the pointcuts generated for the Kettle class. The `traceMethods` pointcut matches the constructor and any method defined inside the Kettle class. The `traceAdvice` pointcut uses the AspectJ designator, `adviceexecution`, which matches every advice execution. Adding the `within` designator limits the scope of the pointcut to match only executions of advices within the aspects, `HeatControl` and `SafetyControl`.

The Method Call Tracing aspect has two before advices: One is called before a method executes and the other before an advice executes. These before advices collect the currently executed method or advice information using AspectJ’s `thisJoinPoint` designator. The gathered information is passed to the Dataflow Coverage aspect. Therefore, Method Call Tracing aspect has precedence over the Dataflow Coverage aspect.

### 4.2.2. Data-flow coverage aspect

This aspect collects dataflow coverage information for a class by tracing the execution of each DUA in the program. DCT-AJ uses an abstract dataflow coverage aspect, which defines three abstract pointcuts and implements four advices. The three pointcuts are:

1. `setting`, which must match every `def` of a state variable within the class or the aspect.
2. `getting`, which must match every use of a state variable within the class or the aspect.
3. `loadTestDriver`, which must match the execution of the test driver.

The abstract aspect contains the following advices:

1. **SetTrace**: This before advice is executed when the setting pointcut is matched. The advice obtains the state variable name and statement in which the variable is defined using the `thisJoinPoint` construct. It uses the currently executing method or advice name found by the method call aspect to find which `def` of the state variable was executed. We implemented the last reaching definition approach for monitoring dataflow execution described by Misurda et al. [22]. In this approach, each `def` of a state variable, `sv`, that is executed is recorded. This `def` is called the last `Def(sv)` and is identified in the SetTrace advice. When `sv` is used, a `use` of `sv` is executed and is recorded by the GetTrace advice. The last `Def(sv)` is the `def` that reaches the use and the DUA < `sv, def, use > is marked as being covered.

2. **GetTrace**: This before advice monitors the execution of statements matched by the getting pointcut. Similar to the SetTrace advice, GetTrace gets information about the used state variable, `sv`, using `thisJoinPoint` and the currently executing method from the `Method Call Tracing Aspect`. Then the GetTrace advice matches the use of `sv` with the `lastDef(sv)` to obtain the covered DUA.

3. **LoadInformation**: This before advice is executed when the `loadTestDriver` pointcut is matched (i.e., before executing the test driver). The LoadInformation advice loads the XML file that contains the DUA information of the class.

4. **SaveInformation**: This after advice is executed when the `loadTestDriver` pointcut is matched (i.e., after executing the test driver). The advice saves the coverage information for the class in an XML file.

DCT-AJ generates a concrete dataflow coverage aspect for each class. The generated aspect inherits from the abstract aspect and provides concrete implementations of the three pointcuts. Fig. 10 shows the Dataflow Coverage aspect generated for the Kettle class. The aspect name is a concatenation of the word, DataCoverage, with the package name and the class name. The Setting pointcut uses the AspectJ pointcut designator, `set`, to match every `def` of a variable while the Getting pointcut uses AspectJ designator to match every use of a variable. Both pointcuts limit the scope of the match in the class Kettle, and aspects `HeatControl` and `SafetyControl`. The `loadTestDriver` pointcut matches the execution of the main method in the test driver of the Kettle class.

### 4.3. Phase 3: Test execution

Given a test suite, the instrumented bytecode of the classes, and the DUA information of the classes under test, the test driver runs the test suite. The Dataflow Coverage Aspect saves coverage information in the form of coverage reports at the end of the run. A report is generated for each class. The report includes the number of state variables, the number of DUs for each DUA type, and the percent of DUs of each type that were covered during testing. The report also computes the coverage for each state variable.

### 4.4. Tool limitations

The current implementation of DCT-AJ has the following limitations:

- DCT-AJ does not trace a DUA when the `def` or the `use` is inside an exception handling code. This is because AJNA does not include exception handling paths in the ICFG.
- Uses of `final` state variables cannot be traced unless they are referenced with `this reference`. This is because Java inlines `final` variables. Therefore, the get pointcut designator cannot find such uses. In order to overcome this limitation,

```
public aspect CallTrace_ekettle_Kettle {
    declare precedence: CallTrace_ekettle_Kettle,
    DataCoverage_ekettle_Kettle;
    pointcut traceMethods(): execution(* ekettle.Kettle.*(..)) ||
        execution(ekettle.Kettle.new(..));
    pointcut traceAdvices(): adviceexecution() &&
        (within(ekettle.HeatControl) ||
        within(ekettle.SafetyControl));
    // advices are not shown due to space limitations
}
```

---

*Fig. 9. Method call tracing aspect for the Kettle class.*
the user can choose one of the following options: (1) reference each use of a final state variable with this reference, or (2) remove the final declaration for state variables in the copy of the source code used for obtaining coverage with DCT-AJ.

- Initialization of intertype state variables cannot be recognized by DCT-AJ. This is because the get and set pointcut designators do not consider initializations of the intertype state variables in aspects as defs or uses of the state variables, respectively. In order to overcome this limitation, the initialization of intertype declared state variables can be performed in an after or a before advise that matches the class constructor.

- The tool cannot trace defs and uses in static methods of the aspects. This is because the ICFG generated by AJANA does not include the calls to the static methods of the aspects.

- The current version of DCT-AJ does not deal with aliasing for state variables.

- AJANA performs a must-alias analysis to identify variables that refer to the objects of the base class in the aspects. Must aliasing analysis is a conservative approach that might miss references to the base class objects in the aspects.

- The current version of DCT-AJ cannot measure coverage in multi-threaded programs. AJANA does not generate the ICFG for multi-threaded programs.

- The current version of DCT-AJ cannot measure coverage for aspects written in annotation style. To overcome this limitation, the user can rewrite the aspects in AspectJ style.

5. Experimental description

This section describes how we evaluated the approach.

5.1. Summary of aspect oriented fault model

Here we summarize the fault types used in the study. For a full description of the fault model, please refer to Wedyan and Ghosh [15].

**F1: Pointcut descriptor faults**

- **F1-1**: The pointcut matches a set of join points that contains only unintended join points [13].
- **F1-2**: The pointcut matches a set of join points that contains unintended join points and some intended join points [13].
- **F1-3**: The pointcut matches all intended join points and some unintended join points [13].

- **F1-4**: The pointcut matches a subset of intended join points and no unintended join points [13].
- **F1-5**: The pointcut does not match any join point.

**F2: Aspect declaration faults**

- **F2-1**: Incorrect method name in introduction, leading to a missing or unanticipated method override [23].
- **F2-2**: Incorrect class name in a member-introduction [23].
- **F2-3**: Incorrect declaration of parent class or interface.
- **F2-4**: Incorrect declaration of error and warning statements [12].
- **F2-5**: Incorrect aspect precedence [23,24].
- **F2-6**: Incorrect aspect instantiation rules and deployment [12].
- **F2-7**: Incorrect advice type specification [12,23].
- **F2-8**: Advice bound to incorrect pointcut [12].

**F3: Advice, aspect method, and intertype method implementation faults**

- **F3-1**: Incorrect guarding statement or missing proceed in around advice [23]: 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 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. [25] for Java methods.
- **F3-4**: Incorrect access to join point static information [12]: The construct, thisJoinPoint, contains information about the current join point for use by the advice. The construct can be used only in the context of the advice. This fault type occurs when the construct is incorrectly used.

**F4: Class implementation related 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 advices 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. [25] 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 [25]. We used the faults described by Ma et al. [25].

5.2. Control flow test criteria

The experiment used two new control flow criteria to compare with the AOSV test criteria. These criteria are modified versions of the traditional block and branch coverage criteria applied to advised classes.

1. AO blocks criterion: Requires exercising (1) all blocks in the methods of the advised class including intertype methods, and (2) all blocks in the advice methods in the advised class.

2. AO branches criterion: Requires exercising (1) all branches in the methods of the advised class including intertype methods, and (2) all branches in the advice methods that advise methods in the advised class.

The above AO control-flow criteria are defined in the scope of an advised class. Thus, they are consistent with the scope of the AOSV test criteria.

Several versions of block and branch coverage criteria have been proposed by other researchers for AO programs (e.g., [6,8,26]). However, these criteria are either defined for aspects [8,26], or for advised classes but without the advice methods and intertype methods [6]. Therefore, these criteria have different scope than the AOSV test criteria and are not directly comparable to the latter.

5.3. Metrics

The Goal-Question-Metric paradigm [27,28] was used for the experiment. The goal and questions are described in the introduction. Here, the questions are restated and the metrics used to answer each question are listed.

1. Cost of using the test criteria:
   - **Question:** What is the cost of generating test cases to achieve full coverage for each test criterion?
   - **Metrics:** Two surrogates for cost were used.
     - (a) Size metric: \( \frac{\text{number of test cases in a test suite needed to satisfy a criterion}}{\text{number of test requirements for the criterion}} \)
     - (b) Effort metric: \( \frac{\text{number of iterations needed to add a new test case until a test suite that satisfies the test criterion is obtained}}{\text{number of test requirements for the criterion}} \)

The number of test cases in a test suite needed to satisfy a criterion is a widely used metric to measure the size of a test suite [29]. However, this number varies in different classes depending on the number of test requirements for a criterion. Therefore, the number of test cases in a suite was divided by the number of test requirements of the criterion to adjust for differences in the number of test requirements among different classes.

The size of a test suite, when used as a measure of cost, does not measure the effort of obtaining the test suite nor does it measure the cost of obtaining the test requirements of the criterion. Briand [29] stated that “regardless of how it is measured, [test suite size] is a very rough cost measure” mainly because it just measures one dimension of the cost. Therefore, we defined an effort metric to measure the effort needed to obtain a test suite that satisfies a criterion.

For the study, each test suite was generated by repeatedly adding test cases to a suite until the desired coverage is obtained. Each iteration added a new test case to the test suite. The test case was not used if coverage remained the same. If coverage increased, the test case was kept and the process of adding test cases was repeated until the desired coverage level was reached. The number of iterations required until the test suite satisfied a criterion is used as a measure of cost. In order to reduce the effect of the variations in test requirements for the criterion in a class, this number is divided by the number of test requirements.

2. Effectiveness of the test criteria:
   - **Question:** With 100% coverage, what is the effectiveness of the test criteria in terms of their ability to detect faults?
   - **Metrics:** the percentage of faults detected by a test suite that satisfies a test criterion.

A number of faulty versions are generated from each program, where each faulty version contains one fault. The fault can be in the base class or in any part of the aspect that interacts with the class (i.e., in any of the advises that advice methods of the class, in aspect methods that these advises call, or in intertype methods introduced in the class). In order to determine what types of faults can be detected using each of the test criteria, effectiveness was computed over all the faults as well as for each fault type defined in our fault model.

3. Effectiveness of the test criteria in detecting fault types:
   - **Question:** What types of faults can be detected by using the AOSV test criteria? Mutation faults were used and classified according to the AO fault model presented in Section 5.1.
   - **Metrics:** the percentage of faults of each type detected by a test suite that satisfies a test criterion.

4. Cost-effectiveness of achieving high coverage of the AOSV criteria:
   - **Question:** What is the cost-effectiveness of achieving high coverage levels for the all-uses, test criterion?
   - **Metrics:**
     - (a) number of test cases in a test suite needed to satisfy the criterion for each measured coverage level.
     - (b) number of iterations needed until a test suite that satisfies the test criterion for each measured coverage level is obtained.
     - (c) percentage of faults detected by a test suite that satisfies a criterion for each measured coverage level.

Achieving high levels of coverage of some test criteria can come with relatively high cost. Therefore, it is of practical importance to know whether achieving high coverage is justified by a significant increase in fault detection. In the study, cost and effectiveness are evaluated for the all-uses, criterion for three coverage levels: 100%, 90%, and 80% coverage levels. The all-uses, criterion was chosen because it subsumes all the other AOSV criteria. Therefore, it is the most important criterion that requires covering all the state variables DUs in the advised class. Moreover, the number of test requirements corresponding to the other test criteria have large variations in the classes, which makes it hard to find a single coverage level that applies to all the test criteria. It...
was also not necessary to take into account the number of test requirements, which is only needed when comparing the criteria with each other.

5.4. Tools for coverage measurement

The DCT-AJ tool was used to measure coverage for the AOSV criteria. CodeCover\(^4\) was used to measure the coverage for the AO control-flow criteria. 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, the aspects were rewritten using the annotation style, which is a feature of AspectJ\(^5\), also known as [AspectJ annotation]. This notation allows writing aspects with regular Java syntax and then annotating the aspect declarations so that they can be interpreted by the AspectJ weaver. Since the aspect written in the annotation style appears as a Java class, CodeCover was able to measure coverage of branches and blocks in the aspect\(^6\).

Unreachable code was manually identified by inspecting the subject programs. Unreachable blocks and branches were dropped from the coverage computation for the AO control-flow criteria. For the AOSV criteria, an XML editor (XML Marker\(^5\)) was used to remove unreachable DUAs from the XML file that contains the DUAs.

5.5. Subject programs

The study used four subject programs. Table 1 summarizes their main characteristics. The characteristics can be present in aspect-oriented programs, including different types of advices, intertype declarations, multiple advices that match the same join point, and different types of data-flow interactions. The programs also cover several applications of aspect-oriented programming, such as contract enforcement, logging, and composition of separate concerns.

The classes that had no interaction with aspects were not tested directly (i.e., JUnit test methods were not invoked on objects of these classes). For this reason we provide the number of tested classes in Table 1.

The Kettle program simulates the functionality of an electric kettle for heating water\(^16\) and was described in Section 2.

Telecom is a simulation of a telephony system that is shipped with the ajc AspectJ compiler\(^18\). This program allows customers to make, accept, merge, and hangup both local and long distance calls. Telecom contains three aspects: (1) Timing aspect, which measures the connection duration for customers by initializing and stopping a timer associated with each connection, (2) Billing aspect, which specifies the payer of each call and ensures that local and long distance calls are appropriately charged, and (3) TimerLog aspect, which implements a log that prints the times whenever a connection is established or dropped.

The Banking program was developed by Laddad\(^1\) as an example of a banking account management system. A customer can have multiple accounts that are managed by two aspects as follows: (1) MinimumBalanceRuleAspect implements a minimum balance banking policy by ensuring that the balance of an account does not go below a minimum required balance, and (2) OverdraftProtectionRuleAspect implements an overdraft coverage policy by allowing a customer to cover overdraft withdrawals from the customer’s other accounts.

Xu et al.\(^30\) developed an aspect-oriented implementation of a legacy cruise control system called CruiseControl. The program contains classes that simulate a car, a cruise controller, and a speed controller. These classes interact with the following three aspects: (1) CarSimulatorFix is an incremental modification aspect that enforces the precondition that the car engine is on for accelerate and brake events, (2) CruiseControlIntegrator adds the implementation of a cruise controller to a car, and (3) SpeedControlIntegrator adds the implementation of a speed controller to a cruise controller.

Table 2 shows the number of test requirements for each of the test criteria in each class in the subject programs. The reported numbers exclude blocks and branches in exception handling segments and unreachable blocks and branches. The table shows that there are five classes that do not contain test requirements with respect to the AO All-Branches criterion. This is because CodeCover only counts branches that are produced by a decision statement. Methods or advices that contain only one path are not counted. This is also the reason why the number of branches is less than the number of blocks in the other six classes.

5.6. Generating mutants

Faults were seeded in the subject programs using mutation operators because a large number of faulty versions can be generated in a systematic and easily replicable manner\(^31\). Moreover, Andrews et al.\(^32,33\) provided evidence that faults generated with mutation operators for procedural programs are similar to real faults in evaluating test effectiveness, while hand-seeded faults are harder to detect than real faults. Just et al.\(^34\) studied 317 real faults in five open-source applications with a total of 321,000 LOC, and show a statistically significant correlation between mutant detection and real fault detection.

Multiple tools were needed to cover the fault categories. Two available tools were used for mutating AspectJ programs, these are: AjMutator\(^35\), and Proteum/AJ\(^36\). Both tools implement a subset of the mutation operators for AO programs proposed by Ferrari et al.\(^12\). While AjMutator implements only operators for mutating the pointcut descriptors, Proteum/AJ implements two operators for mutating the advice declaration and four operators for the advice implementation. AjMutator and Proteum/AJ classify pointcut mutants according to the set of join points they match compared to the set of join points matched by the original pointcut. If the two sets are equal, the mutant is classified as equivalent. The tools also identify non-compilable mutants. Proteum/AJ was not yet available for public use. Therefore, the subject programs were sent to the developers of the tool and they generated the mutants. Both tools were used because the tools can generate different mutants even by using the same operator. This is due to the implementation variations of the operators. When the operators generated identical mutants, the mutants from Proteum/AJ were kept.

---


---

Table 1

<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>2</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>Banking</td>
<td>243</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CruiseControl</td>
<td>1,008</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>2,304</td>
<td>22</td>
<td>10</td>
<td>11</td>
<td>4</td>
<td>18</td>
<td>5</td>
<td>24</td>
</tr>
</tbody>
</table>

---

\(^4\) http://codecover.org.


---

Table 2

<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>2</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>Banking</td>
<td>243</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CruiseControl</td>
<td>1,008</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>2,304</td>
<td>22</td>
<td>10</td>
<td>11</td>
<td>4</td>
<td>18</td>
<td>5</td>
<td>24</td>
</tr>
</tbody>
</table>
Table 2

<table>
<thead>
<tr>
<th>Subject</th>
<th>Class</th>
<th>oDUA</th>
<th>aDUA</th>
<th>cDUA</th>
<th>asDUA</th>
<th>maDUA</th>
<th>allDUA</th>
<th>Blocks</th>
<th>Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kettle</td>
<td>Kettle</td>
<td>16</td>
<td>10</td>
<td>12</td>
<td>10</td>
<td>6</td>
<td>54</td>
<td>21</td>
<td>16</td>
</tr>
<tr>
<td>Banking</td>
<td>Account</td>
<td>11</td>
<td>4</td>
<td>18</td>
<td>2</td>
<td>1</td>
<td>36</td>
<td>21</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Customer</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Telecom</td>
<td>Local</td>
<td>2</td>
<td>0</td>
<td>9</td>
<td>5</td>
<td>1</td>
<td>17</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>LongDistance</td>
<td>2</td>
<td>0</td>
<td>9</td>
<td>5</td>
<td>1</td>
<td>17</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Customer</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Call</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>18</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Timer</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Cruise</td>
<td>CarSimulator</td>
<td>6</td>
<td>0</td>
<td>127</td>
<td>17</td>
<td>0</td>
<td>150</td>
<td>52</td>
<td>23</td>
</tr>
<tr>
<td>control</td>
<td>CruiseController</td>
<td>45</td>
<td>0</td>
<td>65</td>
<td>6</td>
<td>0</td>
<td>116</td>
<td>32</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>SpeedControl</td>
<td>4</td>
<td>8</td>
<td>49</td>
<td>2</td>
<td>0</td>
<td>63</td>
<td>22</td>
<td>10</td>
</tr>
<tr>
<td>Total no. of DUAs</td>
<td>86</td>
<td>22</td>
<td>318</td>
<td>55</td>
<td>9</td>
<td>469</td>
<td>233</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>No. of classes with</td>
<td>7</td>
<td>3</td>
<td>11</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>11</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Both AjMutator and Proteum/AJ did not support operators that seed faults in the advice body (except for four operators implemented by Proteum/AJ that mutate AspectJ constructs), aspect methods, and introduced methods. To seed faults in the body of these constructs as well as in the base classes, \muJava [25] was used. \muJava is a widely used tool for mutating Java programs. \muJava provided two types of mutation operators, class level and method level. Class level mutation operators generate faults related to object-oriented 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 seed faults in the aspects, and to the best of the authors' knowledge, there is no tool that can directly seed faults into the advice, aspect methods, and introduced methods. Therefore, an indirect approach was used to seed faults in the aspects using \muJava. A class was generated from the aspect bytecode with the help of a decompilation tool called Jad (http://www.varanec-kas.com/jad). The decompiled class was mutated with \muJava. The mutated line that resides in the advices, intertype methods, or aspect method was selected and copied back to the aspect. This step was repeated to generate all the mutants, where each mutant contains one fault. \muJava was used directly on the base classes to generate their mutants. \muJava only produces mutants that compile. However, \muJava compiles mutants with a Java compiler by default, not an AspectJ compiler. Therefore, all mutants generated by \muJava were compiled with the AspectJ ajc compiler.

One major problem with seeding faults using mutation operators is that the operators generate a large number of equivalent mutants. In the study, about 31% of the compiled mutants were equivalent. These mutants need to be identified and eliminated before measuring the effectiveness of the test criteria. 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 program. However, this approach only applies to a subset of the pointcut mutants because there are some pointcut mutants that match different set of join points but are still equivalent to the original program (e.g., in some programs, there is no difference between using a call and execution pointcut designators). Therefore, manual inspection and identification of equivalent mutants was necessary. A mutant was considered to be equivalent if the fault seeded by the mutation operator did not propagate to produce a different output from the original program.

Judging whether a mutant is equivalent to the original program or not is an undecidable problem. It is difficult for a tester to know if the mutant is just difficult to kill (also called stubborn mutants [37]) or not possible to kill (equivalent). A study conducted by Yao et al. [38] shows that equivalence is correlated with program size and number of mutants created, but stubbornness is not correlated. There is some relationship between equivalence and stubbornness for mutation operators corresponding to the relational operator replacement (ROR). Mutation operators, such as Absolute Value Insertion (ABS) and post-increment operators, tend to generate many equivalent mutants but few stubborn mutants.

In order to answer research question 3, faults of the types described in the AO fault model were seeded as shown in the programs contained constructs that can be mutated to generate faults of each type. By classifying the mutants generated by the tools, it was observed that faults of type F1-2 were generated in only one class, while there are 7 other classes that can have faults of this type. The reason is that faults of type F1-2 require 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 (FOM) 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. Thus, mutants needed to be generated by applying two operators. The resulting mutant is called a higher order mutant (HOM) [39,40]. Wedyan and Ghosh [15] described how HOMs can be used to seed the faults that FOMs missed.

Tables 3 and 4 show the distribution of the mutants generated from each fault category. Some categories show zeroes because the program did not contain the construct that could be mutated for that category.

5.7. Generating test suites

RANDOOP [41] was used to generate JUnit test cases because it can easily generate a large number of random test cases. RANDOOP is a feedback directed random test generation tool, and is designed to generate test cases for Java programs, not AspectJ, and to the best of the authors' knowledge, there is no tool that can automatically generate test cases for AspectJ programs. The following steps were performed to adapt RANDOOP for use with AspectJ programs:

- The aspects in the subject programs were rewritten using the annotation style. In @AspectJ annotation, 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.
Test cases were generated 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, one can avoid making calls to aspect methods and advices directly from the test cases.

For each subject program, a pool of test cases was generated, starting from 3000 test cases. In order to ensure that the test cases in the pool can satisfy the test criteria, the coverage obtained for each AOSV test criterion using the test cases in the pool was measured. If full coverage was not obtained, another pool with 1000 more test cases was generated (i.e., 4000 test cases in all). This process was stopped when the pool contained at least one test case that covers every test requirement of the test criteria. The test pool sizes for Kettle, Banking, Telecom, and CruiseControl were 3,000, 7,000, 6,000, and 14,000, respectively.

RANDOOP was provided with a set of input values suitable for the subject programs. The observer option of RANDOOP was used for creating custom assertions. In the public version of RANDOOP, the tool generates assertions that check for null values, reflexivity, and symmetry of equality of the variables [42]. However, using the observers option, observer methods can be specified to generate stronger assertions that are application-specific.

Following the approach of generating test suites described in Section 5.3, 30 test suites were generated to satisfy each test criterion in each of the advised classes. The use of 30 test suites provides sufficient statistical power to test for significant differences \((p < 0.05)\) in cost and effectiveness among the test criteria.

5.8. Data analysis

IBM’s SPSS statistics software was used to analyze the data collected for the cost and effectiveness of the test criteria. Since each of the 30 test suites that was generated for each advised class was assessed on all relevant test criteria, repeated measures analysis of variance (ANOVA) was used to test for differences among the criteria. Suitable corrections were made to the F-ratio (i.e., made it more conservative by reducing the degrees of freedom), if there were any violations to the assumption of sphericity [43].

6. Results

Sections 6.1, 6.2, 6.3 and 6.4 present the results of the empirical study to investigate the four research questions presented in Section 5.

6.1. Comparing the cost of the test criteria

The cost of the AOSV criteria are compared with the cost of the control criteria using the two metrics defined in Section 5.

6.1.1. Cost comparison using size metric

Table 5 shows the means of the size metric for each criterion and for each class. The means are computed over the 30 test suites that satisfy each criterion in the class. The results of the statistical analysis are omitted due to space limitations. The results show that the size metric values for each of the AOSV DUAs (rows all-uses, all-uses\(_a\), all-uses\(_b\), all-uses\(_c\), all-uses\(_d\), and all-uses\(_e\)) are significantly higher than those for each of the AO control-flow criteria (rows AO blocks and AO branches). These results confirm the expectations that a test case can cover more blocks or branches than any of the AOSV DUAs in the advised classes. This is because a path in a test case that covers any of the AOSV DUAs contains many branches and blocks. The shortest path that covers a DUA consists of two blocks and one branch. However, most of the paths that cover the AOSV DUAs are longer because these paths exist between \texttt{defs} and \texttt{uses} in different methods or advices.
The size metric values of the all-uses criterion are significantly lower than those of the other AOSV test criteria in most cases. This is because all-uses subsumes the other AOSV test criteria, and therefore, a test case in a test suite that satisfies all-uses covers more of different types of DUAs than it covers a particular type.

The size metric for all-uses is defined as a measure of how many DUAs of the AOSV types can be covered by executing the same path. When covering the AOSV DUAs requires executing more different paths, then the size metric values for all-uses become larger. When the comparison between the size metric for all-uses and that for an AOSV test criterion shows no significant difference, then the DUAs required by the criterion that is compared with all-uses are covered by the test suites that satisfy the other AOSV criteria. For example, in the Account class, there is no significant difference between the size metric for all-uses and that of all-uses. This means that most of the cDUAs in the Account class are covered by the test suites that satisfy the all-uses, all-uses, all-uses, and all-uses criteria.

### 6.1.2. Cost comparison using effort metric

Table 6 shows the means of the effort metric for each criterion and for each class. The means are computed over the 30 test suites that satisfy each criterion in the class. The results show that the effort metric values for each of the AOSV test criteria (rows all-uses, all-uses, all-uses, all-uses, all-uses, and all-uses) are significantly higher than those for each of the AO control-flow criteria (rows AO blocks and AO branches).

The effort of obtaining a test suite that covers a criterion depends on the number of test cases in the pool that can cover the test requirements of a criterion. When the pool of test cases contains many test cases that cover the test requirements of a criterion, the effort in covering the criterion decreases since there is higher likelihood that the randomly selected test case from the pool can cover the requirements. Some classes contain AOSV DUAs that are easily covered by most of the test cases, which therefore, decreases the effort needed to satisfy the criterion. Some classes contain AOSV DUAs that require covering paths that are hard to be executed by the test cases.

The differences in the effort needed to satisfy the AOSV criteria are inconsistent over all the classes.

### 6.2. Comparing the effectiveness of the test criteria

The effectiveness of the AOSV criteria was compared with the two control-flow criteria. Table 7 shows the effectiveness results for each criterion. For each test criterion in column 1 and for each class in row 1, the table shows the average mutation score (as a percentage) obtained by the test suites that satisfy a test criterion. The means are computed over the 30 test suites that satisfy each criterion in the class. The last column in the table shows the means of the mutation scores of the test suites taken over all the classes. The overall mean is computed by dividing the total of the average number of mutants killed by the test suites in each class by the total number of mutants.

Note that all-uses, has the highest overall mutation score while AO blocks has the lowest. The mutation scores of the test suites that satisfy all-uses and all-uses are significantly higher than the mutation scores of the test suites that satisfy each of the AO control-flow criteria (rows AO blocks and AO branches) in all classes. The rest of the AOSV criteria have significantly higher mutation scores in most of the classes (39 out of 48 tests), and there are no significant differences in 9 tests.

The expectation was that criterion all-uses would be more effective than the AO control-flow criteria, and this is matched by the results. For the other AOSV criteria, differences were

### Table 5

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Kettle Account Customer (Banking)</th>
<th>Local LongDistance Customer (Telecom)</th>
<th>Call Timer CarSimulator CruiseController SpeedControl</th>
</tr>
</thead>
<tbody>
<tr>
<td>all-uses</td>
<td>0.79</td>
<td>0.64</td>
<td>0.79</td>
</tr>
<tr>
<td>all-usesa</td>
<td>0.41</td>
<td>0.50</td>
<td>0.50</td>
</tr>
<tr>
<td>all-usesb</td>
<td>0.68</td>
<td>0.34</td>
<td>0.39</td>
</tr>
<tr>
<td>all-usesc</td>
<td>0.68</td>
<td>0.92</td>
<td>0.47</td>
</tr>
<tr>
<td>all-usesd</td>
<td>0.79</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>all-usesa</td>
<td>0.29</td>
<td>0.32</td>
<td>0.49</td>
</tr>
<tr>
<td>AO blocks</td>
<td>0.18</td>
<td>0.10</td>
<td>0.17</td>
</tr>
<tr>
<td>AO branches</td>
<td>0.29</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>

### Table 6

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Kettle Account Customer (Banking)</th>
<th>Local LongDistance Customer (Telecom)</th>
<th>Call Timer CarSimulator CruiseController SpeedControl</th>
</tr>
</thead>
<tbody>
<tr>
<td>all-uses</td>
<td>8.52</td>
<td>625.11</td>
<td>6.97</td>
</tr>
<tr>
<td>all-usesa</td>
<td>2.00</td>
<td>181.14</td>
<td>5.42</td>
</tr>
<tr>
<td>all-usesb</td>
<td>7.74</td>
<td>1.58</td>
<td>60.60</td>
</tr>
<tr>
<td>all-usesc</td>
<td>16.02</td>
<td>501.00</td>
<td>60.60</td>
</tr>
<tr>
<td>all-usesd</td>
<td>7.53</td>
<td>369.37</td>
<td>60.60</td>
</tr>
<tr>
<td>all-usesa</td>
<td>3.96</td>
<td>62.47</td>
<td>60.60</td>
</tr>
<tr>
<td>AO blocks</td>
<td>1.14</td>
<td>11.17</td>
<td>28.3</td>
</tr>
<tr>
<td>AO branches</td>
<td>2.27</td>
<td>27.93</td>
<td>28.3</td>
</tr>
</tbody>
</table>
expected in their effectiveness depending on the paths that they require and the number of test requirements contained in the class. For example, the mutation scores of the test suites that satisfy all-uses, are not significantly higher than the mutation scores of the test suites that satisfy AO blocks in the Account and Customer classes of the Banking program, and the Customer class in the Telecom program. These classes contain many cDUAs between setter and getter methods that are not advised, which means that the criterion does not require executing paths in the advice. However, in the classes that contain advised methods and a high number of cDUAs, test suites that satisfy all-uses are more effective than the test suites that satisfy the AO control-flow criterion.

In the Local and LongDistance classes, the mutation scores of the test suites that satisfy all-uses, are not significantly higher than the mutation scores of the test suites that satisfy AO blocks. These two classes have only two oDUAs that are covered by executing one path. Note also that in the LongDistance class the mutation scores of the test suites that satisfy all-uses, and all-uses, are not significantly higher than the mutation score of the test suites that satisfy AO blocks. These two criteria do not require executing many different paths in the class. In the Local class, for which RANDOOP generated fewer test cases, the test cases that covered them are longer and cover many other DUAs, which increased their effectiveness.

The mutation scores of the test suites that satisfy all-uses, are significantly higher than the mutation scores of the test suites that satisfy the other AOSV test criteria, which are also subsumed by all-uses. The only exception is in the LongDistance class in which the mutation score for the test suites that satisfy all-uses, is higher than the mutation score of the test suites that satisfy all-uses, but not significantly. The reason is that the test suites that satisfy all-uses, in the LongDistance class also cover most of the other AOSV DUAs. Moreover, in this class the sizes of the test suites that satisfy all-uses, are not significantly higher than the sizes of the test suites that satisfy all-uses.

Comparisons between the other AOSV criteria show that there is a significant difference in their effectiveness as measured by the mutation score. However, there is no consistency in the difference between the criteria. The differences are dependent on how many different paths the DUAs cover in the advised class. For example, all-uses, is significantly more effective than all-uses, in the CruiseController class and significantly less effective than all-uses, in the CarSimulator class. That is because the CruiseController class contains 45 aDUAs and only 6 aDUAs, while the CarSimulator class contains 17 aDUAs and 6 oDUAs. The number of DUAs is not the only factor that decides the effectiveness. However, when a criterion requires a high number of DUAs, more paths are covered in the advised class.

### 6.3. Effectiveness of the AOSV test criteria based on fault types

The effectiveness of the proposed AOSV criteria was investigated for the fault types described in Section 5.1. For each fault type, the live mutants were inspected and investigated for determining why the mutants were not killed by the test suites that satisfy all-uses, (i.e., the test suites that cover all types of AOSV DUAs). This section also explains why there are differences between the mutation scores of the test suites that cover each type of AOSV DUAs.

#### 6.3.1. Pointcut descriptor related faults (F1)

Table 8 shows the effectiveness of the AOSV test criteria in detecting faults of the types belonging to category F1. For each fault type in column 1, columns 2 through 7 show the average mutation score (as a percentage) for the test suites that satisfy each of the AOSV criteria averaged over all the classes in the benchmark programs. The symbol "na" for all-uses, is used to indicate that the mutation score for the criterion is not available because the class does not contain aDUAs. The last three rows show for each criterion the total number of mutants that are generated in the classes, the number of classes tested by the criterion, and the overall mutation score of the test suites that satisfy the criterion. The overall mutation score is computed by dividing the total of the average number of mutants killed by the test suites in each class by the total number of mutants in the classes tested by the criterion and have faults in category F1.

Note that all-uses, subsumes all-uses, but all-uses, results in a mutation score of 100% for fault-types F1-2, F1-3, and F1-4 while all-uses, has lower mutation scores. This is a result of how the numbers are calculated in the table. There were three programs where there were aDUAs. In these programs, both all-uses, and all-uses, achieve 100% mutation score. However, for programs where no aDUAs were present, all-uses, was not achievable. In those programs all-uses, achieved less than 100% mutation score. Thus, when we calculated the average mutation scores for Table 8, the average for all-uses, remained 100% while that for all-uses, was less than 100%.

Test suites that satisfy all-uses, killed all mutants of type F1-1 in the LongDistance, Customer, and CruiseController classes. In the Local class, 25 test suites killed all the mutants and the remaining 5 test suites failed to kill one of the 16 mutants.

The criterion all-uses, is more effective than the other AOSV criteria, except all-uses,. This can be explained by the effect of faults of type F1-1 on the advised class. A fault of type F1-1 has two effects: (1) the mutated pointcut misses all the intended join points, and (2) the mutated pointcut matches unintended join points. All the AOSV DUAs require executing paths that contain calls to advised methods and therefore, can help in detecting neglected join points. However, CDUAs also require executing paths where the def and the use are in non-advised methods. Therefore, all-uses, helps more than the other AOSV test criteria in detecting unintended join points.

Covering the AOSV DUAs helps in detecting faults of type F1-2 because they require executing paths that contain calls to advised methods (which the mutant pointcut might miss), and not advised methods (which the mutant pointcut might match). The all-uses, criterion is more effective than all-uses, all-uses, and all-uses,

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Kettle</th>
<th>Account</th>
<th>Customer (Banking)</th>
<th>Local</th>
<th>LongDistance</th>
<th>Customer (Telecom)</th>
<th>Call</th>
<th>Timer</th>
<th>CarSimulator</th>
<th>CruiseController</th>
<th>SpeedControl</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>all-uses</td>
<td>92.9</td>
<td>87.3</td>
<td></td>
<td>62.2</td>
<td>70.6</td>
<td></td>
<td>62.2</td>
<td>89.4</td>
<td>77.2</td>
<td>76.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>all-uses</td>
<td>91.0</td>
<td>86.9</td>
<td></td>
<td>69.3</td>
<td>67.9</td>
<td></td>
<td>75.3</td>
<td>66.5</td>
<td>78.8</td>
<td>74.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>all-uses</td>
<td>92.3</td>
<td>52.3</td>
<td>44.7</td>
<td>74.0</td>
<td>80.1</td>
<td>82.8</td>
<td>70.0</td>
<td>97.9</td>
<td>94.3</td>
<td>90.4</td>
<td>90.0</td>
<td></td>
</tr>
<tr>
<td>all-uses</td>
<td>91.7</td>
<td>59.9</td>
<td></td>
<td>67.0</td>
<td>67.6</td>
<td></td>
<td>73.5</td>
<td>96.5</td>
<td>94.2</td>
<td>95.1</td>
<td>92.4</td>
<td></td>
</tr>
<tr>
<td>all-uses</td>
<td>88.7</td>
<td>57.8</td>
<td></td>
<td>86.6</td>
<td>81.4</td>
<td></td>
<td>49.3</td>
<td>83.6</td>
<td>45.8</td>
<td>57.6</td>
<td>64.6</td>
<td>57.6</td>
</tr>
<tr>
<td>AO blocks</td>
<td>72.6</td>
<td>47.2</td>
<td>46.0</td>
<td>49.0</td>
<td>56.1</td>
<td></td>
<td>73.6</td>
<td>60.0</td>
<td>60.9</td>
<td>70.6</td>
<td>63.2</td>
<td></td>
</tr>
<tr>
<td>AO branches</td>
<td>77.6</td>
<td>40.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
because it requires covering paths between non-advised methods. The differences between the effectiveness of \textit{all-uses}, \textit{all-uses\textsubscript{ma}}, and \textit{all-uses\textsubscript{na}} are small and vary with the number of DUAs required by the criteria.

Table 8 shows that 56% of the mutants of type F1-4 are killed by \textit{all-uses}. All the non-killed mutants are generated by an operator called POAC, which generated two mutants for each after advice by (1) changing \textit{after to after returning}, and thus, matching only normal method returns, and (2) changing \textit{after to after throwing}, and thus, matching only exceptional method returns. Mutants of the latter type are always killed by the test suites that satisfy \textit{all-uses}, because the advices miss all the normal returns. The mutants of the former type are not killed by any of the test suites because they require test cases that cause an exception to be thrown in the advised methods (i.e., test cases that cause the \textit{after throwing} path to be executed).

Finally, the \textit{all-uses} criterion killed all the mutants of type F1-5. The other AOSV criteria killed most of the mutants in all the classes, except in the LongDistance and Local classes, in which all the criteria (except \textit{all-uses\textsubscript{m}}) obtained mutation scores that were less than 70%. This is because in these two classes, named pointcuts are not used (i.e., the advices have their own pointcuts), which caused the fault to propagate in fewer paths. Other than these two classes, mutants of type F1-5 are easy to kill by the AOSV-adequate test suites because when the pointcut does not match any join points, then the \texttt{defs} and \texttt{uses} in the advices are not executed, causing the fault to propagate in many paths.

### 6.3.2. Aspect declaration related faults (F2)

As described in Section 5.6, mutants generated for the programs fall into 3 types: F2-5, F2-7, and F2-8. Table 9 shows the effectiveness results of the AOSV test criteria in detecting faults of the types belonging to category F2.

Faults of type F2-5 (incorrect aspect precedence) are generated by two mutation operators: (1) \texttt{DACP}, which changes the aspect precedence declared in the program, and (2) \texttt{DAPO}, which removes the aspect precedence declaration from the program. Two of the subject programs \texttt{Kettle} and \texttt{Telecom}, have precedence rules. All the AOSV-adequate test suites killed all the mutants of type F2-5, except one subtle mutant in the \texttt{Kettle} class. The AOSV criteria killed all the mutants because swapping the precedence causes faults in all the paths that contain calls to advised methods and the AOSV DUAs require covering these paths.

The mutant that remained alive was generated by operator \texttt{DAPO}. The mutant cannot be killed because in the absence of \texttt{declare precedence} directive, \texttt{Aspect} chooses to execute aspects in an arbitrary order. In the \texttt{Kettle} program, the arbitrary order happened to be the same order as that specified by the original program.

The second row of Table 9 shows the average mutation scores of the AOSV-adequate test suites for faults resulting from incorrectly specifying the advice type (fault type F2-7). The test suites that satisfy \textit{all-uses}, killed all the mutants. This is because swapping an after advice with a \textit{before} advice affects many AOSV DU paths in the program since all the advices have some \texttt{uses} or \texttt{defs} of the state variables.

The data for the \textit{all-uses} criterion show that covering all the AOSV DUAs is effective in killing all the mutants, except for a few subtle mutants. Two mutants in the \textit{Local} and \textit{LongDistance} classes are not always killed by the test suites that satisfy \textit{all-uses}. These two mutants cause the \textit{Timer} object in the \textit{Local} and \textit{LongDistance} classes to log the value of the state variable \texttt{startTime} (i.e., the time when the call stops) at the beginning of the phone call and to log the value of state variable \texttt{startTime} at the end of the call. In order to check the logged values, the test cases need to cover the DUAs in the \textit{Timer} class, which are not in the test requirements of the \textit{Local} and \textit{LongDistance} classes. In other words, to detect such faults, the test criteria also need to cover the DUAs in the classes of the referenced objects.

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

Table 10 shows the effectiveness results of the AOSV test criteria in detecting faults of the types belonging to category F3.

Test suites that satisfy \textit{all-uses}, killed all the mutants resulting from an incorrect guarding statement or missing \texttt{proceed} in \texttt{around} advices (fault type F3-1) except one mutant that is not always killed in the \texttt{CruiseController} class.

All of the AOSV test criteria are effective in detecting faults of type F3-1. \textit{All-uses\textsubscript{m}} is more effective than the other AOSV test criteria because all the \texttt{around} advices have aDUAs between the advice and the original code body of the advised method (i.e., the body of the method called by \texttt{proceed}).

\textit{All-uses\textsubscript{na}}, is not effective in the \texttt{Account} class because most of the cDUAs in the class are between \texttt{getter} and \texttt{setter} methods. The results also show that \textit{all-uses\textsubscript{ma}} in the \texttt{Account} class is not effective. This is because the maDUA that the class contains requires executing a path that does not include the code called by \texttt{proceed}.

The \textit{all-uses\textsubscript{na}} criterion is effective in detecting faults resulting from incorrect altering of base class object state variables (fault type F3-2). The few subtle mutants that are not killed by the test suites that satisfy \textit{all-uses\textsubscript{na}}, do not always produce a failure.

There are small differences in the effectiveness of AOSV criteria in terms of detecting faults of type F3-2. The effectiveness depends on the number of test requirements contained in the aspects. Most of the criteria cover many paths in the aspects and thus, the test suites that satisfy them have high mutation scores, except \textit{all-uses\textsubscript{na}} in the \texttt{Account} class.

Finally, the \textit{all-uses\textsubscript{na}} criterion is effective in detecting intra-advice level faults (fault type F3-3). The few mutants that are not killed by the test suites that satisfy \textit{all-uses\textsubscript{na}}, require test cases that include boundary values for the input domains for the state variables.

There are small differences between the effectiveness of AOSV criteria in detecting faults of type F3-3. The effectiveness of the AOSV test criteria depends on the number of test requirements
Effectiveness of AOSV test criteria in detecting F2 category faults.

<table>
<thead>
<tr>
<th>Type</th>
<th>all-uses</th>
<th>all-uses</th>
<th>all-uses</th>
<th>all-uses</th>
<th>all-uses</th>
<th>all-uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2-5</td>
<td>87.5</td>
<td>87.5</td>
<td>87.5</td>
<td>87.5</td>
<td>87.5</td>
<td>87.5</td>
</tr>
<tr>
<td>F2-7</td>
<td>68.3</td>
<td>68.0</td>
<td>71.0</td>
<td>69.3</td>
<td>55.6</td>
<td>100.0</td>
</tr>
<tr>
<td>F2-8</td>
<td>100.0</td>
<td>71.4</td>
<td>82.4</td>
<td>79.6</td>
<td>78.7</td>
<td>96.2</td>
</tr>
<tr>
<td># Mutants</td>
<td>6.0</td>
<td>93.0</td>
<td>100.0</td>
<td>100.0</td>
<td>46.0</td>
<td>100.0</td>
</tr>
<tr>
<td># Classes</td>
<td>3.0</td>
<td>7.0</td>
<td>8.0</td>
<td>8.0</td>
<td>3.0</td>
<td>8.0</td>
</tr>
<tr>
<td>Avg. mutation score</td>
<td>72.8</td>
<td>71.4</td>
<td>80.5</td>
<td>78.2</td>
<td>73.3</td>
<td>96.3</td>
</tr>
</tbody>
</table>

6.3.4. Class implementation related faults (F4)

Table 10 shows the effectiveness results of the AOSV test criteria in detecting faults of the types belonging to category F4.

The all-uses criterion is effective in detecting faults of type F4-1, which result from passing an object in an unexpected state to an advice. The few subtle mutants that are not always killed by the test suites that satisfy all-uses occurred in getter methods where the mutant altered the value of state variables. All-uses is more effective than all-uses, all-uses, and all-uses in all the classes because it requires covering paths between non-advised methods.

Faults that result from passing arguments to the advice that have incorrect values (fault type F4-2) occurred only in the Kettle program because the advice in the other programs do not receive arguments from the base class. Test suites that satisfy each of the AOSV test criteria are able to kill all the mutants of type F4-2. This is because the arguments passed to the advice in the Kettle program are used in all the paths in the advice. Therefore, covering any of the paths in the advice is sufficient to detect these faults.

Mutation scores of the test suites that satisfy the AOSV test criteria are low for object-oriented faults (fault type F4-3). The average mutation score of the all-uses criterion on all classes is 45.4%. In the Kettle class, none of the mutants of type F4-3 are killed. The reason is that many of the faults of type F4-3 are generated by the class operator JSL of jJava which changes the state variables to static. Killing these mutants requires verifying the values of the static variables with more than one object of the same class. That is, a test case needs to execute a path between the def of the static variable with an object, and the use of the static variable with another object. In other words, killing these mutants requires test suites that cover intra-class data-flow interactions.

Test suites that satisfy all-uses detected most of the intra-method faults (type F4-4). The mutants that are not always killed by the test suites require performing boundary testing. For the other AOSV test criteria, all-uses, is the most effective criterion since it has more test requirements in the class.

6.4. Cost-effectiveness of achieving high levels of coverage for criterion all-uses

The goal was to investigate the cost-effectiveness to achieve different levels of coverage (100%, 90%, and 80%) for the all-uses criterion.

Table 12 shows in columns 2 through 4 the mean number of test cases in the test suites that satisfy all-uses, at 100%, 90%, and 80% coverage levels, respectively. Similarly, columns 5 through 7 show the mean effort metric for obtaining the test suites. Columns 8 through 10 show the means of the mutations scores for the test suites.

The number of test cases at the 100% coverage level is more than 15% higher than the number of test cases at the 90% coverage level. This indicates that the last 10% of the AOSV DUAs require more different paths than the DUAs that are covered at the 90% coverage level. Note that in the Kettle and Local classes, the increase in the number of test cases in the test suites that cover 90% of the AOSV DUAs compared with the number of test cases in the test suites that cover 80% of the AOSV DUAs is only 3%, which means that the costly DUAs in these two classes are less than 10%.

Covering the last 10% of the AOSV DUAs requires 66% higher effort compared with the effort of covering the AOSV DUAs at the 90% level. These results are expected because the hard-to-cover

Table 10

Effectiveness of AOSV test criteria in detecting F3 category faults.

<table>
<thead>
<tr>
<th>Type</th>
<th>all-uses</th>
<th>all-uses</th>
<th>all-uses</th>
<th>all-uses</th>
<th>all-uses</th>
<th>all-uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3-1</td>
<td>100.0</td>
<td>92.4</td>
<td>88.3</td>
<td>88.5</td>
<td>86.8</td>
<td>98.3</td>
</tr>
<tr>
<td>F3-2</td>
<td>90.0</td>
<td>86.3</td>
<td>81.2</td>
<td>86.3</td>
<td>81.9</td>
<td>94.4</td>
</tr>
<tr>
<td>F3-3</td>
<td>89.1</td>
<td>85.0</td>
<td>76.9</td>
<td>81.3</td>
<td>81.0</td>
<td>93.7</td>
</tr>
<tr>
<td># Mutants</td>
<td>37.0</td>
<td>357.0</td>
<td>381.0</td>
<td>381.0</td>
<td>227.0</td>
<td>381.0</td>
</tr>
<tr>
<td># Classes</td>
<td>7.0</td>
<td>7.0</td>
<td>8.0</td>
<td>8.0</td>
<td>4.0</td>
<td>8.0</td>
</tr>
<tr>
<td>Avg. mutation score</td>
<td>91.3</td>
<td>86.7</td>
<td>80.2</td>
<td>84.3</td>
<td>82.3</td>
<td>94.7</td>
</tr>
</tbody>
</table>

Table 11

Effectiveness of AOSV test criteria in detecting F4 category faults.

<table>
<thead>
<tr>
<th>Type</th>
<th>all-uses</th>
<th>all-uses</th>
<th>all-uses</th>
<th>all-uses</th>
<th>all-uses</th>
<th>all-uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>F4-1</td>
<td>94.1</td>
<td>79.5</td>
<td>94.4</td>
<td>70.3</td>
<td>55.9</td>
<td>97.8</td>
</tr>
<tr>
<td>F4-2</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>F4-3</td>
<td>14.6</td>
<td>11.0</td>
<td>43.1</td>
<td>10.4</td>
<td>26.7</td>
<td>45.4</td>
</tr>
<tr>
<td>F4-4</td>
<td>92.6</td>
<td>78.7</td>
<td>96.0</td>
<td>81.5</td>
<td>59.2</td>
<td>98.7</td>
</tr>
<tr>
<td># Mutants</td>
<td>22.0</td>
<td>181.0</td>
<td>199.0</td>
<td>199.0</td>
<td>199.0</td>
<td>199.0</td>
</tr>
<tr>
<td># Classes</td>
<td>3.0</td>
<td>8.0</td>
<td>8.0</td>
<td>8.0</td>
<td>4.0</td>
<td>8.0</td>
</tr>
<tr>
<td>Avg. mutation score</td>
<td>84.5</td>
<td>65.2</td>
<td>73.8</td>
<td>70.2</td>
<td>68.6</td>
<td>85.1</td>
</tr>
</tbody>
</table>
AOSV DUAs require performing many iterations of selecting random test cases from the pool since the test cases that cover them are few in the pool. The results are consistent with the results of covering all-uses in procedural and object-oriented programs, which also show that reaching full all-uses coverage increases the cost considerably.

Only 1% more mutants are killed by the test suites that cover all the AOSV DUAs compared to the test suites that cover 90% of the DUAs. However, the mutation scores for the test suites that cover 80% of the AOSV DUAs are about 15% less than the mutation scores of the suites that cover 90% of the AOSV DUAs. Therefore, it is cost-effective to obtain test suites at the 90% coverage level.

Covering the last 10% of the AOSV DUAs killed only a few more mutants because most faults propagate in paths that are required to be executed by more than one DUA. In other words, covering the AOSV DUAs executes many paths in the advised classes in which the faults might propagate and few mutants are left to be killed by the last 10% of the DUAs. While the results show that obtaining full coverage comes with high cost and small increase in effectiveness, it is the decision of the tester to choose whether detecting 1% more faults is worth spending about 66% more effort.

6.5. Summary

Below we summarize the answers to the four research questions.

RQ1: What are the relative costs of using the different test criteria?

Based on the size and effort metrics, the cost for each of the AOSV test criteria are significantly higher in comparison to the costs for the two AO control-flow criteria. The differences between the average size of the AO blocks and the average size of each of the AOSV test criteria range from a minimum of 0.16 to a maximum of 0.77, while for AO branches the range is between 0.01 and 0.62. Similarly, the differences between the average effort for the AO blocks and the average effort for each of the AOSV test criteria range from a minimum of 33.79 to a maximum of 91.48, while for AO branches the range is between 23.91 and 214.44. Among the AOSV test criteria, all-uses, has, on the average, the smallest size (0.34) and the least effort (38.0).

RQ2: What is the relative effectiveness of each criterion in terms of their ability to detect faults?

AOSV test criteria are more effective in identifying faults in comparison to the AO control-flow criteria. The overall effectiveness scores for AO blocks and AO branches are 57.6 and 63.2, respectively. Among the AOSV test criteria, all-uses, has a highest score of 92.4 and all-uses\textsubscript{m} has the lowest score of 71.8.

RQ3: What types of faults can be detected by using test suites that satisfy the AOSV test criteria?

AOSV test criteria are most effective in detecting F3 (advice, aspect method, and intertype method implementation) category of faults. The average mutation scores for F3 faults are all above 80. Among the AOSV test criteria, all-uses, has the highest average mutation scores across all four fault categories, while all-uses\textsubscript{m} is the least effective for all but the F3 category faults.

RQ4: What is the cost-effectiveness of achieving various coverage levels (80%, 90%, and 100%) for the AOSV test criteria? The marginal benefit of raising the coverage level from 80% to 90% is better (i.e., a 13.6% increase in effort produces a 15.1% improvement in the mean mutation score) than raising the level from 90% to 100% (i.e., a 65.8% increase in effort results in a gain of only 1.2% in the mean mutation score).

7. Threats to validity

Three types of threats to validity of our empirical study are identified: internal validity, external validity, and construct validity. Internal validity is concerned with cause and effect relationships, the extent to which the changes in dependent variables can be stated to be caused by changes in independent variables. Two internal threats to validity are as follows:

1. The test suites that satisfy a test criterion also cover some branches or AOSV DUAs that are not required by the test criterion. Covering these test requirements might increase the fault detection effectiveness of the test criteria. However, this threat is minimized by the use of 30 test suites that satisfy each criterion.

2. RANDOOP can produce long test cases, which contain sequence of methods calls that help achieve high coverage for the AOSV test criteria. Therefore, such test cases reduce the cost of the AOSV test criteria when the number of test cases is used as a measure of cost. On the other hand, having a long sequence of method calls results in covering paths that are not required by the AO control-flow criteria, which results in increasing the effectiveness of the AO control-flow criteria as well. This threat is minimized by the use of 30 test suites that satisfy each criterion. The threat can be further minimized by using other test generation tools.

External validity refers to how well the results can be generalized outside the scope of the study [44]. Two external threats to validity are as follows:

1. Four programs were studied and there is no evidence that the results can be extended or generalized to other aspect-oriented programs. However, as mentioned earlier, the four programs contain many characteristics of aspect-oriented software.
2. The sizes of the studied programs are relatively small (less than 1000 lines of code), which is not adequate to evaluate the scalability of the testing approach. However, the programs contain many different types of data-flow interactions with large variations between the classes. In the future, additional programs with larger sizes will be studied.

Construct validity refers to the meaningfulness of measurements [45]. For the measurement of cost, a notable construct validity issue is the extent to which the metrics that were used are adequate measures for cost. While two dimensions of cost were measured, the cost of applying any test approach includes other components, such as the derivation of the test model (e.g., the CFGs), the identification of the test requirements, the development of test drivers and stubs, the derivation of test oracles for test cases, and the execution of the test [29]. The cost of these components was not measured because automated tools were used to perform them. The difference in the computational time of running the tools is negligible.

The size and effort metrics were normalized with respect to the number of test requirements to account for the size of the programs. However, it can be argued that there is no need to divide these measures with the number of test requirements. In practice, many requirements are redundant and they do not contribute to the selection of test cases. The study was actually conducted using both the normalized and un-normalized metrics but we reported the data for normalized measures in this paper. For results on un-normalized metrics, please refer to Wedyan’s Ph.D. thesis [19]. The results are consistent with both metrics.

Another threat to construct validity for the measurement of effectiveness is how realistic are the seeded faults. The results of the study performed by Andrews et al. [32,33] show that mutants can provide realistic results under the conditions of the removal of equivalent mutants and possibly the selection of a subset of mutants that are neither too easy nor too difficult to detect. Therefore, this threat was reduced by performing an unbiased and systematic seeding of faults using a suitable set of mutant operators.

8. Related work

Researchers have proposed data-flow testing approaches for AO programs, but there is a lack of empirical studies comparing the cost and effectiveness of the approaches. Below we provide a summary of work in data-flow testing and show how it developed from procedural program testing to object-oriented testing, and finally, aspect-oriented program testing.

8.1. Data-flow testing of procedural and object-oriented programs

Rapps and Weyuker [20] define a family of data-flow criteria and examine the relationships among them. They created a CFG annotated with def/use information for each program unit (i.e., main program, procedure, or function) and defined seven data-flow criteria: all-nodes, all-edges, all-defs, all-p-uses, all-c-uses/some-p-uses, all-p-uses/some-c-uses, and all-uses. Rapps and Weyuker [46] added two criteria: all-du-paths which requires exercising every def-clear path with respect to each variable, and all-paths which requires exercising every path in the program. The all-du-paths criterion differs from the all-uses in requiring all def-clear paths to be exercised for each DUA. Therefore, all-du-paths subsumes the all-uses criterion. Since exercising all paths in the CFG includes traversing all nodes, edges, and def-clear paths in the CFG, all-paths subsumes all control and data-flow test criteria.

Dynamic data, such as those referred to by pointers, can lead to two types of problems: (1) the same pointer variable may refer to a number of data storage locations, and (2) a data storage location may have a number of references to it (also called aliasing). Hutchins et al. [47], and Ostrand and Weyuker [48] addressed the problem of dynamic data. Harrold and Soffa [49–51] introduced an approach for testing the data-flow interaction between procedures, and proposed an algorithm for computing interprocedural data dependencies. The algorithm has four steps: (1) construction of subgraphs to abstract control-flow information for each procedure in a program, (2) construction of an Interprocedural Flow Graph (IFG) to represent the interprocedural control-flow in the program, (3) propagation throughout the IFG to obtain interprocedural information, and (4) computation of the interprocedural DUs. Given the interprocedural DUs, the data-flow testing criteria can be extended to support interprocedural data-flow testing. Pande et al. [52] proposed a polynomial-time algorithm, called PLR, for determining interprocedural DUs including dynamic data of single-level pointers for C programs.

Harrold and Rothermel [14] introduced a model for performing data flow testing on classes where they defined three levels of testing DUs for state and local variables of primitive data types: (1) intra-method testing, which tests DUs defined within individual class methods, (2) inter-method testing, which tests DUs that result from a call to a public method of the class along with the methods in its class that the method calls directly or indirectly, and (3) intra-class testing, which tests DUs that result from sequences of calls to public methods of the class. The authors provided a program representation that allows data-flow analysis using a class call graph, a class control-flow graph (CCFG), and a framed CCFG.

Buy et al. [53] developed a technique for the automation of class testing. The approach is based on producing sequences of method calls using data-flow analysis, symbolic execution, and automated deduction. Data-flow analysis is applied to the CCFG to identify pairs of methods that define and use the same instance variable (only primitive variables were considered). Marteno et al. [54] extended the above approach to account for state variables that are instances of, or references to, objects of other classes. Orso [55,56] introduced a technique for testing interactions among classes in the presence of polymorphism.

Chen and Kao [57] describe an approach to testing OO programs called object flow testing. In their approach, they identify and test possible object bindings that can occur within a method. Alexander and Offutt [58,59] described techniques for analyzing and testing the polymorphic relationships that occur in OO software. Their approach is based on previous work by Jin and Offutt [60], which presents an approach to integration testing of OO software based on coupling relationships among procedures, including data coupling.

Souter and Pollock [61] proposed a contextual data-flow analysis algorithm for classes. Contextual def and uses for objects that are part of an aggregation relation are defined as a chain of method calls leading from the original call site to the def or use of the object. Rountev et al. [62] presented an approach for data-flow analysis for OO programs that are built on top of pre-existing library components.

8.2. Data-flow testing of aspect-oriented programs

Zhao [9] introduced a data-flow unit testing approach for AO programs which is based on the data-flow model of Harrold and Rothermel [14]. However, instead of considering the class as the unit of testing, Zhao [9] defined two units of testing in AO programs: (1) an aspect together with those methods whose behavior may be affected by the aspect’s advices (called clustering aspect), and (2) a class together with the advices that may affect its
behavior and introductions that may introduce some new members to the class (called clustering class).

Zhao [9] did not define specific criteria for any of the levels or provide any implementation for program representation or test generation. Although not explicitly stated, the testing approach seems to consider only variables of primitive types. Zhao [9] did not show how to handle around advices, dynamic pointcuts, multiple advices applied to the same join point, and pointcuts that depends on the control-flow context (e.g., pointcuts that use designators like cflow, cfrownbelow). In more recent work, Zhao [63] added more details about constructing the AO program CFG but did not provide any solution to the above problems.

Rinard et al. [64] presented a classification system and analysis for AO programs. They identify four types of interactions that occur between methods and advices executed after a join point: (1) augmentation, where the entire body of the method is always executed, (2) narrowing, where either the entire body of the method executes or none executes, (3) replacement, where the method does not execute at all, and (4) combination, where the method and aspect combine in some way to produce new behavior (e.g., around advices with proceed, cflow).

Lemos et al. [6] proposed three control and data-flow criteria for testing AO programs. They developed a tool called JaBUTi/A[1] that parses the bytecode of the class under test and derives a data-flow graph called aspect-oriented def-use (AODU) graph for each module, where a module can be a method, an advised method, a constructor, an advice, or an intertype method.

Xu and Rountev [17] proposed a framework for source-code interprocedural data-flow analysis of AspectJ programs called AJANA. An interprocedural control flow graph is built for AspectJ programs that is modified by adding data-flow information. The graph contains: (1) CFGs that model the control flow within classes, within aspects, and between aspects and classes through non-advice method calls, and (2) interaction graphs (IGs) that model the interactions between methods and advices at join points. The graph is capable of modeling multiple advices that apply at the same join point and modeling dynamic advices. The AJANA framework provides the essential representation for performing data-flow testing on the three levels suggested by Zhao [9]. In particular, using object effect analysis, data-flow analysis can be performed on the object that is passed to an advice at a join point (i.e., the object that has an advised method at the join point).

9. Conclusions and future work

An approach for testing aspect-oriented programs using aspect-oriented state variable (AOSV) data-flow test criteria was presented along with a prototype implementation. An experimental study was performed that shows that even if the AOSV test criteria require more paths, having few test requirements in the classes makes the size of the test suites that satisfy them not significantly higher than those that satisfy the AO control-flow criteria. These results were shown in the classes that have few AOSV DUAs. However, three of the AOSV test criteria always required larger test suites than each of the AO control-flow criteria. These are (1) all-uses, because it requires executing different paths for each of the aDUAs, (2) all-uses, because it has a high number of test requirements in all the classes, and (3) all-uses, because it requires covering all the AOSV DUAs in the advised classes, and therefore, has a high number of test requirements that require different paths.

A test case can cover more blocks or branches in an advised class than it can cover any of the AOSV DUAs. In other words, the number of different paths that the AOSV DUAs require to be executed are higher than the number of different paths needed to cover all blocks or branches in an advised class.

For the classes that have few AOSV DUAs of any type, the effort needed to obtain the test suites that cover these DUAs is not significantly higher than the effort needed to obtain the test suites that satisfy each of the AO control-flow criteria. However, three of the AOSV test criteria required higher effort than the AO control-flow criteria. These are all-uses, all-uses, and all-uses. When the AOSV test criteria were compared with each other, the results do not show that a certain type of AOSV DUAs always requires more effort to satisfy. However, if a certain type of AOSV DUAs in a class requires more effort to satisfy than the other types of AOSV DUAs, the effort needed to obtain test suites that cover that type of DUAs becomes close to the effort needed to obtain test suites that cover all the AOSV DUAs.

The effort needed to obtain a test case that covers any type of AOSV DUAs is higher than the effort needed to obtain a test case that covers a block or a branch in an advised class. The results also show that aDUAs require more effort than the other types of AOSV DUAs. These results are consistent with the results for procedural and object-oriented programs: Satisfying strong test criteria, such as all-uses, requires covering paths that are hard to generate with test case generation tools.

The AOSV test criteria are more effective than the control-flow criteria. Test suites that cover all-uses, detected 38% and 31% more faults than the test suites that cover the blocks and branches, respectively. Moreover, the test suites that cover any type of the AOSV DUAs were also more effective than the test suites that cover the blocks and branches in most of the classes.

Test suites that satisfy the all-uses criterion obtained an average mutation score of 92.4% over all the classes. The all-uses is effective in detecting faults that result from incorrect data-flow interactions in the advised classes. The mutation scores obtained by the test suites that satisfy all-uses, for these fault types range from 94.4% to 100%. The live mutants in these types are subtle mutants. Among the AOSV criteria that are subsumed by all-uses, the all-uses criterion is more effective than the other criteria for the faults that result from passing an object an unexpected state to an advice (i.e., fault type F4-1). This is because all-uses, requires covering more paths in the base class. For the other two data-flow interactions fault types (i.e., Fault types F3-1 and F4-2), the AOSV criteria have close effectiveness results, except all-uses, which is more effective than the criteria it subsumes.

Covering the data-flow for state variables can detect most types of faults in AspectJ programs. The mutation scores of the test suites that satisfy all-uses, range from 87.5% to 100% on the different fault types. However, two types of faults require targeting different test requirements. These are the faults that occur in the exception handling code and faults that require covering data-flow interactions between classes (i.e., inter-class data-flow level testing).

Test suites that satisfy all-uses, at 100% coverage level are only 1% more effective than the test suites that cover 90% of the AOSV DUAs, and need 66% more effort and 15% more test cases. However, the test suites that cover 80% of the AOSV DUAs are 16% less effective than the suites at full coverage. Therefore, it is cost-effective to obtain test suites at the 90% coverage level.

Future work includes plans to investigate other types of data-flow interactions in aspect-oriented programs, such as interactions within the aspects, interactions between classes, and interactions with the objects referenced by classes and aspects. The results for state variable DUAs show that there is a type of fault that requires covering such data-flow interactions.

The use of mutation testing for aspect-oriented programs needs to be investigated in more detail. Cost and effectiveness studies need to be performed for aspect-oriented programs. The use of HOMs for aspect-oriented programs in our work can be extended by using 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), investigating orders higher than two, and subsuming HOMs. Tool support is also needed for mutating aspect-oriented programs.

Appendix A. Usage instructions

Running DCT-AJ requires the following software applications and packages to be installed:

1. AspectBench compiler. The compiler is required by AJANA.
2. An AspectJ 1.6 compiler.
3. Apache Bytecode Engineering Library (BCEL).  
4. The DCTGraph graph library. This is an extension of a graph library called GraPhIT. We made the extensions to the original library in order to allow the framed ICFG to hold the DUAs information.

In order to use DCT-AJ, the user needs to execute the three phases of the tool. We describe below the commands required to execute each phase:

1. DUA identification: Executing the phase requires running the following two commands from the command line:

(a) `java analysis.Main <InputDirName> <ClassName> <OutputDirName>` The command creates an XML file which contain the DUA information with Jimple method names. The command takes the following arguments:
   i. The directory that contains the program `<InputDirName>`.
   ii. The targeted class name `<ClassName>`.
   iii. The output directory `<OutputDirName>` where the XML that contain the DUA information is to be saved.

(b) `java mapper.Main <InputDirName> <ClassName> <OutputDirName>` The command maps Jimple method names to bytecode method names in the XML file created by the previous command.

2. Instrumentation: Creating the tracing aspects described in Section 4.2 requires executing the following command from the command line:

`java instrument.Main <InputDirName> <PackageName> <XMLDirName> <TestDriverName>`

The command takes the following arguments:

(a) The directory that contains the program `<InputDirName>`.
(b) The highest level package name that contains the targeted class and the aspects `<PackageName>`.
(c) The directory `<XMLDirName>` where the XML file that contains the DUA information is saved.

(d) The test driver class name `<TestDriverName>`.

3. Test Execution. In order to execute the test suites and obtain coverage information for the AOSV test criteria, the user needs to run the test driver with the following command:

`java <TestDriverName> <XMLDirName> <ClassName>`

DCT-AJ requires providing the test driver with two parameters: (1) The directory of the XML file that contains the DUA information, and (2) the targeted class name.

References


