DISSERTATION

TESTING WITH STATE VARIABLE DATA-FLOW CRITERIA FOR ASPECT-ORIENTED PROGRAMS

Submitted by

Fadi Wedyan

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2011

ABSTRACT

TESTING WITH STATE VARIABLE DATA-FLOW CRITERIA FOR ASPECT-ORIENTED PROGRAMS

Data-flow testing approaches have been used for procedural and object-oriented (OO) programs, and empirically shown to be effective in detecting faults. However, few such approaches have been proposed for aspect-oriented (AO) programs. In an AO program, data-flow interactions can occur between the 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.

In this research, we propose a data-flow testing approach for AO programs. In an AO program, an aspect and a base class interact either through parameters passed from advised methods in the base class to the advice, or by the direct reading and writing of the base class state variables in the advice. We identify a group of def-use associations (DUAs) that are based on the base class state variables and propose a set of data-flow test criteria that require executing these DUAs. We identify fault types that result from incorrect data-flow interactions in AO programs and extend an existing AO fault model to include these faults. We implemented our approach in a tool that identifies the targeted DUAs by the proposed criteria, runs a test suite, and computes the coverage results.

We conducted an empirical study that compares the cost and effectiveness of the proposed criteria with two control-flow criteria. The empirical study is performed using four subject programs. We seeded faults in the programs using three mutation tools, *AjMutator*, *Proteum/AJ*, and μJava. We used a test generation tool, called RANDOOP, to generate a pool of random test cases. To produce a test suite that satisfies a criterion, we randomly selected test cases from the test pool until required coverage for a criterion is reached.

We evaluated three dimensions of the cost of a test criterion. The first dimension is the size of a test suite that satisfies a test criterion, which we measured by the number of test cases in the test suite. The second cost dimension is the density of a test case which we measured by the number of test cases in the test suite divided by the number of test requirements. The third cost dimension is the time needed to randomly obtain a test suite that satisfies a criterion, which we measured by (1) the number of iterations required by the test suites generator for randomly selecting test cases from a pool of test cases until a test criterion is satisfied, and (2) the number of the iterations per test requirement. Effectiveness is measured by the mutation scores of the test suites that satisfy a criterion. We evaluated effectiveness for all faults and for each fault type.

Our results show that the test suites that cover all the DUAs of state variables are more effective in revealing faults than the control-flow criteria. However, they cost more in terms of test suite size and effort. The results also show that the test suites that cover state variable DUAs in advised classes are suitable for detecting most of the fault types in the revised AO fault model.

Finally, we evaluated the cost-effectiveness of the test suites that cover all state variables DUAs for three coverage levels: 100%, 90%, and 80%. The results show that the test suites that cover 90% of the state variables DUAs are the most cost-effective.

ACKNOWLEDGEMENTS

I owe a special note of gratitude to my advisor, Dr. Sudipto Ghosh, for his invaluable guidance and friendship. He patiently supported and encouraged me during every moment, good or bad. My appreciation also goes to my advisory committee members, Dr. James Bieman, Dr. Yashwant Malaiya, and Dr. Leo Vijayasarathy, for taking the time to give me valuable feedback. Special thanks for Dr. Leo Vijayasarathy for the help in performing the statistical analysis of the data. I thank the entire Computer Science Department staff for always smiling when they helped me with paperwork.

This dissertation is dedicated to my family for their limitless love.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 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 [37]. Aspect-oriented software development (AOSD) is a programming paradigm that supports the modularization of crosscutting concerns. Promised benefits of AOSD include increasing modularity and reducing explicit coupling, thereby increasing understandability and easing software maintenance [48].

AspectJ [37] is an aspect-oriented (AO) extension to Java and is considered as 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 class) 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 [37].

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. The 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 advices. Faults resulting from incorrect data-flow interactions might be difficult to detect unless such interactions are considered by the testing approach.

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 (e.g., [44, 74, 75]), structural (e.g., [27, 39, 40, 72, 80]), or mutation (e.g., [6, 16, 22, 38]). Several other approaches for fault modeling of AO program have been proposed (e.g., [3, 14, 67]). 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 [28].

Structural testing has been widely used for procedural and object-oriented (OO) programs. Existing work in structural testing of AO programs concentrates on defining control and data-flow test criteria, and addressing challenges specific to testing AO programs (e.g., [40, 72]). However, there are several open problems in AO structural testing, which are as follows:

1. Existing data-flow test criteria do not consider all types of data-flow interactions in an AO program. For example, Lemos et al. [39] 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 advices. However, existing approaches do not consider data-flow interactions that are based on state variables, whether interactions between the aspects and the base classes, or between advices in the same or different aspects.

2. Existing fault models for AspectJ contain overlapping fault types and do not include all types of faults that result from incorrect data-flow interactions in the program.

3. Existing structural test criteria do not target any specific type of AO-specific faults.

4. There are few empirical studies that evaluate the cost and effectiveness of the test criteria.

This dissertation describes a testing approach that aims to provide solutions to these problems. We propose a set of test criteria that require covering the data-flow interactions that are based on state variables. We classify state variable def-use associations (DUAs) into five types and propose data-flow test criteria called aspect-oriented state variable (AOSV) test criteria, which require covering these interactions.

We implemented a tool, called Data-flow Coverage Tool for AspectJ (DCT-AJ), which measures coverage for the AOSV test criteria. DCT-AJ is built on top of an existing framework called AJANA [78], which creates the interprocedural control flow graph (ICFG) of a given AO program. In order to obtain the DUAs between public methods, we extended the ICFG 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, in which we run the test suites that satisfy the AOSV test criteria, and generate coverage reports.

We conducted cost-effectiveness studies. The studies compare the cost and effectiveness of 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 empirical studies aim to answer the following questions:

1. What are the relative costs of using the criteria?
2. What is the relative effectiveness of each criterion in terms of its 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?

In order to answer the first research question, we generated a pool of random test cases using a test generation tool called RANDOOP. To produce a test suite that satisfies a criterion, we randomly selected test cases from the test pool until required coverage for a criterion is reached. For answering the second research question, we seeded faults in the subject programs using mutation operators. We used three mutation tools, *AjMutator* [17], *Proteum/AJ* [23], and μJava [41].

Adequate coverage of fault types is needed in order to answer the third research question. We investigated the types of faults that can be generated by using AspectJ mutation operators. First, we solved problems with existing fault models of AspectJ including overlapping fault types and missed fault types. We revised the fault models and classified the generated mutants according to the types of faults in the revised fault model that they represented. The available operators were able to generate faults in all the types except one that required performing more than one change to the pointcut. Therefore, we propose that higher order mutants (e.g., see Jia and Harman [34]) be used to obtain faults of the remaining types.

We evaluated three dimensions of the cost of a test criterion. The first dimension is the size of a test suite that satisfies a test criterion, which we measured by the number of test cases in the test suite. The second cost dimension is the density of of a test case which we measured by the number of test cases in the test suite divided by the number of test requirements. The density shows how many test requirements can be covered by a test case. The third cost dimension is the time needed to randomly obtain a test suite that satisfies a criterion, which we measured by (1) the number of iterations required by the test suites generator to randomly select test cases from a pool of test cases until a test criterion is satisfied, and (2) the number of the iterations per test requirement. The use of the density metric and the number of requirements per test criterion metric allows comparing the cost of test criteria in different advised classes that have large variations in the number of DUAs (test requirements) they contain. Effectiveness is measured by the mutation scores of the test suites that satisfy a criterion. We evaluated effectiveness for all faults and for each fault type.

Our results show that the AOSV test criteria are more effective than the control-flow criteria. Test suites that cover all types of DUAs for state variables in the advised classes detected 38% and 31% more faults than the test suites that cover the blocks or branches, respectively. Moreover, the test suites that cover a single type of DUAs for state variables were also more effective than the test suites that cover the blocks and branches in most of the classes. However, the results also show that covering all types of DUAs for state variables requires more test cases and more effort than covering the branches and blocks in the advised classes.

Our results show that the covering the DUAs for state variables can detect most types of faults in AspectJ programs. The mutation scores of the test suites range from 87.5% to 100% on the different fault types. For the faults that result from incorrect data-flow interactions, the mutations scores for the test suites range from 94.4% to 100%. The non-killed mutants in these types are subtle mutants that cannot be killed.

In order to answer the fourth research question, we evaluated cost and effectiveness of the test criterion that cover all DUAs for state variables for three coverage levels: 100%, 90% and 80% coverage levels. Our results show that the test suites that cover all the state variables DUAs are only 1% more effective than the test suites that cover 90% of the state variables DUAs, and need 40% more effort and 13% more test cases. However, the test suites that cover 80% of the state variables DUAs are 16% less effective than the suites at 100% coverage level. Therefore, it is cost-effective to obtain test suites at the 90% coverage level.

Testing is a vital phase of software development. While AOSD is gaining more popularity, it is important to develop testing approaches for AO programs that help in producing more reliable software. Our work shows that data-flow testing is an effective approach for AO programs because the data-flow interactions can effect both the behavior of the base classes and the aspects. Our approach for generating mutants and test suites for AO programs can be further used to evaluate other testing approaches for AO programs. With the help of DCT-AJ for measuring data-flow coverage, other testing approaches can be compared with ours.

Chapter 2

Background

The following sections describe the main concepts and terminology that are relevant to the dissertation. Section 2.1 describes concepts in structural testing and Section 2.2 summarizes basic concepts in AspectJ.

2.1 Structural Testing Concepts

In structural testing, also called white-box testing, test cases are derived from the program source or executable code [10]. Structural test criteria are defined in terms of elements in the program covered by test cases. A control flow graph (CFG) is a model that describes the structure of a program or a procedure and is used to define structural test criteria. A CFG of a program or a procedure is a directed graph, $G = (N, E)$, where a node $n \in N$ is a block of instructions that are always executed together. An edge, $e(i, j) \in E$, represents a possible transfer of control after executing the last statement of the block represented by node i to the first statement in the block represented by node j. A *path* is a finite sequence of nodes $(n_l, n_2, ..., n_k)$, $k > 1$, such that there is an edge from n_i to n_{i+1} for $i = l, 2, ..., k-1$.

Control-flow based criteria aim to verify the flow of control between the CFG nodes. The most common criteria are: (1) *block (node)* test criterion, which requires executing each node in the CFG at least once, (2) *branch (edge)* test criterion, which requires executing each edge in the CFG at least once, and (3) *all-paths* test criterion, which requires executing all paths in the CFG at least once [10].

Data flow testing verifies that the definition of variables and their subsequent uses are exercised. A definition, *def*, of a variable v occurs in a node 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 $<v,d,u>$ 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 $(d, n_l, n_2, ..., n_k, u)$, $k \geq 0$, containing no *defs* of v in nodes $(n_l, n_2, ..., n_k)$. Uses of a variable can be computation uses *(c-uses)* or predicate uses *(p-uses)*. A *c-use* occurs when the variable is used in a computation or output statement; a *p-use* occurs when a variable is used in a predicate statement [59].

Data-flow based 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. *All p-uses* and *all c-uses* criteria require exercising all *p-uses* or all *c-uses* of each definition of a variable, respectively. The *all-uses* criterion requires satisfying both all *p-uses* and all *c-uses* criteria [59].

2.2 AspectJ Concepts

AspectJ [66] 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* [66].

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.

```
public class Kettle {
    public int waterAmount;
    public int size;
    States status;
    public Kettle(int size){
        this.size = size;
        waterAmount = 0;
        status = States.ON;
    }
    public Kettle(int size, int amount){
        this.size = size;
        waterAmount = 0;
        status = States.ON;
        addWater(amount);
    }
    public void addWater(int amount) {
        this.waterAmount += amount;
    }
    public void pourWater(int amount){
        this.waterAmount -= amount;
    }
}
```

Figure 2.1: Kettle class.

Figure 2.1 shows the Java class *Kettle*, which simulates the functionality of an electric kettle for heating water. The class contains methods for adding water and pouring water from the kettle. Kettle objects can be in one of 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 Figures 2.2 and 2.3 *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.

Figure 2.2 shows the *HeatControl* aspect, which optimizes the power consumption of the kettle. The aspect introduces to the *Kettle* class a state variable called

```
public aspect HeatControl {
   public int Kettle.temperature;
   pointcut pcConstructors(Kettle t): execution(Kettle.new(..)) && target(t);
   pointcut pcPour(Kettle t): execution(* Kettle.pourWater(..)) && target(t);
   pointcut pcAdd(Kettle t): execution(* Kettle.addWater(..)) && target(t);
   after(Kettle t): pcConstructors(t) || pcPour(t)|| pcAdd(t) {
      //afterHeat
      if (t.status != States.OFF) {
         if (t.temperature >= 100)
             t.status = States.HOT;
         else
            t.status = States.HEATING;
      }
   }
   void around(Kettle t, int amt): pcPour(t) && args(amt) {
      //aroundPour
      if ( amt > t.waterAmount )
         t.waterAmount = 0;
      else
         proceed(t, amt);
   }
   void around(Kettle t, int amt): pcAdd(t) && args(amt) {
      //aroundAdd
      if (t.waterAmount + amt > t.size)
         t.waterAmount = t.size;
      else
         proceed(t, amt);
   }
   public void Kettle.readTemperature( ){
      return temperature;
   }
   public void Kettle.setTemperature(int value){
      temperature = value;
   }
}
```

Figure 2.2: HeatControl aspect.

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.

```
public aspect SafetyControl {
   declare precedence:  SafetyControl, HeatControl;
   after(Kettle t): HeatControl.pcConstructors(t) || HeatControl.pcPour(t)||
                    || HeatControl.pcAdd(t) {
   //afterSafety
      if (t.waterAmount == 0 && t.status != States.OFF)
         t.status= States.OFF;
   }
}
```

Figure 2.3: SafetyControl aspect.

The *SafetyControl* aspect shown in Figure 2.3 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* is advised by 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.

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. Pointcuts *pcConst*, *pcAdd*, and *pcPour* declared in the aspect *HeatControl* are also used in the *SafetyControl* aspect to match the same set of join points. 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.

Chapter 3

Related Work

This chapter is organized as follows. Approaches for testing AO programs are described in Section 3.1. Section 3.2 summarizes existing data-flow testing approaches for procedural programs, object-oriented programs, and AO programs.

3.1 Testing AO Programs

Research on testing AO programs can be classified in many ways based on: (1) the artifact used to derive the test cases (model-based, structural, or mutation testing), (2) the targeted software component (either unit or integration testing), or (3) the testing problem it targets (test generation, test measurement, defining coverage criteria, fault modeling, or testing evaluation). In this section, approaches are organized in four groups: fault modeling, model-based testing, structural testing, and mutation testing. This is because all existing model-based testing approaches are used for integration testing, while all structural approaches are used for unit testing of aspects (except for the data-flow approaches which are discussed in Section 3.2.3). Fault models for AO programs are summarized in Section 3.1.1, while model-based and structural testing approaches are discussed in Section 3.1.2 and Section 3.1.3, respectively. Existing mutation testing approaches are summarized in Section 3.1.4 for both unit and integration testing.

3.1.1 Fault Models for AO Programs

Binder [10] describes a fault model as "an assumption about where faults are likely to be found". A fault model describes where most of the faults are for a specific programming paradigm or language. A testing approach can be used to design a test suite that exercises the program sufficiently to find most faults.

Alexander et al. [3] proposed a candidate fault model for AO programs with six classes of faults. These fault classes are as follows:

- *Incorrect strength in pointcut patterns.* This fault occurs when the pointcut expression is incorrectly written. If the pointcut expression is too strong, some intended join points will be missed. If the pattern is too weak, some unintended join points will be selected. Either case is likely to cause incorrect behavior of the AO program. Missing intended join points results in an incorrect behavior of the crosscutting concern. Selecting unintended join points might incorrectly change the behavior of the core concerns.

- *Incorrect aspect precedence.* This fault can occur when two or more advices are woven into the same join point and a successor advice depends on an object state that is set by a predecessor advice. If the order of weaving is not properly specified, the successor advice might produce an incorrect behavior.

- *Failure to establish expected post-conditions.* Clients expect method post-conditions to be satisfied if the pre-conditions hold prior to calling the method. A faulty aspect might change the post-conditions of the advised method resulting in incorrect method behavior.

- *Failure to preserve state invariants.* Aspects have access to the state variables of the base classes. When an aspect changes the state variables in a way that violates the state invariants, then the integrity of the base class gets violated.

- *Incorrect focus on control flow.* Aspects can be woven depending on the dynamic context of the base class methods. For example, designators like *cflowbelow* or *cflow* specify in what context an advice can be woven. An incorrect use of dynamic context can result in weaving advices in an unintended context. The fault effect is similar to the first class of faults.

- *Incorrect changes in control dependencies.* An advice can alter the control and data dependencies of the advised method. An incorrect advice might force the advised method to flow in incorrect control paths.

Ceccato et al. [14] extended the above fault model with three more fault types:

- *Incorrect changes in exceptional control flow.* An advice that throws an exception might cause a modification of the control flow because the exception triggers the execution of a catch statement, either in the aspect itself or in the base program. Aspects can also modify the system exception handling mechanism using constructs like *declare soft*. A faulty aspect can incorrectly change the exceptional control flow of the base program.

- *Failures due to inter-type declarations.* An aspect can introduce new methods and state variables to a class. If the control flow of a method depends on the structure of the class, then an incorrect or unexpected change to the class structure causes the method to behave incorrectly.

- *Incorrect changes in polymorphic calls.* This type of fault occurs when methods introduced to a base class override a method inherited from a super class. Before weaving the aspect, any invocation to such a method was redirected to the method in the super class, while after weaving, the same invocation, is dispatched to the introduced method. Such a modification in the system behavior may cause faults.

van Deursen et al. [67] proposed an AO fault model that is based on the location of the faults. Accordingly, they classified faults for AOP as follows:

- *Faults due to inter-type declarations.* This category includes any fault that occurs as a result of an introduced method or a state variable. Examples include the two types of faults suggested by Ceccato et al. [14]

- *Faults in pointcuts.* Faults in the pointcut pattern include incorrect strength in pointcut pattern and incorrect focus on control flow as suggested by Alexander et al. [3].

- *Faults in advice.* Faults in advice implementation can result in an incorrect behavior of the advised method. The effect might cause both the core and crosscutting concerns to behave incorrectly.

Baekken and Alexander [9] presented a fault model for pointcuts in AspectJ programs. Each fault is described by a name, a fault category, a syntactic form, and a semantic impact. They described four fault categories:

- *Incorrect patterns.* A pattern describes the syntax of the pointcut expression that matches a particular set of join points. For example, an AspectJ pattern for matching methods consists of a modifier, a return type, a method name, a list of parameters in parenthesis, and a *throws* exception statement. All these elements can be replaced by a wildcard that can be used to match any value, name, number and type of parameters, and exception type. Developers might incorrectly write a method pattern that does not match the intended one.

- *Incorrect choice of pointcut designator.* Developers might incorrectly use a *designator* that has an impact different from the intended one. The resulting faults depend on the *designator* used.

- *Incorrect matching of dynamic circumstances.* Dynamic join points depend on data collected from the context (i.g., from the matching method). For example, using the *args* designator, parameters of the advised method can be passed to the advice. Developers might incorrectly specify the type or order of the parameters which might result in missing the intended method or matching an unintended method.

- *Incorrect pointcut composition.* Pointcut descriptors contain one or more expressions combined using logical operators. Developers might use the incorrect operator, for example, the logical *and* operator (&&) instead of the logical *or* operator (||), which leads to missed join points.

3.1.2 Model-based Testing

In functional testing, also called specification-based or black-box testing, test cases are derived from the program specifications. Model-based testing is a type of functional testing where the program specifications are given in the form of a modeling language (e.g., in UML). Advantages of model-based testing are as follows: (1) testing can start at an early stage of the software development process, (2) faults in the specifications may be detected before these specifications are implemented, (3) it is implementation-independent, and (4) it potentially supports the automation of test generation [12].

Xu and Xu [74] presented a state-based approach to incremental testing of AO programs, which considers aspects as incremental modifications to the base classes. The approach contains an AO extension to state models, which facilitates the specification of the impact of aspects on the states and transitions of base class objects and generation of abstract test cases. Xu and Xu [74] also investigated reusing base class tests for testing AO programs. Their results show that some of the base class tests can be fully or partially reused, thereby reducing the cost of testing aspects. However, the state model is defined only for individual classes and aspects. In general, states are hard to model for multiple classes even if no aspects are included. Xu and Xu [74] did not describe any implementation of their approach, nor did they provide any empirical results.

Xu et al. [75] presented an AO model that consists of *class/aspect* diagrams and sequence diagrams that describe the static structure and dynamic behavior, respectively, of an AO program. To derive test cases, Xu et al. [75] presented a procedure to weave the sequence diagrams for the methods of classes and the advice of aspects and then generate an Aspect-Object Flow (AOF) tree from the woven sequence diagram. In an AOF tree, each path from the root to a leaf represents a sequence of messages between objects and aspects. A backtrack reasoning procedure is applied to derive test cases for each path that satisfy the collective constraints along the path. The message sequence is used as the oracle for each concrete test case. Using the AOF graph, Xu et al. [75] defined three test criteria:

1. *Branch* coverage criterion: Satisfied if and only if $\forall e \in E$, where E is the set of edges in the graph, there exists at least one path p that contains the edge e.

2. *Polymorphic* coverage criterion: Satisfied when for each abstract advised method m of class Abs that is called in the sequence model, each implementation of m in the subclasses of Abs is exercised.

3. *Loop* coverage criterion: Satisfied if each loop in the sequence model is executed for zero (bypassing the loop), one, or the maximum number of iterations.

The work of Xu et al. [75] contributes in two directions, modeling AO programs for testing purposes and using sequence models for test generation. However, the sequence models developed by Xu et al. [75] contain only messages to objects that directly interact with the method under test. Their justification was that there is no need to include indirect interactions since these will be tested using the sequence models of the other methods. This decision limits the ability to test advices that are woven depending on the calling context. However, the approach reduces the number and length of paths, which in turn makes the test generation process easier.

Massicotte et al. [44] proposed an approach that generates test cases for AO programs using three inputs: the collaboration models for system operations, the aspects, and the base classes models. The approach consists of two phases. In phase one, base classes are tested before weaving. The authors proposed to derive test cases that satisfy two of the test criteria defined by Abdurazik and Offutt [1, 50], (1) *transition* coverage, which requires exercising each transition in the collaboration model, and (2) *complete sequence* coverage, satisfied by testing each message sequence in the model where a message sequence corresponds to a scenario in the collaboration model. Massicotte et al. [44] developed a graph called *message control flow* (MCF) graph which shows method calls in the collaboration model. In phase two, aspects are added one by one by integrating the MCF graphs of the advices in the message sequences that use advised methods. Based on the resulting MCF graph, Massicotte et al. [44] defined three test criteria: (1) *modified sequences* criterion, which requires exercising each path that include messages to intertype methods, (2) *simple integration* criterion, which requires exercising paths that contain messages to advised methods, and (3) *multi-aspects integration* criterion, which requires exercising each path that contain messages to methods advised by more than one advice. Massicotte et al. [44] show examples on how satisfying the above criteria can help in detecting faults in AO programs.

Xu et al. [76] presented an approach to automate test code generation from finite state machines for classes. The authors implemented their approach in a tool called Model-based Aspect/Class Testing (MACT). The tool first generates a transition tree from the class state model for a coverage criterion. The tester then provides the tool with arguments for method invocation. The tool uses these arguments and the transition tree to generate concrete test cases. The tool supports generating test cases for state coverage, transition coverage, and round-trip coverage. Xu and Ding [73] used MACT to develop an approach for prioritizing test cases generated by the tool. In their approach, test cases are prioritized by identifying the extent to which an aspect modifies the base classes. Modifications are measured by the number of new and changed components in a transition (i.e., start state, event, precondition, postcondition, and end state). Test cases that execute higher number of modified or new state transitions have higher priority. The authors evaluated their approach using mutation analysis. They manually produced a number of mutants in each fault type of Alexander et al.'s [3] fault model. Their results show that their prioritization of test cases can perform better than arbitrary running the test cases and without decreasing effectiveness. The authors measured performance by the index of the first test case that found the fault.

Although there exists research on modeling AO programs (e.g., [33, 65, 75]), a major concern about model-based testing for AO programs is the need of defining constructs that are not part of any existing specification languages or modeling languages. Thus, it is hard to automate the process of test generation and coverage measurement without having modeling tools for AO programs. Moreover, the problem of using model-based testing remains 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 [28].

3.1.3 Structural Testing

Lopes and Ngo [40] presented a framework that supports unit testing of the aspectual behavior of aspects implemented in AspectJ. Aspectual behavior refers to the behavior encapsulated in the advices and introduction parts of the aspect. The framework consists of two components, Java aspect markup language (JAML) and JamlUnit. JAML is an extensible language framework for programming aspects. In JAML, aspectual behavior is encapsulated in regular Java classes. This allows aspects to exist independent of any base classes, which might not be available when aspects are developed. JamlUnit is a framework that facilitates building JUnit test cases and provides a testing context for the aspects. JamlUnit uses mock objects to provide a context in which aspects (written in JAML) can be tested. Lopes and Ngo [40] provided examples on how their proposed approach can be used. However, they did not define any testing requirements or test generation approaches.

Xie and Zhao [72] developed a framework called *Aspectra*, which automates the generation of test inputs for unit testing of aspects. The use of Aspectra consists of three steps: (1) produce bytecode that is *suitable* for the test generation tools, (2) apply test generation tools on the bytecode and produce test cases, and (3) define *suitable* test criteria and provide the means for measuring test coverage.

In the first step, *Aspectra* requires the tester to implement a base class in which the aspect under testing (AUT) can be woven. However, in order to be able to test the aspect, *Aspectra* has to ensure that all advices and advice methods have join points in the base class. To do so, Aspectra synthesizes a wrapper class for the base class. The wrapper class produces a wrapper method for each base class public method (including intertype methods) and each public non-advice method of the aspect. The wrapper method invokes the corresponding class or aspect method.

In the second step, Xie and Zhao used two tools in order to generate state-based test cases. The first tool, called *Parasoft Jtest* [58], is a unit testing tool that generates JUnit test cases for obtaining structural coverage. The authors used *Parasoft Jtest* to generate arguments for the wrapper class public methods. The second tool, called Rostra [71], uses the test cases provided by *Parasoft Jtest* to generate test cases that exercise each possible combination of non-equivalent object states.

In the third step, Xie and Zhao defined two test criteria for testing aspectual behavior: *aspectual branch* criterion, which requires exercising all branches in the aspect bytecode at least once, and (2) *interaction* criterion, which requires exercising three types of interactions (i.e, calls) at least once: (a) from advised methods to aspect methods, (b) from aspect methods to aspect methods, and (c) from aspect methods to advised methods. Xie and Zhao performed an empirical study on 12 AspectJ benchmarks to assess how the wrapper synthesis mechanism helps in test generation. No results were given for the cost or effectiveness of the suggested coverage criteria.

Harman et al. [27] presented an automated test generation approach for unit testing of aspects. The approach relies on *Aspectra* [72] to provide a wrapper class where the AUT is woven and aims to generate test cases that cover the *aspectual branch* criterion using an evolutionary algorithm (i.e., a genetic algorithm). The evolutionary algorithm repeatedly generates sets of test cases, called *generations*, that have better fitness value (i.e., have better chance to achieve 100% *aspectual branch* coverage) until either a set that covers the criterion is produced or an upper limit of iterations is reached. Harman et al. [27] provided an implementation of the test generation and *aspectual branch* coverage measurement, which they used to conduct an empirical study on a suite of 14 AspectJ programs. The study shows that test cases produced by the evolutionary algorithm can achieve higher *aspectual branch* coverage with less cost than random test case generation (in terms of the number of iterations of test set generation required to achieve 100% coverage). The results also show that covering *aspectual branches* requires less test cases than covering the *all-branches* criterion in the AO program. The authors did not explain the impact of the last result, nor did they evaluate the effectiveness of covering *aspectual branch* in revealing faults.

3.1.4 Mutation Testing

Lemos and Lopes [38] presented an approach for testing pointcut expressions. The authors identified four types of pointcut strength faults: (1) selection of a superset of intended join points, (2) selection of a subset of intended join points, (3) selection of a set that has has some intended and some unintended join points, and (4) selection of a set that includes only unintended join points. They suggested using structural testing for revealing unintended join points and performing mutation testing for detecting neglected join points. The authors, however, did not define any mutation operators or test criteria that can help in detecting such faults.

Ferrari et al. [22] introduced a mutation testing approach for AO programs. The authors summarized AspectJ fault types based on previous works on AspectJ fault modeling into four categories: (1) pointcut related faults, (2) intertype declarations related faults, (3) advice related faults, and (4) base program related faults. They defined a set of mutation operators for each fault type in the first three categories and discussed the ability to generalize the AspectJ fault types to other Java-based AO implementations. They conducted a cost analysis in which they measured the cost of mutation testing using the number of generated mutants. Generally, mutation testing is considered to be an expensive technique where the cost consists of three factors, the number of generated mutants, the cost of analyzing the equivalent mutants, and the size of the test suite needed to kill the mutants [45]. Ferrari et al. [22] also did not evaluate their mutants in terms of the ability to develop effective test cases.

Anbalagan and Xie [6] proposed a framework that generates mutants for testing the *strength* of a pointcut expression. Their approach generates mutants from two sources, the pointcut under test, and a set of *candidate* join points in the base class bytecode (i.e. statements or blocks of statements that can possibly be matched by any pointcut). The mutants are generated by (1) inserting a wildcard in various locations of the candidate join point or the pointcut naming part and parameters, and (2) splitting the naming part of the join point into words that begin with uppercase letters. A mutant is classified as *weak* if it matches more join points than the original pointcut, *neutral or equivalent* if it matches the same join points as the original pointcut, or *strong* otherwise. In order to reduce the number of mutants, they defined a distance measure that is used to rank the mutants so testers can use the mutants that are more *relevant*. However, the authors did not perform a cost and efficiency analysis for the selected mutants.

Delamare et al. [16] presented a test-driven approach for developing pointcut expressions in AspectJ, where the test cases can be used to validate that the join points matched by the pointcut expressions are the intended ones. To do so, the authors implemented a tool called *AdviceTracer* which can be used with JUnit to determine at runtime which advice is executed and at which place in the base program. This information is then used to build oracles that specifically target the presence or absence of an advice. Delamare et al. [16, 17] also developed a mutation tool, called *AjMutator*, which inserts seven different types of faults based on seven mutation operators, defined by Ferrari et al. [22]. Delamare et al. [16] evaluated the ability of their approach for specifying expected join points and for detecting pointcut expressions faults on two AspectJ systems. JUnit test cases were produced using *AdviceTracer* and mutations were produced using the *AjMutator* tool. The results show that the test cases were able to kill all the mutants.

In a recent study, Ferrari et al. [23] developed a tool that implements a subset of the mutation operators they suggested in their previous work [22]. The tool, called *Proteum/AJ*, implements 3 more pointcut mutation operators than *AjMutator*, and also implements 2 advice declaration operators, 4 advice implementation operators, and 5 intertype declaration operators. *Proteum/AJ* allows choosing what operators to apply. The tool also runs JUnit test cases and computes mutation score for a given test suite. The authors showed an example of how their tool can be used to generate mutants, identify some of the equivalent mutants, run test cases, and report mutation scores.

3.2 Data-flow Testing

Data-flow testing approaches for procedural, OO, and AO programs are summarized in Sections 3.2.1, 3.2.2, and 3.2.3, respectively. Section 3.2.4 discusses existing empirical studies on evaluating data-flow test criteria.

3.2.1 Data-flow Testing of Procedural Programs

Rapps and Weyuker [59] 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 [60] 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.

Rapps and Weyuker [59, 60] did not make a distinction between atomic data such as integers, and structured or aggregate data such as arrays and records. *Defs* or *uses* of an element of a structured datum are regarded as *defs* or *uses* to the whole datum. Hamlet et al. [26] argued that treating arrays as aggregate values leads to mistakes of commission when DUAs are identified that are not present for any array element, and mistakes of omission when a path is missed because of a false intermediate assignment (e.g., when swapping array elements). Treating elements of structured data as independent variables can correct the mistakes. Such an extension seems to add no complexity when the references to the elements of structured data are static (i.e., fields of records or objects). However, treating arrays element-by-element may potentially introduce an infinite number of DUAs to be tested. Moreover, as Rapps and Weyuker [59, 60] pointed out, whether two references to array elements are references to the same element is an undecidable problem. Hamlet et al. [26] proposed a partial solution to this problem by using symbolic execution and a symbolic equation solver to determine whether two occurrences of array elements can be occurrences of the same element.

Hutchins et al. [32] addressed the problem of data-flow analysis of dynamic data, which was not taken into account by the early work of Rapps and Weyuker [59, 60]. One of the difficulties of dynamic data such as those referred to by pointers is that a pointer variable may actually refer to a number of data storage locations. On the other hand, a data storage location may have a number of references to it (i.e., the existence of *aliasing*). They defined the *all-DUs* criterion which requires exercising all DUAs in the CFG (similar to the *all-uses* criterion). However, Hutchins et al. [32] defined a *use* as every access to a memory location regardless of being a *p-use* or a *c-use*, a *def* is every write to a memory location. Both *defs* and *uses* are defined for named or dynamically allocated memory locations. Since Hutchins et al. [32] do not distinguish between *c-uses* and *p-uses*, satisfaction of *all-DUs* criterion does not not subsume the *all-edges* criterion (unlike *all-uses*) and is not comparable (in terms of subsumption relations) to the data-flow criteria of Rapps and Weyuker [60, 59].

Ostrand and Weyuker [54] addressed the problem of dynamic data by introducing the concepts of *possible* or *definite def* or *use* of a variable. For a given variable v, *def* (or *use*) of v is *definite* if static analysis determines that the object being defined (or used) is unambiguously variable v. Otherwise, the *def* or *use* is considered *possible*. Similarly, a path may be *definitely def-clear* or *possibly def-clear* with respect to a variable. Ostrand and Weyuker [54] extended the DUA relation on the occurrences of variables to a hierarchy of relations. A DUA is *strong* if there is a *definite def* of a variable and a *definite use* of the variable and every *def-clear* path from the *def* to the use is definitely *def-clear* with respect to the variable. The association is *firm* if both the *def* and the *use* are *definite* and there is at least one path from the *def* to the *use* that it is definitely *def-clear*. The association is *weak* if both the *def* and the *use* are definite, but there is no path from the *def* to the *use* which is definitely *def-clear*. An association is *very weak* if the *def* or the *use* or both of them are *possible* instead of *definite*. Ostrand and Weyuker [54] then defined four versions of the *all-uses* criterion: *strong all-uses*, *firm all-uses*, *weak all-uses*, or *very weak all-uses* depending on the type of associations they measure. Since the four types of DUAs are disjoint, the four types of criteria can be satisfied independently.

Harrold and Soffa [29, 30, 31] introduced an approach for testing the data-flow interaction between procedures. They identified two types of data dependency across procedure interfaces. *Direct dependencies* exist when either a *def* of an actual parameter in one procedure reaches a *use* of the corresponding formal parameter in a called procedure, or a *def* of a formal parameter in a called procedure reaches a *use* of the corresponding actual parameter in the calling procedure. *Indirect dependencies* are similar to direct dependencies except that multiple levels of procedure calls and returns are considered. Indirect data dependencies can be determined by considering the possible *uses* of a *def* along the calling sequences. When a formal parameter is passed as an actual parameter at a call site, an indirect data dependence may exist. Harrold and Soffa [29, 30, 31] 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 DUAs.

Given the interprocedural DUAs, the data-flow testing criteria can be extended to support interprocedural data-flow testing. Pande et al. [56] proposed a polynomial-time algorithm, called PLR, for determining interprocedural DUAs including dynamic data of single-level pointers for C programs.

3.2.2 Data-flow Testing of OO Programs

Harrold and Rothermel [28] introduced a model for performing data flow testing on classes where they defined three levels of testing DUAs for state and local variables of primitive data types: (1) *intra-method* testing, which tests DUAs defined within individual class methods, (2) *inter-method* testing, which tests DUAs 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 DUAs 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.

A class call graph represents the call relationships among class methods where vertices represent methods and arcs represent method calls. The class call graph is enclosed in a frame that represents a driver for the class. The frame provides calls to the public methods of the class. The CCFG, also called interprocedural control flow graph (ICFG) [46, 78], is constructed by replacing the call vertices by a method entry *vertex* and a method *return* vertex in the class call graph. Finally, a framed CCFG is constructed by connecting each method *entry* vertex to the corresponding method CFG and each *return* vertex to the corresponding method *exit* vertex. The authors suggested using the PLR algorithm [56] to find the DUAs for the three testing levels.

Buy et al. [13] 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 of a class are considered. Symbolic execution is used to find the set of conditions related to the execution of a path. Finally, automated deduction is applied to identify sequences of method calls that exercise the DUAs found by data-flow analysis using the preconditions and post-conditions found by the symbolic execution phase.

Marteno et al. [42] extended the above approach to account for state variables that are instances of, or references to, objects of other classes. In the data-flow analysis phase, for each class, starting from simplest ones (i.e., classes that do not contain any aggregated objects), each method of the class is classified into three types: (1) *inspector* methods, which only access the state variables, (2) *modifier methods*, which define state variables, and (3) *inspector-modifier methods*, which can both define and access the state variable. Using the inheritance hierarchy of each class that contains instances of other classes, calls to modifier methods are considered to be *defs* while calls to inspector methods are considered to be *uses*. Calls to inspector-modifier methods might be *defs* or *uses* depending on the path taken when the method is called. As pointed by the authors, even for pure inspector or modifier methods, the approach needs to determine whether the path taken when the method is executed contains a *def* or a *use* of a state variable. However, such information cannot be obtained by static analysis alone since it depends on the state of the object that can only be known at run time.

Orso [52, 53] introduced a technique for testing interactions among classes in the presence of polymorphism. Testing OO programs in the presence of polymorphism and dynamic binding is challenging since the actual bound methods cannot be statically identified. Faults can result in either isolated polymorphic calls or in combined polymorphic calls along the execution path. The technique presented by Orso [52, 53] is composed of two steps: the identification of an integration order, and the incremental testing of the polymorphic interactions while adding classes to the system. Orso [52] defined DUAs that occur in polymorphic methods, and used the DUAs to define different test criteria (i.e., *all-uses*, *all-defs*).

Chen and Kao [15] 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. The idea is to identify the DUAs that occur within a method, and also between pairs of methods that are invoked from the same caller. They defined two criteria that impose testing requirements on a method: (1) *all-bindings*, requires executing each possible binding of each object at least once, and (2) *all-du-pairs* criterion which requires that every *def-clear* path between every *def* of an object and every use of that object be tested at least once.

Alexander and Offutt [4, 5] 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 [35], which presents an approach to integration testing of OO software based on coupling relationships among procedures. Coupling-based testing (CBT) uses three types of coupling relations between methods: (1) parameter couplings, which occur whenever one procedure passes parameters to another, (2) shared data couplings, which occur when two procedures reference the same non-local variable, and (3) external device couplings, which occur when two procedures access the same external storage device [5]. Coupling sequences are calls to methods that have coupling relations. Alexander and Offutt [5] defined four data-flow test criteria that handle coupling and polymorphism relations.

Souter and Pollock [63] proposed a contextual data-flow analysis algorithm for classes. Contextual *defs* 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. Contextual DUAs can add increased coverage since multiple contextual DUAs can be defined for the same context-free association.

Denaro et al. [19, 20] suggested performing contextual testing of state variables by using the algorithm of Souter and Pollock [64] to define the DUAs of state variables as defined by Harrold and Rothermel [28] (i.e., by considering only *defs* and *uses* that reach method boundaries, thereby reducing the complexity of computing the contextual DUAs).

Rountev et al. [62] presented an approach for data-flow analysis for OO programs that are built on top of pre-existing library components. Performing data-flow testing on such OO programs requires having the control and data flow information available for the tester. Even if such information is available, the cost of combining the CFGs of library methods with the analyzed program is likely to be high. Rountev et al. [62] suggested building summary information for the pre-existing library methods that capture the effects of all relevant library-local ICFG paths. Depending on the interaction between the analyzed program and the library, library-local ICFG paths are integrated with the analyzed program ICFG.

3.2.3 Data-flow Testing of AO Programs

Zhao [80] introduced a data-flow unit testing approach for AO programs which is based on the data-flow model of Harrold and Rothermel [28]. However, instead of considering the class as the unit of testing, Zhao [80] 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*). A CFG is built for each method in a clustering aspect or class. For advised methods, the CFG of the advice is merged into the CFG of the method. Algorithms similar to the ones introduced by Harrold and Rothermel [28] are used to produce a class call graph, CCFG, and framed CCFG.

Zhao [80] suggested three levels of testing: (1) *Intra-module* level testing, which requires testing DUAs within the advised method, class methods, and aspect non-advice methods, (2) *Inter-module* testing, which requires testing DUAs that results from a call to a public module along with some other modules (methods) it calls, directly or indirectly, and (3) *Intra-aspect* or *intra-class* level, which requires testing DUAs that results from sequences of calls to the class or aspect public modules. Zhao [80] 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 [80] 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*, *cflowbelow*). In more recent work, Zhao [81] added more details about constructing the AO program CFG but did not give any solution to the above problems.

Rinard et al. [61] 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*).

Rinard et al. [61] associated a *scope* with each advice and method. A scope identifies the correspondence between the concern (method or advice) and accessed object fields. The authors defined four types of scopes: (1) *independent*, where the advice does not write a state variable that the method reads and the method does not write a state variable that the advice reads, (2) *observation*, where the advice reads state variables that the method writes, (3) *actuation*, where the advice may write state variables that the method may read, and (4) *interference*, where the advice and method may write the same state variable. This classification helps in understanding how the advice affects the base class methods and whether object states can be modified by the aspects. The authors performed escape and pointer analysis to determine how objects are affected when passed to advices. Escape analysis determines all the places where a pointer or object can be stored and whether the lifetime of the pointer or object can be proven to be restricted only to the current advice or method [63]. While Rinard et al. [61] did not present a testing approach, their work helps in understanding the data-flow interaction between aspects and base classes.

Lemos et al. [39] proposed three control and data flow criteria for testing AO programs. They developed a tool called *JaBUTi/AJ* 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.

A node in the AODU graph represents a block of bytecode instructions that are always executed together. Edges represent either normal control-flow (called *regular* edges), or edges that are executed when an exception is triggered (called *exception* edges). Sets of nodes and edges that can be reached by paths that do not contain any *exception* edge are called *exception-independent* nodes or edges, respectively; otherwise, the nodes or edges are called *exception-dependent*. A node in the set of *exception-independent* nodes is called a *crosscutting* node if it contains bytecode instructions that represent a join point. Similarly, *exception-dependent* edges are called *crosscutting edges (c-edges)* when either the source or the target node is *crosscutting*.

The AODU graph is used to define the test criteria. The authors divided the traditional *all-edges* and *all-nodes* control-flow criteria into three types each corresponds to an edge or node type. Thereby, Lemos et al. [39] suggested the following two control-flow criteria:

- *All-crosscutting-nodes (all-nodes$_c$)* Criterion: Requires executing each *crosscutting* node in the AODU graph at least once.
- *All-crosscutting-edges (all-edges$_c$)* Criterion: Requires executing each *crosscutting* edge in the AODU graph at least once.

Similarly, the authors defined the following data-flow criterion.

- *All-crosscutting-uses (all-uses$_c$)*: Requires exercising each def-use pair whose uses are in a crosscutting node at least once.

The *all-uses$_c$* criterion requires testing the advised method's local variables that have uses in the advice, which can only occur if the advice receives parameters from the method. The criterion does not require testing local variables that might be modified by the advice since this is required by the *all-uses* criterion of the method (i.e., if the advice returns a value that is assigned to a method local variable). Class state variables that have *uses* or *defs* in the advices are beyond the scope of Lemos et al. [39] work. Also their *JaBUTi/AJ* tool can only handle variables of a scalar type.

Xu and Rountev [78] proposed a framework for source-code interprocedural data-flow analysis of AspectJ programs called AJANA. The framework contains an algorithm that is based on an earlier work of Xu and Rountev [77] that builds an ICFG for AspectJ programs and is modified by adding data-flow information. An ICFG 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 ICFG 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 [80]. 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).

As pointed by Xu and Rountev [77, 78], performing data-flow analysis at the source code level has several advantages over doing it at the bytecode level. First, the mapping between the source code and bytecode entities depends on the weaving compiler used; different compilers or even different versions of the same compiler can create different mappings. Second, source code analysis can be performed before weaving and can thus provide information about the effects advices have on the base code. When performing analysis at the bytecode level, the bytecode of the base classes before weaving cannot be obtained unless the Java code is compiled again. Third, source code level analysis produces more accurate and easier to visualize results. In the bytecode level, there are many details added by the compiler, which are not needed for the data-flow analysis. Finally, the settings of the Java and AspectJ compilers have an effect on the produced bytecode, which complicates the bytecode level analysis. However, performing bytecode level analysis is especially useful when the source code of the software is partially or completely not available [68].

3.2.4 Empirical Studies on Evaluating Data-flow Test Criteria

In this section, empirical studies on evaluating data-flow test criteria are discussed. Since there are no such studies for AO programs, the discussion is limited to studies on procedural and OO programs. The empirical studies aim to evaluate the cost and effectiveness of the coverage criteria. In the following discussion, unless other measures are specified, cost is measured in terms of the size of the test suite required to satisfy the criterion and effectiveness is measured by the number of faults detected by the test suite.

Frankl and Weiss [25] performed one of the first empirical studies where the *all-uses* and *all-edges* criteria are compared with each other and with random test suites (*null* criterion). The study was performed on nine small programs written in Pascal for which the authors had access to real faults. Comparisons are done using hypothesis testing using the proportion of adequate test suites that expose a specific fault as the dependent variable. Test suites were generated from a large test pool developed for each subject program. Logistic regression was used to model the relationship between the probability of finding a fault and two covariates: coverage level and test suite size. Results showed that the *all-uses* criterion was significantly more effective than *all-edges* in five of the nine subjects, and significantly more effective than the *null* criterion for six of the nine subjects. In contrast, in those subjects in which *all-edges* was more effective than the null criterion, it was usually only a little more effective. The results also show that reaching 100% coverage for the *all-uses* criterion after excluding unexecutable DUAs can significantly improve the effectiveness of the criterion.

Hutchins et al. [32] performed a study to investigate the effectiveness of both *all-DUs* and *all-edges* on seven moderate size C programs. Ten experienced programmers manually seeded faults (130 overall) for which test pools of sizes ranging from 1067 to 5548 were generated to ensure that each reachable coverage unit (edge or DUA) was covered by at least 30 test cases. Test cases were randomly selected from the test pool. If a selected test case increased the coverage achieved by the previously selected tests, it was added to the test set. Fault detection effectiveness was measured as the proportion of test suites, within each 2% coverage interval or 2 size units, that detected the fault in a faulty program version. Hutchins et al. [32] reported that fault detection for both *all-DUs* and *all-edges* increased exponentially with the coverage level. The gain in fault detection is particularly significant in the last 10-20% coverage. The *All-DUs* criterion is more effective than *all-edges* criterion but is also more expensive.

Mathur and Wong [2, 69, 70] studied the subsumption relationship, and relative cost and effectiveness of the *all-uses* and mutation criteria. They show that mutation and *all-uses* cannot be compared using the strict subsumes relation. Therefore, they defined a relation called *ProbSubsumes*, which is a probabilistic definition of the subsumes relation and is restricted to a specific program. Mathur and Wong [2, 69, 70] used a suite of five faulty programs and used a tool for generating test cases. For each criterion, they produced 30 test sets that satisfy the criterion. Their results indicate that *mutation* criterion is stronger than (i.e., *ProbSubsumes*) *all-uses* criterion and is more effective. However, the *mutation* criterion cost more.

Offutt et al. [51] reported the results of a comparison study between the *all-uses* and mutation criteria for unit testing (i.e., methods). The authors compared the subsumption relation, and the relative cost and effectiveness of the two criteria using 10 small Fortran and C programs. They produced five independent test sets for each criteria. Mutation operators, different from the ones used to produce the mutants, were used to inject faults in the programs. The results show that mutation *ProbSubsumed* the *all-uses* criterion. Mutation was also more effective but cost more.

Frankl and Iakounenko [24] reported a sharp increase in fault detection in the last 10% coverage for the *all-uses* and *all-edges* criteria. They tested larger C programs than the ones used in the above studies. Their results complement the results obtained by Frankl and Weiss [25].

Andrews et al. [8] reports the results of an empirical study performed on one industrial program with known system testing faults. The authors investigate the feasibility of using mutation analysis to assess the cost-effectiveness of four control and data flow criteria (*all-nodes*, *all-edges*,*all-p-uses*, and *all-c-uses*). The results show that mutation analysis is potentially useful to assess and compare test suites and criteria in terms of their cost-effectiveness by showing that the results are similar to what would be obtained with actual faults. The authors also investigated the relative cost and effectiveness of the above four criteria in terms of fault detection, test suite size, and control/data flow coverage. The results indicate that none of the four criteria is more cost-effective than the other. More demanding criteria, such as *all-p-uses* and *all-c-uses*, require larger test suites that detect more faults. In other words, their relationships between fault detection and test suite size are similar. However, their cost varies significantly for a given coverage level.

Chapter 4

A Revised Fault Model for AO Programs

Several fault models for aspect-oriented programs have been proposed in the literature [3, 9, 14, 22, 67]. Ferrari et al. [22] presented a summary that combines all the fault types from previous studies as well as faults that the authors identified. However, the resulting fault model can be improved in several ways. First, as the authors themselves stated, inclusion relations between these faults were not studied (i.e., some of the fault types can overlap). Second, the summary did not include fault types corresponding to incorrect data-flow interactions in a program.

We define a fault type using a pattern-like description. With each fault type, we specify fault type name, constructs that can contain the faults, and the effect of these faults on the program.

We classify the fault types into four categories in a similar way as Ferrari et al. [22] did, but with some modifications: (1) pointcut descriptor faults, (2) aspect declaration faults, (3) advice, aspect method, and intertype method implementation faults, and (4) class implementation faults. We made three modifications. First, we moved the advice declaration faults that were present in category 3 to category 2 because we wanted to keep the declaration faults and implementation faults in two separate categories. Different mutation tools are used to generate the faults in these two categories. Second, we included faults that occur in intertype methods and aspect methods in category 3, which already included faults in the advice implementation. Third, we added new fault types resulting from incorrect data-flow interactions to categories 3 and 4 in the fault model.

4.1 Pointcut Descriptor Faults (F1)

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

- (F1-1) The pointcut matches a set of join points that contains only unintended join points [38].
- (F1-2) The pointcut matches a set of join points that contains unintended join points and some intended join points [38].
- (F1-3) The pointcut matches all intended join points and some unintended join points [38].
- (F1-4) The pointcut matches a subset of intended join points and no unintended join points [38].

Lemos et al. [38] identified four types of faults that can occur in a pointcut descriptor. These fault types were also adopted by Ferrari et al. [22] and Delamare et al. [17]. The fault types are defined according to the set of join points matched by the pointcut, and these are the first four types in our revised fault model. A special case of F1-1 and F1-4 occurs when the pointcut does not match any join point. In our study we treat this case separately and refer to it as F1-5.

van Deursen et al. [67] defined three faults types which Ferrari et al. [22] adopted. However, their fault types overlap with the fault types we kept in the revised model. For example, the fault type “incorrect use of primitive pointcut designator” or the fault type “incorrect pointcut composition rules”, can generate a different set of join points than the intended one, and thus, overlap with any of the fault types, F1-1 through F1-5. In order to prevent overlapping faults, we do not include Deursen et al.'s fault types separately . Baekken and Alexander [9] described a fault model for pointcut descriptors based on the constructs that cause the fault. That is, for every construct, the authors enumerated all possible faults that might occur. These fault types are subsumed by our revised fault types.

Pointcut descriptor faults can produce several effects on the program. Matching an unintended join point adds a behavior in an unexpected point in the program, and may cause incorrect changes in the control-flow of the matched method, incorrect changes in the object state, and failure to obey the post-condition of the method where the join point is matched. Missing a join point may result in an incorrect implementation of the crosscutting concern.

Pointcut descriptor faults can be generated by any pointcut designator, logical operator, or parameters used in a pointcut description. Baekken and Alexander [9] described these constructs in detail. The pointcut fault model can be used to define pointcut mutation operators that generate the faults types in the revised model.

4.2 Aspect Declaration Faults (F2)

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

- (F2-1) Incorrect method name in introduction, leading to a missing or unanticipated method override [67].

- (F2-2) Incorrect class name in a member-introduction [67].

- (F2-3) Incorrect declaration of parent class or interface.
- (F2-4) Incorrect declaration of error and warning statements [22].
- (F2-5) Incorrect aspect precedence [67, 3].
- (F2-6) Incorrect aspect instantiation rules and deployment [22].
- (F2-7) Incorrect advice type specification [67, 22].
- (F2-8) Advice bound to incorrect pointcut [22].

Fault type F2-1 occurs when an intertype method unintentionally overrides a method in the class. The fault is limited to inherited methods because AspectJ does not allow overriding methods defined in the class by intertype methods. Ceccato et al. [14] defined a fault type which they described as "Incorrect changes in polymorphic calls" that occurs when an aspect incorrectly overrides a method inherited from a superclass. We do not include Ceccato et al.'s this fault type since it is actually an *effect* of a fault of type F2-1.

F2-2 faults occur when the intertype method is defined in a wrong class. Similar to F2-1, F2-2 faults can also have the effect of incorrect changes in polymorphic calls, and can also have the effect of having a method body in the wrong place in the class hierarchy [67].

Type F2-3 faults occur when the aspect declares incorrect parent classes or interfaces to the classes. The effects of this fault include incorrect changes in the class hierarchy, inconsistent parent declaration, or inconsistent overridden method introduction. The last two effects were defined as fault types by van Deursen et al. [67].

Type F2-4 faults occur in the *declare warning* or *declare error* statements, using which, a developer can instruct the compiler to issue a warning or error at specified locations in the program. An incorrect use of these statements can cause failures in the implementation of both the core and base concerns.

Type F2-5 faults occur in the statement *declare precedence*, which specifies the order in which advices that match the same join point must be executed. This fault type can result in (1) a failure to preserve advised method post-conditions, (2) a failure to obey advice pre-conditions, and (3) a violation of object state invariants.

Type F2-6 through F2-8 faults occur in the advice declaration statement. A type F2-6 fault occurs when the developer incorrectly sets the aspect instantiation rule. By default, an aspect has exactly one instance that cuts across the entire program. However, AspectJ allows using different aspect instantiation rules using the aspect declaration statement (e.g., per executing object using *perthis(Pointcut)*) [22]. An incorrect use of such a declaration causes a failure to preserve the advised method post-condition.

Type F2-7 faults occur when a developer incorrectly specifies the advice type, such as by using *before* instead of *after*. A type F2-8 fault occurs when a developer associates the advice with the incorrect pointcut. For example, instead of associating the advice to the pointcut that matches the class constructors, the advice is associated with the pointcut that matches the class method. Type F2-7 and F2-8 faults might cause a failure in preserving the advised method's post-condition, a violation of a state invariant, an incorrect change in data and control dependencies, and incorrect object states used in advices and methods.

4.3 Advice, Aspect Method, and Intertype Method Implementation Faults (F3)

Several prior fault models contained faults types in category F3, and were also adopted by Ferrari et al. [22]. However, three of the fault types were defined based on the effects of the faults. The first type was defined by Zhang and Zhao [79] as "infinite loop resulting from interactions among advices", which they described as resulting from "the accidental matching of unexpected join point". This fault type is caused by matching unintended join points. The other two types, which were defined by Alexander et al. [3], are "incorrect control or data flow changes", and "violating state invariant and failing to establish expected post-conditions". These are also caused by faults that can occur in a pointcut descriptor, an aspect declaration, or an advice implementation.

The revised fault types are as follows:

- (F3-1) Incorrect guarding statement or missing *proceed* in *around* advice [67]: 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. [41] for Java methods.

- (F3-4) Incorrect access to join point static information [22]: This fault type occurs when the construct, *thisJoinPoint*, is incorrectly used.

The effects of this category of fault types are summarized in Table 4.1.

Table 4.1: Effects of advices, aspect methods, and intertype methods implementation faults

Type	**Effects**
F3-1	Failure to obey advised method's post-condition Violate state invariants Incorrect changes in data dependencies Incorrect object state used in advices and methods
F3-2	Failure to obey advised method post-condition Incorrect changes in data and control dependencies Incorrect object state used in advices and methods
F3-3	Failure to obey advised method post-condition Violate state invariants Incorrect changes in data dependencies Incorrect object state used in advices and methods
F3-4	Failure to obey advised method post-condition

4.4 Class Implementation Related Faults (F4)

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

- (F4-1) Passing an object in an unexpected state to an advice: An advice can expect the objects of the base class at a join point to be in a certain state. Failure to pass an object with the expected state causes an incorrect behavior during advice execution.

- (F4-2) Arguments passed to the 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. [41] 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 [41]. We used the faults described by Ma et al. [41].

Table 4.2 shows the effects that may result from class implementation faults.

Table 4.2: Class implementation faults and their effects

Type	Effects
F4-1	Failure to obey advised method post-condition Incorrect object state used in advice and method
F4-2	Failure to obey advised method post-condition Violate state invariant
F4-3	Incorrect implementation of the core concern
F4-4	Incorrect implementation of the core concern

Chapter 5

Approach

This chapter presents the AOSV test criteria, which require testing different types of data-flow interactions for state variables in an advised class. In Section 5.1, we describe the advised class representation we used to define the different types of the state variable DUAs. We present the AOSV data-flow test criteria in Section 5.2.

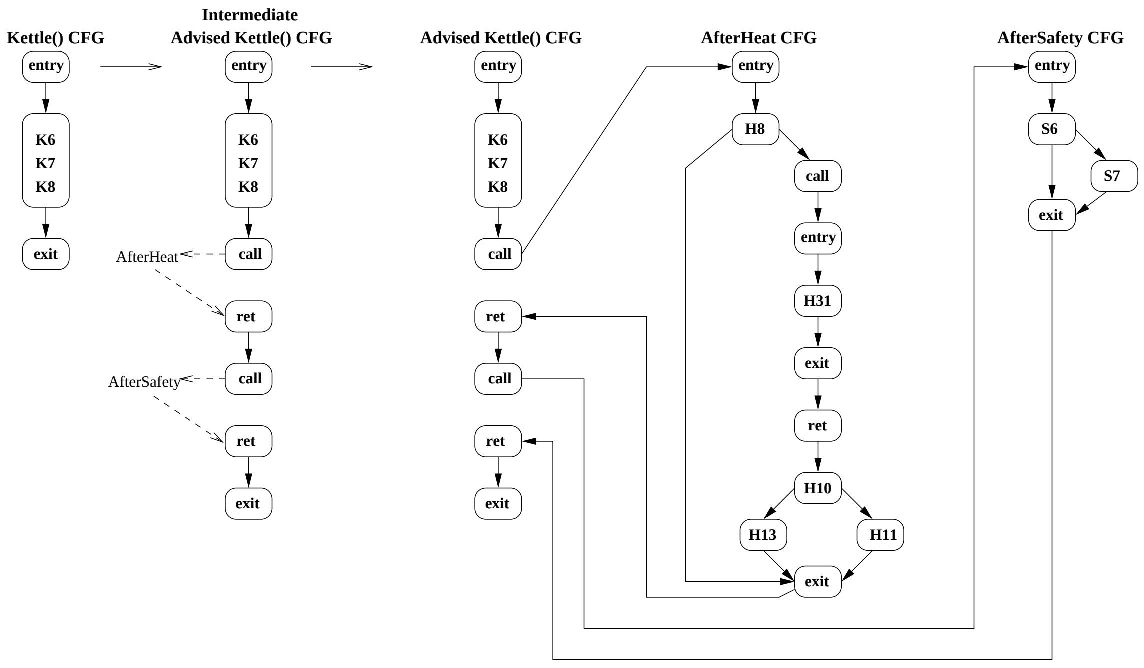

Figure 5.1: Obtaining the CFG of the *Kettle* class default constructor

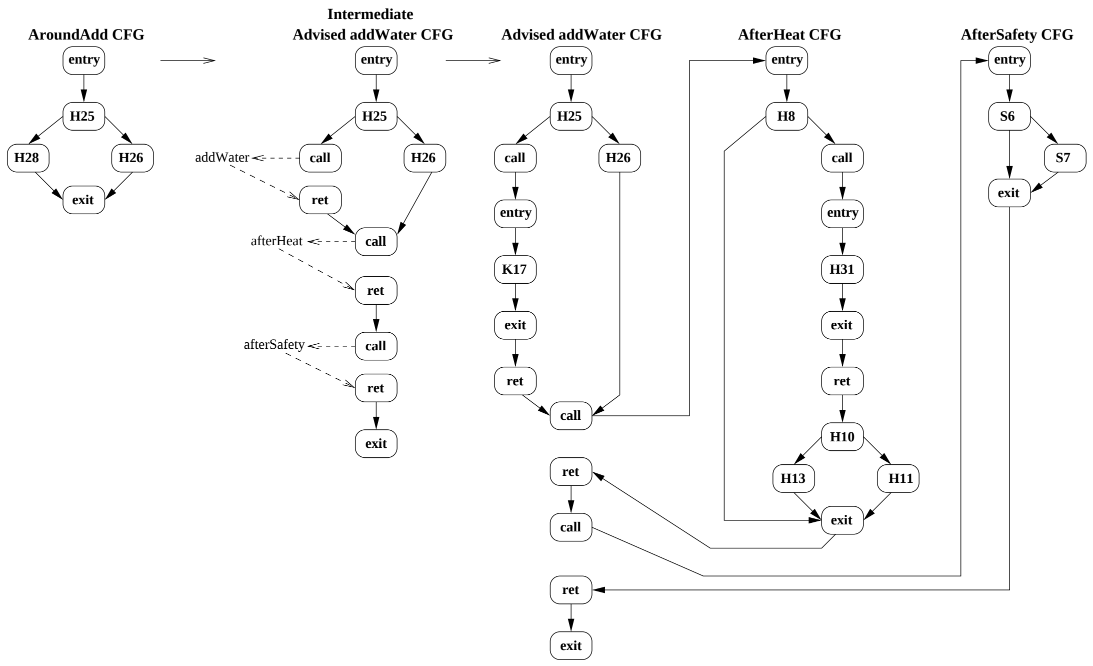

Figure 5.2: Obtaining the CFG of the *addWater* advised method.

5.1 Class Representation

The data-flow test criteria described in this chapter are defined using a framed ICFG for AO programs. The framed ICFG is obtained by adding a call frame to the ICFG produced using AJANA [78]. We begin by describing how AJANA produces the ICFG for AO programs. The description is demonstrated using the *Kettle* program shown in Figures 2.1 through 2.3.

5.1.1 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. Figure 5.1 shows the steps performed to obtain the CFG of the advised default 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. Figure 5.2 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 CFG's of the corresponding advices and method.

5.1.2 Extending AJANA

The ICFG shows what methods and advices are invoked from a single call to each method and constructor of the class. Figure 5.3 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 $< waterAmount, H26, S6 >$ in Figure 5.3). 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 $< waterAmount, H26, H18 >$ in Figure 5.3. This DUA cannot be defined unless we have a path that connects the CFGs of method *addWater* and method *pourWater*. For OO programs, Harrold and Rothermel [28] proposed the use of a

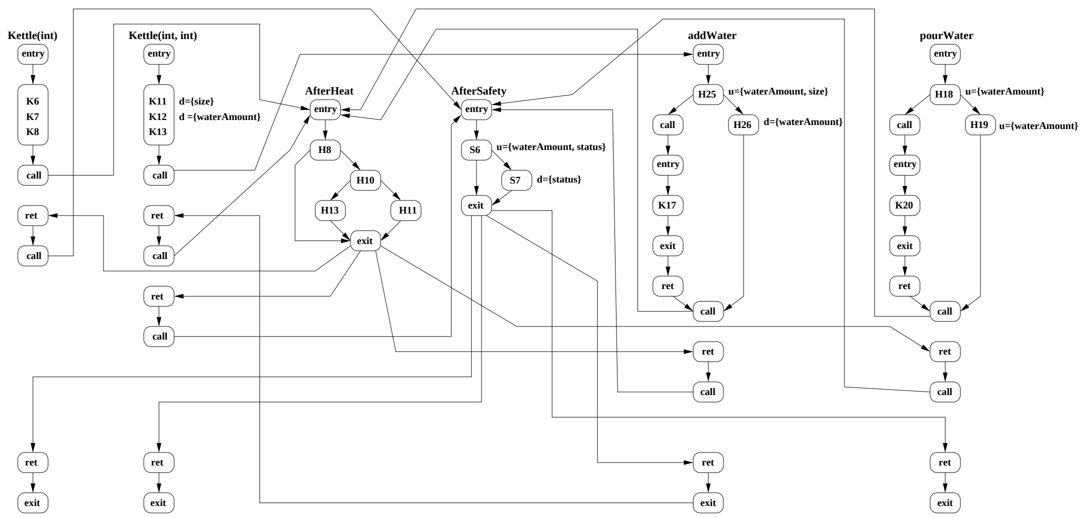

Figure 5.3: ICFG of the *Kettle* AO program.

frame that provides paths between public methods. For AO programs, Zhao [80] 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.

Figure 5.4 shows the framed ICFG for the *Kettle* class. 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 <*waterAmount,H25,H17*> can be defined because of the frame edge that connects the CFGs of the method, *addWater*, and the method, *pourWater*. Methods introduced by aspects (e.g., method *readTemperature*) are treated as any other method of the base class. Due to space limitations, the figure shows only some of the *defs* and *uses* of the kettle state variables.

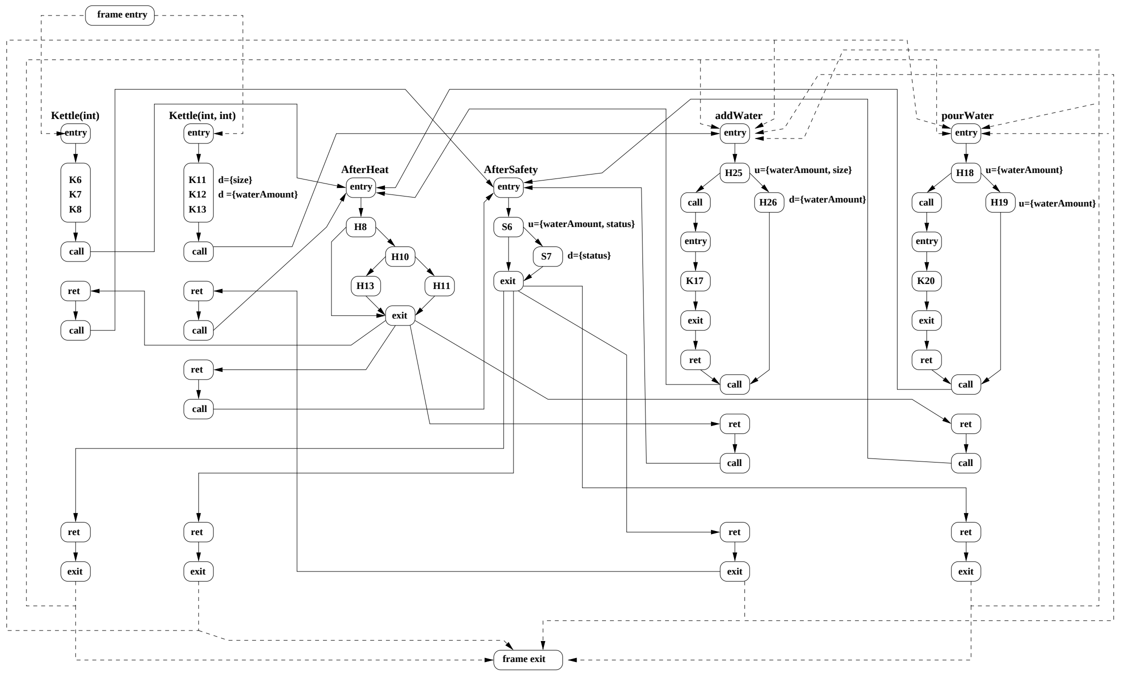

Figure 5.4: Framed ICFG of the *Kettle* AO program.

5.2 AOSV Test Criteria

Data-flow interaction using parameter passing produces DUAs where passed parameters have *defs* in the advised methods and *uses* in the advices (i.e., the passed parameters defined in the advices do not reach uses in the advised methods). The *all-uses$_c$* criterion suggested by Lemos et al. [39] requires covering such interactions. The proposed data-flow test criteria require covering interactions that are based on state variables. The criteria also require covering data-flow interactions between aspects in an AO program.

Taking into account the scope classification described by Rinard et al. [61], we define the following DUAs for state variables in AO programs (called AOSV-DUAs):

1. *Observation DUAs (oDUAs).* Advices might use state variables that the methods define. In Figure 5.4, the *def* of state variables *size* and *waterAmount* at statements K11 and K12 in the *Kettle* class reach their uses in statement H25 in the *HeatControl* aspect. Therefore, <*waterAmount, K12, H25*>, and <*size, K11, H25*> are both oDUAs.

   Formally, an oDUA is a triple $< v, d, u >$ 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 of the AO program.

2. *Activation DUAs (aDUAs).* Methods might use state variables that the advices define. For example, in the framed ICFG in Figure 5.4, both *around* advices have *defs* for state variable *waterAmount* that can be reached in method *addWater*. Therefore, <*waterAmount, H26, K17*>, <*waterAmount, H19, K17*> are both aDUAs.

   Formally, an aDUA is a triple $< v, d, u >$ 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 of the AO program.

3. *Class DUAs (cDUAs).* In an AO program, DUAs of state variables might be defined between nodes that belong to the base class methods only. For

example, in the framed ICFG in Figure 5.4, the *def* of *waterAmount* in statement K12, reaches its *use* in statement K17.

   Formally, a cDUA is a triple $< v, d, u >$ 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 of the AO program.

4. *Aspect DUAs (asDUAs).* In an AO program, DUAs of state variables might be defined between nodes that belong to advices and methods of the same aspect. In the framed ICFG in Figure 5.4, the *def* of *waterAmount* in statement H26, reaches its *use* in statement H25 in aspect *HeatControl*.

   Formally, an asDUA is a triple $< v, d, u >$ 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 for v in the framed ICFG of the AO program.

5. *Multiple Aspects DUAs (maDUAs).* An AO program might contain DUAs for state variables between advices and aspect methods that belong to different aspects. In the framed ICFG in Figure 5.4, the *def* of *waterAmount* in statement H26 of aspect *HeatControl*, reaches the use in aspect *SafetyControl*, statement S6.

   Formally, an maDUA is a triple $< v, d, u >$ where d is a node in the CFG of an advice or method of aspect *s1* that contains a *def* of state variable v; u is a node in the CFG of an advice or method of aspect *s2* 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.

Giving the above types of DUAs for state variables in an AO program, we define the following AOSV test criteria:

1. *All-uses-observation (all-uses$_o$)* criterion. Requires exercising all oDUAs in the AO program at least once.

2. *All-uses-activation (all-uses$_a$)* criterion. Requires exercising all aDUAs in the AO program at least once.

3. *All-uses-class (all-uses$_c$)* criterion. Requires exercising all cDUAs in the AO program at least once.

4. *All-uses-aspect (all-uses$_{as}$)* criterion. Requires exercising all asDUAs in the AO program at least once.

5. *All-uses-multiple-aspects (all-uses$_{ma}$)* criterion. Requires exercising all maDUAs in the AO program at least once.

6. *All-uses-state (all-uses$_s$)* criterion. Requires satisfying the *all-uses$_o$*, *all-uses$_a$*, *all-uses$_c$*, *all-uses$_{as}$*, and *all-uses$_{ma}$* criteria.

Chapter 6

Tool Implementation

In this chapter, we describe the tool that we developed to measure the coverage of the AOSV test criteria. Our tool, called Data-flow Coverage Measurement Tool for AspectJ Programs (DCT-AJ), works in three phases: DUA identification, program instrumentation, and test execution. Figure 6.1 shows the steps involved in using DCT-AJ and how the components inside the tool interact. In Sections 6.1 through 6.3, we describe each of the three phases of DCT-AJ. We describe the usage instructions and the limitations of DCT-AJ in Sections 6.4 and 6.5, respectively.

6.1 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 5.2). 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 5.2 to

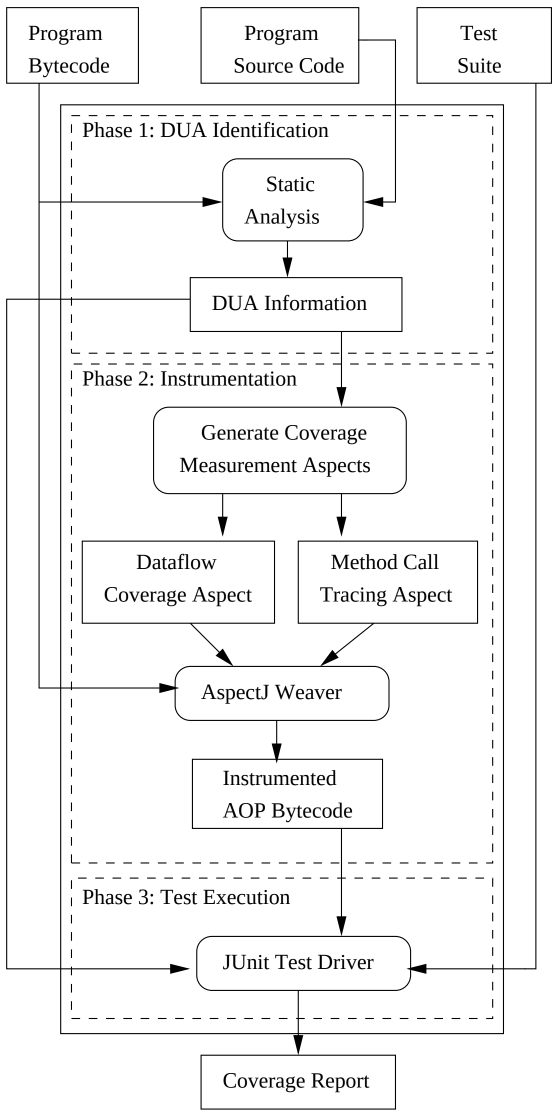

Figure 6.1: DCT-AJ: Data-flow coverage measurement tool.

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. The first statement loads (reads) the array into an intermediate variable. The second statement accesses the array using the intermediate variable. When we parse the bytecode, we do not considering the first statement as a *use* of the array. Instead, we treat *def* and *use* of the intermediate variables as *def* and *use* for the array.

4. **List the DUAs.** We implemented the iterative data-flow algorithm proposed by Pande et al. [57] to identify the DUAs of the state variables. Our implementation does not deal with aliasing.

5. **Map Jimple method names to Bytecode method names.** AJANA uses the *abc*[1] AspectJ compiler and uses the Jimple representation produced by the static weaving component of the *abc* compiler. Jimple is an intermediate representation suitable for optimization produced by Soot[2], a framework that the *abc* compiler is built on.

   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 Li-

[1]`http://abc.comlab.ox.ac.uk`

[2]`http://www.sable.mcgill.ca/soot/`

brary (BCEL)[3], and maps Jimple methods and advices names to their corresponding bytecode names.

6. **Classifying and saving DUAs.** DUAs are classified according to the types described in Section 5.2. 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.

6.2 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.

6.2.1 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 name. Therefore, identical aspect names will never be

[3]`http://jakarta.apache.org/bcel`

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.

Figure 6.2 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.

```
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
}
```

Figure 6.2: Method call tracing aspect for the *Kettle* class.

6.2.2 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. [47]. In this approach, each *def* of a state variable, *sv*, that is executed is recorded. This *def* is called the *lastDef(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 *lastDef(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. Figure 6.3 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 *get* 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.

6.3 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 num-

```
public aspect DataCoverage_ekettle_Kettle extends DataCoverage {
       pointcut getting(): ( get(* *)) &&
                           ( this(ekettle.Kettle) ||
                             this(ekettle.HeatControl) ||
                             this(ekettle.SafetyControl));
       pointcut setting(): ( set(* *)) &&
                           ( this(ekettle.Kettle) ||
                             this(ekettle.HeatControl) ||
                             this(ekettle.SafetyControl));
       pointcut loadTestDriver(String a[]):
                           execution(public static void
                           testcases.KettleTest.main(..) )
                           && args(a);
}
```

Figure 6.3: Dataflow coverage pointcuts for the *Kettle* class.

ber of state variables, the number of DUAs for each DUA type, and the percent of DUAs of each type that were covered during testing. The report also computes the coverage for each state variable.

6.4 Usage Instructions

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

1. *AspectBench* compiler[4]. The compiler is required by AJANA.

2. An AspectJ 1.6 compiler.

3. Apache Bytecode Engineering Library (BCEL)[5].

[4]http://abc.comlab.ox.ac.uk

[5]http://jakarta.apache.org/bcel/

4. The *DCTgraph* graph library. This is an extension of a graph library called *JGraphT* [6]. 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 6.2 requires executing the following command from the command line:

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

[6]http://jgrapht.sourceforge.net/

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.

6.5 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 AJANA 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, 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.

Chapter 7

Empirical Study Approach

This chapter describes how the empirical study was set up for evaluating the cost and effectiveness of the AOSV test criteria. Section 7.1 defines control criteria, which we used to compare with the AOSV test criteria. Section 7.2 states the research questions. Section 7.3 describes the metrics used to measure cost and effectiveness. Section 7.4 states the hypotheses for the study. Section 7.5 describes the approach of measuring coverage. Section 7.6 summarizes the characteristics of the subject programs. Section 7.7 and Section 7.8 describe the approach for seeding faults in the programs, and the approach used to generate test suites that satisfy the test criteria, respectively. Finally, Section 7.9 explains the statistical analysis approach used to analyze the results.

7.1 Control Criteria

We used two control 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 advices that advise 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 advices 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., [39, 48, 72]). However, these criteria are either defined for aspects (e.g., Mortensen and Alexander [48], Xie and Zhao [72]), or for advised classes but without the advices and intertype methods (e.g., Lemos et al. [39]). Therefore, these criteria have different scope than the AOSV test criteria and cannot be compared with them.

7.2 Research Questions

Our empirical studies provide answers to the following questions:

1. What is the cost of achieving full coverage for each test criterion? How does a criterion compare with another in this respect? The AOSV criteria require satisfying only a subset of the DUAs required by the *all-uses* criterion. Nevertheless, this subset of DUAs requires executing paths that involve calling one or more methods of the base class, and executing advices that advise different methods of the base class. Therefore, we expect that covering these paths to be hard and, therefore, the cost of the AOSV criteria to be more than the cost of the *AO control-flow* criteria. Among the AOSV criteria, we expect the *all-uses*$_s$ criterion to cost more than the other AOSV criteria because it subsumes all of them. We expect the other AOSV criteria to have different costs because they require executing different paths.

2. With 100% coverage, what is the effectiveness of the test criteria in terms of their ability to detect faults? How do the criteria compare with each other in this respect? The AOSV criteria target faults related to data-flow interactions in the program, which are hard to detect by unit testing the aspects or the classes. Therefore, we expect the AOSV criteria to be more effective in detecting faults than the AO control-flow criteria. Among the AOSV criteria, we expect the *all-uses*$_s$ criterion to be more effective than the other AOSV criteria because it subsumes all of them.

3. What types of faults can be detected by using the AOSV test criteria? We classify the mutants in the study according to the revised AO fault model presented in Section 4. We expect a variation in the effectiveness of the test criteria when detecting different types of faults because covering the different types of AOSV DUAs requires executing different paths in the advised classes.

4. What is the cost-effectiveness of achieving high coverage levels for the *all-uses*$_s$ test criterion? 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 our study, we evaluate cost and effectiveness of the *all-uses*$_s$ criterion for three coverage levels: 100%, 90%, and 80% coverage levels. We choose the *all-uses*$_s$ criterion because it subsumes all the other AOSV criteria. Therefore, it is the most important criterion that requires covering all the state variables DUAs 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.

7.3 Metrics

In this section, we define the metrics for measuring the cost and effectiveness of the test criteria. We measured three dimensions of the cost of a test criterion. These are as follows:

1. The size of a test suite that satisfies a criterion.

2. The density of a test case

3. The effort of obtaining a test suite that satisfies a criterion.

We measured these dimensions of cost using four cost metrics named c_1 through c_4. Metric c_1 refers to the number of test cases in a test suite needed to satisfy a criterion. The metric is used to measure the size of a test suite.

The number of test cases has been widely used to measure the size of a test suite [11]. However, we noticed that c_1 varies in different classes depending on the number of test requirements for a criterion. For example, two classes *class1* and *class2* might have a large difference in the number of test cases in a suite that satisfy the same criterion in each class because the two classes have a large difference in the number of test requirements for the criterion. When comparing two test criteria A and B in two classes, A might have a large number of test requirements in *class1* and few in *class2*, while B might have the opposite. Therefore, we used metric c_2 in order to measure the cost regardless of the number of requirements a criterion might have in a certain class.

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 [11] 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 define the following two metrics to measure the time needed to obtain a test suite that satisfies a criterion:

c_3. The number of test suites we generate until a test suite that satisfies the test criterion is obtained.

c_4. The number of test suites we generate until a test suite that satisfies the test criterion is obtained divided by the number of test requirements for the criterion that the advised class contains.

The time needed to generate a test suite depends on the approach and whether or not the approach is automated. In our study, we generate test suites by repeatedly adding test cases to a suite until the desired coverage is obtained. In each iteration, we produce a new test suite by adding a test case to the existing test suite. If coverage is increased, we save the new test suite and repeat the process of adding test cases until the desired coverage level is reached. In the process, we produce many test suites. Therefore, we propose to use the number of test suites (c_3) generated until a test suite that satisfies a criterion is obtained as a measure of cost.

Metric c_3 depends on the number of test requirements for the test criterion in a class. Therefore, we used metric c_4, which measures the effort of covering a test requirement for a criterion.

We did not measure the cost of obtaining the test requirements because we used tools to obtain the test requirements for the AOSV criteria and the AO control-flow criteria (i.e., DCT-AJ and *CodeCover*). The difference in the computational time of running the tools is negligible.

We measure the effectiveness of a test criterion by the percentage of faults detected by a test suite that satisfies a test criterion. We generate a number of faulty versions of each class, 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 advices 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, we computed the effectiveness over all the faults as well as for each fault type defined in Chapter 4.

7.4 Hypotheses

In this section, the hypotheses of the study are presented. The hypotheses reflect the expectations that (1) there is a difference in the cost and effectiveness between the test suites that satisfy the AOSV test criteria and those that satisfy the AO control-flow test criteria, and (2) there is a difference in the cost and effectiveness between the test suites that satisfy the AOSV test criteria themselves.

We group the hypotheses into 5 groups, where each group corresponds to a metric described in Section 7.3.

7.4.1 Group A: Comparing the Cost of the Test Criteria Using the Number of Test Cases in the Test Suites

Hypotheses in group A reflect our expectations that (1) the number of test cases in the test suites that satisfy the AOSV test criteria differ from the number of test cases in the test suites that satisfy the AO control-flow test criteria, and that (2) the number of test cases in the test suites that satisfy the AOSV test criteria differ from each other. We classify comparisons in this group into two types, which are as follows:

1. Table 7.1 states the null hypotheses for comparisons between the number of

test cases in the test suites that satisfy the AOSV test criteria and the number of test cases in the test suites that satisfy the AO control-flow test criteria. There are 12 hypotheses for these comparisons numbered $A1$ through $A12$.

2. Table 7.2 states the null hypotheses for comparisons between the number of test cases in the test suites that satisfy the AOSV test criteria with each other. There are 15 hypotheses for these comparisons numbered $A13$ through $A27$.

The alternative hypotheses for the hypotheses in group A state that:

- The number of test cases in the test suites that satisfy each of *all-uses*$_a$, *all-uses*$_c$, and *all-uses*$_s$ is higher than the number of test cases in the test suites that satisfy each of the AO control-flow criteria.

- There is a difference between the number of test cases in the test suites that satisfy each of *all-uses*$_o$, *all-uses*$_{as}$, and *all-uses*$_{ma}$, and the number of test cases in the test suites that satisfy each of the AO control-flow criteria.

- The number of test cases in the test suites that satisfy *all-uses*$_s$ is higher than the number of test cases in the test suites that satisfy each of the AOSV criteria it subsumes.

- The number of test cases in the test suites that satisfy *all-uses*$_c$ is higher than the number of test cases in the test suites that satisfy each of *all-uses*$_o$, *all-uses*$_{as}$, and *all-uses*$_{ma}$.

- There is a difference between the number of test cases in the test suites that satisfy each of *all-uses*$_o$, *all-uses*$_{as}$, and *all-uses*$_{ma}$ compared with each other.

Table 7.1: Null hypotheses for comparing the cost of the AOSV test criteria with the cost of the AO control-flow test criteria using the number of test cases in the test suites that satisfy the criteria (c_1)

$H0_{A1}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_a$ and the number of test cases in the test suites that satisfy *AO blocks*.
$H0_{A2}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_a$ and the number of test cases in the test suites that satisfy *AO branches*.
$H0_{A3}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_o$ and the number of test cases in the test suites that satisfy *AO blocks*.
$H0_{A4}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_o$ and the number of test cases in the test suites that satisfy *AO branches*.
$H0_{A5}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_c$ and the number of test cases in the test suites that satisfy *AO blocks*.
$H0_{A6}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_c$ and the number of test cases in the test suites that satisfy *AO branches*.
$H0_{A7}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_{as}$ and the number of test cases in the test suites that satisfy *AO blocks*.
$H0_{A8}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_{as}$ and the number of test cases in the test suites that satisfy *AO branches*.
$H0_{A9}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_{ma}$ and the number of test cases in the test suites that satisfy *AO blocks*.
$H0_{A10}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_{ma}$ and the number of test cases in the test suites that satisfy *AO branches*.
$H0_{A11}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_s$ and the number of test cases in the test suites that satisfy *AO blocks*.
$H0_{A12}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_s$ and the number of test cases in the test suites that satisfy *AO branches*.

7.4.2 Group B: Comparing the Cost of the Test Criteria Using Size Metric c_2

Hypotheses in group B reflect our expectations that (1) the number of test requirements for each of the AOSV test criteria that can be covered by a test case differ from the number of blocks or branches that can be covered by a test case, and (2) the number of DUAs that can be covered by a test case differ depending on the different types of the AOSV DUAs. We classify comparisons in this group into two types, which are as follows:

1. Table 7.3 states the null hypotheses for comparisons between the AOSV test

Table 7.2: Null hypotheses for comparing the cost of the AOSV test criteria with each other using the number of test cases in a test suite that satisfies a criterion (c_1)

$H0_{A13}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_s$ and the number of test cases in the test suites that satisfy *all-uses*$_a$.
$H0_{A14}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_s$ and the number of test cases in the test suites that satisfy *all-uses*$_o$.
$H0_{A15}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_s$ and the number of test cases in the test suites that satisfy *all-uses*$_c$.
$H0_{A16}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_s$ and the number of test cases in the test suites that satisfy *all-uses*$_{as}$.
$H0_{A17}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_s$ and the number of test cases in the test suites that satisfy *all-uses*$_{ma}$.
$H0_{A18}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_a$ and the number of test cases in the test suites that satisfy *all-uses*$_o$.
$H0_{A19}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_a$ and the number of test cases in the test suites that satisfy *all-uses*$_c$.
$H0_{A20}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_a$ and the number of test cases in the test suites that satisfy *all-uses*$_{as}$.
$H0_{A21}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_a$ and the number of test cases in the test suites that satisfy *all-uses*$_{ma}$.
$H0_{A22}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_o$ and the number of test cases in the test suites that satisfy *all-uses*$_c$.
$H0_{A23}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_o$ and the number of test cases in the test suites that satisfy *all-uses*$_{as}$.
$H0_{A24}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_o$ and the number of test cases in the test suites that satisfy *all-uses*$_{ma}$.
$H0_{A25}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_c$ and the number of test cases in the test suites that satisfy *all-uses*$_{as}$.
$H0_{A26}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_c$ and the number of test cases in the test suites that satisfy *all-uses*$_{ma}$.
$H0_{A27}$: There is no difference between the number of test cases in the test suites that satisfy *all-uses*$_{as}$ and the number of test cases in the test suites that satisfy *all-uses*$_{ma}$.

criteria and the AO control-flow test criteria using size metric c_2. There are 12 hypotheses for these comparisons numbered $B1$ through $B12$.

2. Table 7.4 states the null hypotheses for comparing the AOSV test criteria with each other using size metric c_2. There are 15 hypotheses for these comparisons numbered $B13$ through $B27$.

Table 7.3: Null hypotheses for comparing the cost of the AOSV test criteria with the cost of the AO control-flow test criteria using size metric (c_2)

$H0_{B1}$: There is no difference between the value of the size metric c_2 for the *all-uses$_a$* criterion and the value of the size metric c_2 for the *AO blocks* criterion.
$H0_{B2}$: There is no difference between the value of the size metric c_2 for the *all-uses$_a$* criterion and the value of the size metric c_2 for the *AO branches* criterion.
$H0_{B3}$: There is no difference between the value of the size metric c_2 for the *all-uses$_o$* criterion and the value of the size metric c_2 for the *AO blocks* criterion.
$H0_{B4}$: There is no difference between the value of the size metric c_2 for the *all-uses$_o$* criterion and the value of the size metric c_2 for the *AO branches* criterion.
$H0_{B5}$: There is no difference between the value of the size metric c_2 for the *all-uses$_c$* criterion and the value of the size metric c_2 for the *AO blocks* criterion.
$H0_{B6}$: There is no difference between the value of the size metric c_2 for the *all-uses$_c$* criterion and the value of the size metric c_2 for the *AO branches* criterion.
$H0_{B7}$: There is no difference between the value of the size metric c_2 for the *all-uses$_{as}$* criterion and the value of the size metric c_2 for the *AO blocks* criterion.
$H0_{B8}$: There is no difference between the value of the size metric c_2 for the *all-uses$_{as}$* criterion and the value of the size metric c_2 for the *AO branches* criterion.
$H0_{B9}$: There is no difference between the value of the size metric c_2 for the *all-uses$_{ma}$* criterion and the value of the size metric c_2 for the *AO blocks* criterion.
$H0_{B10}$: There is no difference between the value of the size metric c_2 for the *all-uses$_{ma}$* criterion and the value of the size metric c_2 for the *AO branches* criterion.
$H0_{B11}$: There is no difference between the value of the size metric c_2 for the *all-uses$_s$* criterion and the value of the size metric c_2 for the *AO blocks* criterion.
$H0_{B12}$: There is no difference between the value of the size metric c_2 for the *all-uses$_s$* criterion and the value of the size metric c_2 for the *AO branches* criterion.

The alternative hypotheses for the hypotheses in group B state that:

- The size metric c_2 for each of the AOSV test criteria is higher than the size metric c_2 for each of the AO control-flow criteria.
- The size metric c_2 for *all-uses$_s$* is higher than the size metric c_2 for each of the AOSV criteria it subsumes.
- The size metric c_2 for *all-uses$_a$* is lower than the size metric c_2 for *all-uses$_o$* and *all-uses$_c$*.
- The size metric c_2 for *all-uses$_{ma}$* is higher than the size metric c_2 for *all-uses$_o$*, *all-uses$_c$*, and *all-uses$_{as}$*.

Table 7.4: Null hypotheses for comparing the cost of the AOSV test criteria with each other using size metric (c_2)

$H0_{B13}$: There is no difference between the value of the size metric c_2 for the *all-uses*$_s$ criterion and the value of the size metric c_2 for the *all-uses*$_a$ criterion.
$H0_{B14}$: There is no difference between the value of the size metric c_2 for the *all-uses*$_s$ criterion and the value of the size metric c_2 for the *all-uses*$_o$ criterion.
$H0_{B15}$: There is no difference between the value of the size metric c_2 for the *all-uses*$_s$ criterion and the value of the size metric c_2 for the *all-uses*$_c$ criterion.
$H0_{B16}$: There is no difference between the value of the size metric c_2 for the *all-uses*$_s$ criterion and the value of the size metric c_2 for the *all-uses*$_{as}$ criterion.
$H0_{B17}$: There is no difference between the value of the size metric c_2 for the *all-uses*$_s$ criterion and the value of the size metric c_2 for the *all-uses*$_{ma}$ criterion.
$H0_{B18}$: There is no difference between the value of the size metric c_2 for the *all-uses*$_a$ criterion and the value of the size metric c_2 for the *all-uses*$_o$ criterion.
$H0_{B19}$: There is no difference between the value of the size metric c_2 for the *all-uses*$_a$ criterion and the value of the size metric c_2 for the *all-uses*$_c$ criterion.
$H0_{B20}$: There is no difference between the value of the size metric c_2 for the *all-uses*$_a$ criterion and the value of the size metric c_2 for the *all-uses*$_{as}$ criterion.
$H0_{B21}$: There is no difference between the value of the size metric c_2 for the *all-uses*$_a$ criterion and the value of the size metric c_2 for the *all-uses*$_{ma}$ criterion.
$H0_{B22}$: There is no difference between the value of the size metric c_2 for the *all-uses*$_o$ criterion and the value of the size metric c_2 for the *all-uses*$_c$ criterion.
$H0_{B23}$: There is no difference between the value of the size metric c_2 for the *all-uses*$_o$ criterion and the value of the size metric c_2 for the *all-uses*$_{as}$ criterion.
$H0_{B24}$: There is no difference between the value of the size metric c_2 for the *all-uses*$_o$ criterion and the value of the size metric c_2 for the *all-uses*$_{ma}$ criterion.
$H0_{B25}$: There is no difference between the value of the size metric c_2 for the *all-uses*$_c$ criterion and the value of the size metric c_2 for the *all-uses*$_{as}$ criterion.
$H0_{B26}$: There is no difference between the value of the size metric c_2 for the *all-uses*$_c$ criterion and the value of the size metric c_2 for the *all-uses*$_{ma}$ criterion.
$H0_{B27}$: There is no difference between the value of the size metric c_2 for the *all-uses*$_{as}$ criterion and the value of the size metric c_2 for the *all-uses*$_{ma}$ criterion.

- There is a difference between c_2 for each of *all-uses*$_o$, *all-uses*$_c$, and *all-uses*$_{as}$ compared with each other.

7.4.3 Group C: Comparing the Cost of the Test Criteria Using the Effort Metric c_3

Hypotheses in group C reflect our expectations that (1) the effort of generating test suites that satisfy the AOSV test criteria differ from the effort of generating test suites that satisfy the AO control-flow test criteria, and (2) the effort of generating

Table 7.5: Null hypotheses for comparing the cost of the AOSV test criteria with the cost of the AO control-flow criteria using effort metric (c_3)

$H0_{C1}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_a$ criterion and the value of the effort metric c_3 for the *AO blocks* criterion.
$H0_{C2}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_a$ criterion and the value of the effort metric c_3 for the *AO branches* criterion.
$H0_{C3}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_o$ criterion and the value of the effort metric c_3 for the *AO blocks* criterion.
$H0_{C4}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_o$ criterion and the value of the effort metric c_3 for the *AO branches* criterion.
$H0_{C5}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_c$ criterion and the value of the effort metric c_3 for the *AO blocks* criterion.
$H0_{C6}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_c$ criterion and the value of the effort metric c_3 for the *AO branches* criterion.
$H0_{C7}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_{as}$ criterion and the value of the effort metric c_3 for the *AO blocks* criterion.
$H0_{C8}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_{as}$ criterion and the value of the effort metric c_3 for the *AO branches* criterion.
$H0_{C9}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_{ma}$ criterion and the value of the effort metric c_3 for the *AO blocks* criterion.
$H0_{C10}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_{ma}$ criterion and the value of the effort metric c_3 for the *AO branches* criterion.
$H0_{C11}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_s$ criterion and the value of the effort metric c_3 for the *AO blocks* criterion.
$H0_{C12}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_s$ criterion and the value of the effort metric c_3 for the *AO branches* criterion.

test suites that satisfy the AOSV test criteria differ from each other, where effort is measured using metric c_3. We classify comparisons in this group into two types, which are as follows:

1. Table 7.5 states the null hypotheses for the comparisons between the effort of generating test suites that satisfy the AOSV test criteria and the effort of generating test suites that satisfy the AO control-flow test criteria. There are 12 hypotheses for these comparisons numbered $C1$ through $C12$.

2. Table 7.6 states the null hypotheses for comparing the cost of the AOSV test criteria with each other using effort metric c_3. There are 15 hypotheses for these comparisons numbered $C13$ through $C27$.

Table 7.6: Null hypotheses for comparing the cost of the AOSV test criteria with each other using effort metric c_3

$H0_{C13}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_s$ criterion and the value of the effort metric c_3 for the *all-uses*$_a$ criterion.
$H0_{C14}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_s$ criterion and the value of the effort metric c_3 for the *all-uses*$_o$ criterion.
$H0_{C15}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_s$ criterion and the value of the effort metric c_3 for the *all-uses*$_c$ criterion.
$H0_{C16}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_s$ criterion and the value of the effort metric c_3 for the *all-uses*$_{as}$ criterion.
$H0_{C17}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_s$ criterion and the value of the effort metric c_3 for the *all-uses*$_{ma}$ criterion.
$H0_{C18}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_a$ criterion and the value of the effort metric c_3 for the *all-uses*$_o$ criterion.
$H0_{C19}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_a$ criterion and the value of the effort metric c_3 for the *all-uses*$_c$ criterion.
$H0_{C20}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_a$ criterion and the value of the effort metric c_3 for the *all-uses*$_{as}$ criterion.
$H0_{C21}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_a$ criterion and the value of the effort metric c_3 for the *all-uses*$_{ma}$ criterion.
$H0_{C22}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_o$ criterion and the value of the effort metric c_3 for the *all-uses*$_c$ criterion.
$H0_{C23}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_o$ criterion and the value of the effort metric c_3 for the *all-uses*$_{as}$ criterion.
$H0_{C24}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_o$ criterion and the value of the effort metric c_3 for the *all-uses*$_{ma}$ criterion.
$H0_{C25}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_c$ criterion and the value of the effort metric c_3 for the *all-uses*$_{as}$ criterion.
$H0_{C26}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_c$ criterion and the value of the effort metric c_3 for the *all-uses*$_{ma}$ criterion.
$H0_{C27}$: There is no difference between the value of the effort metric c_3 for the *all-uses*$_{as}$ criterion and the value of the effort metric c_3 for the *all-uses*$_{ma}$ criterion.

The alternative hypotheses for the hypotheses in group C state that:

- The effort metric c_3 for *all-uses*$_a$ and *all-uses*$_s$ is higher than the effort metric c_3 for each of the AO control-flow criteria.
- There is a difference between c_3 for *all-uses*$_o$, *all-uses*$_c$, *all-uses*$_{as}$, and *all-uses*$_{ma}$ and c_3 for each of the AO control-flow criteria.
- The effort metric c_3 for *all-uses*$_s$ is higher than the effort metric c_3 for each

of the AOSV criteria it subsumes.

- There is a difference between c_3 for each of *all-uses*$_a$, *all-uses*$_o$, *all-uses*$_c$, *all-uses*$_{as}$, and *all-uses*$_{ma}$ compared with each other.

7.4.4 Group D: Comparing the Cost of the Test Criteria Using the Effort Metric c_4

Table 7.7: Null hypotheses for comparing the cost of the AOSV test criteria with the cost of the AO control-flow test criteria using effort metric (c_4)

$H0_{D1}$: There is no difference between the value of the effort metric c_4 for the *all-uses*$_a$ criterion and the value of the effort metric c_4 for the *AO blocks* criterion.
$H0_{D2}$: There is no difference between the value of the effort metric c_4 for the *all-uses*$_a$ criterion and the value of the effort metric c_4 for the *AO branches* criterion.
$H0_{D3}$: There is no difference between the value of the effort metric c_4 for the *all-uses*$_o$ criterion and the value of the effort metric c_4 for the *AO blocks* criterion.
$H0_{D4}$: There is no difference between the value of the effort metric c_4 for the *all-uses*$_o$ criterion and the value of the effort metric c_4 for the *AO branches* criterion.
$H0_{D5}$: There is no difference between the value of the effort metric c_4 for the *all-uses*$_c$ criterion and the value of the effort metric c_4 for the *AO blocks* criterion.
$H0_{D6}$: There is no difference between the value of the effort metric c_4 for the *all-uses*$_c$ criterion and the value of the effort metric c_4 for the *AO branches* criterion.
$H0_{D7}$: There is no difference between the value of the effort metric c_4 for the *all-uses*$_{as}$ criterion and the value of the effort metric c_4 for the *AO blocks* criterion.
$H0_{D8}$: There is no difference between the value of the effort metric c_4 for the *all-uses*$_{as}$ criterion and the value of the effort metric c_4 for the *AO branches* criterion.
$H0_{D9}$: There is no difference between the value of the effort metric c_4 for the *all-uses*$_{ma}$ criterion and the value of the effort metric c_4 for the *AO blocks* criterion.
$H0_{D10}$: There is no difference between the value of the effort metric c_4 for the *all-uses*$_{ma}$ criterion and the value of the effort metric c_4 for the *AO branches* criterion.
$H0_{D11}$: There is no difference between the value of the effort metric c_4 for the *all-uses*$_s$ criterion and the value of the effort metric c_4 for the *AO blocks* criterion.
$H0_{D12}$: There is no difference between the value of the effort metric c_4 for the *all-uses*$_{ma}$ criterion and the value of the effort metric c_4 for the *AO branches* criterion.

Hypotheses in group D reflect our expectations that (1) there is a difference between the effort of obtaining a test case that cover any type of AOSV DUAs and the effort of obtaining a test case that covers a block or a branch in advised classes, and (2) there is a difference in the effort of obtaining a test case that covers

Table 7.8: Null hypotheses for comparing the cost of the AOSV test criteria with each other using effort metric (c_4)

$H0_{D13}$: There is no difference between the value of the normalized effort metric c_4 for the *all-uses*$_s$ criterion and the value of the normalized effort metric c_4 for the *all-uses*$_a$ criterion.
$H0_{D14}$: There is no difference between the value of the normalized effort metric c_4 for the *all-uses*$_s$ criterion and the value of the normalized effort metric c_4 for the *all-uses*$_o$ criterion.
$H0_{D15}$: There is no difference between the value of the normalized effort metric c_4 for the *all-uses*$_s$ criterion and the value of the normalized effort metric c_4 for the *all-uses*$_c$ criterion.
$H0_{D16}$: There is no difference between the value of the normalized effort metric c_4 for the *all-uses*$_s$ criterion and the value of the normalized effort metric c_4 for the *all-uses*$_{as}$ criterion.
$H0_{D17}$: There is no difference between the value of the normalized effort metric c_4 for the *all-uses*$_s$ criterion and the value of the normalized effort metric c_4 for the *all-uses*$_{ma}$ criterion.
$H0_{D18}$: There is no difference between the value of the normalized effort metric c_4 for the *all-uses*$_a$ criterion and the value of the normalized effort metric c_4 for the *all-uses*$_o$ criterion.
$H0_{D19}$: There is no difference between the value of the normalized effort metric c_4 for the *all-uses*$_a$ criterion and the value of the normalized effort metric c_4 for the *all-uses*$_c$ criterion.
$H0_{D20}$: There is no difference between the value of the normalized effort metric c_4 for the *all-uses*$_a$ criterion and the value of the normalized effort metric c_4 for the *all-uses*$_{as}$ criterion.
$H0_{D21}$: There is no difference between the value of the normalized effort metric c_4 for the *all-uses*$_a$ criterion and the value of the normalized effort metric c_4 for the *all-uses*$_{ma}$ criterion.
$H0_{D22}$: There is no difference between the value of the normalized effort metric c_4 for the *all-uses*$_o$ criterion and the value of the normalized effort metric c_4 for the *all-uses*$_c$ criterion.
$H0_{D23}$: There is no difference between the value of the normalized effort metric c_4 for the *all-uses*$_o$ criterion and the value of the normalized effort metric c_4 for the *all-uses*$_{as}$ criterion.
$H0_{D24}$: There is no difference between the value of the normalized effort metric c_4 for the *all-uses*$_o$ criterion and the value of the normalized effort metric c_4 for the *all-uses*$_{ma}$ criterion.
$H0_{D25}$: There is no difference between the value of the normalized effort metric c_4 for the *all-uses*$_c$ criterion and the value of the normalized effort metric c_4 for the *all-uses*$_{as}$ criterion.
$H0_{D26}$: There is no difference between the value of the normalized effort metric c_4 for the *all-uses*$_c$ criterion and the value of the normalized effort metric c_4 for the *all-uses*$_{ma}$ criterion.
$H0_{D27}$: There is no difference between the value of the normalized effort metric c_4 for the *all-uses*$_{as}$ criterion and the value of the normalized effort metric c_4 for the *all-uses*$_{ma}$ criterion.

a DUA of each type of the AOSV DUAs. We classify comparisons in this group into two types, which are as follows:

1. Table 7.7 states the null hypotheses for comparisons between c_4 of each of the AOSV test criteria and c_4 for each of the AO control-flow test criteria. There are 12 hypotheses for these comparisons numbered $D1$ through $D12$.

2. Table 7.8 states the null hypotheses for comparing c_4 for the AOSV test

criteria with each other. There are 15 hypotheses for these comparisons numbered $D13$ through $D27$.

The alternative hypotheses for the hypotheses in group D state that:

- The effort metric c_4 for *all-uses*$_a$, *all-uses*$_{as}$, *all-uses*$_{ma}$, and *all-uses*$_s$ is higher than the effort metric c_4 for each of the AO control-flow criteria.
- There is a difference between c_4 for *all-uses*$_o$, *all-uses*$_c$, and c_4 for each of the AO control-flow criteria.
- There is a difference between c_4 for each of the AOSV criteria compared with each other.

7.4.5 Group E: Comparing the Effectiveness of the Test Criteria

Hypotheses in group E reflect our expectations that (1) the effectiveness of the test suites that satisfy the AOSV test criteria differ from the effectiveness of the test suites that satisfy the AO control-flow test criteria, and (2) the effectiveness of the test suites that satisfy the AOSV test criteria differ from each other, where effectiveness is measured by the percentage of faults detected by the test suites that satisfy the criterion. We classify comparisons in this group into two types, which are as follows:

1. Table 7.9 states the null hypotheses for comparisons between the effectiveness of the test suites that satisfy the AOSV test criteria and the effectiveness of the test suites that satisfy the AO control-flow test criteria. There are 12 hypotheses for these comparisons numbered $E1$ through $E12$.

Table 7.9: Null hypotheses for comparing the effectiveness of the AOSV test criteria with the effectiveness of the AO control-flow test criteria

$H0_{E1}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_a$ criterion and that satisfy the *AO blocks* criterion.
$H0_{E2}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_a$ criterion and that satisfy the *AO branches* criterion.
$H0_{E3}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_o$ criterion and that satisfy the *AO blocks* criterion.
$H0_{E4}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_o$ criterion and that satisfy the *AO branches* criterion.
$H0_{E5}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_c$ criterion and that satisfy the *AO blocks* criterion.
$H0_{E6}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_c$ criterion and that satisfy the *AO branches* criterion.
$H0_{E7}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_{as}$ criterion and that that satisfy the *AO blocks* criterion.
$H0_{E8}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_{as}$ criterion and that satisfy the *AO branches* criterion.
$H0_{E9}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_{ma}$ criterion and that satisfy the *AO blocks* criterion.
$H0_{E10}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_{ma}$ criterion and that satisfy the *AO branches* criterion.
$H0_{E11}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_s$ criterion and that satisfy the *AO blocks* criterion.
$H0_{E12}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_s$ criterion and that satisfy the *AO branches* criterion.

2. Table 7.2 states the null hypotheses for comparisons between the effectiveness of the test suites that satisfy the AOSV test criteria with each other. There are 15 hypotheses for these comparisons numbered $E13$ through $E27$.

The alternative hypotheses for the hypotheses in group E state that:

- The percentage of faults detected by test suites that satisfy each of the AOSV test criteria is higher than the percentage of faults detected by test suites that satisfy each of the AO control-flow criteria.

- The percentage of faults detected by test suites that satisfy *all-uses*$_s$ is higher than the percentage of faults detected by test suites that satisfy each of the AOSV criteria it subsumes.

Table 7.10: Null hypotheses for comparing the effectiveness of the AOSV test criteria with each other

$H0_{E13}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_s$ criterion and that satisfy the *all-uses*$_a$ criterion.
$H0_{E14}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_s$ criterion and that satisfy the *all-uses*$_o$ criterion.
$H0_{E15}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_s$ criterion and that satisfy the *all-uses*$_c$ criterion.
$H0_{E16}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_s$ criterion and that satisfy the *all-uses*$_{as}$ criterion.
$H0_{E17}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_s$ criterion and that satisfy the *all-uses*$_{ma}$ criterion.
$H0_{E18}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_a$ criterion and that satisfy the *all-uses*$_o$ criterion.
$H0_{E19}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_a$ criterion and that satisfy the *all-uses*$_c$ criterion.
$H0_{E20}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_a$ criterion and that satisfy the *all-uses*$_{as}$ criterion.
$H0_{E21}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_a$ criterion and that satisfy the *all-uses*$_{ma}$ criterion.
$H0_{E22}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_o$ criterion and that satisfy the *all-uses*$_c$ criterion.
$H0_{E23}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_o$ criterion and that satisfy the *all-uses*$_{as}$ criterion.
$H0_{E24}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_o$ criterion and that satisfy the *all-uses*$_{ma}$ criterion.
$H0_{E25}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_c$ criterion and that satisfy the *all-uses*$_{as}$ criterion.
$H0_{E26}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_c$ criterion and that satisfy the *all-uses*$_{ma}$ criterion.
$H0_{E27}$: There is no difference between the percentage of faults detected by test suites that satisfy the *all-uses*$_{as}$ criterion and that satisfy the *all-uses*$_{ma}$ criterion.

- There is a difference between the percentage of faults detected by test suites that satisfy each of *all-uses*$_a$, *all-uses*$_o$, *all-uses*$_c$, *all-uses*$_{as}$, and *all-uses*$_{ma}$ criteria compared with each other.

7.5 Coverage Measurement

Coverage of test suites is measured using two tools (1) DCT-AJ, which we developed to measure coverage for the AOSV test criteria, and (2) *CodeCover*[7], which we used to measure the coverage for the AO control-flow criteria. There are two limitations of DCT AJ that can effect measuring coverage in the subject programs (Section 6.5). These are (1) initialization of intertype state variables cannot be recognized by DCT-AJ, and (2) DCT-AJ cannot trace *defs* and *uses* in static methods of the aspects. Therefore, We made the following two modifications to the original programs:

1. Initialization of intertype state variables is performed in an advice that runs after the class constructor.

2. In the *Telecom* program, *getter* methods for intertype state variables are implemented as static methods of the aspect methods. We changed the static methods to intertype methods.

CodeCover is an open source tool that measures *statement* and *branch* coverage in Java programs. However, the tool does not support AspectJ programs. In order to measure the coverage in the aspects, we rewrote the aspects 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 annotate 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. *CodeCover*

[7]`http://codecover.org`

creates separate coverage reports for each class and aspect. In order to measure the coverage of advised classes, we measure coverage only within the advices that advise methods in a class.

We manually identified unreachable code by inspecting the subject programs. We dropped unreachable blocks and branches from the coverage computation for the *AO control-flow* criteria. For the AOSV criteria, we used an XML editor (XML Marker[8]) to remove unreachable DUAs from the XML file that contains the DUAs.

7.6 Subject Programs

We used 4 subject programs in our study. Table 7.11 shows the main characteristics of these programs. For each program shown in column 1, column 2 shows the lines of code (LOC). The third and fourth columns show the number of classes and the number of aspects in each program, respectively. The number of tested classes in each program is shown in column 5. Columns 6, 7, and 8 show the number of *before*, *after*, and *around* advices, respectively. The last column in the table shows the number of intertype methods. The subject programs contain a variety of characteristics that 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. We provide brief descriptions of each program below, except for the *Kettle* program, which we described in Chapter 2.

Telecom is a simulation of a telephony system that is shipped with the *ajc* AspectJ compiler [66]. This program allows customers to make, accept, merge,

[8]`http://xml-marker.en.softonic.com/`

Table 7.11: Main characteristics of the subject programs

Program	LOC	#Classes	#Aspects	#Tested Classes	#Before advices	#After advices	#Around advices	#ITM
Kettle	125	1	2	1	0	2	2	2
Telecom	928	10	3	5	0	9	0	9
Banking	243	2	2	2	0	0	1	1
CruiseControl	1008	9	3	3	4	7	2	12
All	2304	22	10	11	4	18	5	24

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.

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 [37] 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.

2. *OverdraftProtectionRuleAspect* implements an overdraft coverage policy by allowing a customer to cover overdraft withdrawals from the customer's other accounts.

Xu et al. [76] 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

3. *SpeedControlIntegrator* adds the implementation of a speed controller to a cruise controller.

Table 7.12: Number of test requirements for the AOSV test criteria in the classes of the subject programs.

Subject	Class	oDUA	aDUA	cDUA	asDUA	maDUA	allDUA
Kettle	Kettle	16	10	12	10	6	54
Banking	Account	11	4	18	2	1	36
	Customer	0	0	5	0	0	0
Telecom	Local	2	0	9	5	1	17
	LongDistance	2	0	9	5	1	17
	Customer	0	0	8	8	0	16
	Call	0	0	9	0	0	0
	Timer	0	0	7	0	0	0
Cruise Control	CarSimulator	6	0	127	17	0	150
	CruiseController	45	0	65	6	0	116
	SpeedControl	4	8	49	2	0	63
Total No. of DUAs		86	22	318	55	9	469
No. of Classes with this DUA type		7	3	11	8	4	8

Table 7.12 shows the number of test requirements for the AOSV test criteria in each class in the subject programs. The first and second columns of the table show the programs and the classes names, respectively. Columns 3 through 8 show the

number of DUAs of each of the AOSV DUAs. The last two rows show the number of DUAs of each type in all the classes and the number of classes that have DUAs of the type in each column, respectively. For example, the fourth column in the last two rows shows that there are 318 *cDUAs* that occurred in 11 classes. The reported numbers exclude DUAs that have *defs* and *uses* in exception handling segments and unreachable DUAs. As the table shows, two of the advised classes have all types of DUAs. These are the *Kettle* class of the *Kettle* program and the *Account* class of the *Banking* program. Note that the *maDUAs* are the least frequent in terms of the number of DUAs in the classes. This is because *maDUAs* occur only when there are data dependencies between advices in different aspects, which did not occur frequently in the tested classes.

DUAs of type *aDUA* occurred only in 3 classes, while *oDUAs* occurred in 7 of the 8 advised classes. These are about 4 times more *oDUAs* than there are *aDUAs*, which shows that aspects use data from variables provided by base classes more than they alter their values.

asDUAs occurred in all the advised classes because the aspects in the subject programs needed to introduce state variables in the base classes in order to implement the crosscutting concern. Since these introduced state variables are defined and used only in the aspect that introduced them, we had this high number of *asDUAs*, compared to other types of AOSV DUAs.

cDUAs occurred in all the 11 classes, whether the classes were advised or not. Recall that cDUAs are defined between *defs* and *uses* of the state variable that occur in the base class. Therefore, such DUAs can occur in non-advised classes. In our studies, we tested three non-advised classes, which are *Customer* in the *Banking* program, and *Call* and *Timer* in the *telecom* program. We tested these classes in order to evaluate the cost and effectiveness of the cDUAs in non-advised classes.

Table 7.13: Number of test requirements for the AO *control-flow* test criteria in the classes of the subject programs.

Subject	Class	Blocks	Branches
Kettle	Kettle	21	16
Banking	Account	21	12
	Customer	6	0
Telecom	Local	19	0
	LongDistance	19	0
	Customer	18	0
	Call	18	9
	Timer	5	0
Cruise Control	CarSimulator	52	23
	CruiseController	32	25
	SpeedControl	22	10
Sum No. Test Requirements		233	95
No. Classes		11	6

Table 7.13 shows the number of test requirements for the *AO control-flow* test criteria in each class of 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.

7.7 Mutant Generation

Experimental studies in software testing require the evaluation of the ability of a testing approach to detect known faults in subject programs. However, researchers often have a problem finding subject programs that meet their research requirements and also contain real faults. Even when such programs are available, the number of real faults is often not large enough to allow achieving statistically significant results [7]. Therefore, researchers typically seed faults in the subject programs, either manually or with mutation operators. The latter approach has several advantages. A large number of faulty versions can be generated in a systematic and easily replicatable manner [49]. Moreover, Andrews et al. [7, 8] provided evidence that faults generated with mutation operators are similar to real faults in evaluating test effectiveness, while hand-seeded faults are harder to detect than real faults. Therefore, we chose to seed faults in the subject programs using mutation operators.

7.7.1 Mutant Generation Using Mutation Tools

We used three mutation tools *AjMutator*, *Proteum/AJ*, and μJava. *AjMutator* was developed by Delamare et al. [17]. It implements a subset of the operators for pointcut descriptors proposed by Ferrari et al. [22]. *AjMutator* parses a pointcut descriptor from the aspect source code and performs the mutations. The modified pointcut is then used to generate a mutant. *AjMutator* classifies the mutants according to the set of join points they match compared to the set of join points matched by the original pointcut. If the two sets are equal, the mutant is classified as equivalent. The tool also identifies non-compilable mutants and runs JUnit test cases.

Proteum/AJ [23] implements three more pointcut mutation operators than *AjMutator* from the same set of operators proposed by Ferrari et al. [22]. The tool also implements two advice declaration operators, four advice implementation operators, and 5 intertype declaration operators. *Proteum/AJ* is not yet available for public use. Therefore, we sent the subject programs to the developers of the tool and they generated the mutants for us.

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

Table 7.14 lists the operators from *AjMutator* and *Proteum/AJ* that generated mutants for the subject programs. Operators that apply to constructs not used in the programs are not shown (e.g., operators that mutate annotations). The first and second columns of the table show the operator name and description, respectively. The third and fourth columns specify whether the operator is implemented in *Proteum/AJ* and *AjMutator*, respectively. The fifth column shows the fault category of the generated mutant classified according to the revised AO fault model described in Chapter 4. There are eight pointcut operators (category F1) from both the tools. Note that some operators are implemented in both the tools but in different ways. For example, POPL in *Proteum/AJ* is implemented by adding or removing the parameter list, which is how the operator is defined by Ferrari et al. [22]. However, *AjMutator* only removes the parameter list from the descriptor if the list is specified. In our studies, we used the mutants from both tools. When the operators generate identical mutants, we keep the mutants from *Proteum/AJ*.

To mutate the rest of the code, we used μJava [41]. μJava is a widely used tool for mutating Java programs. μJava 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. μJava relies on the tester to manually identify equivalent mutants.

Table 7.14: Mutation operators implemented by *AjMutator* and *Proteum/AJ* that generated mutants in the subject programs

Operator	Description	*Proteum/AJ*	*AjMutator*	Fault Category
PCCE	Replace call/execution/ initialization/preinitialization	yes	only call/execution	F1
PCLO	Change logical operators in pointcut descriptor	yes	yes	F1
PCTT	Replace *this/target*	yes	yes	F1
POEC	change exception throwing	yes	yes	F1
POAC	change after[returning |throwing]	yes	no	F1
POPL	Change the parameter list	yes	Delete only	F1
PSWR	Remove wildcard	yes	yes	F1
PWIW	adds wildcard	yes	yes	F1
ABAR	Changes advice kind	yes	No	F2
ABPR	change pointcut advice binding	yes	No	F2
DAPC	swap aspects precedence	yes	No	F2
DAPO	Remove declare precedence	yes	No	F2
ABHA	remove advice	yes	No	F3
APER	change guard condition of *around*	yes	No	F3
APSR	remove *proceed* statement	yes	No	F3

The current version of μJava does not support AspectJ. That is, the tool cannot seed faults in the aspects, and to the best of our knowledge, except for the 4 operators implemented in *Proteum/AJ*, there is no tool that can directly seed faults into the advice, aspect methods, and introduced methods. Therefore, we used an indirect approach to seed faults in the aspects using μJava. We generate a class from the aspect bytecode with the help of a decompilation tool (Jad[9]). We mutate the decompiled class with μJava. We choose the mutated line that resides in the

[9]`http://www.varaneckas.com/jad`

advices, intertype methods, or aspect method and copy it back to the aspect. We repeat this step to generate all the mutants, where each mutant contains one fault. Note that since the mutated class is produced from the bytecode, it contains some extra methods that are not present in the source code of the aspect. These methods include those that the aspect inherits from the AspectJ base class (e.g., methods *aspectOf*, *hasAspect*), and methods that correspond to some AspectJ constructs. For example, the *declare precedence* statement is compiled into a method in the bytecode. We discard mutants that reside in these extra methods.

We use μJava directly on the base classes to generate their mutants. μJava only produces mutants that compile. However, μJava compiles mutants with a Java compiler by default, not an AspectJ compiler. Therefore, we compile all mutants generated by μJava 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 our 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. This concept only applies for a subset of the pointcut mutants because there are some pointcut mutants match different set of join points and 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, we manually inspected all the mutants and identified the equivalent ones. We consider a mutant to be equivalent if the fault seeded by the mutation operator did not propagate to produce a different output from the original program.

7.7.2 Generation of Higher Order Mutants

In order to answer research question 3, we need to seed faults of the types described in the revised AO fault model as long as the programs contain constructs that can be mutated to generate faults of each type. By classifying the mutants generated by the tools, we found 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 operators) perform one change in the code, which does not guarantee that F1-2 mutants can be generated, especially when the pointcuts are bound to objects of specified types, like the pointcuts in the subject programs. This observation leads us to consider generating mutants by applying two operators. The resulting mutant is called a higher order mutant (HOM) [18, 34]. Mutants can be classified into first order mutants (FOMs), which are created by applying a mutation operator once, and higher order mutants (HOMs) of degree k, which are generated by applying mutation operators more k times [18, 34]. A HOM of degree two (or a second order mutant) is constructed by applying two mutation operators.

Our goal is to generate faults of the types that the FOMs missed. Given two mutants m_1 and m_2, where m_1 produces a fault of type f_1 and m_2 produces a fault of type f_2, the following four conditions must hold before we generate the HOMs:

1. Condition 1: $S_{m_1} \neq S_{m_2}$, where S_{m_1} and S_{m_2} are the sets of join points matched by mutants m_1 and m_2, respectively.

2. Condition 2: $S_{orig} \neq \emptyset$, where S_{orig} is the set of join points matched by the original pointcut.

3. Condition 3: $S_{m_1} \neq S_{org}$ and $S_{m_1} \neq S_{org}$, i.e., the mutants m_1 and m_2 are not equivalent to the original program.

4. Condition 4: m_1 and m_2 mutate the same pointcut descriptor.

Given conditions 1 through 4, we take a mutant corresponding to fault type F1-3 and one mutant of type F1-4 and combine them to get a mutant of type F1-2.

In the *Kettle* program, we have 2 mutants of type F1-3 and 4 mutants of type F1-4. We could generate 4 HOMs of type F1-2 because every mutant of type F1-3 mutates the same pointcut in two of F1-4 mutants. In the *telecom* program, we have 5 mutants of type F1-3, and 18 mutants of type F1-4. We were able to generate 6 HOMs of type F1-2 because 3 of the 5 F1-3 mutants mutate the same pointcut of 2 mutants of type F1-4. For the rest of the classes that do not have mutants of type F1-2, we could not generate HOM for them since conditions 1 through 4 did not hold.

Figure 7.1 shows a higher order mutant generated for the *telecom* program. Mutant 1 generated by operator PWIW is of type F1-3 because the mutant matches a superset of the intended join points. Mutant 2 generated by operator POAC is of type F1-4 because it matches a subset of the intended join points (i.e., it misses the join point when the method *Timer.start* throws an exception). The generated HOM is of type F1-2 because it matches a subset of the intended join points (i.e., it misses the join point after an exception is thrown), and some unintended join points (i.e., after returning from other methods of class *Timer*).

7.7.3 Classification of Mutants

In this section, we show the distribution of the generated mutants on the types of faults of the revised AO fault model presented in Chapter 4. Table 7.15 shows the distribution of the mutants generated from the aspect pointcuts (category 1).

```
//original
after(Timer t): target(t) && call(* Timer.start())

//Mutant 1
after(Timer t): target(t) && call(* Timer.*())

//Mutant 2
after(Timer t) returning: target(t) && call(* Timer.start())

//HOM
after(Timer t) returning: target(t) && call(* Timer.*())
```

Figure 7.1: HOM for *telecom*

The first and second columns of the table show the program and class names, respectively. Columns 3 through 7 show the number of mutants in each type for each of the classes. The last column shows the total number of pointcut mutants generated for each class, while the last row shows the total of the number of mutants in each type for all the classes. In classes *Kettle*, *Local*, and *LongDistnace*, mutants of all fault types are present. Class *Account* has only one pointcut mutant. The reason is that the pointcut in aspect *OverdraftProtectionRuleAspect*, which matches one method in class *Account*, is written to exactly match the signature of the advised method. Therefore, all the mutants that the tools generated for the pointcut either did not compile or were equivalent, except for one mutant generated by operator POEC. We also could not generate HOMs for the *Account* class since we need at least two non-equivalent FOMs.

Table 7.16 shows the classification of the mutants generated from aspect declarations (category F2). We have faults of types F2-5, F2-7, and F2-8 in the programs. We could not seed faults of types F2-1 and F2-2 because Proteum/AJ does not have operators than can seed such types of faults. Other fault types (i.e., F2-4 and F2-5), do not occur in the subject programs. For class *Account* in the *Banking* program, and the 3 classes of the *CruiseControl* program, we could not

Table 7.15: Classification of mutants generated from aspect pointcuts

Subject	Class	F1-1	F1-2	F1-3	F1-4	F1-5	All
Kettle	Kettle	0	4	2	4	6	16
Banking	Account	0	0	0	0	1	1
Telecom	Local	16	6	5	14	20	61
	LongDistance	16	6	5	14	20	61
	Customer	6	0	0	4	8	18
Cruise Control	CarSimulator	0	2	6	10	5	23
	CruiseController	2	0	3	8	1	14
	SpeedControl	0	0	0	3	2	5
All		40	18	21	57	63	199

Table 7.16: Classification of mutants generated from aspect declarations

Program	Class	F2-5	F2-7	F2-8	All
Kettle	Kettle	2	2	2	6
Banking	Account	0	0	0	0
Telecom	Local	2	5	13	20
	LongDistance	2	5	13	20
	Customer	2	0	5	7
Cruise Control	CarSimulator	0	0	20	20
	CruiseController	0	8	19	27
	SpeedControl	0	0	0	0
All		8	20	72	100

Table 7.17: Classification of mutants generated from aspect implementation

Program	Class	F3-1	F3-2	F2-3	All
Kettle	Kettle	27	38	30	95
Banking	Account	10	28	34	72
Telecom	Local	0	8	22	30
	LongDistance	0	8	22	30
	Customer	0	20	4	24
Cruise Control	CarSimulator	3	8	6	17
	CruiseController	16	17	34	67
	SpeedControl	0	17	29	46
All		56	144	181	381

seed faults of type F2-5 since there are no precedence rules for the aspects in these programs. For class *Account*, we could not generate any fault of type F2 because type F2-7 requires having *before* and *after* advices while the program only has an *around* advice. Fault type F2-8 requires having more than one advice in the aspect, which is not the case in the *Banking* program.

Table 7.18: Classification of mutants generated from class implementation

Program	Class	F4-1	F4-2	F4-3	F4-4	All
Kettle	Kettle	31	5	3	6	45
Banking	Account	33	0	3	6	42
	Customer	2	0	3	0	5
Telecom	Local	8	0	1	5	14
	LongDistance	8	0	1	5	14
	Customer	17	0	6	12	35
	Call	6	0	4	12	22
	Timer	14	0	2	7	23
Cruise Control	CarSimulator	106	0	17	128	251
	CruiseController	76	0	10	72	158
	SpeedControl	84	0	20	84	188
All		385	5	70	337	797

Table 7.19: Summary of the generated mutants for the subject programs

Program	Class	F1	F2	F3	F4	All
Kettle	Kettle	16	6	95	45	162
Banking	Account	1	0	72	46	119
	Customer	0	0	0	5	5
Telecom	Local	61	20	30	14	125
	LongDistance	61	20	30	14	125
	Customer	18	7	24	35	84
	Call	0	0	0	22	22
	Timer	0	0	0	23	23
Cruise Control	CarSimulator	23	20	17	251	311
	CruiseController	14	27	67	158	266
	SpeedControl	5	0	46	188	239
All		199	100	381	801	1481

Table 7.17 shows the classification of the mutants generated from aspect implementations (category F3). The table combines the mutants generated by *Proteum/AJ* and μJava. Classes of the *telecom* program and the *SpeedControl* class of the *CruiseControl* program do not have faults of type F3-1 since these classes are not advised by *around* advices.

Table 7.18 shows the classification of the mutants generated from Java classes (category F4). Note that only class *Kettle* has faults of type F4-2. This is because only the *Kettle* program contains advices that receive arguments from the base classes.

Table 7.19 summarizes the number of mutants of all types generated for the classes in the subject programs. Note that non-advised classes only have faults in category F4 since these classes do not directly interact with the aspects.

7.8 Test Suite Generation

We used RANDOOP [55], which generates JUnit test cases for Java programs. RANDOOP generates new test cases by randomly selecting a method to call and finding arguments from among previously found inputs. Since RANDOOP is not designed for AspectJ, we performed the following steps to use RANDOOP for AspectJ programs:

- We rewrote the aspects in the subject programs 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.

- We generate test cases only for the classes in the subject programs (i.e., not for the aspects). RANDOOP allows the tester to specify which classes to test and which methods can be called by the test cases. Using this feature, we can avoid having calls to aspect methods and advices directly from the test cases.

For each subject program, we generated a pool of test cases, starting from 3000 test cases. In order to ensure that the test cases in the pool can satisfy the test criteria, we measured the coverage the AOSV test criteria obtained by the test cases in the pool. If the criteria were not fully covered, we generated a another pool with 1000 more test cases (i.e., 4000 test cases in all). We stopped when the pool contains at least one test case that covers each of requirements of the test criteria. The sizes of the pools for each subject program are given in Table 7.20.

Table 7.20: Size of the test pool for each of the subject programs

Subject Program	Test Pool Size
Kettle	3,000
Banking	7,000
Telecom	6,000
CruiseControl	14,000

We provided RANDOOP with a set of input values suitable for the subject programs. For example, for the *telecom* program, we provided RANDOOP with a set of customer names to chose from, and values for call durations that must be used. We also used an option of RANDOOP called the observers option, which is not yet available for public use, 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 [43]. However, using the observers option, we can specify observer methods that help generate stronger assertions that are application-specific. RANDOOP considers a method as an observer method if all of the following hold: (1) the method has no parameters, (2) the method is public and non-static, (3) the method returns values of primitive types or Strings, and (4) the name of the method is `size`, `count`, `length`, `toString`, or begins with `get` or `is`.

Following the approach of generating test suites described in Section 7.3, we generated 30 test suites that satisfy each test criterion in each of the advised class. The use of 30 test suites allows analyzing the results for cost and effectiveness at a significant level (i.e., $p < 0.05$).

7.9 Data Analysis

Using the SPSS[10] package, we performed a repeated measures ANOVA analysis on the data collected for the cost and effectiveness of the test criteria. For each of the classes, we compared each pair of test criteria that apply to the class. Each of the repeated measures ANOVA tests was performed using 30 observations. We used repeated measures ANOVA because the measurement of the independent variable is repeated. We used repeated measures ANOVA to test the hypotheses in all of the 5 groups. Normality is tested using Kolmogorov-Smirnov test the sample size is small (less than 50 observations). The data in our experiments passed the normality We used the Mauchly's test which tests the hypothesis that the variances of the differences between conditions are equal (i.e., test the assumption of sphericity). When data violated the sphericity assumption, we applied the suitable correction to correct the degrees of freedom.

[10]`http://www.spss.com`

Chapter 8

Empirical Study Results

We present the results of the empirical study in this chapter. Sections 8.1 through 8.4 address each of the four research questions in turn. Section 8.5 discusses the threats to the validity of the study.

8.1 Comparing the Cost of the Test Criteria

We address the first research question in this section. Our answer is based on the results of testing the hypotheses in groups A through D. Table 8.1 shows statistics about the sizes of the test suites that satisfy the criteria. For each criterion shown in column 1, and for each class in row 1, Table 8.1 shows: (1) the means of the number of test cases in the test suites that satisfy a test criterion (top line), (2) the standard deviations of the number of test cases in the test suites that satisfy a test criterion (second line), (3) the minimum and maximum of the number of test cases in the test suites that satisfy a test criterion (third line), and (4) the number of test requirements for a test criterion in the class (fourth line). For any class, the symbol (**) on the first line indicates that the average size of the test suite that satisfies the criterion is the smallest, while the symbol (*) indicates that it is the largest. For example, the entry in the second row and second column shows that in the *Kettle* class (1) the mean number of test cases in the test suites that satisfy

Table 8.1: Statistics for the number of test cases in the test suites that satisfy the test criteria

Criteria	Kettle	Account	Customer (Banking)	Local	LongDistance	Customer (Telecom)	Call	Timer	CarSimulator	CruiseController	SpeedControl
all-uses$_a$	7.9 0.99 5-10 10	2.3 0.5 2-3 4									6.4 0.77 4-7 8
all-uses$_o$	6.6 1.16 5-9 16	5.5 1.04 4-7 11		1** 0 1-1 2	1** 0 1-1 2				4.8** 0.74 3-6 6	26.2 1.85 23-30 45	2.5 0.57 1-3 4
all-uses$_c$	8.1 1.48 6-12 12	6.1 1.28 4-8 18	2* 0.67 1-3 5	7 0.93 5-8 9	4.7 1 2-7 9	2.1 0.68 1-4 8	5.4* 0.93 3-7 9	3.4* 0.97 2-5 7	51.4 6.09 37-61 127	26.4 1.83 23-29 65	15.7 2.05 11-21 49
all-uses$_{as}$	6.8 1.1 4-9 10	1.8 0.38 1-2 2		2.8 0.41 2-3 5	1.9 0.52 1-3 5	4.3 1.03 2-6 8			11.7 1.62 9-16 17	4** 0.89 2-6 6	2** 0 2-2 2
all-uses$_{ma}$	4.8 1.04 3-6 6	1** 0 1-1 1		1** 0 1-1 1	1** 0 1-1 1						
all-uses$_s$	15.8* 2.7 10-21 54	11.4* 1.3 9-15 36		8.3* 1.61 4-11 17	4.8* 0.97 3-7 17	4.9* 1.09 3-7 18			54.7* 4.85 44-63 150	34.3* 2.16 30-38 116	21.2* 2.29 17-27 63
AO blocks	3.7** 1.72 1-6 21	2 0 2-2 21	1** 0 1-1 6	2.1 0.89 1-3 19	2.2 0.79 1-3 19	1.6** 0.5 1-2 18	2.1** 0.35 2-3 18	3.2** 1.22 1-4 5	5.1 0.94 3-8 52	4.8 1.18 3-8 32	4.3 0.91 3-6 22
AO branches	4.7 1.73 2-7 16	3 0.85 2-4 12					4.1 1.52 2-6 9		5.8 1.55 3-8 23	5.2 1.47 3-8 25	5.2 0.95 3-7 10

the *all-uses$_a$* criterion is 7.9, (2) the standard deviation is 0.99, (3) the number of test cases in the test suites that satisfy *all-uses$_a$* ranges from one to three test cases, and (4) there are 10 aDUAs in the class. Column 2 shows that for the *Kettle* class, the number of test cases that satisfies *all-uses*$_s$ is the largest while the test suite that satisfies *AO blocks* is the smallest. Empty cells in the table indicate that the criterion does not have test requirements in the class.

All the test suites that satisfy the *all-uses*$_{ma}$ criterion in *Account*, *Local*, and *LongDistance* classes contain only one test case. This is because these classes contain only one maDUA. Similarly, the test suites that satisfy the *all-uses*$_o$ criterion in *Local* and *LongDistance* classes also contain only one test case. This is because the 2 oDUAs in the classes are covered by executing the same path (i.e., they are always covered together).

The maximum number of test cases in a test suite does not exceed the number of test requirements for a criterion. This is because of the iterative procedure we performed to generate the test suites (i.e., adding a test case only if it improves coverage).

Columns 5 and 6 show that the average number of test cases in the test suites that satisfy each of the test criteria in the *Local* class is higher than in the *LongDistance* class, even though both classes implement the same super class (class *Connection*), and differ only in the constructors. RANDOOP test cases exercised more *LongDistance* calls than *Local* calls because RANDOOP tends to randomly generate different values for area codes. This results in more calls to different area codes than calls to the same area code.

Table 8.1 shows that in all the advised classes, the average number of test cases in the test suites that satisfy *all-uses*$_s$ is higher than c_1 for all other test criteria. In the non-advised classes, the average number of test cases in the test suites that satisfy *all-uses*$_c$ is higher than c_1 for the AO control-flow criteria. However, the smallest c_1 in each class is not consistent for one of the test criteria.

8.1.1 Comparing the Cost of the Test Criteria Using Size Metric c_1

We present the results from testing the hypotheses in group A, in which the cost of the test criteria is compared using the number of test cases in a test suite that satisfies a criterion. Section 8.1.1.1 presents the results of testing hypotheses $H0_{A1}$ through $H0_{A12}$, in which c_1 of the AOSV test criteria is compared with c_1 of the AO control-flow criteria. Section 8.1.1.2 presents the results of testing hypotheses $H0_{A13}$ through $H0_{A27}$, in which the AOSV test criteria are compared with each other using c_1.

8.1.1.1 Comparing the Cost of the AOSV Test Criteria with the Cost of the AO Control-Flow Criteria Using Size Metric c_1

Table 8.2 shows the results of testing hypotheses $H0_{A1}$ through $H0_{A12}$. Column 1 shows which criteria are being compared and row 1 shows the class names. Columns 2 through 12 show the results of testing the hypotheses in each class. For each hypothesis, we report the difference between the means of number of test cases in the test suites sizes that satisfy the compared criteria and the p-value of the test. We reported the p-value using the symbol $**$ when $p < 0.01$, the symbol $*$ when $p < 0.05$, "ns" when there is no significant difference to reject the null hypothesis, and "na" when testing the hypothesis is not applicable because repeated measures ANOVA requires that the variances of the compared observations to be not equal to zero, while all the observations for the compared criteria have the same value. For example, the entry in the second row and second column shows that in the *Kettle* class (1) there is a significant difference between the number of test cases in the test suites that satisfy *all-uses*$_a$ and the number of test cases in the test suites that satisfy *AO blocks*, (2) the number of test cases in the test suites that satisfy *all-uses*$_a$ is significantly higher than the number of test cases in the test

Table 8.2: Hypotheses test results for comparing the cost of the AOSV test criteria with the cost of the AO control-flow criteria using the number of test cases in a test suite that satisfies a test criterion

Hypotheses	Kettle	Account	Customer (Banking)	Local	LongDistance	Customer (Telecom)	Call	Timer	CarSimulator	CruiseController	SpeedControl
$H0_{A1}$: all-uses$_a$-blocks	4.2 **	0.8 **									2.1 **
$H0_{A2}$: all-uses$_a$-branches	3.2 **	-0.4 ns									1.2 **
$H0_{A3}$: all-uses$_o$-blocks	2.9 **	3.5 **		-1.1 **	-1.2 **				-0.3 ns	21.4 **	-1.8 **
$H0_{A4}$: all-uses$_o$-branches	2.0 **	2.5 **							-1.0 **	21.0 **	-2.7 **
$H0_{A5}$: all-uses$_c$-blocks	4.4 **	4.1 **	0.9 **	4.9 **	2.2 **	0.5 **	3.3 **	0.2 ns	46.3 **	21.5 **	11.5 **
$H0_{A6}$: all-uses$_c$-branches	3.5 **	3.2 **					1.3 **		45.6 **	21.2 **	10.6 **
$H0_{A7}$: all-uses$_{as}$-blocks	3.0 **	-0.2 ns		.7 **	-.2 ns	2.7 **			6.6 **	0.9 ns	-2.3 **
$H0_{A8}$: all-uses$_{as}$-branches	2.1 **	-1.1 **							5.9 **	-1.2 *	-3.2 **
$H0_{A9}$: all-uses$_{ma}$-blocks	1.0 ns	-1.0 na		-1.1 **	-1.2 **						
$H0_{A10}$: all-uses$_{ma}$-branches	0.1 ns	-2.0 **									
$H0_{A11}$: all-uses$_s$-blocks	12.1 **	9.4 **		6.2 **	2.6 **	3.3 **			49.6 **	29.4 **	16.9 **
$H0_{A12}$: all-uses$_s$-branches	11.2 **	8.4 **							49.9 **	29.1 **	16.0 **

suites that satisfy *AO block* ($p < 0.01$), and (3) the difference between the mean of the number of test cases in the test suites that satisfy *all-uses*$_a$ and the means of the number of test cases in the test suites that satisfy *AO blocks* is 4.17. Empty cells in the table indicate that the hypothesis cannot be tested in the class because the class does not contain test requirements for the two criteria compared in the hypothesis.

The results from testing hypotheses $H0_{A1}$ through $H0_{A12}$ support the alternative hypotheses and show that there is a significant difference between c_1 for each of the AOSV test criteria and each of the AO control-flow criteria in 58 out of 66 hypothesis tests performed in the classes. The results from testing hypotheses $H0_{A1}$, $H0_{A2}$, $H0_{A5}$, $H0_{A6}$, $H0_{A11}$, and $H0_{A12}$ show also that c_1 for each of the *all-uses*$_a$, *all-uses*$_c$, and *all-uses*$_s$ criteria, is significantly higher than c_1 for each of the AO control-flow criteria. For the other AOSV test criteria (i.e., *all-uses*$_o$, *all-uses*$_{as}$, and *all-uses*$_{ma}$), the difference in c_1 is inconsistent over all the classes. In the classes that contain few DUAs and DUAs that can be covered by any test case, c_1 for the *AO control-flow* is higher. For example, the *Local* and *LongDistance* classes each contain only two oDUAs that are covered by one path, and therefore, one test case is enough to satisfy *all-uses*$_o$. However, each of these two classes contain 19 blocks, 11 of which are in the aspects. Therefore, c_1 for *AO blocks* in *Local* and *LongDistance* is significantly higher than c_1 for *all-uses*$_o$. On the other hand, the *CruiseController* class contains 45 oDUAs, of which 17 require paths that do not cover any other oDUAs (i.e., a path covers only one oDUA). Therefore, c_1 for the *all-uses*$_o$ criterion in the *CruiseController* class is significantly higher than c_1 for *AO blocks* and *AO branches*.

The reason why c_1 for *all-uses*$_a$ is significantly higher than c_1 for each of the AO control-flow criteria is that in the tested classes, the paths that cover the aDUAs always require two consecutive calls to the advised methods that contain the *def* and the method that contain the *use*. This is because all the *defs* are in either an *after* or an *around* advice and in order to reach a *use* (even if the *use* is in the same method), another call to the method that contains the *use* is needed. The other types of AOSV DUAs also require consecutive calls for some of the DUAs, but not for all of them (i.e., they have DUAs than can be satisfied by a path through one method).

There are two reasons why c_1 for the *all-uses*$_c$ criterion is significantly higher than c_1 for each of the AO control-flow criteria. First, the classes have a higher percentage of cDUAs than other types of AOSV DUAs as can be seen in Table 8.3. The table shows each class in column 1, the number of cDUAs in column 2, the total number of AOSV DUAs in column 3, and the percentage of cDUAs from the total number of AOSV DUAs in column 4. Except for the *Kettle* class, the cDUAs contribute at least 50% of all ASOV DUAs. In the *CarSimulator* class, the cDUAs contribute about 86% of all the AOSV DUAs. The second reason is that some cDUAs require covering paths that are easily executed by the test cases (e.g., paths between *setter* and *getter* methods), while some cDUAs require covering paths that are not frequently executed by the test cases.

We encountered four types of cDUAs in the tested classes, which are as follows:

- The *def* and *use* are in the same method.
- The *def* and *use* are in non-advised methods.
- Either the *def* or the *use* but not both are in advised methods.
- Both the *def* and *use* are in advised methods.

When a class has more of the last two types of cDUAs, then covering *all-uses*$_c$ requires more paths and therefore, more test cases. For example, in the *Kettle* class, in order to cover the cDUA <*WaterAmount, K26, K26*>, a test case needs to make two consecutive calls to the method *addWater* so that the *proceed* statement gets executed in both calls. Covering the cDUA <*Size, K15, K36*> requires only calling the *getSize* method after the constructor.

The *Timer*, which is not advised by any aspect, is the only class in which there is no significant difference between c_1 for *all-uses*$_c$ and c_1 for *AO blocks*.

This is because objects of type *Timer* are only referenced by classes *Local* and *LongDistance*, and a test case does not contain calls to all methods of classes that are referenced by other classes. Thus, the number of test cases required to cover the blocks in the class is increased.

Table 8.3: Percentage of cDUAs in the classes

Class	# cDUAs	# AOSV DUAs	Percentage of cDUAs
Kettle	12	54	22
Account	18	36	50
Local	9	17	53
LongDistance	9	17	53
Customer	8	16	50
CarSimulator	127	150	85
CruiseController	65	116	56
SpeedControl	49	63	78
All	297	469	63

The results from testing hypotheses $H0_{A9}$ and $H0_{A10}$ show that c_1 for *all-uses*$_{ma}$ is significantly lower than c_1 for each of the AO control-criteria, except in the *Kettle* class, in which the difference is not significant. This is because all the classes, except the *Kettle* class, contain only one maDUA and, therefore, the test suites that satisfy *all-uses*$_{ma}$ contain only one test case. The *Kettle* class contains 6 maDUAs which require at least 3 different paths to be covered, and therefore, increases c_1 for *all-uses*$_{ma}$.

Testing hypothesis $H0_{A7}$ in the *Account* class is not applicable. c_1 for *all-uses*$_{ma}$ and c_1 for *AO blocks* have the same value for all the 30 observations (1 and 2, respectively). However, the difference between the means of c_1 for *all-uses*$_{ma}$ and the mean of c_1 for *AO blocks* shows that c_1 for *all-uses*$_{ma}$ is higher, which supports the alternative hypothesis.

Finally, the results from testing hypotheses $H0_{A11}$ and $H0_{A12}$ show that c_1 for *all-uses*$_s$ is significantly higher than c_1 for each for the *AO control-flow* criteria in all the classes. These results match our expectations.

8.1.1.2 Comparing the Cost of the AOSV Test Criteria with Each Other Using Size Metric c_1

Table 8.4 shows the results from testing hypotheses $H0_{A13}$ through $H0_{A27}$. The table is organized in the same way as Table 8.2 but does not show the non-advised classes since they do not contain any of the AOSV criteria other than *all-uses*$_c$. The results from testing hypotheses $H0_{A13}$ through $H0_{A27}$ support the alternative hypotheses and show that there is a significant difference between c_1 for the AOSV test criteria compared to each other in 66 out of 73 hypothesis tests performed in the classes.

The results from testing hypotheses $H0_{A13}$ through $H0_{A17}$ show that c_1 for the *all-uses*$_s$ criterion is significantly higher than c_1 for the other AOSV criteria in all the classes, except for *all-uses*$_c$ in the *LongDistance* class, and *all-uses*$_{as}$ in the *Customer* class, in which it is higher but not significantly. These results match our expectations since *all-uses*$_s$ subsumes all other AOSV criteria. In the *LongDistance* class, the test suites that covered *all-uses*$_c$ also covered most of the other AOSV DUAs. Therefore, these test suites were close in terms of c_1 to the ones that cover *all-uses*$_s$. In the *Customer* class, which contains only DUAs of the types cDUA and asDUA, the test cases that covered the asDUAs also covered most of the cDUAs, making c_1 for *all-uses*$_{as}$ close to c_1 of *all-uses*$_s$.

The results from testing hypotheses $H0_{A19}$, $H0_{A22}$, $H0_{A25}$, and $H0_{A26}$ show that:

- $H0_{A19}$: c_1 for *all-uses*$_c$ is significantly higher than c_1 for *all-uses*$_a$ in 2 out of 3 classes in which these criteria are compared. In the *Kettle* class, it is higher but not significantly.

Table 8.4: Hypotheses test results for comparing the cost of the AOSV criteria with each other using the number of test cases in a test suite that satisfies a test criterion

Hypotheses	Kettle	Account	Local	LongDistance	Customer (Telecom)	CarSimulator	CruiseController	SpeedControl
$H0_{A13}$: all-uses$_s$-all-uses$_a$	7.9 **	8.8 **						14.8 **
$H0_{A14}$: all-uses$_s$-all-uses$_o$	9.2 **	5.9 **	7.3 **	3.8 **		49.9 **	8.1 **	18.7 **
$H0_{A15}$: all-uses$_s$-all-uses$_c$	7.7 **	5.3 **	1.3 *	0.4 ns	2.8 **	3.3 **	7.9 **	5.4 **
$H0_{A16}$: all-uses$_s$-all-uses$_{as}$	9.1 **	9.6 **	5.5 **	2.8 **	0.3 ns	43.0 **	30.3 **	19.2 **
$H0_{A17}$: all-uses$_s$-all-uses$_{ma}$	11.1 **	10.4 **	7.3 **	3.8 **				
$H0_{A18}$: all-uses$_a$-all-uses$_o$	1.3 **	-2.9 **						3.9 **
$H0_{A19}$: all-uses$_a$-all-uses$_c$	-0.2 ns	-3.6 **						-9.3 **
$H0_{A20}$: all-uses$_a$-all-uses$_{as}$	1.1 *	0.7 **						4.4 **
$H0_{A21}$: all-uses$_a$-all-uses$_{ma}$	3.1 **	1.6 **						
$H0_{A22}$: all-uses$_o$-all-uses$_c$	-1.5 *	-0.6 ns	-6.0 **	-3.4 **		-46.6 **	-0.2 ns	-13.3 **
$H0_{A23}$: all-uses$_o$-all-uses$_{as}$	-0.1 ns	3.7 ns	-1.8 **	-0.9 **		-6.9 **	22.2 **	0.5 **
$H0_{A24}$: all-uses$_o$-all-uses$_{ma}$	1.9 **	4.5 **	0 na	0 na				
$H0_{A25}$: all-uses$_c$-all-uses$_{as}$	1.4 *	4.3 **	4.22 **	2.4 **	-2.2 **	39.7 **	22.4 **	13.7 **
$H0_{A26}$: all-uses$_c$-all-uses$_{ma}$	3.4 **	5.1 **	6.0 **	3.4 **				
$H0_{A27}$: all-uses$_{as}$-all-uses$_{ma}$	2.0 **	0.8 **	1.8 **	0.9 **				

- $H0_{A22}$: c_1 for *all-uses$_c$* is significantly higher than c_1 for *all-uses$_o$* in 5 out of 7 classes in which these criteria are compared. In the *Account* and the *CruiseController* classes, it is higher but not significantly.

- $H0_{A25}$: c_1 for *all-uses$_c$* is significantly higher than c_1 for *all-uses$_{as}$* in all the classes in which these criteria are compared, except in the *Customer* class, in which it is significantly lower.

- $H0_{A26}$: c_1 for *all-uses$_c$* is significantly higher than c_1 for *all-uses$_{ma}$* in all the classes in which these criteria are compared.

As explained in the previous section, the classes have a high percentage of cDUAs compared with the other types of AOSV DUAs. Therefore, c_1 for *all-uses$_c$* is significantly higher than c_1 for all the other AOSV criteria, except when the cDUAs in the class are of type that is easy to cover by any test case. In the *Account* class, 50% of the cDUAs are either in the same method or between non-advised methods. In the *CruiseController* class, there is a high percentage of oDUAs compared with the cDUAs (45 oDUAs, and 65 cDUAs). Therefore, the difference between c_1 for *all-uses$_c$* and c_1 for *all-uses$_o$* is not significant. The *Kettle* class contains fewer cDUAs than other classes but more aDUAs, and thus, the difference between c_1 for *all-uses$_c$* and c_1 for *all-uses$_a$* is not significant. In the *Customer* class, the cDUAs are covered by any test case because most of them require executing paths between *setter* and *getter* non-advised methods.

The results from testing hypotheses $H0_{A20}$ and $H0_{A21}$ show that c_1 for *all-uses$_a$* is significantly higher than c_1 for *all-uses$_{as}$* and *all-uses$_{ma}$*. As explained earlier, aDUAs require more paths to be covered than other AOSV DUAs. The results from testing hypotheses $H0_{A24}$ and $H0_{A27}$ show that c_1 for *all-uses$_o$* and c_1 for *all-uses$_{as}$* are significantly higher than c_1 for *all-uses$_{ma}$*. This is because 3 out of 4 of the tested classes contain only one maDUA.

8.1.2 Comparing the Cost of the Test Criteria Using Size Metric c_2

We present the results from testing the hypotheses in group B, in which the cost of the test criteria is compared using the number of test requirements that can be covered by a test case. Section 8.1.2.1 presents the results from testing hypotheses $H0_{B1}$ through $H0_{B12}$, in which c_2 for the AOSV test criteria is compared with c_2 for the AO control-flow criteria. Section 8.1.2.2 presents the results from testing hypotheses $H0_{B13}$ through $H0_{B27}$, in which the AOSV test criteria are compared with each other using c_2.

8.1.2.1 Comparing the Cost of the AOSV Test Criteria with the Cost of the AO Control-Flow Criteria Using Size Metric c_2

Table 8.5 shows the results from testing hypotheses $H0_{B1}$ through $H0_{B12}$. Column 1 shows which criteria are being compared while row 1 shows the class names. Columns 2 through 12 show the results of testing the hypotheses in each class. For each hypothesis, we report the difference between the means of c_2 for the compared test criteria rounded to the nearest two decimal digits, and the p-value of the test. We reported the p-value using the symbol $**$ when $p < 0.01$, the symbol $*$ when $p < 0.05$, "ns" when there is no significant difference to reject the null hypothesis, and "na" when testing the hypothesis is not applicable. Empty cells in the table indicate that the hypothesis cannot be tested in the class because the class does not contain test requirements compared in the hypothesis.

The results from testing hypotheses $H0_{B1}$ through $H0_{B12}$ support the alternative hypotheses and show that c_2 for each of the AOSV test criteria is significantly higher than c_2 for each of the *AO control-flow* criteria in 62 out of 66 hypothesis tests performed in the classes. These result confirm our expectations that a test case can cover more blocks or branches than it can cover any of the AOSV DUAs

Table 8.5: Hypotheses test results for comparing the cost of the AOSV criteria with the cost of the AO control-flow criteria using size metric c_2

Hypotheses	Kettle	Account	Customer (Banking)	Local	LongDistance	Customer (Telecom)	Call	Timer	CarSimulator	CruiseController	SpeedControl
$H0_{B1}$: all-uses$_a$-blocks	.61 **	.55 **									.61 **
$H0_{B2}$: all-uses$_a$-branches	.50 **	.39 **									.28 **
$H0_{B3}$: all-uses$_o$-blocks	.24 **	.41 **		.39 **	.39 **				.70 **	.43 **	.42 **
$H0_{B4}$: all-uses$_o$-branches	.12 **	.25 **							.54 **	.37 **	.10 ns
$H0_{B5}$: all-uses$_c$-blocks	.50 **	.25 **	.23 **	.67 **	.37 **	.18 **	.48 **	.15 **	.31 **	.26 **	.13 **
$H0_{B6}$: all-uses$_c$-branches	.39 **	.09 **					.14 **		.15 **	.20 **	-.20 **
$H0_{B7}$: all-uses$_{as}$-blocks	.50 **	.82 **		.36 **	.27 **	.45 **			.59 **	.51 **	.81 **
$H0_{B8}$: all-uses$_{as}$-branches	.39 **	.67 **							.44 **	.45 **	.48 **
$H0_{B9}$: all-uses$_{ma}$-blocks	.62 **	.91 na		.89 **	.89 **						
$H0_{B10}$: all-uses$_{ma}$-branches	.50 **	.75 **									
$H0_{B11}$: all-uses$_s$-blocks	.12 **	.22 **		.38 **	.17 **	.22 **			.27 **	.14 **	.14 **
$H0_{B12}$: all-uses$_s$-branches	0 ns	.07 **							.11 **	.09 **	-.18 **

in the advised classes. This is because a path in a test case that covers any of the AOSV DUA 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 *defs* and *uses* in different methods or advices. We discuss below the 4 tests which do not confirm the alternative hypotheses.

The *SpeedControl* class contains branches that require the execution of a high number of different paths in order to be covered. This can be observed from the results from testing hypotheses $H0_{B4}$, $H0_{B6}$, and $H0_{B12}$, which show that (1) c_2 for *AO branches* is not significantly different from c_2 for *all-uses*$_o$, and (2) c_2 for *AO branches* is significantly higher than c_2 for *all-uses*$_c$ and c_2 for *all-uses*$_s$, respectively. These branches are part of the paths that cover the aDUAs and the asDUA in the class. This is the reason why *all-uses*$_a$ and *all-uses*$_{as}$ are the only AOSV criteria for which c_2 is significantly higher that c_2 for *AO branches* in the *SpeedControl* class. Similarly, in the *Kettle* class, c_2 for *AO branches* is high due to the branches in the aspects, but it is not significantly different from c_2 for *all-uses*$_s$.

Finally, testing hypothesis $H0_{B9}$ in the *Account* class is not applicable. In the *Account* class, c_2 for the *all-uses*$_{ma}$ and c_2 for AO blocks have the same value for all 30 observations (1 and 0.095, respectively). However, the difference between the means of c_2 for the test criteria shows that c_2 for *all-uses*$_{ma}$ is higher, which supports the alternative hypothesis.

8.1.2.2 Comparing the Cost of the AOSV Test Criteria with Each Other Using Size Metric c_2

Table 8.6 shows the results from testing hypotheses $H0_{B13}$ through $H0_{B27}$. The table is organized in the same way as Table 8.5 but does not show the non-advised classes since they do not contain any of the AOSV criteria other than the *all-uses*$_c$. The results from testing hypotheses $H0_{B13}$ through $H0_{B27}$ are in favor of the alternative hypotheses and show that there is a significant difference between c_2 for the AOSV test criteria compared with each other in 61 out of 73 hypothesis tests performed in the classes.

The results from testing hypotheses $H0_{B13}$ through $H0_{B17}$ show that c_2 of the *all-uses*$_s$ criterion is significantly lower than c_2 of the other AOSV test criteria in 27 out of 32 tests performed for the hypotheses. This is because *all-uses*$_s$ subsumes

Table 8.6: Hypotheses tests results for comparing the cost of the AOSV criteria with each other using size metric c_2

Hypotheses	Kettle	Account	Local	LongDistance	Customer (Telecom)	CarSimulator	CruiseController	SpeedControl
$H0_{B13}$: all-uses$_s$-all-uses$_a$	-.50 **	-.33 **						-.46 **
$H0_{B14}$: all-uses$_s$-all-uses$_o$	-.12 **	-.18 **	-.01 ns	-.22 **		-.43 **	-.29 **	-.28 **
$H0_{B15}$: all-uses$_s$-all-uses$_c$	-.49 **	-.02 ns	-.29 **	-.21 **	.04 ns	-.04 **	-.11 **	.02 ns
$H0_{B16}$: all-uses$_s$-all-uses$_{as}$	-.38 **	-.60 **	.02 ns	-.11 **	-.24 **	-.32 **	-.37 **	-.66 **
$H0_{B17}$: all-uses$_s$-all-uses$_{ma}$	-.50 **	-.68 **	-.51 **	-.72 **				
$H0_{B18}$: all-uses$_a$-all-uses$_o$	.38 **	.14 **						.18 **
$H0_{B19}$: all-uses$_a$-all-uses$_c$	.11 *	.30 **						.48 **
$H0_{B20}$: all-uses$_a$-all-uses$_{as}$	.11 *	-.28 **						-.20 **
$H0_{B21}$: all-uses$_a$-all-uses$_{ma}$	0 ns	-.36 **						
$H0_{B22}$: all-uses$_o$-all-uses$_c$	-.26 **	.16 **	-.28 **	.02 ns		.39 **	.18 **	.30 **
$H0_{B23}$: all-uses$_o$-all-uses$_{as}$	-.26 **	-.42 **	.03 ns	.11 **		.11 *	-.08 ns	-.38 **
$H0_{B24}$: all-uses$_o$-all-uses$_{ma}$	-.38 **	-.50 **	-.50 na	-.50 na				
$H0_{B25}$: all-uses$_c$-all-uses$_{as}$	0 ns	-.58 **	.32 **	.10 *	-.28 **	-.28 **	-.26 **	-.68 **
$H0_{B26}$: all-uses$_c$-all-uses$_{ma}$	-.12 *	-.66 **	-.22 **	-.52 **				
$H0_{B27}$: all-uses$_{as}$-all-uses$_{ma}$	-.12 ns	-.08 ns	-.53 **	-.61 **				

the other AOSV test criteria, and therefore, a test case in a test suite that satisfies *all-uses*$_s$ covers different types of DUAs than it covers a particular type.

Metric c_2 for *all-uses*$_s$ is 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 c_2 for *all-uses*$_s$ becomes larger. When the comparison between c_2 for *all-uses*$_s$ and c_2 for an AOSV test criterion shows no significant difference, then the DUAs required by the criterion that is compared with *all-uses*$_s$ are covered by the test suites that satisfy the other AOSV criteria. For example, testing hypothesis $H0_{B13}$ in the *Account* class shows that there is no significant difference between c_2 for *all-uses*$_s$ and c_2 of *all-uses*$_c$. This means that most of the cDUAs in the *Account* class are covered by the test suites that satisfy the *all-uses*$_a$, *all-uses*$_o$, *all-uses*$_{as}$, and *all-uses*$_{ma}$ criteria.

The results from testing hypotheses $H0_{B18}$ and $H0_{B19}$ show that the means for c_2 for *all-uses*$_a$ is significantly higher than c_2 for *all-uses*$_o$ and *all-uses*$_c$ in all classes in which the hypotheses are tested, respectively. This is because, as mentioned in Section 8.1.1.2, the aDUAs require more different paths than the other AOSV DUAs. However, the result from testing hypothesis $H0_{B20}$ shows that c_2 for *all-uses*$_a$ is significantly different than c_2 for *all-uses*$_{as}$ but not higher. In the *Account* and *SpeedControl* classes, c_2 for *all-uses*$_{as}$ is significantly higher than c_2 for *all-uses*$_a$. This is because in these classes, all the asDUAs required executing two consecutive calls to the *debit* method of the *Account* class. Thus, the number of test cases needed to cover the asDUAs in the class is increased.

The results from testing hypotheses H_{B25}, H_{B26}, and H_{B27} show that c_2 for *all-uses*$_{ma}$ is significantly higher than c_2 for each of *all-uses*$_o$, *all-uses*$_c$, and *all-uses*$_{as}$ (except for H_{B27} in the *Kettle* and *Account* classes, in which it is higher but not significantly). This is because a path between a *def* in an aspect and a *use* in a different aspect do not cover many other maDUAs. Testing hypothesis H_{B24} is not applicable in the *Local* and *LongDistance* classes because the variances of the compared observations are zero. However, the difference between the means of c_2 for the criteria shows that c_2 of *all-uses*$_{ma}$ is higher, which supports the alternative hypothesis.

The results from testing hypotheses $H0_{B22}$, $H0_{B23}$, and $H0_{B25}$ show that there is a significant difference between c_2 for *all-uses*$_o$, *all-uses*$_c$, and *all-uses*$_{as}$, but none of these criteria costs more than the others. In some classes, the oDUAs, cDUAs, and asDUAs are easily covered by any test case (e.g., oDUAs, asDUAs in the *Local* class, and cDUAs in the *Account* class). In other classes, they require different paths, and therefore, more test cases (e.g., oDUAs in the *CruiseController* class, asDUAs in the *SpeedControl* class, and cDUAs in the *CarSimulator* class).

8.1.3 Comparing the Cost of the Test Criteria Using Effort Metric c_3

We present the results from testing the hypotheses in group C. These hypotheses compare the effort needed to obtain the test suites that satisfy the various test criteria. Section 8.1.3.1 presents the results from testing hypotheses $H0_{C1}$ through $H0_{C12}$, in which c_3 for the AOSV test criteria is compared with c_3 for the AO control-flow criteria. Section 8.1.3.2 presents the results from testing hypotheses $H0_{C13}$ through $H0_{C27}$, in which the AOSV test criteria are compared with each other using c_3.

In Table 8.7, we show statistics for the effort needed to obtain the test suites that satisfy the test criteria. We use these statistics to explain the results from testing the hypotheses in group C and group D. For each test criterion shown in column 1, and for each class in row 1, Table 8.7 shows: (1) the means of c_3 for the test criteria rounded to the nearest decimal digit (top line), (2) the standard deviations of c_3 rounded to the nearest decimal digit (second line), and (3) the minimum and maximum values of c_3 (third and fourth lines, respectively). For example, the entry in the second row and second column shows that in the *Kettle* class (1) the mean value of c_3 for generating the test suites that satisfy *all-uses*$_a$ is

Table 8.7: Statistics for measuring the cost of the test criteria using effort metrics

Criteria	kettle	Account	Customer (Banking)	Local	LongDistance	Customer (Telecom)	Call	Timer	CarSimulator	CruiseController	SpeedControl
all-uses$_a$	85.2 8.3 23 216	2500.4 174.7 426 3981									415.6 313.8 100 1289
all-uses$_o$	32 3.5 7 75	1992.5 173.5 458 3736		13.9 16.2 1 78	4 2.7 1 10				32.5 17.3 6 86	10567.6 1650.2 7389 13404	6.8 4 1 16
all-uses$_c$	92.8 11.2 25 322	28.5 3 4 71	2.8 1.6 1 8	646.2 498.1 45 1941	24.3 15 4 69	3.7 3.5 1 19	619.5 473.6 1656 63	15.3 9.8 30 1	10236.8 2627.5 4919 13940	11170.1 1718.9 7661 14023	951.8 572.3 300 2529
all-uses$_{as}$	160.2 20.2 22 413	1002 126.2 6 2276		463.8 333 44 1438	3.4 2.7 1 8	22 16.6 2 63			220.2 128.9 38 512	281.7 267.6 15 1045	39 42.3 3 217
all-uses$_{ma}$	45.2 7.9 3 202	369.4 54.6 31 1060		3.2 2.3 1 10	2.7 2 1 9						
all-uses$_s$	213.8 30.3 40 751	2248.8 148.2 793 3878		911.3 490.5 156 2458	21.2 17.6 5 79	22 12.7 74 4			10670.5 2414.2 5828 13982	11081.9 1377.2 8997 13823	925.2 490.3 217 2253
AO blocks	23.9 2.5 3 48	234.7 22.3 43 486	2.1 0.8 1 3	125.9 32.8 80 196	17 5.7 1 26	1.6 0.5 1 2	214.5 111.9 41 399	13.3 8.7 45 3	140.2 32.8 73 186	158.4 100.9 46 405	69.8 16.4 42 98
AO branches	36.3 3.1 7 58	335.1 30 63 546					2547 130.5 477 41		267.7 114.3 88 512	172.3 98.4 15 405	77.8 17.7 51 121

85.2, (2) the standard deviation is 8.3, and (3) c_3 ranges from 23 to 216. Empty cells in the table indicate that the class does not contain test requirements for the criterion.

8.1.3.1 Comparing the Cost of the AOSV Test Criteria with the Cost of the AO Control-Flow Criteria Using Effort Metric c_3

Table 8.8: Hypotheses test results for comparing the cost of the AOSV criteria with the cost of the AO control-flow criteria using effort metric c_3

Hypotheses	Kettle	Account	Customer (Banking)	Local	LongDistance	Customer (Telecom)	Call	Timer	CarSimulator	CruiseController	SpeedControl
$H0_{C1}$: all-uses$_a$-Nodes	61 **	2266 **									346 **
$H0_{C2}$: all-uses$_a$-branches	48 **	2165 **									338 **
$H0_{C3}$: all-uses$_o$-Nodes	8 ns	1758 **		-112 **	-13 **				-108 **	10409 **	-63 **
$H0_{C4}$: all-uses$_o$-branches	-4 ns	1657 **							-235 **	10395 **	-71 **
$H0_{C5}$: all-uses$_c$-Nodes	69 **	-206 **	1 ns	520 **	7 ns	-2 ns	405 **	2 ns	10097 **	11012 **	882 **
$H0_{C6}$: all-uses$_c$-branches	57 **	-307 **					365 **		9969 **	10998 **	874 **
$H0_{C7}$: all-uses$_{as}$-Nodes	136 **	767 **		338 **	-14 **	17 **			80 *	123 ns	-31 *
$H0_{C8}$: all-uses$_{as}$-branches	124 **	667 **							-48 ns	109 ns	-39 **
$H0_{C9}$: all-uses$_{ma}$-Nodes	21 ns	135 ns		-123 **	-14 **						
$H0_{C10}$: all-uses$_{ma}$-branches	9 ns	34 ns									
$H0_{C11}$: all-uses$_s$-Nodes	190 **	2014 **		785 **	4 **	17 **			10530 **	10924 **	855 **
$H0_{C12}$: all-uses$_s$-branches	178 **	1914 **							10401 **	10910 **	847 **

Table 8.8 shows the results from testing hypotheses $H0_{C1}$ through $H0_{C12}$. The table is organized in the same way as Tables 8.2 and 8.5. We rounded the difference between the means of the effort metric c_3 for the compared criteria to the nearest integer.

The results from testing hypotheses $H0_{C1}$ through $H0_{C12}$ are in favor of the alternative hypotheses except for hypothesis $H0_{C10}$, for which the results show that there is no significant difference between c_3 for *all-uses$_{ma}$* and c_3 for *AO branches*. The results from testing hypotheses $H0_{C1}$, $H0_{C2}$, $H0_{C11}$, and $H0_{C12}$ also show that c_3 for *all-uses$_a$* and c_3 for *all-uses$_s$* are significantly higher than c_3 for each of the AO control-flow criteria, respectively. As explained in the previous sections, the aDUAs require more different paths that are hard to be exercised by the test cases than the other types of AOSV DUAs. This is because aDUAs always require two consecutive calls to the advised methods that contain the *def* and the method that contain the *use*. Covering all AOSV DUAs in the classes requires significantly more effort than covering the branches and blocks because (1) the classes contain a higher number of AOSV DUAs than they contain of one type of DUAs, and (2) the DUAs require executing more paths that are hard to be exercised by the test cases than the paths required to cover the branches and blocks.

The results from testing hypotheses $H0_{C3}$ through $H0_{C8}$ show that there are significant differences between c_3 for *all-uses$_o$*, *all-uses$_c$*, and *all-uses$_{as}$*, and c_3 for each of the AO control-flow criteria, but these differences are not consistent (i.e., the results do not show that c_3 for any pair of compared criteria is higher in all classes).

The two factors that contribute to the effort of obtaining a test suite that covers a criterion are (1) the number of test cases in the pool that can cover the test requirements of a criterion, and (2) the number of test requirements a class contains for the 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 criteria. Some classes contain AOSV DUAs that require covering paths that are hard to be executed by in the test cases.

We can make the same conclusions for the AO control-flow criteria. For example, satisfying *AO blocks* in the *Customer* class of the *Telecom* program and the *Customer* class of the *Banking* program requires an average c_3 of only 2.1 and 1.6 iterations, respectively. This is because these two classes are simple (i.e., they do not contain branches and the aspects do not contain requirements). On the other hand, satisfying *AO blocks* in the *CruiseController* class requires an average c_3 of 158.4 iterations because there are blocks in the aspects that are part of hard to cover paths.

The results for the *all-uses*$_c$ criterion show how much there are variations in the effort needed to satisfy an AOSV test criterion in different classes. In the non-advised *Customer* class in the *Banking* program, and the non-advised *Timer* class in the *Telecom* program, satisfying *all-uses*$_c$ does not require more effort than satisfying *AO blocks*. Note that these two classes do not contain branches, which makes the paths between the cDUAs easy to cover. In the non-advised *Call* class, *all-uses*$_c$ requires more effort to satisfy than both of the AO control-flow criteria because the class contains *defs* and *uses* of state variables inside loops. In the *Account* class, as mentioned in Section 8.1.1, most of the cDUAs occur between *setter* and *getter* methods, while the AO control-flow criteria have test requirements in the aspects. Therefore, satisfying *all-uses*$_c$ requires less effort than satisfying the AO control-flow criteria in the *Account* class. In the classes of the *CruiseControl* program (*CarSimulator*, *CruiseController*, and *SpeedControl*), the effort needed to satisfy *all-uses*$_c$ is significantly higher than the effort of satisfying the AO control-flow criteria because the classes contain many cDUAs where the *def* and *use* are in advised methods. These cDUAs and also some oDUAs are the reason why we needed to generate a pool that contains 14,000 test case in order to satisfy the criteria in the classes of the *CruiseControl* program.

8.1.3.2 Comparing the Cost of the AOSV Test Criteria with Each Other Using Effort Metric c_3

Table 8.9: Hypotheses test results for comparing the cost of the AOSV test criteria with each other using the effort metric c_3

Hypotheses	Kettle	Account	Local	LongDistance	Customer (Telecom)	CarSimulator	CruiseController	SpeedControl
$H0_{C13}$: all-uses$_s$-all-uses$_a$	129 **	252 ns						510 **
$H0_{C14}$: all-uses$_s$-all-uses$_o$	182 **	256 ns	898 **	17 **		10638 **	514 ns	918 **
$H0_{C15}$: all-uses$_s$-all-uses$_c$	121 **	2220 **	265 ns	3 ns	18 **	434 ns	-88 ns	-27 ns
$H0_{C16}$: all-uses$_s$-all-uses$_{as}$	54 ns	1247 **	448 **	18 **	-1 ns	10450 **	10800 **	886 **
$H0_{C17}$: all-uses$_s$-all-uses$_{ma}$	169 **	1879 **	908 **	19 **				
$H0_{C18}$: all-uses$_a$-all-uses$_o$	53 **	507 ns						409 **
$H0_{C19}$: all-uses$_a$-all-uses$_c$	-8 ns	2472 **						-536 **
$H0_{C20}$: all-uses$_a$-all-uses$_{as}$	-75 **	1498 **						377 **
$H0_{C21}$: all-uses$_a$-all-uses$_{ma}$	40 ns	2131 **						
$H0_{C22}$: all-uses$_o$-all-uses$_c$	-61 **	1964 **	-632 **	-20 **		-10204 **	-603 ns	-945 **
$H0_{C23}$: all-uses$_o$-all-uses$_{as}$	-128 **	991 **	-450 **	1 ns		-188 **	10286 **	-32 **
$H0_{C24}$: all-uses$_o$-all-uses$_{ma}$	-13 ns	1623 **	11 *	1 ns				
$H0_{C25}$: all-uses$_c$-all-uses$_{as}$	-67 ns	-974 **	182 ns	21 **	-19 **	10017 **	10888 **	913 **
$H0_{C26}$: all-uses$_c$-all-uses$_{ma}$	48 ns	-341 **	643 **	22 **				
$H0_{C27}$: all-uses$_{as}$-all-uses$_{ma}$	115 **	633 **	2 **	1 ns				

Table 8.9 shows the results from testing hypotheses $H0_{C13}$ through $H0_{C27}$. The table is organized in the same way as Table 8.8 but does not show the non-advised classes. The results from testing hypotheses $H0_{C13}$ through $H0_{C27}$ are in favor of the alternative hypotheses, except for hypothesis $H0_{C15}$, which shows that there is no significant difference between c_3 for *all-uses*$_s$ and c_3 for *all-uses*$_c$ in most of classes.

The results from testing hypotheses $H0_{C13}$, $H0_{C14}$, $H0_{C16}$, $H0_{C17}$ show that c_3 for *all-uses*$_s$ is significantly higher than c_3 for *all-uses*$_a$, *all-uses*$_{as}$, *all-uses*$_{as}$, and *all-uses*$_{ma}$, respectively, in 17 out of 22 tests performed for theses hypotheses. The remaining 5 tests show that the difference is not significant. The differences occurred in the classes that contain a type of AOSV DUAs that require significantly higher effort to cover than the other types of AOSV DUAs in the class. For example, in the *Local* class, the results show that selecting the test cases that satisfy the 9 cDUAs in the pool of test cases for the *Telecom* program, which contains 6000 test cases, requires 265 fewer iterations than that required to find the test cases that cover all the 17 DUAs in the class. In the classes where the effort needed to cover a particular type of AOSV DUAs is not significantly different than the effort needed to cover all AOSV DUAs, the effort of covering that type is also significantly higher than the effort needed to cover the remaining AOSV DUAs in the class. For example, in the *SpeedControl* class, there is no significant difference between c_3 for *all-uses*$_s$ and c_3 for *all-uses*$_c$, but c_3 of *all-uses*$_c$ is significantly higher than c_3 for the other AOSV criteria in the class (*all-uses*$_a$, *all-uses*$_o$, and *all-uses*$_{as}$).

The results from testing hypothesis $H0_{C15}$ supports the null hypothesis in 5 out of 8 classes. In the remaining 3 classes, c_3 for *all-uses*$_s$ is significantly higher than c_3 for *all-uses*$_c$. In the 5 classes in which the results of testing $H0_{C15}$ show no significant difference, c_3 for *all-uses*$_c$ is significantly higher than c_3 for the other

AOSV test criteria. In other words, the results show that whenever there is a type of AOSV DUAs that requires higher effort to cover in the class compared with other AOSV DUAs, the effort of covering this type of DUAs becomes close to the effort of covering all the AOSV DUAs in the class.

The results from testing hypotheses $H0_{C18}$ through $H0_{C23}$ show significant differences between c_3 for the AOSV test criteria in 34 out of the 45 tests performed for the hypotheses, but the differences in c_3 between each pair of compared criteria are not consistent in all classes. This is because c_3 for any of the AOSV DUAs varies from class to class depending on how many DUAs of each type a class contains and whether the test cases that execute the paths that cover the DUAs are hard to find in the pool of test cases.

8.1.4 Comparing the Cost of the Test Criteria Using Effort Metric c_4

We present the results from testing the hypotheses in group D. Section 8.1.4.1 presents the results from testing hypotheses $H0_{D1}$ through $H0_{D12}$, in which c_4 for the AOSV test criteria is compared with c_4 for the AO control-flow criteria. Section 8.1.4.2 shows the results from testing hypotheses $H0_{D13}$ through $H0_{D27}$, in which the AOSV test criteria are compared with each other using c_4.

8.1.4.1 Comparing the Cost of the AOSV Test Criteria with the Cost of the AO Control-Flow Criteria Using Effort Metric c_4

Table 8.10 shows the results from testing hypotheses $H0_{D1}$ through $H0_{D12}$. The results are in the favor of the alternative hypotheses and show that there is a significant difference between c_4 for each of the AOSV test criteria and each of the AO control-flow criteria in 58 out of 67 hypothesis tests in the classes. The results also show that c_4 for *all-uses*$_a$ is significantly higher than c_4 for each of the AO control-flow criteria in all classes.

Table 8.10: Hypotheses test results for comparing the cost of the AOSV test criteria with the cost of the AO control-flow criteria using effort metric c_4

Hypotheses	Kettle	Account	Customer (Banking)	Local	LongDistance	Customer (Telecom)	Call	Timer	CarSimulator	CruiseController	SpeedControl
$H0_{D1}$: all-uses$_a$-blocks	7.4 **	613.9 **									48.8 **
$H0_{D2}$: all-uses$_a$-branches	6.3 **	597.2 **									44.2 **
$H0_{D3}$: all-uses$_o$-blocks	0.9 ns	170.0 **		0.34 ns	1.1 **				2.7 **	229.9 **	-1.5 **
$H0_{D4}$: all-uses$_o$-branches	-0.3 ns	153.2 **							-6 **	227.9 **	-6.1 **
$H0_{D5}$: all-uses$_c$-blocks	6.6 **	-9.6 **	0.2 **	65.2 **	1.8 **	0.2 ns	56.9 **	1.2 **	77.9 **	166.9 **	16.3 **
$H0_{D6}$: all-uses$_c$-branches	5.5 **	-26.3 **					40.5 **		69.2 **	165 **	11.7 **
$H0_{D7}$: all-uses$_{as}$-blocks	14.9 **	489.8 **		70.7 **	-0.2 ns	2.5 **			10.3 **	42 **	16.3 **
$H0_{D8}$: all-uses$_{as}$-branches	13.8 **	473.1 **							1.6 ns	40.1 **	11.7 ns
$H0_{D9}$: all-uses$_{ma}$-blocks	6.4 **	358.2 **		-3.4 **	1.8 **						
$H0_{D10}$: all-uses$_{ma}$-branches	5.3 *	341.4 **									
$H0_{D11}$: all-uses$_s$-blocks	2.8 **	51.3 **		47.0 **	0.4 ns	1.1 **			68.4 **	90.6 **	11.5 **
$H0_{D12}$: all-uses$_s$-branches	1.7 ns	34.5 **							59.8 **	88.6 **	6.9 **

The result from testing hypothesis $H0_{D7}$ shows that c_4 for *all-uses*$_{as}$ is significantly higher than c_4 for *AO blocks* in 7 out of 8 classes in which the hypothesis is tested. In the *LongDistance* class, there is no significant difference. This is because objects of type *LongDistance* are frequently used in the test cases, and this class contains asDUAs that are covered by most of the test cases. The result from testing hypothesis $H0_{D8}$ show that c_4 for *all-uses*$_{as}$ is significantly higher than c_4

for *AO branches* in 3 out of 5 classes in which the hypothesis is tested. In the remaining two classes (*CarSimulator* and *SpeedControl*), the effort is higher but not significantly. The test cases that execute the branches in the aspects for these two advised classes are hard to find in the pool of test cases.

The result from testing hypothesis $H0_{D10}$ shows that the effort needed to cover an maDUA is significantly higher than the effort needed to cover a branch in all the advised classes in which $H0_{D10}$ is tested. The results from testing hypotheses $H0_{D3}$ through $H0_{D6}$ show that there is a significance difference between c_4 for *all-uses*$_o$ and c_4 for *all-uses*$_c$, and c_4 for each of the AO control-flow criteria, respectively. However, the differences are inconsistent in the different classes.

Finally, the results from testing hypothesis $H0_{D11}$ and $H0_{D12}$ show that the effort needed to cover any type of AOSV DUAs is significantly higher than the effort needed to cover a block or a branch in 11 out of 13 tests performed for the hypothesis. In the other two tests, there is no significant difference. These are the test of hypothesis $H0_{D11}$ in the *LongDistance* class, and the test of $H0_{D12}$ in the *Kettle* class. As discussed earlier, satisfying a criterion for the *LongDistance* class requires less effort due to high use of the class in the test cases, and the test cases that execute the branches in the aspects of the *Kettle* class are hard to find in the pool of test cases.

8.1.4.2 Comparing the Cost of the AOSV Test Criteria with Each Other Using Effort Metric c_4

Table 8.11 shows the results from testing hypotheses $H0_{D13}$ through $H0_{D27}$. The results show that the differences in the effort needed to satisfy the AOSV criteria are inconsistent over all the classes, except for the results from testing hypotheses $H0_{D18}$ and $H0_{D19}$, which show that c_4 for *all-uses*$_a$ is significantly higher than c_4 for *all-uses*$_o$ and c_4 for *all-uses*$_c$, respectively.

Table 8.11: Hypotheses test results for comparing the cost of the AOSV criteria with each other using effort metric c_4

Hypotheses	Kettle	Account	Local	LongDistance	Customer (Telecom)	CarSimulator	CruiseController	SpeedControl
$H0_{D13}$: all-uses$_s$-all-uses$_a$	-4.6 **	-562.6 **						-37.3 **
$H0_{D14}$: all-uses$_s$-all-uses$_o$	2.0 ns	-118.7 **	46.6 **	-0.7 ns		65.7 **	-139.3 **	13 **
$H0_{D15}$: all-uses$_s$-all-uses$_c$	-3.8 *	60.9 **	-18.2 ns	-1.5 **	0.9 **	-9.5 ns	-76.3 **	-4.7 ns
$H0_{D16}$: all-uses$_s$-all-uses$_{as}$	-12.1 **	-438.5 **	-23.7 ns	0.6 ns	-1.4 *	58.2 **	48.6 **	-4.8 ns
$H0_{D17}$: all-uses$_s$-all-uses$_{ma}$	-3.6 *	-306.9 **	50.4 **	-1.4 ns				
$H0_{D18}$: all-uses$_a$-all-uses$_o$	6.5 **	444.0 **						50.3 **
$H0_{D19}$: all-uses$_a$-all-uses$_c$	0.8 ns	623.5 **						32.5 **
$H0_{D20}$: all-uses$_a$-all-uses$_{as}$	-7.5 *	124.1 ns						32.4 **
$H0_{D21}$: all-uses$_a$-all-uses$_{ma}$	1.0 ns	255.7 **						
$H0_{D22}$: all-uses$_o$-all-uses$_c$	-5.7 **	179.6 **	-64.8 **	-0.7 ns		-75.2 **	63 **	-17.7 **
$H0_{D23}$: all-uses$_o$-all-uses$_{as}$	-14.0 **	-319.9 **	-70.3 **	1.3 **		-7.5 **	187.9 **	-17.8 **
$H0_{D24}$: all-uses$_o$-all-uses$_{ma}$	-5.5 **	-188.7 *	3.8 ns	-0.7 ns				
$H0_{D25}$: all-uses$_c$-all-uses$_{as}$	-8.3 **	-499.4 **	-5.5 ns	2.0 **	-2.3 **	67.7 **	124.9 **	-0.1 ns
$H0_{D26}$: all-uses$_c$-all-uses$_{ma}$	0.2 ns	-367.8 **	68.6 **	0.0 ns				
$H0_{D27}$: all-uses$_{as}$-all-uses$_{ma}$	8.3 **	131.6 ns	74.1 **	-0.2 ns				

Metric c_4 for criterion *all-uses$_s$* is an average of the effort needed to cover the different types of AOSV DUAs. In the classes in which the effort of satisfying a particular type of AOSV DUAs is significantly higher than satisfying the other types, c_4 for the criterion that requires satisfying that particular type of DUAs is higher than c_4 for the *all-uses$_s$*. The results from testing hypotheses $H0_{D13}$ through $H0_{D17}$ show that there is no particular type of the AOSV DUAs that always requires above average effort except the aDUAs. This is also confirmed by the results of testing hypotheses $H0_{D18}$, $H0_{D19}$, $H0_{D20}$, and $H0_{D21}$, which show that c_4 for *all-uses$_a$* is higher than c_4 for *all-uses$_o$*, *all-uses$_c$*, *all-uses$_{as}$*, and *all-uses$_{ma}$* in 10 out of 11 tests performed for these hypotheses.

8.2 Comparing the Effectiveness of the Test Criteria

Table 8.12: Means and standard deviations of the mutations scores of the test suites that satisfy the test criteria

Criteria	Kettle	Account	Customer (Banking)	Local	LongDistance	Customer (Telecom)	Call	Timer	CarSimulator	CruiseController	SpeedControl	All
all-uses$_a$	92.9 2.5	87.3 2.6									84.8 5	87.8
all-uses$_o$	91 3	86.9 2.4		62.2 8.2	70.6 4.1				62.2 5.9	89.4 2.2	77.2 4.9	76.6
all-uses$_c$	92.3 1.4	52.3 8.7	44.7 10.1	74 6.8	80.1 5.7	82.8 8.4	70 30.6	97.9 3.3	94.3 0.9	90.4 2	90 4.2	85.3
all-uses$_{as}$	91.7 3.7	59.9 10.4		69.3 6.8	67.9 8.6	91.7 3			75.3 6.8	66.5 6.4	78.8 3.9	74.8
all-uses$_{ma}$	88.7 6.2	57.8 11.2		67 9.5	67.6 8.7							71.8
all-uses$_s$	94.4 5.7	89.6 0		86.6 6.9	81.4 5.2	93.5 2			96.5 0.7	94.2 1.5	95.1 2.3	92.4
AO blocks	72.6 12.5	47.2 6.8	46 0.8	49 26.3	56.1 20.6	73.6 14.6	49.3 29.1	83.6 22.5	45.8 5.1	57.6 4.7	64.6 4.9	57.6
AO branches	77.6 10.9	40.1 10.8					73.7 33.2		60 8	60.9 4	70.6 5.2	63.2

We address the second research question in this section. Table 8.12 shows the effectiveness results for the test criteria. For each test criterion in column 1 and for each class in row 1, the table shows: (1) the average mutation score obtained by the test suites that satisfy a test criterion (top line), and (2) the standard deviation of the mutation scores obtained by the test suites that satisfy a test criterion (bottom line). The means and standard deviations are computed over the 30 test suites that satisfy each criterion in the class. For example, the entry in the second row and second column shows that in the *Kettle* class (1) the mean mutation score of the test suites that satisfy *all-uses*$_a$ is 92.9%, and (2) the standard deviation is 2.5.

The standard deviations of the mutations scores for the AOSV test criteria are small compared with the standard deviations of the AO control-flow criteria, which means that the variations in mutation scores between the 30 test suites that satisfy each of the AOSV test criteria are small. The last column in the table shows the means of the mutation scores of the test criteria 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*$_s$ has the highest overall mutation score while *AO blocks* has the lowest.

Section 8.2.1 presents the results from testing hypotheses $H0_{E1}$ through $H0_{E12}$, in which the effectiveness of the AOSV test criteria is compared with the effectiveness of the AO control-flow criteria. Section 8.2.2 presents the results of testing hypotheses $H0_{E13}$ through $H0_{E27}$, in which the effectiveness of the AOSV test criteria is compared with each other.

8.2.1 Comparing the Effectiveness of the AOSV Test Criteria with the Effectiveness of the AO Control-Flow Criteria

Table 8.13 shows the results from testing hypotheses $H0_{E1}$ through $H0_{E12}$. The results of testing hypotheses $H0_{E1}$, $H0_{E2}$, $H0_{E11}$, and $H0_{E12}$ show that the mutation scores of the test suites that satisfy *all-uses*$_a$ and *all-uses*$_s$ are significantly

Table 8.13: Effectiveness results for comparing the AOSV test criteria with the AO control-flow criteria

Hypotheses	Kettle	Account	Customer (Banking)	Local	LongDistance	Customer (telecom)	Call	Timer	CarSimulator	CruiseController	SpeedControl
$H0_{E1}$: all-uses$_a$-Nodes	20.3 **	40.1 **									20.2 **
$H0_{E2}$: all-uses$_a$-branches	15.3 **	47.2 **									14.2 **
$H0_{E3}$: all-uses$_o$-Nodes	18.5 **	39.7 **		13.1 ns	14.5 ns				16.3 **	31.8 **	12.6 **
$H0_{E4}$: all-uses$_o$-branches	13.4 **	46.9 **							2.5 ns	28.5 **	6.6 **
$H0_{E5}$: all-uses$_c$-Nodes	19.8 **	5.1 ns	-1.3 ns	25 **	24 **	9.2 ns	20.7 **	14.3 **	48.5 **	32.8 **	25 **
$H0_{E6}$: all-uses$_c$-branches	14.8 **	12.3 **					-3.7 ns		34.7 **	29.5 **	19 **
$H0_{E7}$: all-uses$_{as}$-Nodes	19.1 **	12.7 **		20.2 **	11.8 ns	18 **			29.6 **	8.9 **	14.3 **
$H0_{E8}$: all-uses$_{as}$-branches	14.1 **	19.8 **							15.8 **	5.6 **	8.3 **
$H0_{E9}$: all-uses$_{ma}$-Nodes	16.1 **	10.6 **		17.9 *	11.4 ns						
$H0_{E10}$: all-uses$_{ma}$-branches	11.1 **	17.8 **									
$H0_{E11}$: all-uses$_s$-Nodes	21.8 **	42.4 **		37.6 **	25.2 **	19.9 **			49.7 **	36.6 **	30.6 **
$H0_{E12}$: all-uses$_s$-branches	16.8 **	49.5 **							35.9 **	33.3 **	24.6 **

higher than the mutation scores of the test suites that satisfy each of the *AO control-flow* criteria in all the classes in which these hypotheses are tested. For the rest of the AOSV criteria, the results show that these criteria have significantly higher mutation scores in most of the classes (39 out of 48 tests), and there is no significant difference in 9 tests.

Our expectations are that criterion *all-uses$_s$* is more effective than the AO control-flow criteria, and this is matched by the results. For the other AOSV criteria, we expected differences 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*$_c$ 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* in *Telecom* program. Recall from Section 8.1.1 that 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 advices. However, in the classes that contain advised methods and a high number of cDUAs, test suites that satisfy *all-uses*$_c$ are more effective than the test suites that satisfy the AO control-flow criteria.

In the *Local* and *LongDistance* classes, the mutation scores of the test suites that satisfy *all-uses*$_o$ are not significantly higher than the mutation scores of the test suites that satisfy *AO blocks*. Recall from previous sections that these two classes have only 2 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*$_{as}$ and *all-uses*$_{ma}$ 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 (Section 8.1.1). 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.

8.2.2 Comparing the Effectiveness of the AOSV Test Criteria with Each Other

Table 8.14 shows the results from testing hypotheses $H0_{E13}$ through $H0_{E27}$. The results show that the mutation scores of the test suites that satisfy *all-uses*$_s$ 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*$_s$. The only exception is in the *LongDistance* class in which the mutation scores for the test suites that satisfy *all-uses*$_s$ is higher than the mutation scores of the test suites that satisfy *all-uses*$_c$, but not significantly. The reason is that the test suites that satisfy *all-uses*$_c$ in the *LongDistance* class also cover most of the other AOSV DUAs. Recall that also in this class the sizes of the test suites that satisfy *all-uses*$_s$ are not significantly higher than the sizes of the test suites that satisfy all-uses$_c$.

Comparisons between the other AOSV criteria show that there is a significant difference in their effectiveness as measured by the mutation score. Out of 45 tests, 17 show an insignificant difference. However, there is no consistency in the difference between the criteria, except for the result from testing hypothesis $H0_{E21}$, which shows that *all-uses*$_a$ is more effective than *all-uses*$_{ma}$. The reason for these differences is effectiveness depend on how many different paths the DUAs cover in the advised class. For example, *all-uses*$_o$ is significantly more effective than *all-uses*$_{as}$ in the *CruiseConttroller* class and significantly less effective than *all-uses*$_{as}$ in the *CarSimulator* class (i.e., result of testing hypothesis $H0_{E23}$). That is because the *CruiseController* class contains 45 oDUAs and only 6 asDUAs, while the *CarSimulator* class contains 17 asDUAs 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.

8.3 Effectiveness of the AOSV Test Criteria Based on Fault Types

We present the answer to research question 3 in this section. We organize this section according to the fault categories of the revised AO fault model given in

Table 8.14: Effectiveness results for comparing the AOSV test criteria with each other

Hypotheses	Kettle	Account	Local	LongDistance	Customer (telecom)	CarSimulator	CruiseController	SpeedControl
$H0_{E13}$: all-uses$_s$-all-uses$_a$	1.5 **	2.3 **						10.3 **
$H0_{E14}$: all-uses$_s$-all-uses$_o$	3.4 **	2.6 **	24.5 **	10.7 **		33.4 **	4.8 **	18 **
$H0_{E15}$: all-uses$_s$-all-uses$_c$	2 **	37.2 **	12.6 **	1.2 ns	10.6 **	1.2 **	3.8 **	5.6 **
$H0_{E16}$: all-uses$_s$-all-uses$_{as}$	2.7 *	29.7 **	17.4 **	13.5 **	1.8 *	20.1 **	27.7 **	16.3 **
$H0_{E17}$: all-uses$_s$-all-uses$_{ma}$	5.7 **	31.7 **	19.7 **	13.8 **				
$H0_{E18}$: all-uses$_a$-all-uses$_o$	1.9 ns	0.3 ns						7.6 **
$H0_{E19}$: all-uses$_a$-all-uses$_c$	0.5 ns	35 **						-4.8 *
$H0_{E20}$: all-uses$_a$-all-uses$_{as}$	1.2 ns	27.4 **						6 **
$H0_{E21}$: all-uses$_a$-all-uses$_{ma}$	4.2 **	29.4 **						
$H0_{E22}$: all-uses$_o$-all-uses$_c$	-1.3 **	34.6 **	-11.8 **	-9.5 **		-32.2 **	-1 ns	-12.4 **
$H0_{E23}$: all-uses$_o$-all-uses$_{as}$	-0.7 **	27.1 **	-7.1 **	2.7 ns		-13.3 **	22.9 **	-1.7 ns
$H0_{E24}$: all-uses$_o$-all-uses$_{ma}$	2.3 **	29.1 **	-4.8 ns	3.1 ns				
$H0_{E25}$: all-uses$_c$-all-uses$_{as}$	0.7 ns	-7.5 ns	4.8 ns	12.2 **	-8.8 **	18.9 **	23.9 **	10.7 **
$H0_{E26}$: all-uses$_c$-all-uses$_{ma}$	3.7 ns	-5.5 ns	7.1 *	12.6 **				
$H0_{E27}$: all-uses$_{as}$-all-uses$_{ma}$	3 ns	2 ns	2.3 ns	-13.5 **				

Chapter 4. For each fault type, we inspected the live mutants and identified the reasons why the mutants were not killed by the test suites that satisfy *all-uses$_s$* (i.e., the test suites that cover all types of AOSV DUAs). We also explain why there are differences between the mutation scores of the test suites that cover each

type of AOSV DUAs.

8.3.1 Pointcut Descriptor Related Faults (F1)

We discuss in this section the effectiveness of the test criteria in terms of detecting faults of each type in category F1. Table 8.15 shows the average mutation scores of the test suites that satisfy each of the AOSV criteria for faults resulting from a pointcut that matches only unintended join points (fault type F1-1). In the table, column 1 shows the class names. Column 2 shows the number of mutants of type F1-1 in each class. Columns 3 through 8 show the average mutation score for the test suites that satisfy each of the AOSV criteria. The average is taken over the 30 test suites that satisfy a criterion in each class. The symbol “na” means that the mutation score for the criterion is not available because the class does not contain test requirements for the criterion. The last row in the table shows the overall averages of the mutation scores of a criterion taken overall the classes. The overall average 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 F1-1 mutants. Classes that do not contain fault of type F1-1 are not shown in the table. Note that *all-uses$_a$* is not shown in Table 8.15 because the classes that contain faults of type F1-1 do not contain aDUAs.

Table 8.15: Effectiveness of the AOSV test criteria in detecting faults of type F1-1

Class	# Mut	all-uses$_o$	all-uses$_c$	all-uses$_{as}$	all-uses$_{ma}$	all-uses$_s$
Local	16	72.3	79.2	77.1	78.5	99
LongDistance	16	67.5	87.5	76.9	79.6	100
Customer	6	na	97.2	100	na	100
CruiseController	2	100	100	100	na	100
All	**40**	**75.9**	**86.3**	**81.6**	**79.1**	**99.6**

The results show that test suites that satisfy *all-uses$_s$* killed all mutants of type F1-1 in the *LongDistance*, *Customer*, and *CruiseController* classes. In the *Local*

```
//original
after(Connection conn, Customer cust): this(conn) && args(cust, ..) &&
                                     execution(Connection+.new(..)){
     conn.payer = cust; }

//Mutant 1
after(Connection conn, Customer cust): this(conn) && args(cust, ..) &&
                                     !execution(Connection+.new(..)){
     conn.payer = cust; }

//class Connection constructor
public Connection(Customer a, Customer b) {
     this.caller = a;      // matched by mutant 1
     this.receiver = b;    // matched by mutant 1
}
```

Figure 8.1: A hard to kill mutant of type F1-1. The mutant is from the *Billing* aspect which is woven with the *Local* class

class, 25 test suites killed all the mutants and the remaining 5 test suites failed to kill one of the 16 mutants.

Figure 8.1 shows a mutant, which is not always killed by the test suites that satisfy the *all-uses*$_s$ criterion. The mutant is generated by operator PCLO by negating the condition in the poinctut descriptor. The pointcut of the mutant matches statements in methods whenever the running object is of type *Connection*, and the first argument is of type *Customer*. The original pointcut matches the constructor of the *Connection* class. The mutant pointcut matches the two statements in the *Connection* class constructor. Matching the last statement in the constructor results in defining the receiver of the call as the payer (instead of the caller). However, some test cases use the same object for the caller and the callee, and therefore, these test cases are not able to kill the mutant. In the other test cases, the mutant is killed by covering the asDUA between the *def* in the *after* advice and the *use* in the *getPayer* intertype method. This mutant is also always killed by the test suites that satisfy *all-uses*$_s$ in the *LongDistance* class because the caller and the receiver are always different objects.

The results show that *all-uses*$_c$ is more effective than the other AOSV criteria, except *all-uses*$_s$. 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*$_c$ helps more than the other AOSV test criteria in detecting unintended join points.

Table 8.16 shows the average mutation scores of the ASOV-adequate test suites for faults resulting from pointcut that match a subset of the intended join points and some unintended join points (fault type F1-2). The table is organized in the same way as table 8.15. The *all-uses*$_s$ criterion killed all the mutants of type F1-2 in the *Kettle* and *CarSimulator* classes. In the *Local* and *LongDistance* classes, 1 of the 6 mutants is never killed. This mutant shown in Figure 8.2 is a case when the fault generated by operator PWIW which replaces the class name with the match all symbol (*), is combined with the fault generated by operator POAC, which replaces *after* with *after returning.* The mutant pointcut matches the constructors of the *Connection* class, and also the constructors of the subclasses *Local* and *LongDistance*, which causes the *after* advice to be executed 3 times instead of only once. This fault is not detected by the AOSV criteria since DCT-AJ does not measure coverage in exception handling code, and executing the advice 3 times instead of once does not cause a failure.

Covering the AOSV DUAs helps in detecting faults of type F1-2 because they

Table 8.16: Effectiveness of the AOSV test criteria in detecting faults of type F1-2

Class	# Mut	all-uses$_a$	all-uses$_o$	all-uses$_c$	all-uses$_{as}$	all-uses$_{ma}$	all-uses$_s$
Kettle	4	100	100	100	100	100	100
Local	6	na	62.2	72.2	63.3	65.6	83.3
LongDistance	6	na	68.9	79.4	62.8	66.1	83.3
CarSimulator	2	na	100	100	100	na	100
All	**18**	**100**	**77**	**83.9**	**75.4**	**74.4**	**88.9**

```
//original
after (Connection c): target(c) && execution(Connection.new(..)){
      c.timer = new Timer();
}

//Mutant
after (Connection c) returning: target(c) && execution(*.new(..)){
      c.timer = new Timer();
}
```

Figure 8.2: A subtle mutant of type F1-2 from the *Timing* aspect, which is woven with the *Local* and *LongDistance* classes

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*$_c$ criterion is more effective than *all-uses*$_o$, *all-uses*$_{as}$, and *all-uses*$_{ma}$ because it requires covering paths between non-advised methods. The differences between the effectivenesses of *all-uses*$_o$, *all-uses*$_{as}$, and *all-uses*$_{ma}$ are small and vary with the number of DUAs required by the criteria. Finally, since *all-uses*$_a$ only requires the coverage of DUAs in one class that have faults of type F1-2 (the *Kettle* class), the effectiveness of *all-uses*$_a$ is compared with the effectiveness of the other AOSV criteria only in the *Kettle* class, in which all the criteria killed all the mutants.

Table 8.17 shows the average mutation scores of the AOSV-adequate test suites for faults resulting from pointcut that match a superset of the intended join points (fault type F1-3). The table is organized in the same way as Table 8.15. The table shows that for all the classes, *all-uses$_s$* test suites killed all the mutants of type F1-3, except one mutant in the *Local* and *LongDistance* classes. The live mutant, which is shown in Figure 8.3, is generated by operator PWIW and is the same mutant that is used to generate the HOM of type F1-2 shown in Figure 8.3. The mutant caused the *after* advice to be executed 3 times instead of only once.

Table 8.17: Effectiveness of the AOSV test criteria in detecting faults of type F1-3

Class	# Mut	all-uses$_a$	all-uses$_o$	all-uses$_c$	all-uses$_{as}$	all-uses$_{ma}$	all-uses$_s$
Kettle	2	100	100	100	100	100	100
Local	5	na	53.3	62	63.3	64	80
LongDistance	5	na	66	77.3	63.3	64	80
CarSimulator	6	na	66.7	100	66.7	na	100
CruiseController	3	na	78.9	83.3	72.2	na	100
All	**21**	**100**	**68.3**	**83.2**	**69**	**70**	**90.5**

```
//original
after (Connection c): target(c) && execution(Connection.new(..)){
       c.timer = new Timer();
}

//Mutant
after (Connection c): target(c) && execution(*.new(..)){
       c.timer = new Timer();
}
```

Figure 8.3: A subtle mutant of type F1-3 from the *Timing* aspect which is woven with the *Local* and *LongDistance* classes

The results of the other AOSV criteria show that *all-uses$_c$* is more effective than *all-uses$_o$*, *all-uses$_{as}$*, and *all-uses$_{ma}$* because it requires covering paths between non-advised methods. The *CarSimulator* class contains an example of 2 faults that involve unintended join points and require covering the cDUAs in the class. In the

CarSimulator class, both *all-uses$_o$* and *all-uses$_{as}$* were unable to kill 2 mutants of type F2-3, which *all-uses$_c$* killed. These 2 mutants caused advices that accelerate the car speed and turn the brakes on to execute *after* the class constructor. Killing these two mutants requires covering the cDUAs between the constructor and the *getter* methods of the 2 state variables.

Table 8.18 shows the average mutation scores of the AOSV-adequate test suites for faults resulting from pointcut that match a subset of the intended join points (fault type F1-4). The table is organized in the same way as Table 8.15. The table shows that about 56% of the mutants are killed by *all-uses$_s$* and 50% in 5 out of 7 classes. In these 5 classes, all mutants of type F1-4 are generated by operator POAC, which generated two mutants for each *after* advice by (1) changing *after* to *after returning*, and thus, matching only normal method returns, and (2) changing *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 *all-uses$_s$* 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 *after throwing* path to be executed).

Table 8.18: Effectiveness of the AOSV test criteria in detecting faults of type F1-4

Class	# Mut	all-uses$_a$	all-uses$_o$	all-uses$_c$	all-uses$_{as}$	all-uses$_{ma}$	all-uses$_s$
Kettle	4	50	50	50	50	50	50
Local	14	na	30.7	50	50	50	50
LongDistance	14	na	46.2	50	50	50	50
Customer	4	na	na	45.8	50	na	50
CarSimulator	10	na	41	50	46.3	na	50
CruiseController	8	na	71.7	73.8	68.3	na	87.5
SpeedControl	3	66.3	54.4	66.7	66.7	na	66.7
All	**57**	**57.1**	**54.4**	**53.9**	**52.8**	**50**	**56.1**

The effectiveness scores of the remaining AOSV criteria are close to each other. In some classes that have few test requirements of a criterion (e.g., *all-uses*$_o$ in the *Local* and *LongDistance* classes), the criterion effectiveness decreases since the criterion requires executing few paths in the advised class.

Table 8.19 shows the average mutation scores of the AOSV-adequate test suites for faults resulting from pointcut that does not match any join point (fault type F1-5). The table is organized in the same way as Table 8.15. The *all-uses*$_s$ 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 *all-uses*$_s$) 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 *defs* and *uses* in the advices are not executed, causing the fault to propagate in many paths.

Table 8.19: Effectiveness of the AOSV test criteria in detecting faults of type F1-5

Class	# Mut	all-uses$_a$	all-uses$_o$	all-uses$_c$	all-uses$_{as}$	all-uses$_{ma}$	all-uses$_s$
Kettle	6	93.3	94.4	100	91.7	89.9	100
Account	1	100	100	100	100	100	100
Local	20	na	63	65.3	67	67.5	100
LongDistance	20	na	67.5	66.3	66.8	69.8	100
Customer	8	na	na	93.8	100	na	100
CarSimulator	5	na	82	100	93.3	na	100
CruiseController	1	na	100	100	100	na	100
SpeedControl	2	100	100	100	100	na	100
All	**63**	**95.6**	**72.5**	**77.5**	**77.7**	**71.9**	**100**

Table 8.20 shows the average mutation scores obtained by the test suites that satisfy each of the AOSV criteria for all faults in the pointcut descriptor category.

Column 1 in the table shows the AOSV test criteria. For each criterion, column 2 shows the total number of mutants that are generated in the classes. Column 3 shows the number of classes tested by the criterion. Column 4 shows 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.

Table 8.20: Effectiveness of the AOSV test criteria in detecting faults in category F1

Criteria	# Mutants	# Classes	Average Mutation Score (%)
all-uses$_a$	22	3	84.5
all-uses$_o$	181	8	65.2
all-uses$_c$	199	8	73.8
all-uses$_{as}$	199	8	70.2
all-uses$_{ma}$	139	4	68.6
all-uses$_s$	199	8	85.1

The *all-uses$_a$* criterion has a higher mutation score than the other AOSV criteria (excluding *all-uses$_s$*). However, *all-uses$_a$* is tested only on three classes in which the mutation scores of all the AOSV criteria are high. Therefore, we cannot conclude that the test suites that cover the aDUAs are more effective than the test suites that cover other types of AOSV DUAs. However, test suites that satisfy *all-uses$_c$* are more effective than the test suites that satisfy *all-uses$_o$*, *all-uses$_{as}$*, and *all-uses$_{ma}$*. We summarize the effectiveness results of the AOSV criteria in detecting faults in category F1 as follows:

- Covering all the AOSV DUAs can help detect faults of the types in category F1, especially for faults of types F1-1, F1-2, F1-4, and F1-5, for which the mutation score of the test suites that satisfy *all-uses$_s$* ranges from 88.9% to 100%.

- A set of faults of type F1-4 are not detected by the AOSV criteria because killing these mutants requires executing paths that include exceptions. These mutants contribute 25 out of the 26 mutants that the test suites that satisfy the *all-uses*$_s$ criterion never killed.

- The *all-uses*$_c$ criterion is more effective than *all-uses*$_o$, *all-uses*$_{as}$, and *all-uses*$_{ma}$ in detecting faults in the types that causes the pointcut to match unintended join points (i.e., faults of type F1-1, F1-2, and F1-3). This is because *all-uses*$_c$ requires executing paths between non-advised methods.

- The differences between the mutation scores of *all-uses*$_o$, *all-uses*$_{as}$, and *all-uses*$_{ma}$ (and also *all-uses*$_a$ in the *Kettle* class) are small and depend whether the criteria require covering DUAs that require executing the missed intended advices or the matched unintended ones.

8.3.2 Aspect Declaration Related Faults (F2)

We discuss in this section the effectiveness of the test criteria in detecting faults of each type in category F2. As we showed in Section 7.7.3, mutants generated for the programs fall into 3 types: F2-5, F2-7, and F2-8.

Faults of type F2-5 (incorrect aspect precedence) are generated by two mutation operators: (1) DAPC, which changes the aspect precedence declared in the program, and (2) DAPO, which removes the aspect precedence declaration from the program. Two of the subject programs *Kettle* and *Telecom*, have precedence rules. The average mutation scores of the AOSV-adequate test suites in detecting faults of type F2-5 are given in Table 8.21. The table is organized in the same way as Table 8.15. All the AOSV-adequate test suites killed all the mutants of type F2-5, except one subtle mutant in the *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.

Table 8.21: Effectiveness of the AOSV test criteria in detecting faults of type F2-5

Class	# Mut	all-uses$_a$	all-uses$_o$	all-uses$_c$	all-uses$_{as}$	all-uses$_{ma}$	all-uses$_s$
Kettle	2	50	50	50	50	50	50
Local	2	100	100	100	100	100	100
LongDistance	2	100	100	100	100	100	100
Customer	2	100	100	100	100	100	100
All	**8**	**87.5**	**87.5**	**87.5**	**87.5**	**87.5**	**87.5**

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

Table 8.22 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 table is organized in the same way as Table 8.15. The test suites that satisfy *all-uses*$_s$ killed all the mutants. This is because swapping an *after* advice with a *before* advice affects many AOSV DU paths in the program since all the advices have some *uses* or *defs* of the state variables.

Table 8.22: Effectiveness of the AOSV test criteria in detecting faults of type F2-7

Class	# Mut	all-uses$_a$	all-uses$_o$	all-uses$_c$	all-uses$_{as}$	all-uses$_{ma}$	all-uses$_s$
Kettle	2	68.3	100	83.3	90	63.3	100
Local	5	na	38.7	56	54	52.7	100
LongDistance	5	na	37.3	54.7	57.3	55.3	100
CruiseController	8	na	97.5	87.5	81.3	na	100
All	**20**	**68.3**	**68**	**71**	**69.3**	**55.6**	**100**

Results for the remaining AOSV criteria show small differences in their effectivenesses in detecting faults of type F2-7. *All-uses$_o$* is more effective than the remaining AOSV criteria in two classes: *Kettle* and *CruiseControl*. This is because executing the path between a *def* in a method and a *use* in the advice that advises the method guarantees that the path that contains the fault is executed. We also found that when the mutant of type F2-7 is in an advice that contains a *use* of an oDUA, then the fault is always detected. In the *LongDistance* and *Local* classes, the effectiveness of all-uses$_o$ is low because the classes have only 2 oDUAs that are covered by executing one path.

Table 8.23 shows the average mutation scores of the AOSV-adequate test suites for faults resulting from binding an advice to an incorrect pointcut (fault type F2-8). Faults of type F2-8 were generated by operator ABPR that swaps the pointcut bound to the advice with other pointcuts in the aspect. Depending on the pointcut selected by the operator, the mutant might match any set of join points as described in category F1. This is reflected in the results of the AOSV test criteria in detecting faults of type F2-8. Note that *all-uses$_c$* is more effective than the other criteria (except *all-uses$_s$*) because it requires covering paths between non-advised methods.

Table 8.23: Effectiveness of the AOSV test criteria in detecting faults of type F2-8

Class	# Mut	all-uses$_a$	all-uses$_o$	all-uses$_c$	all-uses$_{as}$	all-uses$_{ma}$	all-uses$_s$
Kettle	2	100	73.3	80	100	90	100
Local	13	na	65.1	79.2	77.4	77.4	92.3
LongDistance	13	na	78.2	86.2	77.7	78.2	91
Customer	13	na	na	66.7	100	na	100
CarSimulator	20	na	71.8	91.3	85.7	na	97.2
CruiseController	27	na	70	77	68.6	na	100
All	**72**	**100**	**71.4**	**82.4**	**79.6**	**78.7**	**96.2**

The results of the *all-uses$_s$* criterion shows that covering all the AOSV DUAs is effective in killing all the mutants, except 3 subtle mutants, which we discuss below. Two mutants in the *Local* and *LongDistance* classes are not always killed by the test suites that satisfy *all-uses*$_s$. These two mutants cause the *Timer* object in the *Local* and *LongDistance* classes to log the value of the state variable *stopTime* (i.e., the time when the call stops) at the beginning of the phone call and to log the value of state variable *startTime* at the end of the call. In order to check the logged values, the test cases need to cover the DUAs in the *Timer* class, which are not in the test requirements of the *Local* and *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.

Table 8.24: Effectiveness of the AOSV test criteria in detecting faults in category F2

Criteria	# Mutants	# Classes	Average Mutation Score (%)
all-uses$_a$	6	3	72.8
all-uses$_o$	93	7	71.4
all-uses$_c$	100	8	80.5
all-uses$_{as}$	100	8	78.2
all-uses$_{ma}$	46	3	73.3
all-uses$_s$	100	8	96.3

Table 8.24 shows the average mutation scores obtained by the test suites that satisfy each of the AOSV criteria for all faults in category F2. The table is organized in the same way as Table 8.20. We summarize the results for the effectiveness of the AOSV criteria in detecting faults in category F2 as follows:

- Covering all the AOSV DUAs can help detect faults of the types in category F2. The test suites that satisfy *all-uses*$_s$ obtained an average mutation score of 96.6% in all faults of category F2.

- A subtype of faults of type F2-8 are not always detected by the test suites that satisfy *all-uses*$_s$ because detecting these faults requires covering DUAs in the classes of the referenced objects.

- The *all-uses*$_c$ criterion is more effective than the *all-uses*$_o$, *all-uses*$_{as}$, *all-uses*$_{ma}$ criteria in detecting faults of type F2-8 because the criterion requires executing paths between non-advised methods.

8.3.3 Advice, Aspect Method, and Intertype Method Implementation Faults (F3)

We discuss in this section the effectiveness of the AOSV test criteria in detecting faults of the types in category F3. Table 8.25 shows the average mutation scores of the AOSV-adequate test suites for faults resulting from an incorrect guarding statement or missing *proceed* in *around* advices (fault type F3-1). The table is organized in the same way as Table 8.15. The table does not show classes from the *Telecom* program because the program does not contain *around* advices.

The results show that the test suites that satisfy *all-uses*$_s$ killed all the mutants of type F3-1 except one mutant that is not always killed in the *CruiseController* class. This mutant is shown in Figure 8.4. The mutant changes the logical *and* operator (&&) to the logical *or* operator (||). The mutant cannot be killed when the two conditions evaluate to true.

```
//original
if (controller.controlState != controller.INACTIVE
   && sc.isMinCruiseSpeedReached() ) {

//Mutant
if (controller.controlState != controller.INACTIVE
    || sc.isMinCruiseSpeedReached() ) {
```

Figure 8.4: A hard to kill mutant of type F3-1 from the *SpeedControlIntegrator* aspect which is woven with the *CruiseController* class

The results show that all of the AOSV test criteria are effective in detecting faults of type F3-1. *All-uses*$_a$ is more effective than the other AOSV test criteria

Table 8.25: Effectiveness of the AOSV test criteria in detecting faults of type F3-1

Class	# M.	all-uses$_a$	all-uses$_o$	all-uses$_c$	all-uses$_{as}$	all-uses$_{ma}$	all-uses$_s$
Kettle	27	100	94.7	96.8	97.7	94.1	100
Account	10	100	95.3	53.5	68.7	67.3	100
CarSimulator	3	na	75.6	100	93.3	na	100
CruiseController	16	na	91.7	93.8	84.6	na	94
All	**56**	**100**	**92.4**	**88.3**	**88.5**	**86.8**	**98.3**

because all the *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 *proceed*).

All-uses$_c$ is not effective in the *Account* class because most of the cDUAs in the class are between *getter* and *setter* methods. The results also show that *all-uses$_{ma}$* in the *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 *proceed*.

Table 8.26: Effectiveness of the AOSV test criteria in detecting faults of type F3-2

Class	# M.	all-uses$_a$	all-uses$_o$	all-uses$_c$	all-uses$_{as}$	all-uses$_{ma}$	all-uses$_s$
Kettle	38	95	93.8	93.9	94.3	91.7	96.8
Account	28	82.6	80.7	31.5	70.6	70.1	85.7
Local	8	na	79.2	90	82.1	80	100
LongDistance	8	na	80	88.3	81.3	78.8	100
Customer	20	na	na	99.2	100	na	100
CarSimulator	8	na	70	91.7	92.9	na	91.7
CruiseController	17	na	89.4	92.4	85.7	na	93.9
SpeedControl	17	93.1	89.6	90	80	na	93.3
All	**144**	**90**	**86.3**	**81.2**	**86.3**	**81.9**	**94.4**

Table 8.26 shows the average mutation scores of the AOSV-adequate test suites for faults resulting from incorrect altering of base class object state variables (fault type F3-2). The table is organized in the same way as Table 8.15. The results show that *all-uses$_s$* is effective in detecting faults of type F3-2. The few subtle mutants that are not killed by the test suites that satisfy *all-uses$_s$* do not always

produce a failure. We show an example from the *Kettle* program of such mutants in Figure 8.5. The figure shows a mutant that alters the value of state variable *status* in the *after* advice in the *HeatControl* aspect. Note that when the state variable *status* is not OFF, the fault does not propagate because of the *defs* that get executed if the condition holds. If the status is OFF, then the fault might not also propagate depending on the method called after the advice is executed. We found 8 such mutants in the programs.

```
//original
if (t.status != OFF) {
    if (t.temperature >= 100)
         t.status = HOT;
    else t.status = HEATING;

//Mutant
if (t.status-- != OFF) {
    if (t.temperature >= 100)
         t.status = HOT;
    else t.status = HEATING;
```

Figure 8.5: A hard to kill mutant of type F3-2 from the *HeatControl* aspect which is woven with the *Kettle* class

Figure 8.6 shows another example of a mutant which is not always killed by the test suites that satisfy *all-uses*$_s$. The example represent a type of mutants in which the fault occurs in intertype *getter* methods for intertype state variables. The AOSV DUAs require executing paths to the *use* in the intertype method but there is no requirement to execute any method after the test case calls the *getter* method. In the test cases, the assertions are always executed at the end, and therefore, there is no guarantee that the intertype state variable will have a *use* after the mutated *getter* method is called. Moreover, the intertype methods cannot be called from the methods in the class. Detecting such faults requires executing the intertype methods by other classes (i.e., inter-class data-flow testing).

```
//original
public int Kettle.readTemperature()
{
     return temperature;
}

//Mutant
public int Kettle.readTemperature()
{
     return temperature++;
}
```

Figure 8.6: A hard to kill mutant of type F3-2 from an intertype method in the *HeactControl* aspect which is woven with the *Kettle* class

The results of the other AOSV test criteria show that there are small differences in their effectivenesses 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 *all-uses*$_c$ in the *Account* class.

Table 8.27: Effectiveness of the AOSV test criteria in detecting faults of type F3-3

Class	# M.	all-uses$_a$	all-uses$_o$	all-uses$_c$	all-uses$_{as}$	all-uses$_{ma}$	all-uses$_s$
Kettle	30	94	89.3	92	91	89.1	94.7
Account	34	91	91.1	26.7	80.7	78	94.1
Local	22	na	80.3	84.5	77.4	77	91.4
LongDistance	22	na	78.9	86.5	80	78.3	90
Customer	4	na	na	100	100	na	100
CarSimulator	6	na	82	94.4	90.6	na	94.4
CruiseController	34	na	90.7	92.9	82.6	na	94.3
SpeedControl	29	81.7	75.2	81.1	70	na	95.2
All	**181**	**89.1**	**85**	**76.9**	**81.3**	**81**	**93.7**

Table 8.27 shows the average mutation scores of the AOSV-adequate test suites for intra-advice level faults (fault type F3-3). The table is organized in the same way as Table 8.15. The results show that *all-uses*$_s$ is effective in detecting faults

of type F3-2. The few mutants that are not killed by the test suites that satisfy *all-uses*$_s$ require test cases that include boundary values for the input domains for the state variables. For example, killing the mutant shown in Figure 8.7 requires testing the value of state variable *temperature* of the *Kettle* class to have the value 100 that reach the mutated *if* statement. While we provided RANDOOP with s set of input values to be used by the test cases, RANDOOP does not guarantee that all these input values will be used for each state variable and there is no guarantee that the test suites that satisfy the AOSV test criteria will contain the test cases with the boundary values.

```
//original
if (t.temperature >= 100)
    t.status = Hot;
else t.status = HEATING;

//Mutant
if (t.temperature > 100)
    t.status = HOT;
else t.status = HEATING;
```

Figure 8.7: A mutant of type F3-3 that require testing with boundary input values. The mutant is from the *HeatControl* aspect which is woven with the *Kettle* class

The results of the other AOSV test criteria show that there are small differences between their effectivenesses in detecting faults of type F3-3. The effectiveness of the AOSV test criteria 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. Finally, we show the average mutation scores obtained by the test suites that satisfy each of the AOSV test criteria for all faults in category F3 in Table 8.28. The table is organized in the same way as Table 8.20.

Our results for the effectiveness of the AOSV criteria in detecting faults in

Table 8.28: Effectiveness of the AOSV test criteria in detecting faults in category F3

Criteria	# Mutants	# Classes	Average Mutation Score (%)
all-uses$_a$	37	3	91.3
all-uses$_o$	357	7	86.7
all-uses$_c$	381	8	80.2
all-uses$_{as}$	381	8	84.3
all-uses$_{ma}$	227	4	82.3
all-uses$_s$	381	8	94.7

category F3 show the following:

- The average mutation scores of the test suites that satisfy the *all-uses$_s$* criterion range from 93.7% for fault type F3-3 to 98.3% for fault type F3-1. Therefore, covering the AOSV DUAs can help in detecting a high percentage of the faults in category F3.

- Covering the aDUAs helps more than covering the other AOSV DUAs in detecting faults of type F3-1 because covering the aDUAs requires executing the paths that contain the call to *proceed*.

- Detecting faults of type F3-2 that occur in *getter* intertype methods requires performing intra-class data-flow level testing, which is not a requirement for the AOSV criteria.

- Detecting some faults of type F3-3 requires performing boundary testing.

8.3.4 Class Implementation Related Faults (F4)

We discuss in this section the effectiveness of the AOSV test criteria in terms of detecting faults of the types in category F4. Table 8.29 shows the average mutation scores of the AOSV-adequate test suites for faults that result from passing an object in an unexpected state to an advice (fault type F4-1). The table is organized in the same way as Table 8.15. The results show that the *all-uses*$_s$ criterion is effective in detecting faults of type F4-1. The few subtle mutants that are not always killed by the test suites that satisfy *all-uses*$_s$ occurred in *getter* methods where the mutant altered the value of state variables. The results of the other AOSV test criteria show that *all-uses*$_c$ is more effective than *all-uses*$_o$, *all-uses*$_{as}$, and *all-uses*$_{ma}$ in all the classes because it requires covering paths between non-advised methods.

Table 8.29: Effectiveness of the AOSV test criteria in detecting faults of type F4-1

Class	# Mut	all-uses$_a$	all-uses$_o$	all-uses$_c$	all-uses$_{as}$	all-uses$_{ma}$	all-uses$_s$
Kettle	31	97.4	98	99.8	96	94.2	100
Account	33	85.8	87.6	87.8	30.6	28	87.9
Customer (B)	2	na	na	100	na	na	na
Local	8	na	50	84.6	62.5	36.3	87.5
LongDistance	8	na	51.3	86.3	55.8	42.5	88.8
Customer (T)	17	na	na	85.7	98.8	na	na
Call	6	na	na	100	na	na	na
Timer	14	na	na	100	na	na	na
CarSimulator	106	na	62.5	96.9	69.2	na	99.2
CruiseController	76	na	95.6	93.7	54.2	na	98.4
SpeedControl	84	96.2	83.2	94.5	88.7	na	100
All	**385**	**94.1**	**79.5**	**94.4**	**70.3**	**55.9**	**97.8**

Table 8.30 shows the average mutation scores of the AOSV-adequate test suites for faults that result from passing arguments to the advices that have incorrect values (fault type F4-2). The table is organized in the same way as Table 8.15. Faults of type F4-2 occurred only in the *Kettle* program because the advices in the other programs do not receive arguments from the base class. The results show that the 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 advices in the *Kettle* program are used in all the paths in the advices. Therefore, covering any of the paths in the advices is sufficient to detect these faults.

Table 8.31 shows the average mutation scores of the AOSV-adequate test suites

Table 8.30: Effectiveness of the AOSV test criteria in detecting faults of type F4-2

Class	# Mut	all-uses$_a$	all-uses$_o$	all-uses$_c$	all-uses$_{as}$	all-uses$_{ma}$	all-uses$_s$
Kettle	5	100	100	100	100	100	100

in detecting object-oriented faults (fault type F4-3). The table is organized in the same way as Table 8.15. The table shows that mutation scores of the test suites that satisfy the AOSV test criteria are low. The average mutation score of the *all-uses*$_s$ 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 JSI of μJava 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.

The mutation scores in Table 8.31 show that the test suites that satisfy the cDUAs are more effective than the test suites that cover other types of AOSV DUAs. Covering the cDUAs helped in killing the mutants of type F4-3 other than the ones generated by the JSI operator. These include mutants that remove or insert the keyword *this*, mutants that incorrectly initialize the state variables, and mutants that assign references with other comparable variables. These mutants are killed by the test suites that satisfy *all-uses*$_c$ because the criterion requires covering the paths that have *defs* and *uses* of the state variables in the class.

Table 8.32 shows the average mutation scores of the AOSV-adequate test suites for intra-method faults (fault type F4-4). The table is organized in the same way as Table 8.15. The results show that the test suites that satisfy *all-uses*$_s$ killed

Table 8.31: Effectiveness of the AOSV test criteria in detecting faults of type F4-3

Class	# Mut	all-uses$_a$	all-uses$_o$	all-uses$_c$	all-uses$_{as}$	all-uses$_{ma}$	all-uses$_s$
Kettle	3	0	0	0	0	0	0
Account	3	66.7	66.7	66.7	66.7	66.7	66.7
Customer (B)	3	na	na	5.6	na	na	na
Local	1	na	16.7	20	10	10	56.7
LongDistance	1	na	10	13.3	3.3	3.3	40
Customer (T)	6	na	na	18.9	15	na	18.9
Call	4	na	na	35	na	na	na
Timer	2	na	na	76.7	na	na	na
CarSimulator	17	na	16.8	60.6	16.8	na	60.6
CruiseController	10	na	2	13	2	na	13
SpeedControl	20	na	4	60	15	na	60
All	**70**	**14.6**	**11**	**43.1**	**10.4**	**26.7**	**45.4**

most of the mutants of 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$_c$* is the most effective criterion since it has more test requirements in the class.

Table 8.32: Effectiveness of the AOSV test criteria in detecting faults of type F4-4

Class	# Mut	all-uses$_a$	all-uses$_o$	all-uses$_c$	all-uses$_{as}$	all-uses$_{ma}$	all-uses$_s$
Kettle	6	100	98.9	100	47.8	90.6	100
Account	6	100	98.9	100	43.9	39.4	100
Local	5	na	16	82	70.7	50.7	77.3
LongDistance	5	na	36	77.3	47.3	54	80.5
Customer (T)	12	na	na	76.6	100	na	100
Call	13	na	na	66.7	na	na	na
Timer	7	na	na	100	na	na	na
CarSimulator	128	na	64.8	100	86.3	na	97
CruiseController	72	na	94.2	95.8	63.6	na	92.6
SpeedControl	84	91.5	86	96	91.2	na	100
All	**389**	**92.6**	**78.7**	**96**	**81.5**	**59.2**	**98.7**

Finally, we show the average mutation scores obtained by the test suites that satisfy each of the AOSV test criteria for all faults in category F4 in Table 8.28. The

Table 8.33: Effectiveness of the AOSV test criteria in detecting faults in category F4

Criteria	# Mutants	# Classes	Average Mutation Score (%)
all-uses$_a$	22	3	84.5
all-uses$_o$	181	8	65.2
all-uses$_c$	199	8	73.8
all-uses$_{as}$	199	8	70.2
all-uses$_{ma}$	139	4	68.6
all-uses$_s$	199	8	85.1

table is organized in the same way as Table 8.20. Our results for the effectiveness of the AOSV criteria in detecting faults in category F4 show the following:

- Test suites that cover all the AOSV DUAs can kill most of the mutants of type F4-1, F4-2, and F4-4. Faults of type F4-2 require test suites that satisfy intra-class data-flow interactions.

- Covering cDUAs can help in detecting faults of the types in category F4 more than the other AOSV DUAs. This is because covering the cDUAs requires executing more paths in a class.

8.4 Cost-Effectiveness of Achieving High Levels of Coverage for Criterion all-uses$_s$

We address the fourth research question in this section. The goal is to find whether it is cost-effective to achieve high levels of coverage for the *all-uses$_s$* criterion. We computed the cost and effectiveness of *all-uses$_s$* at 3 coverage levels: 100%, 90%, and 80% coverage levels. We measured the cost using only metrics c_1 and c_3 only because the other metrics (c_2 and c_4) are designed to compare the cost with other test criteria.

Table 8.34 shows in columns 2 through 4 the mean number of test cases in

Table 8.34: Means of the number of test cases in the test suites that satisfy *all-uses*$_s$ at 100%, 90% and 80% coverage levels

Class	Mean Test Suites Size			Increase (%)	
	100%	90%	80%	90% to 100%	80% to 90
Kettle	15.8	12.4	12	22	3
Account	11.4	10.2	9.2	11	10
Local	8.3	6.6	6.4	20	3
LongDistance	4.7	4.1	3.3	12	20
Customer	4.9	3.9	3.3	20	15
CarSimulator	54.7	48.7	43.9	11	10
CruiseController	34.3	30.2	25.7	12	15
SpeedControl	21.1	18.1	14.8	14	19
All	156.2	135.2	119.3	13	12

the test suites that satisfy *all-uses*$_s$ at 100%, 90%, and 80% coverage levels, respectively. Column 5 shows the increase in the number of test cases required to reach 100% coverage compared with the number of test cases that cover 90% of the AOSV DUSA. Column 6 shows the increase in the number of test cases required to reach 90% coverage compared with the number of test cases that cover 80% of the AOSV DUAs. For example, the second row shows that for the *Kettle* class (1) the number of test cases in the test suites at 100% coverage level is 22% higher than the number of test cases in the test suites at 90% coverage level, and (2) the number of test cases at the 90% coverage level is 3% higher than the number of test cases at the 80% coverage level.

The results in Table 8.34 show that the number of test cases at 100% coverage level is more than 10% higher than the number of test cases at 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. This is expected since most of the DUAs that can be executed by common are covered with test suites with lower coverage levels. In other words, the DUAs that get covered only at high coverage levels are the most costly in terms of the number of test cases. Note than 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%.

Table 8.35 shows in columns 2 through 4 the mean effort metric c_3 for obtaining the test suites at 100%, 90%, and 80% coverage levels, respectively. Column 5 shows the increase in the effort of obtaining the test suites at 100% coverage level compared with the effort of obtaining test suites that cover 90% of the AOSV DUAs. Column 6 shows the increase in the effort of obtaining the test suites at 90% level compared with the effort of obtaining test suites at the 80% coverage level. For example, the second row shows that for the *Kettle* class (1) the effort of obtaining the test suites at 100% coverage level is 59% more than the effort of obtaining the test suites at 90% coverage level, and (2) the effort of obtaining the test suites at the 90% coverage level is 2% more than the effort of obtaining the test suites at the 80% coverage level. The results show that covering the last 10% of the AOSV DUAs requires 40% higher effort compared with the effort of covering the AOSV DUAs at the 90% level. These results are expected because the hard to cover 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. Our 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 highly increases the cost.

Table 8.35 shows in columns 2 through 4 the means of the mutations scores for the test suites at 100%, 90%, and 80% coverage levels, respectively. Column 5 shows the increase in the means of the mutations scores for the test suites at

Table 8.35: Means of the effort of obtaining the test suites that satisfy *all-uses*$_s$ at 100%, 90% and 80% coverage levels

Class	Mean Effort Metric c_3			Increase (%)	
	100%	90%	80%	90% to 100%	80% to 90
Kettle	214	88	87	59	2
Account	2249	1542	1071	31	31
Local	911	256	232	72	10
LongDistance	21	10	8	55	12
Customer	22	9	8	58	16
CarSimulator	10671	6402	5634	40	12
CruiseController	11082	6982	6423	37	8
SpeedControl	925	444	395	52	11
All	3262	1967	1732	40	12

Table 8.36: Means of the mutation scores for the test suites that satisfy *all-uses*$_s$ at 100%, 90% and 80% coverage levels

Class	Mean Mutation Score			Increase (%)	
	100%	90%	80%	90% to 100%	80% to 90
Kettle	94.4	93.4	85.6	1	8
Account	89.6	88.3	80.5	1	9
Local	86.6	83.2	63.8	4	23
LongDistance	81.4	77.8	58.9	4	24
Customer	93.5	90.8	73.7	3	19
CarSimulator	96.5	94.8	89.6	2	6
CruiseController	94.2	93.1	82.9	1	11
SpeedControl	95.1	94.3	76.2	1	19
All	92.4	91.3	79.3	1	15

100% coverage level compared with the means of the mutations scores of test suites that cover 90% of the AOSV DUAs. Column 6 shows the increase in the means of the mutation score of the test suites at 90% coverage compared with the means of the mutation scores at 80% coverage. The results shows that only 1% more mutants are killed by the test suites that cover all the AOSV DUAs compared with 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.

The reason that explains why covering the last 10% of the AOSV DUAs killed only a few more mutants is that most of the faults in the mutants 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. Note that the *Local*, and *LongDistance* classes have more mutants that were only killed by the test suites that cover all AOSV DUAs compared to the other classes. These are the mutants that have faults in the method that drops a call. Detecting these mutants requires checking the state of the call after it has been dropped. Note that few test cases called the *drop* method, and thus, the test suites need to include the hard- to-cover AOSV DUAs in order to detect these faults.

While our 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 40% more effort.

8.5 Threats to Validity

We identify three types of threats to the validity of our empirical study: internal validity, external validity, and construct validity. Internal validity is concerned with cause and effect relationships, the extent to which we can state that the changes in dependent variables are caused by changes in independent variables. We have five dependent variables in our study, which are the metrics that we used to measure the cost and effectiveness of the test criteria. We recognize two internal threats to validity, which 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 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 [21]. We recognize two external threats to the validity of our study, which are as follows:

1. We studied four programs 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, we plan to study additional programs with larger sizes.

Construct validity refers to the meaningfulness of measurements [36]. For the measurement of cost, a notable construct validity issue is the extent to which the four metrics we used to measure cost are adequate measures for cost. We measure two dimensions of the cost: the size of the test suites and the effort required to obtain a test suite that satisfies a test criterion. However, 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 [11]. We did not measure the cost of these components because we used tools to obtain them. The difference in the computational time of running the tools is negligible.

One 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. [7, 8] 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, we reduced this threat by performing an unbiased and systematic seeding of faults using a suitable set of mutant operators.

Chapter 9

Conclusions and Future Work

We presented a data-flow testing approach for aspect-oriented programs. The approach classifies five types of DUAs based on class state variables and proposes six test criteria, called AOSV test criteria, which require covering these DUAs.

We implemented a tool called DCT-AJ, which measures coverage of five types of DUAs. 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, in which we run the test suites that cover the test criteria, and generate coverage reports.

We revised existing fault models for AspectJ programs and proposed a fault model that (1) eliminates overlapping between fault types, and (2) includes 3 fault types that result from incorrect data-flow interactions in the program. These fault types are: (1) incorrect altering of the base class state variables, (2) passing an object in an unexpected state to an advice, and (3) passing arguments to the advices that have incorrect values.

We performed a cost-effectiveness study for the AOSV test criteria using four subject programs. The programs contain a variety of characteristics that can be present in aspect-oriented programs and cover different application domains. We seeded faults in the subject programs using mutation operators. We used three mutation tools, *AjMutator*,*Proteum/AJ*, and μJava. Since the current version of μJava does not support AspectJ, we used an indirect approach to seed faults in the aspect using μJava by decompiling the aspects into a class and mutating the decompiled class with μJava. The study shows that automated test generation tools for Java programs can be used for AspectJ programs if aspects are written using the @AspectJ annotation style and with suitable tool settings. Seeding of faults using mutation operators, however, requires applying more than one tool and manual intervention. Therefore, a tool that automates this process is needed for AO programs.

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

We used a test generation tool called RANDOOP to generate a pool of random test cases. To produce a test suite that satisfies a criterion, we randomly selected test cases from the test pool until required coverage for a criterion is reached.

We evaluated two dimensions of the cost of a test criteria and measured each dimension by two metrics. The first dimension is the size of the test suites that satisfies a test criterion, which we measured by (1) the number of test cases in the test suite, and (2) the number of test cases in the test suite divided by the number of test requirements for the criterion. The second cost dimension is the effort of obtaining a test suite that satisfies a criterion which we measured by (1) the number of iterations needed for randomly selecting test cases from the pool of test cases until a test criterion is satisfied, and (2) the number of iterations needed for randomly selecting test cases from the pool of test cases in order to cover a test requirement. Measuring the size and effort per requirement allows comparing the cost of the test criteria over all classes, which have a large variation in the number of test requirements for a criterion. We measured effectiveness by the mutation scores of the test suites that satisfy a criterion. We evaluated effectiveness for all faults and for each fault type in the revised fault model.

The empirical study compared the cost and effectiveness of the AOSV test 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 results for measuring the cost of the test criteria are as follows:

- The number of test cases in a test suite that satisfies a criterion depends on two factors: (1) the number of test requirements in a class for a criterion, and (2) the number of paths needed to cover the test requirements. Our results show that even if the AOSV test criteria require more paths, having few test requirements in the classes makes the number of test cases in the test suites that satisfy them not significantly higher than the number of the test cases 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 a higher number of test cases than each of the AO control-flow criteria. These are (1) *all-uses*$_a$, because it requires executing different paths for each of the aDUAs, (2) *all-uses*$_c$, because it has high number of test requirements in all the classes, and (3) *all-uses*$_s$, 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.

- The results of using metric c_2, which measures size by the number of test requirements that can be covered by a test case, show that 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.

- The effort of satisfying a test suite as measured by metric c_3 depends on two factors: (1) the number of test requirements in a class for a criterion, and (2) the number of test cases in the pool that can cover the test requirements of a criterion. Our results show that 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*$_a$, *all-uses*$_c$, and *all-uses*$_s$. When we compared the AOSV test criteria with each other, the differences in c_3 do not show that a certain type of AOSV DUAs require more effort to satisfy. However, the results show that 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 results of measuring the effort of covering a test criterion (i.e., metric c_4), show that 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. Our 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. RANDOOP was able to generate test cases that cover all the criteria with the help of two features of the tool. First, we did not set up a limit on the size of test cases, which allowed the tool to produce test cases that contain long sequences of different method calls. Second, we allowed RANDOOP to produce large number of test cases by running for a long time. This let RANDOOP produce test cases that can execute hard to cover paths.

Our results show that the AOSV test criteria are more effective than the control-flow criteria. Test suites that cover *all-uses*$_s$ detected 38% and 31% more faults than the test suites that cover the blocks and branches, respectively. Moreover, the test suites that 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.

The test suites that satisfy the *all-uses*$_s$ criterion obtained an average mutation score of 92.4% over all the classes. The *all-uses*$_s$ 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*$_s$ 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*$_s$, the *all-uses*$_c$ 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$_c$* 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$_s$*, which is more effective than the criteria it subsumes.

The results also show that the covering the data-flow for the state variable can detect most types of faults in AspectJ programs. The mutation scores of the test suites that satisfy *all-uses$_s$* 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).

In order to evaluate the cost-effectiveness of reaching high levels of coverage for the *all-uses$_s$* criterion, we evaluated cost and effectiveness of *all-uses$_s$* for three coverage levels: 100%, 90% and 80% coverage levels. Our results show that the test suites that cover satisfy *all-uses$_s$* at 100% coverage level are only 1% more effective than the test suites that cover 90% of the AOSV DUAs, and need 40% more effort and 13% 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.

We identified three directions to extend this work, which are as follows:

1. Investigation of other types of data-flow interactions in aspect-oriented programs: These include data-flow interactions within the aspects, data-flow interactions between classes, and data-flow interactions for the objects referenced by classes and aspects. Our results for state variable DUAs show that there is a type of faults that require covering such data-flow interactions.

2. Investigation of the use of mutation testing for aspect-oriented programs: Mutation testing has been widely used for procedural and object-oriented programs. However, the cost and effectiveness of mutation testing have not been evaluated for testing aspect-oriented programs.

3. Investigation of the use of HOMs for aspect-oriented programs: HOM is a promising approach that has not been evaluated for aspect-oriented programs. We used HOMs in our work to produce mutants of types that FOMs were not able to generate. Our work can be extended by investigating the use of 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 2, and subsuming HOMs.

REFERENCES

[1] Aynur Abdurazik and Jeff Offutt. Using UML Collaboration Diagrams for Static Checking and Test Generation. In *UML'00: The Third International Conference on the Unified Modeling Language*, pages 383–395, York, UK, October 2000.

[2] Aditya P. Mathur and Weichen E. Wong. An empirical comparison of mutation and data flow-based test adequacy criteria. *The Journal of Software Testing, Verification, and Reliability*, 4(1):9–31, March 1994.

[3] Roger T. Alexander, James M. Bieman, and Anneliese A. Andrews. Towards the Systematic Testing of Aspect-Oriented Programs. Technical Report CS-4-105, Department of Computer Science, Colorado State University, Fort Collins, Colorado, 2004.

[4] Roger T. Alexander and Jeff Offutt. Analysis techniques for testing polymorphic relationships. In *TOOLS 30: Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 104–114, Santa Barbara, California, August 1999.

[5] Roger T. Alexander and Jeff Offutt. Criteria for testing polymorphic relationships. In *ISSRE 2000: Proceedings of 11th International Symposium on Software Reliability Engineering*, pages 15–23, San Jose, California, October 2000.

[6] Prasanth Anbalagan and Tao Xie. Efficient Mutant Generation for Mutation Testing of Pointcuts in Aspect-Oriented Programs. In *ISSRE '08: 19th International Symposium on Software Reliability Engineering*, Seattle, Washington, November 2008.

[7] James H. Andrews, Lionel C. Briand, and Yvan Labiche. Is Mutation an Appropriate Tool for Testing Experiments. In *ICSE 05: Proceedings of the 27th international conference on Software engineering*, pages 402–411, St. Louis, MO, USA, May 2005.

[8] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, Augest 2006.

[9] Jon S. Baekken and Roger T. Alexander. A Candidate Fault Model for AspectJ Pointcuts. In *ISSRE'06: Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 169–178, Raleigh, North Carolina, November 2006.

[10] Robert Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.

[11] Lionel C. Briand. A critical analysis of empirical research in software testing. In *ESEM '07: First International Symposium on Empirical Software Engineering and Measurement*, pages 1–8, Madrid, Spain, September 2007.

[12] Lionel C. Briand, Yvan Labiche, and Jim (Jingfeng) Cui. Automated support for deriving test requirements from UML statecharts. *Software and Systems Modeling*, 4(4):399–423, November 2005.

[13] Ugo Buy, Alessandro Orso, and Mauro Pezze. Automated Testing of Classes. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 39–48, Portland, Oregon, 2000.

[14] Mariano Ceccato and Paolo Tonella Filippo Ricca. Is AOP code easier or harder to test than OOP code? In *WTAOP: Proceedings of the 1st Workshop on Testing Aspect-Oriented Programs, held in conjunction with the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, 2005.

[15] Mei-Hwa Chen and Howard M. Kao. Testing object-oriented programs - an integrated approach. In *ISSRE '99: Proceedings of the 10th International Symposium on Software Reliability Engineering*, pages 73–82, Boca Raton, Florida, November 1999.

[16] Romain Delamare, Benoit Baudry, Sudipto Ghosh, and Yves Le Traon. A Test-Driven Approach to Developing Pointcut Descriptors in AspectJ. In *ICST '09: Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, pages 376–385, Denver, Colorado, April 2009.

[17] Romain Delamare, Benoit Baudry, and Yves Le Traon. AjMutator: A Tool for the Mutation Analysis of AspectJ Pointcut Descriptors. In *ICSTW '09: Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, pages 200–204, Denver, Colorado, April 2009.

[18] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, 11(4):34–43, April 1978.

[19] Giovanni Denaro, Alessandra Gorla, and Mauro Pezzè. Contextual Integration Testing of Classes. In *FASE '08: Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering*, pages 246–260, Budapest, Hungary, April 2008.

[20] Giovanni Denaro, Alessandra Gorla, and Mauro Pezzè. DaTeC: Dataflow Testing of Java Classes. In *ICSE'09: Proceedings of the International Conference on Software Engineering (Tool Demo)*, Vancouver, Canada, May 2009.

[21] N. Fenton and S. Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* PWS Publishing Co., Boston, MA, second edition, 1999.

[22] Fabiano C. Ferrari, Jose C. Maldonado, and Awais Rashid. Mutation Testing for Aspect-Oriented Programs. In *ICST 08: International Conference on Software Testing, Verification, and Validation*, pages 52–61, Lillehammer, Norway, April 2008.

[23] Fabiano C. Ferrari, Elisa Y. Nakagawa, Awais Rashid, and José C. Maldonado. Automating the Mutation Testing of Aspect-Oriented Java Programs. In *AST '10: Proceedings of the 5th Workshop on Automation of Software Test*, pages 51–58, Cape Town, South Africa, May 2010.

[24] Phyllis G. Frankl and Oleg Iakounenko. Further empirical studies of test effectiveness. In *FSE '98: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 153–162, Lake Buena Vista, Florida, 1998.

[25] Phyllis G. Frankl and Stewart N. Weiss. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *TAV 4: Proceedings of the 4th symposium on Testing, analysis, and verification*, pages 154–164, Victoria, British Columbia, Canada, 1991.

[26] Dick Hamlet, Bruce Gifford, and Borislav Nikolik. Exploring dataflow testing of arrays. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*, pages 118–129, Baltimore, Maryland, May 1993.

[27] Mark Harman, Fayezin Islam, Tao Xie, and Stefan Wappler. Automated test data generation for aspect-oriented programs. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 185–196, Charlottesville, Virginia, USA, April 2009.

[28] Mary Jean Harrold and Gregg Rothermel. Performing data flow testing on classes. In *FSE '94: Proceedings of the ACM Symposium on the Foundations of Software Engineering*, pages 154–163, New Orleans, Louisiana, December 1994.

[29] Mary Jean Harrold and Mary Lou Soffa. Interprocedual data flow testing. In *TAV3: Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification*, pages 158–167, Key West, Florida, December 1989.

[30] Mary Jean Harrold and Mary Lou Soffa. Computation of interprocedural definition and use dependencies. In *ICCL 1990: International Conference on Computer Languages*, pages 297–306, New Orleans, Louisiana, March 1990.

[31] Mary Jean Harrold and Mary Lou Soffa. Selecting and using data for integration testing. *IEEE Software*, 8(2):58–65, March 1991.

[32] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 191–200, Sorrento, Italy, 1994.

[33] Ivar Jacobson. Use Cases and Aspects Working Seamlessly Together. *Journal of Object Technology*, 2(4):7–28, August 2003.

[34] Yue Jia and Mark Harman. Constructing Subtle Faults Using Higher Order Mutation Testing. In *SCAM '08: Proceedings of Eighth International Working Conference on Source Code Analysis and Manipulation*, pages 249 –258, Beijing, China, September 2008.

[35] Zhenyi Jin and Jefferson Offutt. Coupling-based criteria for integration testing. *Software Testing, Verification and Reliability*, 8(3):133–154, 1998.

[36] F. Kerlinger. *Foundations of Behavioral Research.* Harcourt Brace Jovaonvich College Publishers, Orlando, Florida, third edition edition, 1986.

[37] Ramnivas Laddad. *AspectJ In Action: Practical Aspect-Oriented Programming.* Manning Publications Co., 2003.

[38] Otávio A. Lazzarini Lemos, Fabiano C. Ferrari, Paulo C. Masiero, and Cristina V. Lopes. Testing Aspect-oriented Programming Pointcut Descriptors. In *Proceedings of the 2nd workshop on Testing aspect-oriented programs, held in conjunction with the International Symposium on Software Testing and Analysis (ISSTA'06)*, pages 33–38, Portland, Maine, July 2006.

[39] Otavio A. Lazzarini Lemos, Auri M. Rizzo Vincenzi, Jose C. Maldonado, and Paulo C. Masiero. Control and Data Flow Structural Testing Criteria for Aspect-Oriented Programs. *Journal of Systems and Software*, 80(6):862–882, June 2007.

[40] Cristina V. Lopes and Trung C. Ngo. Unit-Testing Aspectual Behavior. In *WTAOP: Proceedings of the 1st Workshop on Testing Aspect-Oriented Programs, held in conjunction with the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, Chicago, Illinois, March 2005.

[41] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. MuJava: An Automated Class Mutation System. *Journal of Software Testing, Verification and Reliability*, 15(2):97–133, June 2005.

[42] Vincenzo Martena, Politecnico Di Milano, Alessandro Orso, and Mauro Pezzé. Interclass Testing of Object Oriented Software. In *ICECCS '2: Proceedings of the International Conference on Engineering of Complex Computer Systems*, pages 135–144, Greenbelt, Maryland, December 2002.

[43] Massachusetts Institute of Technology. Randoop 1.2: The Randomized Unit Test Generator for Java. `http://people.csail.mit.edu/cpacheco/randoop`, April 2010.

[44] Philippe Massicotte, Linda Badri, and Mourad Badri. Towards a Tool Supporting Integration Testing of Aspect-Oriented Programs. *Journal of Object Technology*, 6(1):67–89, January-February 2007.

[45] Aditya P. Mathur and Weichen E. Wong. Reducing the Cost of Mutation Testing: An Empirical Study. *The Journal of Systems and Software*, 31:185–196, December 1995.

[46] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, January 2005.

[47] Jonathan Misurda, James Clause, Juliya Reed, and Bruce Childers Mary Lou Soffa. Demand-driven structural testing with dynamic instrumentation. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 156–165, St. Louis, Missouri, May 2005.

[48] Michael Mortensen and Roger T. Alexander. An approach for adequate testing of AspectJ programs. In *WTAOP: Proceedings of the 1st Workshop on Testing Aspect-Oriented Programs, held in conjunction with the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, 2005.

[49] Akbar S. Namin, James H. Andrews, and Duncan J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *ICSE 08: Proceedings of the 30th international conference on Software engineering*, pages 351–360, Leipzig, Germany, May 2008.

[50] Jeff Offutt and Aynur Abdurazik. Generating Tests from UML Specifications. In *UML'00: The Second International Conference on the Unified Modeling Language*, pages 416–429, Fort Collins, Colorado, October 1999.

[51] Jeff Offutt, Jie Pan, Kanupriya Tewary, and Tong Zhang. An experimental evaluation of data flow and mutation testing. *Software Practice and Experience*, 26(2):165–176, February 1996.

[52] Alessandro Orso. *Integration Testing of Object-Oriented Software*. PhD thesis, Politecnico Di Milano, 1998.

[53] Alessandro Orso and Mauro Pezzè. Integration testing of procedural object-oriented languages with polymorphism. In *TCS '99: Proceedings of the 16th International Conference on Testing Computer Software: Future Trends in Testing*, Washington, D.C., USA, June 1999.

[54] Thomas J. Ostrand and Elaine J. Weyuker. Data flow-based test adequacy analysis for languages with pointers. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 74–86, Victoria, British Columbia, Canada, October 1991.

[55] Carlos Pacheco, Shuvendu Lahiri, Michael Ernst, and Thomas Ball. Feedback-Directed Random Test Generation. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 75–84, Minneapolis, MN, USA, May 20-26 2007.

[56] Hemant D. Pande, William A. Landi, and Barbara G. Ryder. Interprocedural Def-Use associations in C programs. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 139–153, Victoria, British Columbia, Canada, October 1991.

[57] Hemant D. Pande, William A. Landi, and Barbara G. Ryder. Interprocedural def-use associations for c systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[58] Parasoft Jtest. Jtest manual version 4.5. online manual. `http://www.parasoft.com/`, October 2009.

[59] Sandra Rapps and Elaine J. Weyuker. Data flow analysis techniques for test data selection. In *ICSE '82: Proceedings of the 6th international conference on Software engineering*, pages 272–278, Tokyo, Japan, September 1982.

[60] Sandra Rapps and Elaine J. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

[61] Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *FSE '04: Proceedings of the Twelfth International Symposium on the Foundations of Software Engineering*, pages 147–158, Newport Beach, California, November 2004.

[62] Atanas Rountev, Scott Kagan, and Thomas Marlowe. Interprocedural Dataflow Analysis in the Presence of Large Libraries. In *CC '06:International Conference on Compiler Construction*, pages 2–16, Vienna, Austria, March 2006.

[63] Amie L. Souter and Lori L. Pollock. The Construction of Contextual Def-Use Associations for Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 29(11):1005–1018, 2003.

[64] Amie L. Souter, David Shepherd, and Lori L. Pollock. Testing with Respect to Concerns. In *ICSM 2003: Proceedings of the 19th IEEE International Conference on Software Maintenance*, pages 54–63, September 2003.

[65] Dominik Stein, Stefan Hanenberg, and Rainer Unland. A uml-based aspect-oriented design notation for aspectj. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 106–112, 2002.

[66] The AspectJ Team. AspectJ Compiler 1.5.4. `http://www.eclipse.org/aspectj/`. October 2008.

[67] Arie van Deursen, Marius Marin, and Leon Moonen. A Systematic Aspect-Oriented Refactoring and Testing Strategy, and its Application to JHotDraw. Technical Report SEN-R0507, Delft University of Technology, Delft, Netherlands, 2005.

[68] Fadi Wedyan and Sudipto Ghosh. A Joinpoint Coverage Measurement Tool for Evaluating the Effectiveness of Test Inputs for AspectJ Programs. In *ISSRE '08: 19th International Symposium on Software Reliability Engineering*, pages 207–212, Seattle, Washington, November 2008.

[69] Weichen E. Wong and Aditya P. Mathur. Fault Detection Effectiveness of Mutation and Data Flow Testing. *Software Quality Journal*, 4(1):69–83, 1995.

[70] Weichen E. Wong. *On Mutation and Data Flow.* PhD thesis, Purdue University, West Lafayette, Indiana, December 1993.

[71] Tao Xie, Darko Marinov, and David Notkin. Rostra: a framework for detecting redundant object-oriented unit tests. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, pages 196–205, September 2004.

[72] Tao Xie and Jianjun Zhao. A Framework and Tool Supports for Generating Test Inputs of AspectJ Programs. In *AOSD '06: Proceedings of the 5th International Conference on Aspect Oriented Software Development*, pages 190–201, Bonn, Germany, March 2006.

[73] Dianxiang Xu and Junhua Ding. Prioritizing State-Based Aspect Tests. In *ICST '10: Proceedings of the 2010 International Conference on Software Testing Verification and Validation*, pages 265–274, April 2010.

[74] Dianxiang Xu and Weifeng Xu. State-based incremental testing of aspect-oriented programs. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 180–189, March 2006.

[75] Dianxiang Xu, Weifeng Xu, and W. Eric Wong. Testing Aspect-Oriented Programs with UML Design Models. *International Journal of Software Engineering and Knowledge Engineering*, 18(3):413–437, May 2008.

[76] Dianxiang Xu, Weifeng Xu, and W. Eric Wong. Automated Test Code Generation from Class State Models. *International Journal of Software Engineering and Knowledge Engineering*, 19(4):599–623, June 2009.

[77] Guoqing Xu and Atanas Rountev. Regression Test Selection for AspectJ Software. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 65–74, 2007.

[78] Guoqing Xu and Atanas Rountev. AJANA: a general framework for source-code-level interprocedural dataflow analysis of AspectJ software. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 36–47, March 2008.

[79] Sai Zhang and Jianjun Zhao. On Identifying Bug Patterns in Aspect-Oriented Programs. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 431–438, July 2007.

[80] Jianjun Zhao. Data-Flow-Based Unit Testing of Aspect-Oriented Programs. In *COMPSAC '03: Proceedings of the 27th Annual International Conference on Computer Software and Applications*, pages 188–198, Dallas, Texas, November 2003.

[81] Jianjun Zhao. Control-Flow Analysis and Representation for Aspect-Oriented Programs. In *QSIC '06: Proceedings of the Sixth International Conference on Quality Software*, pages 38–48, Beijing, China, October 2006.