A Dataflow Testing Approach for Aspect-Oriented Programs

Fadi Wedyan, Sudipto Ghosh
Department of Computer Science
Colorado State University
Fort Collins, Colorado, USA
{*wedyan,ghosh*}*@cs.colostate.edu*

***Abstract*—Dataflow testing of programs ensures the execution of data dependencies between locations in the code (1) where variables are assigned values, and (2) where these definitions are used. Such data dependencies are called Def-Use Associations (DUAs). In an aspect-oriented (AO) program, aspects and classes interact in several ways, such as (1) through parameters passed from advised methods in a class to advices in the aspect, or (2) by the reading or writing of class state variables in an advice. In this paper, we present a dataflow testing approach for AO programs that is based on class state variables. We identify various types of DUAs for such variables and propose a set of dataflow test criteria that require executing these DUAs. Our approach is implemented by a tool called DCT-AJ, which identifies the DUAs in an AspectJ program and computes the coverage obtained from a test suite. Preliminary results indicate that the test suites satisfying the proposed dataflow criteria are more effective in detecting faults than the test suites that satisfy block coverage criteria.**

***Keywords*-AspectJ, aspect-oriented programming, dataflow, test criteria, test coverage**

I. Introduction

Aspect-oriented software development (AOSD) is a programming paradigm that supports the modularization of crosscutting concerns. AOSD is used in industrial frameworks, such as JBoss and Spring Framework, which are, in turn used to develop complex, critical applications.

AspectJ [1] 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 *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 [1].

Several structural testing approaches have been proposed for AO programs (e.g., [2], [3], [4], [5], [6]). They focus on defining control and dataflow test criteria, or generating test inputs to satisfy certain criteria. For example, Lemos et al. [3] proposed a test criterion called *All-crosscutting-uses*, which requires covering dataflow interactions based on parameter passing from the advised methods in the base class to the advices.

A key problem with existing dataflow test criteria is that they do not consider all types of dataflow interactions in an AO program. In this paper, we propose covering dataflow interactions that are based on state variables. Alexander et al. [7] recognized one type of faults that are caused by such interactions. An example of this fault is an aspect that changes the state variables in a way that violates the class invariant. Other faults can result from causes such as: (1) arguments passed to the advices have incorrect values, and (2) a base class object used in an advice is in an unexpected state.

Xu and Rountev [8] proposed a framework for inter-procedural dataflow analysis of AspectJ programs called AJANA, which uses an Interprocedural Control Flow Graph (ICFG) for AO programs. The ICFG represents paths containing calls to methods from a given method. However, the graph does not represent dataflow between base class public methods. Zhao [6] created a framed ICFG from the aspect and the base class to define three levels of dataflow testing (intra-module, inter-module, and intra-aspect or intra-class). However, in AspectJ, instances of the base class are passed to the aspect and the state variables of the object may be defined and used in the aspect advice. Zhao's framed ICFG does not support such state variables. Zhao's approach (which to our knowledge is not implemented in a tool) also does not support dynamic pointcuts, around advices, and multiple advices applied to the same join point, while AJANA handles them all.

We extended the ICFG obtained from AJANA by adding frame nodes and edges to represent dataflow interactions between public methods. We used the framed ICFG to define five types of state variable based dataflow interactions. We propose dataflow test criteria called aspect-oriented state variable test criteria, which require covering these interactions. We implemented a tool, called Dataflow Coverage Tool for AspectJ (DCT-AJ), which measures dataflow coverage.

We evaluated the proposed approach by comparing the cost and effectiveness of the proposed dataflow criteria with the *all-nodes* criterion (i.e., block coverage) using a subject program that we developed. We developed a mutation environment for AspectJ using μJava [9], AjMutator [10], and a Java decompiler to insert mutation faults in the subject program. Our results show that while the dataflow criteria are more effective, they also require more test cases.

The rest of this paper is organized as follows. AspectJ concepts and terminology are described in Section II using an illustrative example. In Section III, we describe the proposed approach. The dataflow coverage tool is presented in Section IV. Results of our evaluation are provided in Section V. We summarize related work in dataflow testing of AO programs in Section VI. Finally, conclusions and future work are discussed in Section VII.

II. AspectJ and Running Example

This section describes the concepts and terminology related to AspectJ. We provide a code example in AspectJ, which we use later to describe our dataflow testing approach.

AspectJ provides a set of primitive pointcut expressions, called *designators*, which can be used to target the desired *join points*. A *pointcut* expression consists of one or more pointcut expressions combined using logical operators. A *pointcut* is used by an advice, which is 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* a *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*; *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.

Figure 1 shows the Java class *Kettle*, which simulates the functionality of an electric kettle for heating water. The class has methods for adding water and pouring water from the kettle. Kettle objects can be in 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 and 3 *affect* it. A class is *affected* by an aspect if: (1) an aspect advises one or more methods in the class, (2) an aspect introduces one or more methods or state variables to the class, or (3) the aspect changes the class inheritance hierarchy.

```
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 1. Kettle Class.

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

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

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

```
public aspect HeatControl {
   public int Kettle.temperature;
   pointcut pcConst(Kettle t): target(t) &&
              execution(Kettle.new(..));
   pointcut pcPour(Kettle t): target(t) &&
              execution(* Kettle.pourWater(..));
   pointcut pcAdd(Kettle t): target(t) &&
              execution(* Kettle.addWater(..));
   after(Kettle t): pcConst(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. HeatControl Aspect.

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

Figure 3. SafetyControl Aspect.

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.

III. AO Dataflow Test Criteria

Dataflow testing ensures that the definition of variables and their subsequent uses are exercised. A definition, *def*, of a variable *v* occurs in a node of a control flow graph (CFG) where *v* is given a value; a *use* of *v* occurs in a node where *v* is accessed. For a variable *v*, a *definition-use association* (DUA) is a triple <*d,u,v*> 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 [11].

Dataflow 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 [11].

We define dataflow criteria for AO programs by using a framed Inter-procedural Control Flow Graph (ICFG). The ICFG is constructed from the CFG's of methods and advices in the AspectJ program. We obtain the ICFG of the AspectJ program using AJANA [8].

A. How AJANA Works

AJANA [8] 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. 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 *call-site* and *return-site* nodes.

The ICFG shows what methods and advices are invoked from a single call to each method and constructor of the 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.

B. Extending AJANA

Obtaining intra-class DUAs requires having paths between the CFGs of the public methods of the class (whether advised or not). For OO programs, Harrold and Rothermel [12]

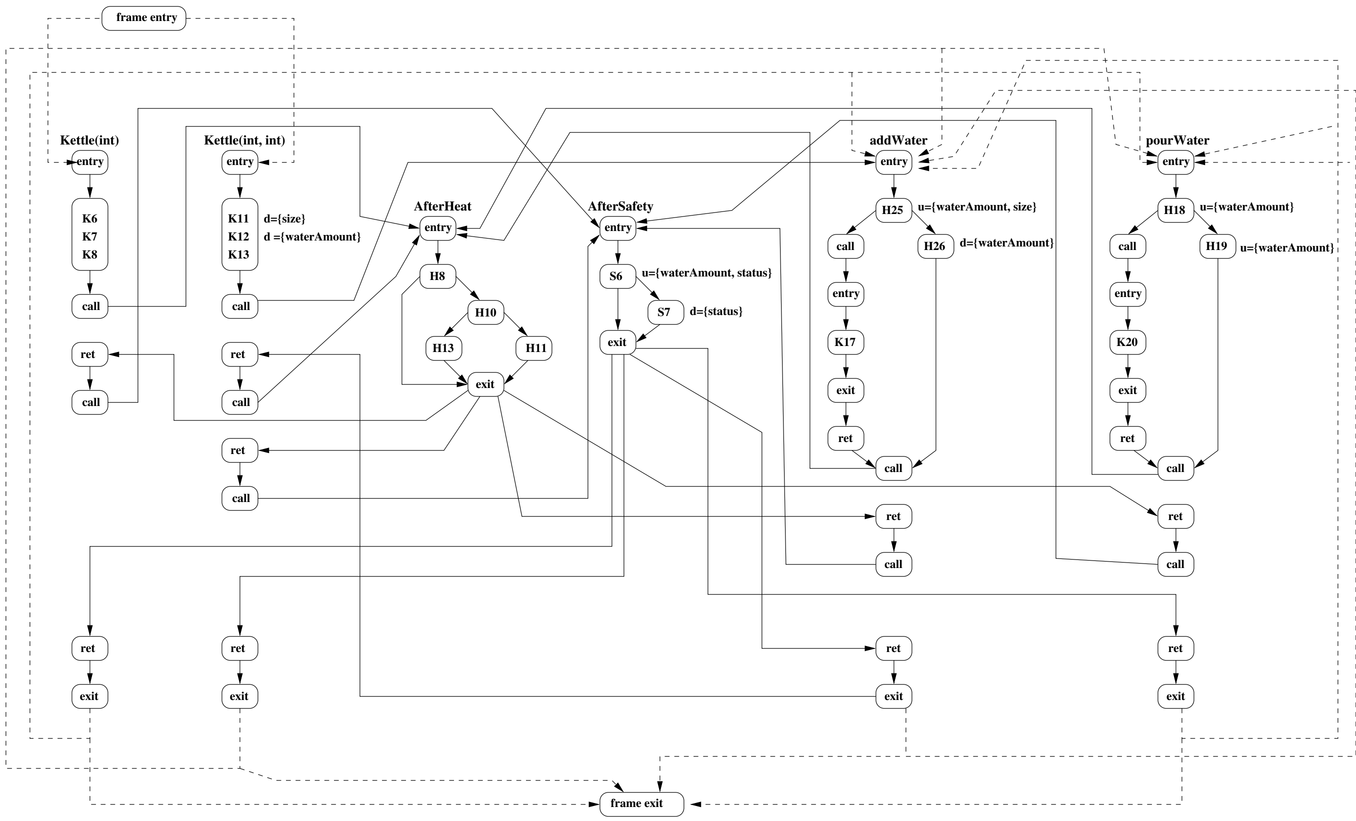

Figure 4. Framed ICFG of the *Kettle* Program.

proposed the use of a frame that provides paths between public methods. For AO programs, Zhao [6] proposed the use of a frame for the ICFG created for the class and the aspect. The frame can be viewed as a test driver that 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 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.

C. Proposed AOP Dataflow Criteria

Our proposed dataflow test criteria require covering interactions that are based on state variables. The criteria also require covering dataflow interactions between aspects in an AO program. We take into account the scope classification described by Rinard et al. [13] and define the following DUAs for state variables in AO programs.

Observation DUAs (*oDUAs*): Advices may use state variables that the methods define. In Figure 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, H24*>, and <*size, K11, H24*> 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.

Activation DUAs (*aDUAs*): Methods may use state variables that the advices define. For example, in Figure 4, both the *around* advices have *defs* for the state variable *waterAmount* that can be reached in method *addWater*. Therefore, *<waterAmount, H25, K17>*, *<waterAmount, H18, 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.

Class DUAs (*cDUAs*): In an AO program, DUAs of state variables may exist in the nodes that only belong to the base class methods. In Figure 4, the *def* of *waterAmount* in K12 reaches its *use* in K17. *<waterAmount, K12, K17>* is a *cDUA*.

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.

Aspect DUAs (*asDUAs*): In an AO program, DUAs of state variables may exist in the nodes that belong to advices and methods of the same aspect. In Figure 4, the *def* of *waterAmount* in H25, reaches its *use* in H24 in aspect *HeatControl*. *<waterAmount, H25, H24>* is an *asDUA*.

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.

Multiple Aspects DUAs (*maDUAs*): An AO program may contain DUAs for state variables in advices and aspect methods that belong to different aspects. In Figure 4, the *def* of *waterAmount* in H25 of aspect *HeatControl* reaches the *use* in statement S5 of aspect *SafetyControl*.

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.

From the above types of DUAs, we define the following AO state variable test criteria:

1) *All-uses-observation (all-uses$_o$)*: Requires exercising all *oDUAs* in the AO program at least once.
2) *All-uses-activation (all-uses$_a$)*: Requires exercising all *aDUAs* in the AO program at least once.
3) *All-uses-class (all-uses$_c$)*: Requires exercising all *cDUAs* in the AO program at least once.
4) *All-uses-aspect (all-uses$_{as}$)*: Requires exercising all *asDUAs* in the AO program at least once.
5) *All-uses-multiple-aspects (all-uses$_{ma}$)*: Requires exercising all *maDUAs* in the AO program at least once.
6) *All-uses-state (all-uses$_s$)*: Requires satisfying the *all-uses$_o$*, *all-uses$_a$*, *all-uses$_c$*, *all-uses$_{as}$*, and *all-uses$_{ma}$* criteria.

IV. Prototype Tool Implementation

Our dataflow coverage tool called DCT-AJ calculates the coverage metrics discussed in Section III. Figure 5 shows the steps involved in using DCT-AJ and how the components inside the tool interact. DCT-AJ works in three phases: DUA identification, program instrumentation, and test execution.

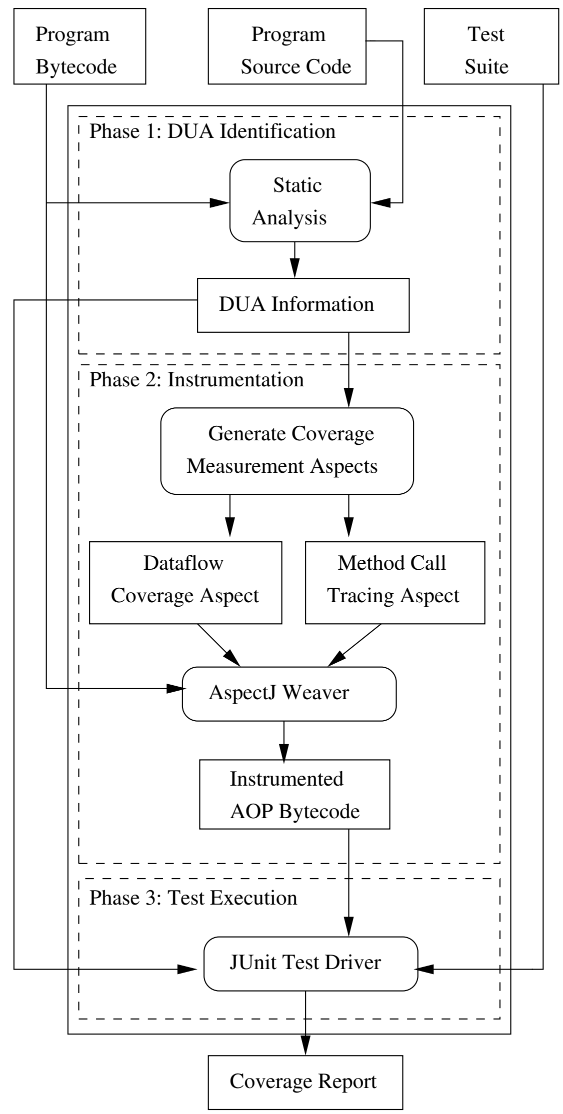

Figure 5. DCT-AJ: Data-flow Coverage Measurement Tool.

A. 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 III).

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 III to the ICFG.

3) Process defs and uses: The ICFG produced by AJANA identifies *defs* and *uses* of variables in each node of the ICFG. DCT-AJ parses this ICFG and builds a list of *defs* and *uses* for each state variable, where a *def* or a *use* is defined by a triple that consists of (1) the class or aspect in which the *def* or *use* resides, (2) the method or advice name that uses or defines the state variable, and (3) the statement number which contains the *def* or *use*. We changed the way AJANA deals with variables of array types; it considers every access to an array element (whether *def* or *use*) as a *use* of the variable. Accessing an array element is handled by two Jimple statements. 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 dataflow algorithm proposed by Pande et al. [14] 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* complier 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 Library (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 III. 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.

B. 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.

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 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 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. Method Call Tracing Aspect for the Kettle class.

2) ***Dataflow 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.

[1]http://abc.comlab.ox.ac.uk
[2]http://abc.comlab.ox.ac.uk
[3]http://jakarta.apache.org/bcel

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. [15]. 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 <*def,use,sv*> 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 7 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.

```
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 7. Dataflow Coverage Pointcuts for the Kettle class.

C. *Phase 3: Test Execution*

Given a test suite, the instrumented bytecode of the classes, and the DUA information of the classes under test, the test driver runs the test suite. The *Dataflow Coverage Aspect* saves coverage information in the form of coverage reports at the end of the run. A report is generated for each class. The report includes the number of state variables, the number of 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.

V. Evaluation

The goal of this study was to evaluate the cost and effectiveness of the proposed dataflow criteria. For comparison, we chose the *all-nodes* (statement coverage) criterion. We used the size of the test suite required to satisfy a criterion as a measure of the cost. The size of a test suite is the number of test cases in a test suite. We measured effectiveness using the number of faults detected in the subject program.

We performed the study on the *Kettle* program described in Section II. Even though the *Kettle* program is small, it contains many important attributes of AO programs (e.g., *around* advices, multiple advices applied on the same join point, and intertype declarations). We added get methods for the *Kettle* class state variables because the test generation tool needed to obtain the values of the state variables.

A. *Fault Seeding*

Fault in an AspectJ program can occur either in (1) the implementation of the base class, (2) the pointcut descriptors, and (3) the implementation of the advices, aspect methods, and introduced methods [16].

1) Seeding of pointcut faults: We used *AjMutator* developed by Delamare et al. [10] to seed pointcut faults. This tool implements 10 of the 15 mutant operators proposed by Ferrari et al. [16]. We used *AjMutator*'s capabilities for identifying equivalent mutants and non-compilable mutants.

2) Seeding of base class faults: We used μJava [9] to seed faults in the base class. μJava uses two types of mutation operators, class level and method level. In our experiment, we applied all the operators implemented by μJava.

3) Seeding of faults in the aspects: Since there is a lack of tools that can directly seed faults into the advice, aspect methods, and introduced methods, we had to use an indirect approach. First, we generated a class from the aspect bytecode using a decompilation tool (e.g., Jad[4]). The generated class was then mutated with μJava. The mutated line was then copied into the aspect. Note that μJava do not support annotations, otherwise, we would had used @AspectJ annotation style.

Table I
SUMMARY OF GENERATED MUTANTS

	AjMutator	μJava
Generated mutants	51	156
Equivalent mutants	17	24
Non-compilable mutants	30	0
Remaining mutants	4	132

Table I summarizes the mutants generated for the *Kettle* program. There were 136 mutants that were used for testing.

Most of the mutants generated with *AjMutator* were either equivalent or non-compilable. Several equivalent mutants were produced because there was only one class in the program. For example, mutation operators that insert the wildcard (*) in the package name, class name, or method names in the *Kettle* program always generated equivalent mutants. Non-compilable mutants were created because of the existence of dynamic pointcuts. For example, mutation operators that replace logical operators or insert a negation (!) in the pointcuts of the *HeatControl* aspect leaves the targeted object unbounded, which is not allowed in AspectJ.

μJava does not report the mutants that failed to compile, and thus manual inspection was used to determine the equivalent mutants. Only 7 from the 15 method level operators produced mutants in the program and class operators produced only 7 mutants. This was because of the simplicity of the *Kettle* program.

B. Test Case Generation

We used RANDOOP [17] to generate a large pool of random test cases. RANDOOP generates new test cases by randomly selecting a method to call and finding arguments from among previously found inputs.

RANDOOP generates JUnit test cases using the bytecode of Java classes. RANDOOP is not designed for AspectJ programs and we encountered two problems. First, since advices, intertype declarations, and declare precedence statements are compiled into methods in the bytecode, RANDOOP would generate calls for these constructs. These calls had to be removed. We solved this problem by setting RANDOOP to generate calls only for methods that belong to the class. Second, RANDOOP did not generate calls to introduced methods because these methods are not declared in the class. This issue was also reported by Xie and Zhao [5], where they used other test generation tools for Java. They suggested using a wrapper class that encapsulates each class and contains a wrapper method for each method in the base class, including intertype methods. However, test cases that use the wrapper mechanism cannot be used for integration testing since other classes use the original class rather than the wrapper one. To solve this problem, we rewrote the aspects in the *Kettle* program using the annotation style, which is a new feature of AspectJ 5, also known as @AspectJ annotation. Using this feature, we can write aspects with regular Java syntax and then annotate the aspect declarations so that they can be interpreted by the AspectJ weaver. In @AspectJ annotation style, intertype methods are declared in an interface that the class implements. Thus, the intertype methods appear in the class declaration and RANDOOP can generate calls for them.

We set RANDOOP to run for 30 seconds and save each five test cases in one file. We provided RANDOOP with a set of input test values. Using these settings, we generated a pool of 2845 test cases. From this pool, we produced 35 test suites such that each criterion is satisfied by 5 different test suites. To produce a test suite, we randomly selected files from the test pool until full coverage for a criterion was reached. Since the current version of RANDOOP does not allow setting the post-conditions for methods, we manually added JUnit assertions inside the selected test cases.

C. Results

Table II
COST SUMMARY

Criterion	No. Req	Test suite sizes					Avg. Size
all-uses$_o$	16	25	15	5	35	30	22
all-uses$_a$	10	40	25	30	20	45	32
all-uses$_c$	12	25	20	30	65	40	36
all-uses$_{as}$	10	60	40	30	25	35	38
all-uses$_{ma}$	6	5	45	10	20	20	20
all-uses$_s$	54	55	75	65	30	60	57
all-nodes	112	20	25	10	5	15	14

Table II summarizes the cost of using a test criterion. Column 2 shows the number of test requirements for each criterion. For the *all-nodes* criterion, this number is the lines of code in the program. For the other criteria, this is the number of DUAs. Columns 3 through 7 give the sizes of

[4]http://www.varaneckas.com/jad

Table III
EFFECTIVENESS RESULTS

Mutant Operator	No. of Mutants	all-uses$_o$		all-uses$_a$		all-uses$_c$		all-uses$_{as}$		all-uses$_{ma}$		all-uses$_s$		all-nodes	
		Mut. Score		Mut. Score		Mut. Score		Mut. Score		Mut. Score		Mut. Score		Mut. Score	
		Avg	Min	Avg	Min	Avg	Min	Avg	Min	Avg	Min	Avg	Min	Avg	Min
AOIS	57	90	60	96	95	97	95	99	96	85	74	99	95	71	49
AOIU	11	95	73	100	100	100	100	100	100	93	82	100	100	67	27
LOI	16	93	63	93	81	93	88	95	88	84	75	99	94	60	44
ASRS	8	85	25	98	88	100	100	100	100	83	63	100	100	60	50
ROR	23	96	91	97	96	96	91	97	96	97	91	100	100	90	87
COI	6	100	100	100	100	100	100	100	100	100	100	100	100	90	67
AORB	4	100	100	100	100	100	100	100	100	100	100	100	100	75	50
Class	7	91	57	94	71	97	86	100	100	91	71	100	100	83	71
PC	4	95	75	100	100	100	100	100	100	95	75	100	100	85	50
all	136	92	68	97	93	97	96	98	97	89	79	99	98	74	59

each test suite. Note that since each file in the test pool consists of 5 test cases, the sizes of the generated test suites are multiples of 5. The last column is the average size of the 5 test suites.

Table II shows that all the dataflow criteria cost more than the *all-nodes* criterion. Among the dataflow criteria, the *all-uses*$_{ma}$ costs the least and the *all-uses*$_s$ costs the most. These two criteria have also the least and the most test requirements, respectively. However, for the other criteria, there is no clear relationship between the number of the test requirements and the cost.

We executed every test suite on each mutant and determined whether or not the mutant was killed. We calculated the mutation score for every test suite with respect to every type of mutation operator, as well as for all the mutants. Table III shows the effectiveness results in terms of the mutation scores (expressed as a percentage). We show the average mutation score for the 5 test suites used for each criterion, as well as the minimum mutation score.

Among the dataflow criteria, *all-uses*$_{ma}$ was the least effective. The *all-uses*$_{ma}$ criterion requires covering DUAs between *HeatControl* and *SafetyControl*. The code in the after advice in *SafetyControl* is executed only when the kettle becomes empty. This state is not frequently produced by the mutants. So for this particular program, the *all-uses*$_{ma}$ criterion is less effective.

For mutants of categories COI (Conditional Operator Replacement) and AORB (Arithmetic Operator Replacement), using test suites for the dataflow criteria always killed all the mutants. These operators change the logical and arithmetic conditions in the code that result in setting the state variables incorrectly.

We observed that each mutant was killed by at least one of the 5 test suites that satisfy any particular dataflow criteria. This observation does not hold for the first five mutant categories when the *all-nodes* criterion was used. The mutants that were not killed required the execution of paths that are not necessary to achieve *all-nodes* coverage.

For lack of space, we are not including data on the length of the test cases (measured by lines of test code). RANDOOP can produce long test cases, which contain sequences of method calls that help achieve high coverage for the dataflow criteria. Therefore, such test cases reduce the cost of the dataflow criteria when the number of test cases is used as a measure of cost. On the other hand, having long sequences of method calls results in covering paths that are not required by the *all-nodes* criterion, which results in increasing the effectiveness of the *all-nodes* criterion as well. It is possible that other test generation techniques will have different costs.

VI. RELATED WORK

Zhao [6] introduced a dataflow unit testing approach for AO programs and defined two units of testing: (1) an aspect together with those methods whose behavior may be affected by the aspect's advices, and (2) a class together with the advices that may affect its behavior. The approach tests all dataflow interactions that occur within three levels of testing: *Intra-module*, *Inter-module*, and *Intra-aspect* or *Intra-class*. Zhao [6] created a framed ICFG for the each class and the aspect. However, the created graph does not consider instances of the base class objects passed to the aspect. Also, Zhao [6] did not define specific criteria or provide any implementation for program representation and test generation. Moreover, the approach does not handle *around* advices, dynamic pointcuts, and multiple advices applied to the same join point.

Rinard et al. [13] presented a classification system for AO programs. They identified four types of interactions that occur between methods and advices executed after a join point. However, the classification did not include interactions that might occur between an aspect and another aspect. We used Rinard et al.'s [13] classification in defining the types of state variable DUAs.

Lemos et al. [3] proposed three intra-procedural control and data flow criteria for testing AO programs. They developed a tool called *JaBUTi/AJ* that generates a dataflow graph for each module, where a module can be a method, an advised method, a constructor, an advice, or an intertype method. Lemos et al. [3] did not evaluate the cost and effectiveness of their criteria.

Delamare et al. [18] implemented a tool, called *AdviceTracer* that can determine at runtime what advice is executed and at which place in the base program. The *traceAdvices* pointcut we used in the *Method Call Tracing* aspect is similar to the pointcut that *AdviceTracer* uses.

Xu and Rountev [8] proposed a framework for source-code interprocedural dataflow analysis of AspectJ programs called AJANA. We used AJANA to generate the ICFG of the AO program.

VII. Conclusions and Future Work

We presented a dataflow testing approach for aspect-oriented programs. The approach classifies five types of DUAs based on class state variables and proposes six test criteria. We implemented a tool, called DCT-AJ, that measures the dataflow coverage for a test suite.

We performed a cost-effectiveness study for the proposed criteria. The results of the study show that the dataflow criteria were more effective than the *all-nodes* criterion but cost more. Our study has several limitations. First, the study was performed using one small program. While the program includes many features of AO programs, this program may not be representative of the general population. Second, the mutation operators for Java programs might not be representative of the faults that occur in AO programs.

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.

In the future, we will extend our evaluation to include larger programs. We will investigate different techniques for test generation and seeding faults in AO programs. We will also investigate the effectiveness of the test criteria on different types of AO related faults.

References

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

[2] M. Harman, F. Islam, T. Xie, and S. Wappler, "Automated test data generation for aspect-oriented programs," in *Proc. of AOSD*, Charlottesville, USA, April 2009, pp. 185–196.

[3] O. A. L. Lemos, A. M. R. Vincenzi, J. C. Maldonado, and P. C. Masiero, "Control and Data Flow Structural Testing Criteria for Aspect-Oriented Programs," *Journal of Systems and Software*, vol. 80, no. 6, pp. 862–882, June 2007.

[4] C. V. Lopes and T. C. Ngo, "Unit-Testing Aspectual Behavior," in *Proc. of the 1st Workshop on Testing Aspect-Oriented Programs, held in conjunction with the AOSD'05*, Chicago, USA, March 2005.

[5] T. Xie and J. Zhao, "A Framework and Tool Supports for Generating Test Inputs of AspectJ Programs," in *Proc. of AOSD*, Bonn, Germany, March 2006, pp. 190–201.

[6] J. Zhao, "Data-Flow-Based Unit Testing of Aspect-Oriented Programs," in *Proc. of COMPSAC*, Dallas, USA, November 2003, pp. 188–198.

[7] R. T. Alexander, J. M. Bieman, and A. A. Andrews, "Towards the Systematic Testing of Aspect-Oriented Programs," Dept. of Computer Science, Colorado State Univ., Fort Collins, USA, Tech. Rep. CS-4-105, 2004.

[8] G. Xu and A. Rountev, "AJANA: a general framework for source-code-level interprocedural dataflow analysis of AspectJ software," in *Proc. of the AOSD*, March 2008, pp. 36–47.

[9] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "MuJava: An Automated Class Mutation System," *Journal of Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, June 2005.

[10] R. Delamare, B. Baudry, and Y. L. Traon, "AjMutator: A Tool for the Mutation Analysis of AspectJ Pointcut Descriptors," in *Proc. of MUTATION workshop at ICST*, Denver, USA, April 2009, pp. 200–204.

[11] S. Rapps and E. J. Weyuker, "Data flow analysis techniques for test data selection," in *Proc. of ICSE*, Tokyo, Japan, September 1982, pp. 272–278.

[12] M. J. Harrold and G. Rothermel, "Performing data flow testing on classes," in *Proc. of FSE*, New Orleans, USA, December 1994, pp. 154–163.

[13] M. Rinard, A. Salcianu, and S. Bugrara, "A classification system and analysis for aspect-oriented programs," in *Proc. of FSE*, Newport Beach, USA, November 2004, pp. 147–158.

[14] H. D. Pande, W. A. Landi, and B. G. Ryder, "Interprocedural def-use associations for C systems with single level pointers," *IEEE Trans. on Software Engr.*, vol. 20, no. 5, pp. 385–403, May 1994.

[15] J. Misurda, J. Clause, J. Reed, and B. C. M. L. Soffa, "Demand-driven structural testing with dynamic instrumentation," in *Proc. of ICSE*, St. Louis, USA, May 2005, pp. 156–165.

[16] F. C. Ferrari, J. C. Maldonado, and A. Rashid, "Mutation Testing for Aspect-Oriented Programs," in *Proceedings of ICST*, Lillehammer, Norway, April 2008, pp. 52–61.

[17] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball, "Feedback-Directed Random Test Generation," in *Proc. of ICSE*, Minneapolis, USA, May 20-26 2007, pp. 75–84.

[18] R. Delamare, B. Baudry, S. Ghosh, and Y. L. Traon, "A Test-Driven Approach to Developing Pointcut Descriptors in AspectJ," in *Proc. of ICST*, Denver, USA, April 2009, pp. 376–385.