Testing is a fundamental issue for ensuring software quality. The characteristics of aspect-oriented programming may be sources for failures due to new kinds of faults in the code of aspects. New testing approaches and criteria, taking into account the aspect-oriented features and constructs, are needed to capture the new kind of program failures.

In this article the authors propose a set of testing coverage criteria based on interactions among the advices and the methods they affect. The proposed criteria are based on the interprocedural aspect control flow graph (IACFG), representing the interprocedural interactions among advices and methods. The results obtained in experiments carried out demonstrated the validity and effectiveness of the proposed criteria.

Key Words
aspect-oriented programming, software quality, software testing

SOFTWARE VERIFICATION AND VALIDATION

Testing Coverage Criteria for Aspect-Oriented Programs

MARIO LUCA BERNARDI
AND GIUSEPPE ANTONIO DI LUCCA
Department of Engineering, Research Center on Software Technology, University of Sannio

INTRODUCTION

Aspect-oriented programming (AOP) explicitly supports cross-cutting concerns by means of program units called aspects (Kiczales et al. 1997). The cross-cutting code, encapsulated in aspects, is woven into the code of traditional program units (for example, classes in object-oriented (OO) code) to build the overall system. The weaving process makes it possible to change structural properties of the basic system (such as the inheritance tree and interface implementation) and to inject the logic needed to catch points (join points) in a program execution in which the code in the aspects must be executed. Join points are defined by pointcuts that specify the conditions triggering the execution of advices (blocks of code in the aspects) that are to be run at the defined join points. Moreover, aspects can also introduce intertype declarations and alter the (normal and exceptional) control flow.

The specific features and constructs of AOP can make the testing of an AO system very difficult, due to the interactions among aspects and traditional modules and the ways these can take place.

On one hand, AOP allows one to simplify and enhance the system design and code structure by separating the cross-cutting concerns from the principal decomposition and encapsulating them in aspects. On the other hand, the ways in which aspects interact with and alter the code of the base system may make it difficult to identify and comprehend which portions of code are directly affected by the aspects, as well as which portions will be executed at a given point in the code.
Thus, the main difficulty in testing AOP systems comes from the interactions among aspects and classes and, in particular, among the advices and methods they affect. Indeed, these interactions are not identifiable when analyzing classes, and it is difficult to identify all the places (join points) an advice can have an effect just by analyzing aspects.

Due to the network of interactions among advices and methods, an AO system could resemble a new kind of “spaghetti bowl” with sources of failures that are difficult to find and remove. Thus, it might be difficult to know if all the interactions among the aspects and traditional modules have been considered (that is, covered) when performing the testing of an AOP system. A complete coverage of these interactions is needed to carry out any testing task to discover failures due to them.

Different kinds of faults may be in the code of pointcuts or advices, or of intertype declarations (Alexander, Biemann, and Andrews 2004), and different testing strategies are needed to identify them.

The definition of novel testing approaches (or the adaptation of existing ones), taking into account the specific features of AOP, is needed to identify the failures due to faults “hidden” into aspects. Such new approaches have to rely on adequate fault and test models allowing one to: 1) make assumptions about where faults are likely to be in the aspects; and 2) define complete and effective coverage criteria of the interactions among aspects and traditional programming units and among the aspects themselves.

In this article the authors propose a set of testing coverage criteria for AO programs based on the interactions among the advices and methods. The coverage criteria exploit the interprocedural aspect control flow graph (IACFG) (Bernardi and Di Lucca 2007), a graph representing the relationships among aspect advices and class methods. An extension of the fault model proposed by Alexander, Biemann, and Andrews (2004) has been defined as a fault model by considering the unique AOP characteristics and language constructs at a finer-grained level. The model is based on the interactions among the aspectual and traditional units and takes into account: 1) the dynamic behavior alterations caused by pointcut expressions and advice injection; and 2) nonsingleton aspect declarations. The testing approach proposed in this article is based on the AspectJ (Kiczales et al. 2001) implementation of AO systems.

RELATED WORK

Some work on the testing of AO programs has been carried out in recent years. Alexander, Biemann, and Andrews (2004) introduced a fault model for AOPs based on six types of faults. The model is not a complete testing approach, but it is a starting point for developing coverage criteria and testing strategies specific to AOP software. Ceccato, Tonella, and Ricea (2005) extend the model proposed by Alexander, Biemann, and Andrews (2004) by considering faults in exceptional control flow due to intertype polymorphic calls.

An approach based on two traditional testing techniques—white box coverage and mutation testing—is proposed by Mortensen and Alexander 2004. The approach mainly consists of discovering failures that are related to the code introduced by advices. The mutation operators are applied to the weaving process.

Massicotte, Badri, and Badri (2007) present an aspects-classes integration testing strategy and a tool supporting it. The approach consists of two main phases:

1) Static analysis: Generating testing sequences based on dynamic interactions between aspects and classes.

2) Dynamic analysis: Verifying the execution of selected sequences. The focus is on the integration of one or more aspects in the control of collaborating classes.

In van Deursen, Marin, and Moonen (2005) a fault model is proposed for identifying several basic faults due to intertype declarations, pointcut choice, pointcut’s conditional logic, and wrong use of patterns. Zhao proposed a data-flow-based approach to unit testing for AO programs (Zhao 2003). The approach allows one to perform testing at intramodule, intermodule, and intratatype levels. The approach is based on testing the aspects and classes that could be affected by one or more aspects. The method analyzes definition-use pairs to determine the interactions between aspects and classes to be tested. Souter, Amie, and Pollok (2000) proposed a concern-based test-selection technique, based on code instrumentation for collecting dynamic information. Xu and Xu (2005) proposed an approach to generate tests for exercising interactions between aspects and classes based on UML models. Moreover, Xu and Xu (2006) proposed a state-based testing approach exploiting aspect flow graphs.
In these proposals, however, no precise testing coverage criterion based on the interactions between advices and methods, like the one the authors propose, is defined. Testing approaches usually exploit some representation forms of programs to define testing methods and criteria. Some representation forms for AOP have been proposed in recent years, mainly based on control flow graphs (CFG) and call graphs (CG). Sereni and de Moor (2003) proposed a call graph for aspect-oriented programs in order to perform static analysis.

Zhao (2002) proposed an AO code representation based on an adaptation of the dependence graph to support the slicing of AO code. Zhao and Rinard (2003) proposed an extension of the same representation. More recently, Zhao proposed an interprocedural representation of AO systems, based on the representation presented in his previous works, to support program comprehension and maintenance (Zhao 2006).

Finally, Aldawud, Elrad, and Bader (2003); Coelho and Murphy (2004); Mostefaoui and Vachon (2006); and Stein, Hanenberg, and Unland (2002) described some models for AOP systems, mainly based on UML diagrams. None of these representation forms, however, is able to represent all together the various features of AOP, each contributing in a different way to making AO program testing difficult. In particular, the AOP constructs and features related to structural modification, field introduction, around advice, and aspect precedence are not considered and represented in some of them. These features are taken into account by the IACFG graph-based representation of an AO system (Bernardi and Di Luccia 2007) exploited by the testing approach proposed in this article.

A FAULT MODEL FOR AO PROGRAMS

A fault model makes assumptions about faults that are likely to be found in a system (Binder 1999). In AO programs the new kind of interactions between aspects and classes are sources for new kinds of faults that can produce system failures. The authors consider the ways in which aspects interact with the classes to alter the behavior or the structure of the system and, in particular, they refer to the following two kinds of alterations:

- **Interprocedural control-flow based alterations**, that is, alterations that happen at join points and are mainly related to the way aspects modify the interprocedural control flow of the original (not weaved) system.

- **System’s structure transformations**, due to intertype declarations that modify the structure of a class (or an interface) or the hierarchical structure among classes or interfaces.

Since all mainstream AOP languages have adopted a black-box join point model in which only interactions at module boundary can be intercepted, the fault model the authors consider is based on the interactions among methods and advices, for what concerns dynamic cross cutting.

These interactions are defined by the code specifying pointcuts and advices. Thus, faults are to be found in the pointcut expressions defining the join points and in the advices declarations. The types of failures due to faults in pointcuts or advice include:

a. **Picking up the wrong kind of elements to affect.** This is due to faults in pointcut expressions that do not select the right intended join points to which advices are to be applied. In this case, a fault usually is in the pointcut expression that could be “too strong” or “too weak.” If it is too strong, some intended join points will not be selected, while if it is too weak, a set of join points larger than the intended ones will be selected by the pointcut. For example, with reference to the code in Figure 1, if one modifies the expression of the pointcut p2() changing the “set(A.s)” predicate to “set(*.s)” the expression becomes weaker and picks up the definitions of “s” fields in the classes declaring a field named “s.”

b. **Wrong identification of the run-time context to apply the advice logic.** The right information to correctly determine the run-time context is known just at run time, while the selection of the join point is made at programming time. This means that it is complex for a programmer to select the right run-time context. Indeed, he or she must comprehend or know a complex dynamic behavior that could not be well known at programming time. Thus, programmers can introduce faults in selecting join points. For example, with reference to Figure 1, if in the p() expression the “AND” operator is changed with the “OR” one, the p()
expression will apply the before and around advices also in wrong contexts. Indeed, when the aspect X is enabled it will be applied in all join points since it always matches. Moreover, when the aspect is disabled, it will match for M2() calls, since the dynamic check fails at each join point.

c. Execution of an unintended advice at a join point. This is due to faults introduced by wrongly mapping pointcuts and advices. As shown in Figure 1, if one modifies the around advice and associates it to the pointcut p2() instead of the pointcut p(), the advice's logic will be applied in a wrong context.

d. Wrong order of advices execution at a join point. This is due to faults in defining the precedence rules among advices to be executed at a join point. The erroneous order of advices' execution will cause a failure in the system with respect to the specified one. In the example in Figure 1, if the declaration order of the before and around advices is inverted, the execution order of those advices also will be inverted. This means that the around advice will check for errors before any status field has been defined. The precedence rules based on the declaration order are very easy to break.

e. Wrong alteration of control and data flow at class scope. This is due to faults introduced in specifying when (that is, before, after, around) an advice has to be applied at the join point, or in the proceed condition of an around advice. In this case, the behavior resulting from the weaving will be different from the expected one. For example, referring to the after advice in line 16 of Figure 1, when changing the “after” declaration to a “before” one, the check is performed against a trivial or even undefined value.

Other sources of failures are due to the intertype and type hierarchy modifications declarations (binding interference). These include the wrong ways aspects can:

- Use privileged definition and use of private state variables
- Restrict, for a client, the access to a member

In this article, the authors deal only with the failures occurring in the interactions among advices and class methods due to faults in pointcut expressions and advices. An adequate test model representing the interactions among advices and methods must be used to support the testing of AO programs to discover failures due to these interactions. The IACFG (Bernardi and Di Luccia 2007) is used as a test model in the authors' approach, and the coverage criteria are defined with reference to it.

**FIGURE 1** A simple aspect-oriented program

```java
1 aspect X {
2 static bool isEnabled() {...}
3 static long l,m;
4 pointcut p() : call(void *.M2(...)) && if (X.isEnabled());
5 pointcut p2() : set(A.s);
6 before() : p() {
7   m = checkForErrors();
8   setEnabled();
9  }
10  void around() : p() {
11   if (m==1) notifyError();
12   proceed();
13  }
14 }
15 void after() : p2() {
16    verifyField();
17  }
18 }
19 }
20 class A {
21  ClassB b;
22  String s;
23  void M1(){
24    b.M2(); // matched by p() associated // to both advices in // aspect A. 
25    s = calculateString();
26  }
27  String calculateString() {...};
28 }
29 class B {
30  String s;
31  void M2() {
32    s = calculateString();
33  }
34 }
```
THE INTERPROCEDURAL ASPECT CONTROL FLOW GRAPH (IACFG)

The IACFG represents the relationships among class methods and aspect advices (from here on the term “operation” will be used to refer to methods and advices indifferently). In particular, the IACFG allows one to highlight the points in the control flow of an operation where the execution control is transferred to another operation. The IACFG is composed of a set of subgraphs, the operation control flow graphs (OCFG), each representing a (reduced) control flow graph of an operation.

A full description of the IACFG and OCFG is provided in Bernardi and Di Luca (2007), along with the rules to recover them by static analysis of the source code. Here, for sake of brevity, it is worthwhile to note that one distinguishes different kinds of nodes in an OCFG. A node in the OCFG can be:

- **An entry node** (EN), representing the first node of the OCFG of an operation. Each OCFG has just one EN.

- **A final node** (FN), representing the last node terminating the OCFG of an operation. Each OCFG has just one FN.

- **An exit node** (XN), representing the statement where the execution of an operation terminates, such as the “return” or nodes potentially terminating the execution of an operation because the control transfer toward advices that “not proceed.”

- **A statement node** (SN), representing traditional code statements (for example, assignment, branching, call (C), field reading (FR), field writing (FW), and so on).

- **A join-point shadow node** (JN) representing the join point where the control may be transferred toward an advice.

JN nodes correspond to no actual statement in the code; they are fictitious nodes inserted in the graph to show where a pointcut has an effect in the code.

An OCFG corresponding to a given operation is linked to the OCFG of a different operation, depending on the presence of statements or events (such as “call” statements, or the matching of pointcuts’ expressions) that transfer the execution control from the first one to the latter. Interprocedural directed arcs depict the control transfer between operations, that is, between the corresponding OCFGs.

**FIGURE 2** The IACFG of the program in Figure 1
Testing Coverage Criteria for Aspect-Oriented Programs

The IACFG is the graph resulting from the composition of all the OCFGs linked by the interoperation arcs. In particular, an interprocedural directed arc is between a JN node and the EN node of the OCFG corresponding to the advice woven at that join point. To reduce the complexity of an IACFG, just the OCFGs of operations (directly or indirectly) affected by aspects are reported in an IACFG.

Figure 1 shows an example of a simple AO program, while Figure 2 shows an excerpt of the IACFG representing it. In Figure 2, the numbers labeling each interprocedural arc indicate the order by which the interprocedural control flow is transferred among the operations.

A SET OF TESTING COVERAGE CRITERIA FOR AO SYSTEMS

The focus of the proposed testing approach is on the interactions among aspects and classes, thus it can be classified as integration testing. It aims to cover all the relationships, due to the implementation of static and dynamic cross-cutting features that exist among:

- Methods and advices affecting them
- Pointcuts and advices referring to them
- Aspects and classes whose structure is altered by intertype declarations in the aspects

Due to space constraints, just the definition of testing coverage criteria related to the dynamic cross-cutting features is presented and discussed in this article. The coverage criteria are defined with respect to the system components represented in the IACFG. In the IACFG the interactions due to dynamic cross cutting are represented by directed arcs from:

- JN nodes to the EN nodes of the OCFG corresponding to the advices that match the pointcut expressions triggering the execution of those advices
- FN nodes of the OCFG of an advice to the operation’s statements to be executed after the advice execution (the authors will call these arcs “returning arcs”)
- EN nodes of the OCFGs corresponding to advices to the pointcut nodes the advices refer to

Both named and implicit pointcuts are considered, that is, pointcuts explicitly declared and identified by a name used in the advices associated with them, and pointcuts whose declarations are embedded in the code of the advice itself. The coverage criteria will involve all of these structural elements of an IACFG.

The proposed coverage criteria have been derived from the traditional criteria for white-box unit testing (for example, node, branch, condition, and multiple conditions coverage criteria) (Beizer 1990) and adapted to the case of testing the interactions among aspects and classes.

- **Criterion C1:** All join-point shadow nodes. Each join-point shadow node in the IACFG must be exercised at least once by a test case. This does not mean that all the advices associated with each join point have to be executed. Indeed, just the advices matching the associated pointcuts’ expressions will be executed.

- **Criterion C2:** All join-point shadow branches. Each arc starting from a join-point shadow node in the IACFG must be exercised at least once by a test case. This means that the advices related to that join point have to be executed. The test case must exercise the returning arc, from the advice to the method. This criterion includes C1.

- **Criterion C3:** All pointcut node expressions. The criterion requires the definition and execution of test cases that make at least once “true” and at least once “false” the Boolean expression defined at each pointcut node. This does not mean that all the join-point shadow nodes or all branches from join-point shadow nodes are to be executed. Since a pointcut can define more join points in the code, it will be sufficient to exercise some of the join points and branches to satisfy the criterion. Satisfying the criterion C2, all the “true” values of pointcuts’ expressions are met.

- **Criterion C4:** All pointcut node conditions. If the Boolean expression defined at a pointcut is a composite one (that is, it is made of more Boolean conditions connected by operators OR, AND, and so on), then the criterion requires the definition and execution of test cases that make at least once “true” and at least once “false,” each condition in the composite expressions. Of course, this criterion includes C3.

- **Criterion C5:** All pointcut node multiple conditions. If the Boolean expression defined at a
pointcut is a composite one, test cases considering all the combinations of the “true” and “false” values for the conditions making up the pointcut expression are to be defined and executed for each composite pointcut node (that is, a truth table reporting all the possible combinations of the conditions involved in the pointcut expression must be built and test cases matching each row of this table are to be defined and executed). Of course, this criterion includes C4.

• Criterion C6: All advices. Each advice, that is, each OCFG corresponding to an advice, must be exercised by at least a test case. The test case must exercise the returning arc, from the advice to the method, as well as all the arcs related to any proceed statement in the advice code. The execution of all the advices does not imply the coverage of all the join-point shadow nodes (criterion C1) and the coverage of all the join-point shadow branches (criterion C2). Indeed, an advice may be associated with different join point shadow nodes, each one picked up by different pointcut expressions.

The defined criteria provide a useful guideline for defining test cases to discover failures when the control is transferred between classes and aspects, and where a possible fault is likely to be when a failure is detected. The criteria also provide a way to assess the quality of a suite of test cases by computing the percentage of coverage the suite is able to get for one or more criteria.

EXPERIMENTING THE PROPOSED TESTING COVERAGE CRITERIA

Some experiments have been carried out to assess the validity and effectiveness of the proposed testing criteria to identify failures in AO systems, and to support the localization of the faults generating the failures.

Some small- and medium-sized AO systems were considered in the experiments. First, the authors began by using small systems to more easily verify and validate the results. Next, they used larger systems and, in particular, they applied the proposed criteria to a portion of the AJHotDraw v0.4 system.

In the following paragraphs the main steps of the testing procedure followed in the experiments are described. First, the IACFG of the considered system was built using a prototype tool that statically analyzes the source code, builds the graph, and visualizes it (Bernardi and Di Lucca 2007). Then, to assess the effectiveness of the coverage criteria in discovering failures, some faults were seeded in the code of aspects.

The injected types of faults were selected among those that produce the type of failures reported earlier. To have failures of type (a), each pointcut expression was modified in two ways: first to miss some intended join points (that is, stronger expressions were generated) and second to include some unintended join points (that is, weaker expressions were generated). To produce failures of type (b) faults were seeded to change the runtime logic where to apply some advices. For failures of type (c), the injected faults were related to the same permutation of advice-pointcuts mapping. To generate failures of type (d), the advices’ precedence order was changed in some aspects to modify the correct sequence of the execution of advices. To generate failures of type (e) some other faults were inserted to modify when (that is, before, after, around) an advice had to be applied at a join point or in the proceed condition.

All kinds of faults were not introduced together at the same time. According to the aim of each test, just one kind of fault was injected each time. To record the coverage of the considered structural elements the systems were instrumented by hand.

Test cases were defined to satisfy the coverage criteria. In some cases it was necessary to compute a path condition to define the test cases reaching specified elements to cover. In other cases the test cases were generated by executing just the original (not fault seeded) system and capturing some values during the running.

In the following paragraphs the authors present and discuss the results of the testing for one of the small considered systems and for the analyzed portion of AJHotDraw version 0.4.

Results From the SeqGen System

The selected small system was developed by a graduate student at the University of Sannio. The system, called seqGen, implements a generator of sequences of “integer” and “char” values. It is based on two aspects to modularize generation policies (random and fibonacci series) and logging concerns. This system was selected because, despite its small size, it includes the major features of the AspectJ language. The system is composed of seven
classes, four aspects, and counts 346 lines of code (LOC). The four aspects define 24 introductions, six pointcuts (two abstract pointcuts, two overridden pointcuts, and two concrete pointcuts), and four advices (two after and two around ones). The AO portion of the system is 233 LOC. (The code of SeqGen is available at http://www.ing.unisannio.it/dilucca/seqgen.zip).

Figure 3 shows an excerpt of the IACFG of this small system. In Figure 3 each of the OCFGs is identified by a label with a capital letter indicating the concern the operation belongs to (“B” is for base system, “L” for logging, “F” and “R” for sequence generation) and a number corresponding to the number of the code line where the operation begins. Also, the numbers labeling the nodes correspond to the code line number. The bold dashed nodes (yellow colored) show the join-point shadows relative to the pointcut expressions in the system.

In particular, Figure 3 shows all the OCFGs corresponding to SequenceGenerator’s main method (label Bsm25) and some operations involved in the main execution. The two concerns—logging (labels La21 and La32) and sequence generation (labels Fa14, Fim37, Fm17, Ra15, and Rim31)—are considered in the excerpt; the graph shows the interactions among traditional methods and fields.

The authors note that the IACFG allows one to easily identify and follow the interactions between traditional class methods and aspect-oriented components, thus allowing the identification of the structural elements to cover.

**FIGURE 3** An excerpt of the IACFG showing a portion of the seqGen system
Table 1 reports the number of each type of fault seeded in the considered aspects.

The rows of Table 1 report the aspect identifier used in the IACFG in Figure 3, while the columns report the type of failure produced by the injected faults. Each column is labeled with the same label used earlier to describe the type of failure. The faults were seeded according to the general rules described previously. In particular:

- For failures of type (e), all possible permutations of the four advices in the system, with respect to the associated pointcuts, were considered (for a total of 12 combinations).
- For failures of type (d), a wrong precedence order was defined in the aspect Fa14, to reverse the correct sequence of advice execution (Figure 3 reports the correct execution sequence of advices).
- To generate type (e) failures, one fault was introduced in advice La32. The proceed statement was conditioned so that it did not always proceed.

In this experiment, to avoid any “noise” disturbing the experiment, it was decided to not introduce faults for (b) failures.

For this system, the test cases were generated by executing just the original (not fault seeded) system and capturing some values during the running. The reason for this was due to the kind of system that is a (random) sequence generator; thus, it was executed and some values captured by using a debugger. Nine test cases were adequately selected to be able to cover each criterion. Table 2 reports the mapping between test cases and the criteria they were applied to. On the rows are the test cases identifiers, and on the columns are the identifiers of the criteria. A “1” in a cell indicates that the test case shown on the row was used to satisfy the criterion on the column. The execution of the nine test cases allowed for a 100 percent coverage of each criterion.

All the failures due to the seeded faults were detected by the execution of the test cases. Moreover, the execution of the test cases also identified two failures due to faults existing in the “native” code of the system (that is, failures not due to the seeded faults). The first failure was of type (b); it was due to a recursive call used to compute the Fibonacci series. Indeed, the monitoring advice (bounded to the pointcut “fibMonitor()”) was applied to a wrong run-time context since fibMonitor() did not filter out recursive calls. Consequently, the monitoring advice (OCFG labeled as La32 in Figure 3) was wrongly invoked for each recursive call and the computed result was not right. The second failure was due to a wrong condition on a proceed statement in the advice reported with the label La21 in Figure 3. While several criteria detected the first of these failures, only criterion C6 was able to detect the second one.

An analysis of the results of this first experiment showed that some criteria were able to detect more failures than others. In particular, criterion C2 was the best at detecting different types of failures. Also, criteria C4 and C5 performed better than the remaining criteria in detecting more failures of different types. Criterion C6 was the best for identifying failures due to advice execution.

Results From the AJHotDraw System

In the following, the main results from the experiment that involved a portion of the AJHotDraw (AJHD) v0.4 system are synthetically reported and discussed. The considered portion of AJHD was composed of seven aspects and 95 classes. The considered aspects implement the concerns related to contracts checking, undo commands, and observing relationships of commands. The code in the aspects included 21 pointcuts, among which six were anonymous pointcuts, 13 advices, and four precedence rules definitions.

<table>
<thead>
<tr>
<th>TABLE 2</th>
<th>Test cases-criteria mapping for the seqGen system</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>C2</td>
</tr>
<tr>
<td>T1</td>
<td>1</td>
</tr>
<tr>
<td>T2</td>
<td>1</td>
</tr>
<tr>
<td>T3</td>
<td>1</td>
</tr>
<tr>
<td>T4</td>
<td>1</td>
</tr>
<tr>
<td>T5</td>
<td>1</td>
</tr>
<tr>
<td>T6</td>
<td>1</td>
</tr>
<tr>
<td>T7</td>
<td>1</td>
</tr>
<tr>
<td>T8</td>
<td>1</td>
</tr>
<tr>
<td>T9</td>
<td>1</td>
</tr>
</tbody>
</table>
First, the IACFG of the parts of interest of the system was produced, and then the considered aspects were seeded with some faults, as reported in Table 3, where the rows report the considered aspects and the columns report the types of faults.

As far as seeded faults of type (a) failures, in the Command Observer aspect the pointcuts’ expressions reacting to the change of selection in the command view were made “too strong” to exclude some commands from the match and this caused, in the right conditions, failures in lacking of execution of the correct update logic. Faults for type (b) failures were seeded in the aspects Command Observer and Contracts to change the kind of a predicate simulating a wrong identification of run-time logic where to apply the advice. Faults for type (c) failures were seeded in the Command Observer and UndoableCommand aspects. In the latter, two bindings among the wrapper around advices were “permuted” with respect to their pointcut expressions. This resulted in the wrong application of the computation logic when the pointcuts conditions matched.

Some aspects in AJHD require a predefined execution order; for instance, initialization advices must be executed before contracts checking, and undoable wrappers must be executed before specific undo commands logic. Thus, several faults for type (d) failures were seeded in the aspects Precedence, AlignCommand Undo, CutCommand Undo, and DuplicateCommand Undo to change precedence rules affecting the execution of advices in Contracts and Align/Cut/Duplicate_Command Undo aspects. For example, faults were injected in these aspects by modifying the lexicographic ordering of the two advices responsible for: 1) creating and configuring the internal UndoActivity; and 2) setting the figures affected by the execution of the command to the associated undo activity.

After the fault seeding, 28 test cases were defined to test the considered portion of AJHD to satisfy the defined coverage criteria. Also in this case the complete coverage of each criterion was obtained and all the failures due to the seeded faults were detected. Table 4 shows, for each considered aspect (corresponding to the table rows), the number of seeded faults (the first column) and the number of faults identified (columns 2 to 7; in the bottom of each column is the total number of the faults seeded or identified by the coverage criterion). Table 5 reports, for each type of failures (columns 1 to 5), the number of seeded faults (the first row), and the number of faults identified by each criterion (rows 2 to 7). The last two columns of the table show, respectively, the total number of seeded faults and faults identified by each criterion, as well as the percentage of the identified faults by each criterion with respect to the total number of seeded faults.

Analyzing the results summarized in Tables 4 and 5, the authors can say that in this case the most effective criteria (for identifying more types and number of failures/faults) were C2 and C5 (discovering, respectively, the 92,59 percent and 81,48 percent of total failures). Criteria C3 and C4 were able to find the same failures. This was due to the kind of pointcuts’ expressions (all compound expressions of “and” of predicates) and the method used to generate test cases to satisfy the criteria.

### TABLE 4 No. of faults seeded and detected in each aspect by the coverage criteria for the AJHD system

<table>
<thead>
<tr>
<th>Seeded</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precedences</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Contracts</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Command observer</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Undoable command</td>
<td>5</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Align command undo</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Cut command undo</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Duplicate command undo</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td>27</td>
<td>13</td>
<td>25</td>
<td>18</td>
<td>18</td>
<td>22</td>
</tr>
<tr>
<td>% Total</td>
<td>100,00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### TABLE 5 No. of faults injected for each type of failures and detected by the coverage criteria for the AJHD system

<table>
<thead>
<tr>
<th>[a]</th>
<th>[b]</th>
<th>[c]</th>
<th>[d]</th>
<th>[e]</th>
<th>Total</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>9</td>
<td>3</td>
<td>3</td>
<td>8</td>
<td>4</td>
<td>27</td>
</tr>
<tr>
<td>%</td>
<td>100,00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>9</td>
<td>3</td>
<td>2</td>
<td>8</td>
<td>3</td>
<td>25</td>
</tr>
<tr>
<td>%</td>
<td>100,00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>%</td>
<td>66,67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>%</td>
<td>66,67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C5</td>
<td>8</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>4</td>
<td>22</td>
</tr>
<tr>
<td>%</td>
<td>87,48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C6</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>4</td>
<td>17</td>
</tr>
<tr>
<td>%</td>
<td>62,96</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Criteria C1 and C6 show the lowest percentage values. Criterion C6 was effective in identifying faults of type (c), (d), and (e), mainly when the number of test cases needed to satisfy this criterion was considered. Indeed, this number was lower than the ones needed to satisfy the other criteria.

Considerations About the Experiments

Summarizing the results of these two experiments one can note that criteria C2 and C5 were the most effective, while the remaining criteria had a similar effectiveness. The criterion C6 was the most effective in identifying failures due to advice execution in both experiments. A last consideration involves the effort needed to define test cases to satisfy the coverage criteria. Criteria C1, C3, C4, and C6 required a lower effort than C2 and C5. Some other experiments carried out on some small- and medium-sized systems confirmed these considerations.

CONCLUSIONS AND FUTURE WORK

The interactions among advices and methods may make the testing of AO systems more complex than traditional ones, because they may be difficult to identify by just reading the source code. A complete coverage of these interactions is needed to verify the presence of failures due to them.

In this article the authors have proposed a set of six testing coverage criteria defining structural elements to be exercised in order to cover advices – methods interactions. The structural elements considered are nodes and arcs of the IACFG (Bernardi and Di Lucca 2007), a graph representing the relationships among advices and methods. The coverage criteria also have been defined with respect to a fault model considering five different types of failures caused by faults in aspects. The experiments carried out to assess the validity and effectiveness of the proposed coverage criteria show that:

- They make up useful guidelines to define effective test cases able to detect failures due to the interactions among aspects and class methods.
- They allow an accurate detection of failures and localization of the faults generating them.
- Some of the criteria have a greater effectiveness than others in detecting more types of failures.
- Some of the criteria were better suited to detect particular types of failures.
- They can be used to assess the effectiveness of a test case suite.

Future work will be addressed mainly to:

- Improve the existing prototype tool generating the IACFG (the improvements are mainly devoted to the usability of the tool)
- Develop a tool, integrated with the previous one, supporting the tester in the definition of test cases satisfying the proposed criteria

These tools will allow a more “comfortable” application of the proposed criteria, and will allow them to be applied to systems larger than those considered in the experiments carried out, thus allowing a further and deeper assessment of the validity and effectiveness of the coverage criteria. This also will allow one to determine if some criteria are more effective than others in detecting specific types of failures.

Finally, some work would be devoted to create an “online” forum about the issues related to the testing of AO systems to involve a large number of interested researchers and practitioners in defining effective approaches to face this problem, by a cooperative way.

REFERENCES


Testing Coverage Criteria for Aspect-Oriented Programs


BIOGRAPHIES

Giuseppe A. Di Lucca received his Laurea degree in electronic engineering from the University of Naples “Federico II,” Italy, in 1987, and his doctorate degree in electronic engineering and computer science from the same university in 1992. He is currently an associate professor of computer science at the Department of “Ingegneria” of the University of Sannio. Since 1987 he has been a researcher in the field of software engineering and his list of publications contains more than 80 papers published in journals and conference proceedings. His main research interests include software engineering, software maintenance, software testing, reverse engineering, software reuse, software reengineering, Web engineering, and software migration. He serves on the program and organizing committees of conferences in the field of software engineering, software maintenance, and program comprehension. He can be reached by e-mail at dilucca@unisannio.it.

Mario Luca Bernardi received his Laurea degree in computer science engineering from the University of Naples “Federico II,” Italy, in 2003 and his doctorate degree in information engineering from the University of Sannio in 2007. Since 2003 he has been a researcher in the field of software engineering, and he is author of several papers published in journals and conference proceedings. His main research interests include software maintenance and testing, software reuse, and software reverse engineering and reengineering, with particular interest on systems developed by the aspect-oriented paradigm. He has served as a reviewer of papers submitted to conferences and journals in the field of software engineering, software maintenance, and program comprehension. He can be reached by e-mail at mlbernar@unisannio.it.

The annual index for Volume 10 can be found online: http://www.asq.org/pub/sqp/.