Integration testing of object-oriented and aspect-oriented programs: a structural pairwise approach for Java

Otávio Augusto Lazzarini Lemos, Ivan Gustavo Franchin and Paulo Cesar Masiero

Depto. de Sistemas de Computação,
ICMC/USP - São Carlos - Caixa Postal 668
13560-970 São Carlos, SP, Brazil

Abstract

Several testing approaches focus on finding faults in software units of implementation. A problem not addressed by unit testing is the interaction among units, with respect to the correctness of their interfaces. In this paper a structural integration testing approach for object-oriented (OO) and aspect-oriented (AO) Java programs is presented. To make the activity more feasible, we address the testing of pairs of units (*i.e.*, methods and pieces of advice). A model called $\mathcal{PWDU}$ (Pair-Wise Def-Use) graph to represent the flow of control and data between pairs of units is proposed. Based on the $\mathcal{PWDU}$, the following family of testing criteria is defined: all-pairwise-integrated-nodes (control-flow based), all-pairwise-integrated-edges (control-flow based), and all-pairwise-integrated-uses (data-flow based). To evaluate the proposed approach, an implementation of the criteria in a testing tool is presented along with an example of usage and an exploratory study. The study with 7 AO programs that are also OO was conducted to investigate the cost of application and usefulness of the approach. Results provided evidence that the criteria are practical and useful for integration testing of OO and AO programs.

Key words: Software Testing, Object-Oriented Programming, Aspect-Oriented Programming, Structural Testing, Integration Testing, Testing Criteria, Testing Object-Oriented Programs, Testing Aspect-Oriented Programs, Java

1 Introduction

Several approaches for testing Object-Oriented (OO) programs are targeted at finding faults in units of implementation. While unit testing supports revealing

faults located in the logic of single units, it may not reveal faults related to the interactions among units (*i.e.*, interface/integration faults). Aspect-Oriented (AO) programs [13] that are extensions of OO programs may also contain interface faults because they also involve method interactions [23]. Moreover, aspects cut new interfaces through the primary decomposition of a system [14] that should also be tested. In this paper we present an approach for structural testing of interfaces present in OO and AO programs.

While unit testing a program, the tester is mainly interested in the algorithmic characteristics of the system, verifying whether each unit performs its function as expected [1] . The anticomposition axiom defined by Weyuker [39] states that testing each piece of a program in isolation is not necessarily sufficient to deem the entire program adequately tested. Therefore, the goal of integration testing is to test units in their intended environment by exercising their interactions as completely as possible. This is important because other types of errors might arise at this level [5].

Integration problems include errors in input-output format, incorrect sequencing of subroutine calls, and misunderstood entry or exit parameter values [10]. In an empirical study reported by Basili and Perricone [2], 39% of the errors found in the system studied were classified as interface errors, which made the authors conclude that interfaces seemed to be a major problem for software. This is an evidence that they should be carefully handled. Moreover, OO and AO programs are composed of several units with simple intraprocedural control-flow that interact with each other [30], which implies more opportunities for integration/interface errors [41]. In this context, quantitative criteria are also useful to evaluate how well interfaces are being exercised.

Following Haley and Zweben [8], we consider that integration errors occur when incorrect values are passed through unit connections. Based on this observation, we adapt the classification of integration errors proposed by Delamaro et al. [5]. Consider a program P and a test case t for P. Suppose that P contains units F and G such that F calls G or is affected by G (when G is an advice – see Section 2). Consider $S_I(G)$ to be the n-tuple of values passed to G and $S_O(G)$ the n-tuple of values returned from G. When executing t on P, an integration error is identified in the interaction of G with F when (a diagram depicting each type of integration error is presented in Figure 1):

(a) Upon entering G, $S_I(G)$ does not have the expected values and these values cause an erroneous output (a failure) before returning from G;
(b) Upon entering G, $S_I(G)$ does not have the expected values and these values lead to an incorrect $S_O(G)$, which in turn causes an erroneous output (a failure) after returning from G;

[1] In this paper, we consider a *unit* to be a method or an advice, the latter being a method-like construct of AO programs (see Section 2).

(c) Upon entering G, $S_I(G)$ has the expected values, but incorrect values in $S_O(G)$ are produced inside G and these incorrect values influence an erroneous output (a failure) after returning from G.

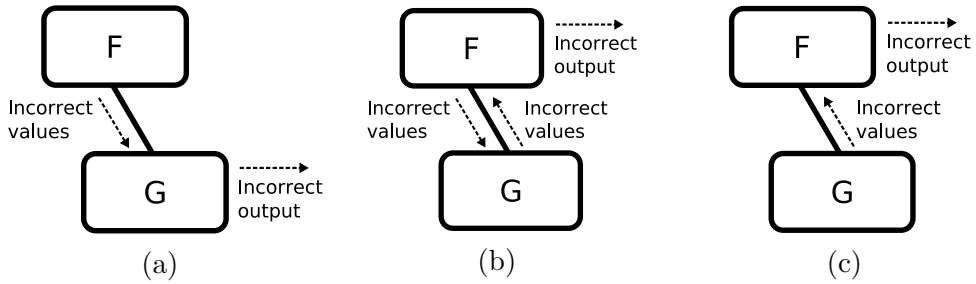


Fig. 1. Types of integration errors (adapted from the approach presented by Delamaro et al. [5]).

To the best of our knowledge few testing criteria were defined for integration testing of OO programs, and even fewer for the integration testing of AO programs. Harrold and Rothermel [11] proposed the first structural integration testing approach for OO programs. They considered the intra-method, inter-method, intra-class and inter-class testing types, which also considered def-use information from call sequences issued to objects of a class. However, no limitation was defined to the call depths, making the authors wonder how well would the technique scale for inter-class testing of large programs. Harrold and colleagues also explored other problems related to OO program testing such as regression testing and incremental testing of OO program structures [9,10,25]. Souter and others [30–32] have also proposed OO testing approaches, but mostly based on other concepts such as Points-to/Escape analysis [40].

Zhao has developed a data-flow testing approach for AO programs [43] based on the approach proposed by Harrold and Rothermel [11]. He also addresses the testing of interfaces between class units and aspect units, but does not limit the depth of interactions. Moreover, until now, no implementation of the approach has been presented.

Vilela et al. also proposed a pairwise integration testing approach, however, their approach targets procedural programs [35]. Based on the work of Linnenkugel and Müllerberg [21] (also used for definitions related to data-flow testing in this paper), they extended the family of Potential-Uses data-flow criteria [22] to the pairwise integration testing of procedural programs. Paradkar [26] uses the idea of pairwise testing to test the integration of classes.

In this paper we propose a family of structural testing criteria to test interacting units of OO and AO Java programs. These criteria help evaluating when test cases issued to F, for instance, are enough to test the interface between F and G. The main problem is that some integration faults can not be discovered unless an adequate coverage of G is obtained in the context of F (Section 3

presents an example of such type of fault).

Since even for small systems there might be several interactions among units, it can be very expensive to test the integration of units in arbitrary call depths. Moreover, for large systems this problem can be exponentially worse [33]. Therefore, the testing of pairs of units helps keeping the activity more feasible. The idea is similar to pairwise specification testing where for each pair of input parameters of a system, every combination of valid values have to be covered by at least one test case [34]. Pairwise specification testing is based on the observation that most faults are caused by interactions of at most two factors. In our case we apply a similar motivation, but for structural testing, considering pairs of structures instead of inputs.

We propose the testing of both intra-module units (units that interact with each other inside classes and aspects) and inter-module units (units of different classes and aspects that interact with each other). We also address a research subject that has not been fully explored yet: adequate testing of AO programs. Based on a Java bytecode control-flow and data-flow model, we define three specific testing criteria to test OO and AO Java programs. The model and criteria are implemented in a testing tool, extended from a family of tools named JaBUTi (Java Bytecode Understanding and Testing) [36,37]. This version of the tool is called JaBUTi/PW-AJ, for *PairWise-AspectJ*.

Since structural testing is usually subject to important cost-effectiveness trade-offs, we performed an exploratory study to analyze the cost of application of the proposed criteria. We selected 7 AO programs and applied the criteria to check whether the cost in number of additional test cases was practical, given that the application was already unit tested with respect to specific unit testing criteria.

Results indicated that, in general, the proposed criteria require a relatively small number of additional test cases to the original unit test set; an evidence of its applicability and usefulness (since unit testing was not enough to cover the criteria). The remainder of this paper is structured as follows. Section 2 presents basic knowledge about Aspect-Oriented Programming (AOP) and the AspectJ language to provide a basis to understand our approach; and Section 3 presents a motivation example for our approach using the tool we have implemented. Section 4 presents the proposed model and criteria for pairwise testing of OO and AO Java programs; and Section 5 presents the implementation of the proposed model and criteria in the JaBUTi/AJ-PW tool. Section 6 presents an exploratory study of the effort required to adequate an initial test set to the pairwise integration testing criteria and when additional test cases are indeed required. Finally, Section 7 concludes the paper and discusses future work.

2 Aspect-Oriented Programming and AspectJ

While traditional programming techniques help separating out the different concerns implemented in a software system, there are some that cannot be clearly mapped to isolated units of implementation. The main idea of AOP is the modularization of these types of concerns [13]. Examples of such concerns are the following: mechanisms to persist objects in relational data bases, access control, quality of services that require fine tuning of system properties, synchronization policies, and logging. These are often called *crosscutting* concerns because they tend to cut across multiple elements of the system instead of being localized within specific structural pieces [6].

AOP supports the construction of separate modules – called aspects – that have the ability to cut across other modules, defining behavior that would otherwise be spread throughout other parts of the code – the *base code*. Generic AOP languages must define four features: (1) a join point model to describe hooks in the program where additional behavior may be defined; (2) a mechanism of identification of these join points; (3) modules that encapsulate both join point specifications and behavior enhancements; and (4) a *weaving* process to combine both base code and aspects [6].

AspectJ is an extension of the Java language to support AOP. In AspectJ, aspects are modules that combine the following types of structures: join point specifications (pointcuts); pieces of advice, which implement the desired behavior to be added at join points; and regular OO structures like methods, fields and inner classes. Also, aspects can declare members (fields and methods) to be owned by other types, which is called inter-type declaration. AspectJ also supports declarations of warnings and errors that arise when certain join points are identified or reached. Before, after and around advice are method-like constructs that can be executed before, after, and in place of the join points selected by a pointcut. These constructs can also pick context information from the join point that caused them to execute.

Figure 2 lists the source code of a logging aspect written in AspectJ that affects the *BinomialDistribution* class presented in Figure 3. By default, pieces of advice in AspectJ are anonymous but the *@AdviceName* annotation can be used to name them. In the example, *printOperands* advises calls to an exponentiation method, printing the type of the base operand (integer or real), its value, and the value of the exponent.

In any AOP language implementation, aspect and non-aspect code must run in a properly coordinated fashion. To do so an important issue is to ensure that pieces of advice run at the appropriate join points as specified by the program. The AspectJ advice weaver statically transforms the program so that at

```
public aspect Logging {
  pointcut expCalls(double b, int exp):
    call(* br.math.BinomialDistribution.exponentiation(..))
    && args(b,exp);

  @AdviceName("printOperands")
  before(double b, int exp) : expCalls(b, exp) {
    double intPart = Math.floor(b);
    double decimalPart = b - intPart;
    if (decimalPart != 0)
      System.out.print("Real base: " + b);
    else System.out.print("Integer base: " + b);
    System.out.println(" Exponent: " + exp);
  }
}
```

Fig. 2. Source code of an aspect with an advice that affects the *exponentiation* method.

runtime it behaves according to the language semantics. The compiler accepts both AspectJ bytecode and source code and produces pure Java bytecode as a result. The main idea is to compile aspect and advice declarations into standard Java classes and methods at bytecode level. Parameters of the pieces of advice become parameters of these bytecode methods (with special treatment when reflective information is needed). To coordinate aspects and non-aspects, the bytecode is instrumented and calls to the 'advice methods' are inserted considering that certain regions of the bytecode represent possible join points (called join point *static shadows*). Furthermore, if a join point cannot be completely determined at compile time, the corresponding advice method calls are guarded by dynamic tests to ensure that the advice runs only at appropriate time. These tests are called *residues* [12].

3 Motivation Example

Consider the Java code presented in Figure 3 that implements a simple probability class named *BinomialDistribution* (the class *Round* is also present to exemplify an exception handling context later in this paper). The method *pmf* calculates the probability of successes in a sequence of n independent yes/no experiments, each yielding success with probability p (*i.e.*, the binomial probability mass function). *BinomialDistribution* also implements the following: a *combination* method that calculates the binomial coefficient function, and two auxiliary exponentiation and factorial methods. An example set of test cases to unit test each of these methods is described in Table 1.

Analyzing the test set using a Java unit testing tool, we conclude that it is adequate – that is, attains 100% of coverage – for the structural criteria all-nodes, all-edges, and all-uses [36]. While it is adequate for unit testing each

```
public class BinomialDistribution {
  public static double pmf(int k, int n, double p) {
0  long firstTerm = combination(k, n);
1  double secondTerm = exponentiation(p, k);
2  double thirdTermBase = (1 - p);
3  int thirdTermExp = n - k;
4  double thirdTerm = exponentiation(thirdTermBase, thirdTermExp);
5  return firstTerm * secondTerm * thirdTerm;
  }

  public static long combination(int k, int n) {
    if (k < 1 || n < 1 || k < n)
       throw new
         IllegalArgumentException("k and n must be greater than 1" +
          " and k must be greater than n.");
    if( k == n) return 1;
    long s = factorial(k);
    s /= factorial(n);
    s /= factorial(k-n);
    return s;
  }

  public static long factorial(int x) {
    if(x < 0 || x > 20)
      throw new
        IllegalArgumentException("x must be between 1 and 20");
    if (x == 0) return 1;
    if (x <= 2) return x;
    long s = 1;
    while (x > 1) {
      s *= x;
      x --;
    }
    return s;
  }

  public static double exponentiation(double base, int exponent) {
    double exp = 1;
    for(int i = 1; i <= exponent; i++)
      exp *= base;
    return exp;
  }
}

public class Round {
  public static double round(double x, int scale, int roundingMethod) {
    try {
        return (new BigDecimal
               (Double.toString(x))
               .setScale(scale, roundingMethod))
               .doubleValue();
    } catch (NumberFormatException ex) {
        if (Double.isInfinite(x)) { return x; }
        else {
            return Double.NaN;
        }
    }
  }
}
```

Fig. 3. Java code of a class that implements simple probability functionality and a rounding class.

method according to the three mentioned criteria, it is not able to reveal a fault present in the interface between methods *pmf* and *combination*; namely

Table 1
Sample test set to test each method of the *BinomialDistribution* class.

Test case	**Method**	**Input**	**Expected Output**
1	*exponentiation*	$10, 2$	100
2	*factorial*	10	3628800
3	*factorial*	0	1
4	*factorial*	1	1
5	*factorial*	21	Exception
6	*factorial*	-1	Exception
7	*combination*	$-1, -10$	Exception
8	*combination*	$10, -1$	Exception
9	*combination*	$5, 10$	Exception
10	*combination*	$5, 5$	1
11	*combination*	$20, 5$	15504
12	*pmf*	$5, 5, 0.5$	0.03125

combination is called with inverted parameters – (k, n) instead of (n, k). The fault is not revealed because, even though the test case issued to *pmf* is adequate for the three mentioned unit testing criteria, the method is called with the same value for k and n. Next we explain how the tool we have implemented with the approach presented in this paper can help revealing such fault.

To test an application using JaBUTi/PW-AJ we need to first create a testing project. In this step, the tester selects the classes and aspects to be tested. For instance, to test the binomial distribution application shown before along with the logging aspect presented in the last section, we need to select the referring class and aspect. After the selection, the tool generates an Aspect-Oriented Def-Use graph (see next section for more details) and derives testing requirements for each unit of each selected module. The tool also calculates and assigns different weights to each testing requirement (identified by different colors) to indicate the requirements that, when covered, enhance the coverage compared to other requirements with respect to the selected criterion.

After assuring that each unit has been tested with test cases that cover 100% of the code for each criterion – such as in the example –, we can select the intra-module pairwise testing environment to test the interfaces inside the selected modules (class or aspect). For our example there are 6 intra-module pairs in *BinomialDistribution*, that is, 6 interfaces between methods of the same class. Let us focus on the *pmf* – *combination* pair of units (see line 0 of

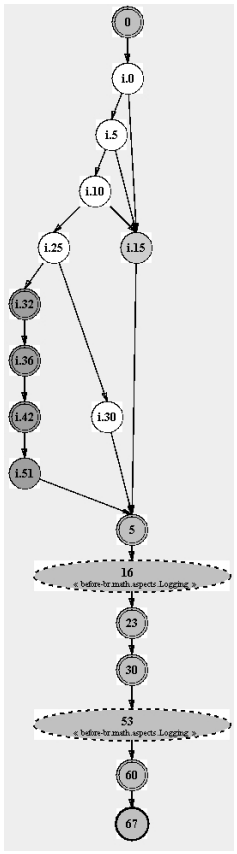
the source code in Figure 2). JaBUTi/PW-AJ supports importing JUnit test cases into the testing project. Since we already had a test set to test each of the *BinomialDistribution*'s methods (Table 1, including the test to the *pmf* method with parameters values 5, 5, 0.5), we can use it as a starting point for the testing of this pair. Figure 4 shows a partial JUnit implementation of the test set.

```
public class MathTest extends TestCase {

...

  public void testExp() {
    assertEquals(new Double(100),
        new Double(
            BinomialDistribution.exponentiation(10, 2)
        ));
  }

  public void testFact1() {
    assertEquals(new Long(3628800),
        new Long(
            BinomialDistribution.factorial(10)
        ));
  }

  public void testFact2() {
    assertEquals(new Long(1),
        new Long(
            BinomialDistribution.factorial(0)
        ));
  }

  public void testFact3() {
    assertEquals(new Long(1),
        new Long(
            BinomialDistribution.factorial(1)
        ));
  }

...

  public void testPmf() {
    assertEquals(new Double(0.03125),
        new Double(
            BinomialDistribution.pmf(5, 5, 0.5)
        ));
  }

}
```

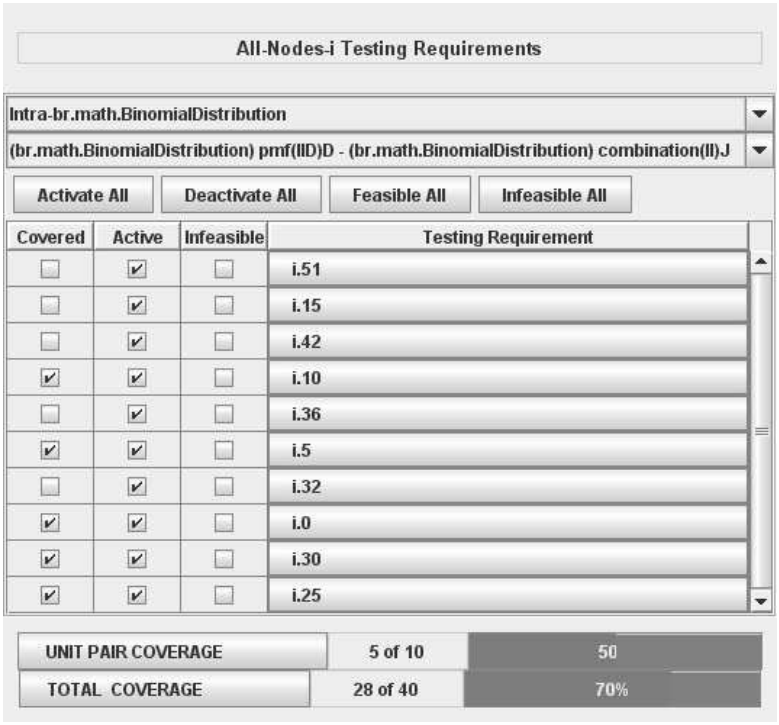
Fig. 4. JUnit implementation of the test set to test the methods of the *Binomial-Distribution* class.

After importing the test cases, we can check the coverage for each pairwise testing criterion for each pair. We can then focus on the requirements that are not covered by the test cases. For instance, in the example, some of the statements – or nodes – of the called units (requirements of the All-PW-nodes$_i$ – see next section) are not covered by the unit test set. The *pmf* - *combination* intra-module pair, for instance, gets a coverage of only 50% of the statements. Figure 5 shows the PairWise Def-Use graph ($\mathcal{PWDU}$ – defined in Section 4.2)

of this pair with the execution of the test cases with respect to the referred criterion (shown as All-Nodes-i in the tool). The nodes of the *combination* method have labels prefixed with 'i.'. White nodes represent the executed paths in the *combination* method by test cases issued to *pmf* (in this case, the single test case mentioned before). Figure 5 also shows a screenshot of the tool with the list of requirements for the same criterion and pair, and information about which requirements have been covered and which have not. The tester can also define a requirement as infeasible if it can not be covered by any test case.



(a) $\mathcal{PWDU}$ for the pair *pmf* - *combination*.



(b) List of requirements for the All-PW-Nodes$_i$ criterion.

Fig. 5. $\mathcal{PWDU}$ and list of requirements of the *pmf* - *combination* pair after the execution of the original test case.

Note that there are five uncovered nodes: $i.15$, $i.32$, $i.36$, $i.42$, and $i.51$. Analyzing the logic of the *combination* method, these nodes are not covered because the test case issued to *pmf* has the same value for parameters k and n, as pointed out before. Thus, *combination* returns 1 in the second if, without executing the other parts. Now, to thoroughly test this interface, we should at least try to cover all nodes of the latter in the context of *pmf*. To do that we need to create test cases for the other combination possibilities. For instance,

we need a test case where $k \neq n$ to cover nodes $i.32$, $i.36$, $i.42$, and $i.51$, where *factorial* is called. One example test case would be calling *pmf* with parameters $5, 10, 0.5$ (which means *what is the probability of getting 5 tails in a sequence of 10 coin flips?*). Now, when we execute this test case, an integration error raises: an exception is thrown telling us that the arguments are illegal. This happens because the call to *combination* is made with inverted parameters: it should be combination of n, k and not k, n, as commented before. When the program is fixed, both test cases that touch the interface being tested are successful.

At this point, the tester should continue covering the remaining parts of the code that were not covered for each pair, to enhance the confidence in the interfaces of the program. For instance, to cover the $i.15$ node of the same pair, there must be a test case where *combination* is called with illegal parameters from *pmf*. The stronger criteria defined in the next section (All-PW-Edges$_i$ and All-PW-Uses$_i$) can also be used to test these integrations more thoroughly. Additionally, after doing the intra-module integration testing, the inter-module pairs generated by the two interactions between *pmf* and the *printOperands* advice should also be tested.

In this example, although *pmf* and *combination* can be considered correct with respect to their isolate implementation, wrong values can be passed from *pmf* to *combination* due to a fault present in the interface. In the classification depicted in Figure 1, this fault can produce an error of type (b), such as the one raised when the additional test case is executed. The problem with these types of errors is that they might not be revealed unless the structure of the called unit is thoroughly covered by test sets issued to the caller unit; such as in this example.

4 Structural testing of OO and AO programs

Testing is the execution of a program with the intent of finding faults [24]. The different testing techniques that were proposed can be classified by the artifact used to derive the testing requirements. *Functional testing* derives its requirements from the specification of the system, without taking into account specific implementation details; *structural testing*, which is the focus of this paper, derives its requirements from the knowledge of characteristics and internal details of the implementation; and *fault-based testing* derives its requirements from typical faults inserted during the development process.

Regardless of the testing technique used, software testing is usually performed in three levels:

(1) Unit testing, where the smallest pieces of a system are tested in isolation with the intent of finding faults in their logic and implementation;
(2) Integration testing, where interactions among units are tested with the intent of finding faults in the logic and implementation of the interfaces; and
(3) System testing, which consists in verifying the integration of all elements of a system to assure that they combine adequately and that expected global functioning/performance is obtained.

In this paper we focus on integration testing, building on top of unit testing approaches described in other papers [18,36,37]. We consider a method and an advice as the smallest units to be tested (*i.e.*, the *unit*) and we address the testing of each pair of interacting units. We call a *module* a part of the program that clusters a number of units together with other structures (like fields). For our purposes, a module can either be a class or an aspect.

In structural testing a representation of the structure of the program is required. The control-flow graph (CFG) is used to represent the flow of control of a program, where each node represents a statement or a block of statements executed sequentially, and each edge represents the flow of control from one statement or block to another. With respect to data-flow information, we use the definition-use (or def-use) graph, which extends the CFG with information about the definition and use of variables in each node and edge of the CFG [28].

For our purposes, the occurrence of a variable is either classified as a definition or use. As to the use occurrences it is called a *predicate* use (or p-use) a use of a variable in a conditional statement – for instance: *if (i == 5)* – and a *computational* use a use of a variable that directly affects a computation – for instance: $j = i + 5$. P-uses are associated to the def-use graph edges and c-uses are associated to the nodes. A definition-clear path (or simply def-clear path) is a path that goes from the definition place of a variable to a subsequent c-use or p-use, such that the variable is not redefined along the way. A def-use pair with respect to (wrt) some variable is then a pair of definition and subsequent use such that there is a definition-clear path wrt that same variable from the definition to the use location [28].

The basic unit testing model for OO and AO Java programs is the aspect-oriented def-use ($\mathcal{AODU}$) graph [18], that was built on top of the approach by Vincenzi et al. for OO programs only. The $\mathcal{AODU}$ is generated for each unit to be tested, both methods and pieces of advice. It is defined as a directed graph with elements (N, E, s, T, C). Informally, N represents the set of nodes – which are composed by blocks of bytecode instructions that are executed sequentially; E represents the set of edges connecting nodes when there is transfer of flow from one to the other; s represents the entry node; T is the

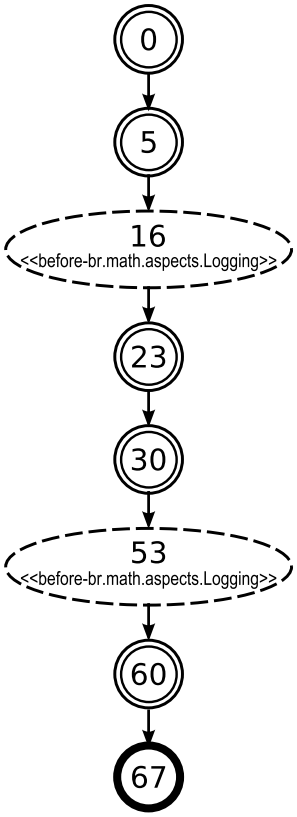
set of exit nodes; and C is the set of nodes affected by pieces of advice (called *crosscutting nodes*). We also differentiate regular edges – edges that connect regular nodes – from exceptional edges – edges that connect regular nodes to nodes that represent exception handling statements. The element C was added to the original def-use model defined by Vincenzi et al. [36,37] to represent the basic interaction that occurs in AO programs.

The $\mathcal{AODU}$ graph is represented by using the following conventions: single circled nodes represent regular blocks of instructions, double circled nodes represent method calls, bold nodes represent exit nodes, dashed elipses (crosscutting nodes) represent advice execution and contain additional information of what kind of advice is affecting that point and to which aspect it belongs, regular edges represent regular control-flow, and dashed edges represent exceptional control-flow. Four examples of $\mathcal{AODU}$ graphs are presented in Figure 6. The units refer to three methods and an advice of the example presented in Figures 2 and 3. An example of exceptional control-flow can be seen in the $\mathcal{AODU}$ of the *round* method in Figure 6(c). The transfer of flow from the *try* block to the *catch* block is represented with a dashed edge.

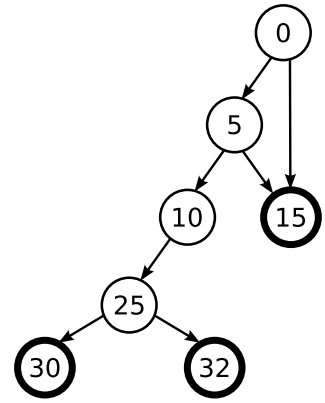
Following the approach by Vincenzi et al. [36,37], the labels of the $\mathcal{AODU}$ correspond to the offsets of the first bytecode instructions of the corresponding blocks of instructions. For instance, consider the *pmf* method presented in Figure 3. Line number 0 of the source code correspond to graph nodes 0 and 5 in Figure 6(a) (the last call is added by the AspectJ compiler to access the instance of the *Logging* aspect). Node 16 corresponds to the execution of the *printOperands* advice, before the execution of the *exponentiation* method at line 1 of the source code; the latter is represented by node 23 of the graph. Node 30 represents the statements at lines 2 and 3 of the source code (plus another call to access the instance of the *Logging* aspect). Node 53 corresponds to the second execution of the *printOperands* advice and nodes 60 and 70 correspond to lines 4 and 5 of the source code. The tool we have implemented – presented in Section 3 and detailed in Section 5 – also maps the bytecode to the source code and to the graph with the aid of colors. In this way, each required element can be visually mapped back to the source code.

4.1 Pairwise integration testing

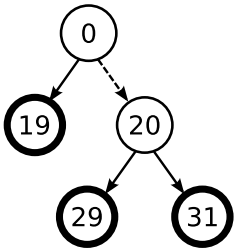
Concerned with integration faults present in OO and AO Java programs that produce the types of errors depicted in Figure 1, we propose the extension of our unit testing approach to integration testing of pairs of units. The main idea is to make sure that test sets cover the structure of the integrated unit in the context of the unit that calls it or is affected by it. In this way, we can enhance the probability of raising the types of integration errors mentioned



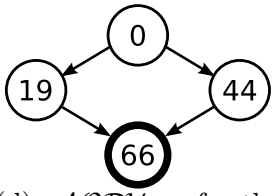
(a) $\mathcal{AODU}$ of the *pmf* method



(b) $\mathcal{AODU}$ of the *combination* method



(c) $\mathcal{AODU}$ of the *round* method



(d) $\mathcal{AODU}$ of the *printOperands* before advice

Fig. 6. Examples of $\mathcal{AODU}$s.

before, helping the tester to find the related integration faults. To keep the activity more feasible, we propose the testing of each pair of units at a time, instead of addressing arbitrary call depths at once.

With that purpose in mind, we propose a model to represent the structure of a pair of units. For OO programs we have a single type of pairs of interacting units: method-method – when a method calls another method. For AO programs, on the other hand, we have four types of pairs of interacting units: method-method, method-advice – when a method is affected by an advice, advice-method – when an advice calls a method, and advice-advice – when an advice is affected by another advice.

To adequately represent the execution flow that occurs inside a pair of units, we need to define a graph that integrates the def-use graphs of the corresponding units. We define the PairWise Def-Use ($\mathcal{PWDU}$) graph, which integrates the $\mathcal{AODU}$s of the interacting units. Before we can define the $\mathcal{PWDU}$ graph, we need to define an extra element in the $\mathcal{AODU}$ graph to represent the set of *interaction* nodes, which is composed of all call nodes and crosscutting nodes. With these types of nodes we are able to identify all interactions among units of OO and AO programs.

The unit in the pair that is either calling a method or being affected by an advice is the *base* unit and the unit to which the control-flow can be transferred to is the *integrated* unit. The $\mathcal{PWDU}$ is then composed by the $\mathcal{AODU}$ of the base unit and the $\mathcal{AODU}$ of the integrated unit. To differentiate the nodes and edges of the two units, we define the *integrated* nodes, which represent the nodes of the integrated unit, and two kinds of edges: the *integrated* edges – edges that connect two integrated nodes – and the *integration* edges – edges that represent the flow of control between a node of the base unit and a node of the integrated unit, and vice versa.

The entry node of the $\mathcal{PWDU}$ graph of a pair of units u_1 and u_2 ($\mathcal{PWDU}(u_1, u_2)$) is the entry node of the base unit u_1. The same applies for the exit nodes, that is, the exit nodes of the $\mathcal{PWDU}$ are the exit nodes of u_1. The $\mathcal{PWDU}(u_1, u_2)$ is defined as a directed graph with elements (N, E, s, T, I, i, R):

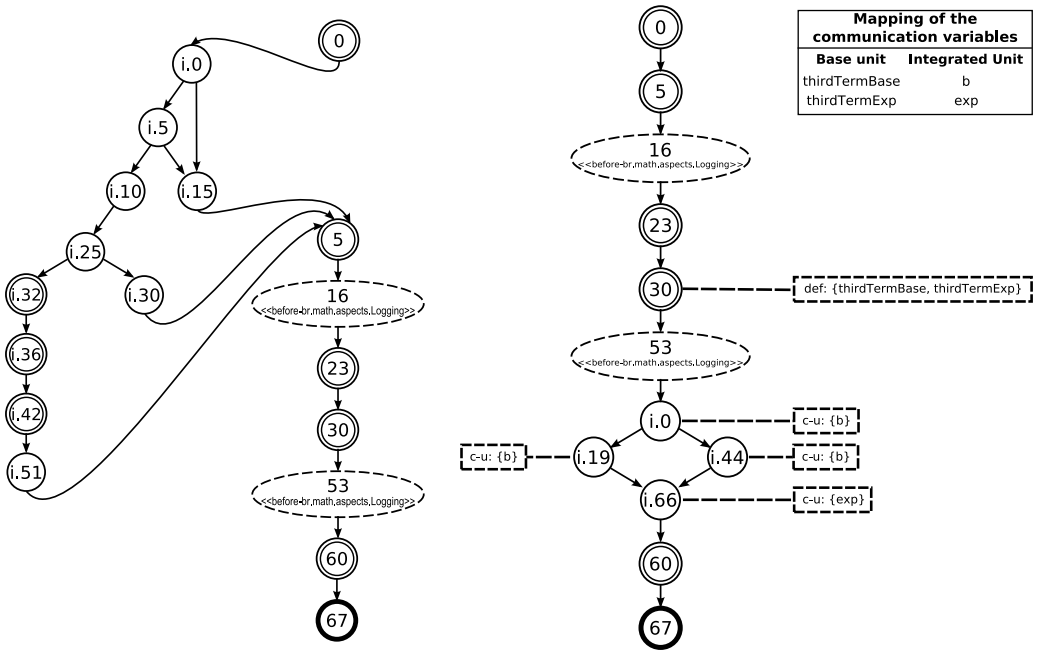
- $N = N_1 \cup N_2$ represents the complete set of nodes of the $\mathcal{PWDU}$ graph, such that:
  - N_1 is the set of nodes of the $\mathcal{AODU}$ of u_1;
  - N_2 is the set of nodes of the $\mathcal{AODU}$ of u_2, also called integrated nodes (also defined as N_i);
- $E = E_1 \cup E_2 \cup E_c - e_d$ is the complete set of edges of the $\mathcal{PWDU}$, such that:
  - $E_1 \subseteq N_1 X N_1$ is the set of edges of u_1;
  - $E_2 \subseteq N_2 X N_2$ is the set of edges of u_2, also called integrated edges (also defined as E_i).
  - E_c is the set of integration edges, created to 'connect' the two $\mathcal{AODU}$ graphs.
  - e_d is the original edge that connected the node where the integration occurs with the subsequent node in the $\mathcal{AODU}$ of the base unit. This edges is removed because the flow is now transferred to the integrated unit, not to the subsequent node.
- $s \in N$ and $s = s_1$ is the entry node of the $\mathcal{PWDU}$, such that $s_1 \in N_1$ is the entry node of u_1;
- $T \subseteq N$ and $T = T_1$ is the set of exit nodes of the $\mathcal{PWDU}$, such that T_1 is

the set of exit nodes of u_1;
- $I = I_1 \cup I_2$ is the set of interaction nodes (that is, crosscutting nodes and call nodes of the $\mathcal{PWDU}$), such that:
  - $I_1 \subseteq N_1$ is the set of interaction nodes of u_1;
  - $I_2 \subseteq N_2$ is the set of interaction nodes of u_2;
- $i \in I_1$ is the node where the transfer of flow from u_1 to u_2 occurs;
- $R \subseteq N_1$ is the set of return nodes of the transfer of flow from u_1 to u_2.

$\mathcal{PWDU}$ graphs are represented using the following conventions:

- Regular, integration, and integrated edges are represented by regular edges;
- Exceptional edges that represent flow of control from a regular node to an exception handler are represented by dashed edges [36];
- Regular nodes are represented by single circled nodes;
- Integrated nodes are represented by regular nodes with the label preceded by 'i.', to avoid label repetition;
- Call nodes are represented by double circled nodes; and
- Crosscutting nodes are represented by dashed elipses with the corresponding label stereotyped by a '<<', the type of the advice, a '-', the full qualified name of the aspect that contains the advice followed by a '>>'; and
- Exit nodes are represented by bold nodes.

Figure 7 presents two $\mathcal{PWDU}$ examples for the *pmf* method interacting with a method inside the same class – *combination* – and an advice of an aspect – *printOperands* (second execution of the advice). The corresponding source code and $\mathcal{AODU}$ graphs were presented in Figures 2, 3, and 6. Note that these graphs are formed by an integration of the $\mathcal{AODU}$s presented in Figure 6 with a slight difference in the *pmf* - *combination* pair: there are additional nodes that were suppressed in the representation of the *combination* $\mathcal{AODU}$ ($i.36$, $i.42$ and $i.51$). Nevertheless, the logic remains the same, because these nodes represent contiguous instructions. The table in the upper right corner of Figure 7(b) shows the mapping of the communication variables used for the data-flow criterion (see Section 4.3) and the notes coming from the nodes show which variables are being defined (def) and computationally-used (c-u) at those places. Note that if there were predicative uses they would be presented along the corresponding edges. No def-use information is presented in Figure 7(a) because there are no communication variables in the corresponding interface: variables used in *combination* are not previously defined in the body of *pmf*.



(a) Intra-module $\mathcal{PWDU}$ for *pmf - combination*.

(b) Inter-module $\mathcal{PWDU}$ for *pmf - printOperands*$_2$.

Fig. 7. Example $\mathcal{PWDU}$s.

4.3 A family of pairwise structural testing criteria

Testing criteria are a very important way to provide systematic selection and evaluation of test sets. To enhance the confidence that two units are combined in a correct way, we propose three structural testing criteria: two control-flow based and one data-flow based. The main idea is to make sure that the integrated unit is thoroughly covered by test cases given to the base unit, stressing the interface between the units.

Let T be a test set for a program P, being $\mathcal{PWDU}$ the graph of a pair of units, and let Π be the set of paths executed by T in P. We say that a node i is included in Π if Π contains a path $(n_1, \ldots, n_m)$ where $i = n_j$ for some j, $1 \leq j \leq m$. Similarly, an edge (i_1, i_2) is included in Π if Π contains a path $(n_1, \ldots, n_m)$ where $i_1 = n_j$ and $i_2 = n_{j+1}$ for some j, $1 \leq j \leq m - 1$.

4.3.1 Control-flow criteria

For the control-flow based criteria, we decided to extend the basic all-nodes and all-edges criteria, revisiting them in the pairwise OO and AO structural testing context. One way of stressing the interface between two units is to try to make sure that each node of the integrated unit – the integrated nodes – is being executed in the context of the base unit. The same idea can also be

applied to the integrated edges. Thus, we define the **all-pairwise-integrated-nodes** and the **all-pairwise-integrated-edges** criteria:

- **all-pairwise-integrated-nodes** (All-PW-Nodes$_i$): Π is adequate wrt the all-pairwise-integrated-nodes criterion if each integrated node $n_i \in N_i$ of the $\mathcal{PWDU}$ graph is included in Π. In other words, this criterion requires that each integrated node in a $\mathcal{PWDU}$ graph be exercised at least once by some test case in T.
- **all-pairwise-integrated-edges** (All-PW-Edges$_i$): Π is adequate wrt the all-pairwise-integrated-edges criterion if each integrated edge $e_i \in E_i$ of a $\mathcal{PWDU}$ graph is included in Π. In other words, this criterion requires that each integrated edge of a $\mathcal{PWDU}$ graph be exercised at least once by a test case in T.

Note that E_i can also contain exceptional edges, that is, edges that connect blocks where an exception might be thrown to the corresponding catch blocks where the exception is handled (such as exemplified in Figure 6(c)). However, in the case of the $\mathcal{PWDU}$, these edges are treated just like regular edges, which does not affect the application of the criterion.

4.3.2 Data-flow criterion

In some cases covering all statements and conditionals of the integrated unit is not enough to raise an integration error. For instance, consider units u_1 and u_2, where u_1 calls u_2. A simplified data-flow graph of their integration is presented in Figure 8. Consider def_x and use_x places where variable x is defined and subsequently used. Note that a test set that traverses paths $1, 2, 4, i.1, i.2, i.4$ and $1, 2, 4, i.1, i.3, i.4$ covers all nodes and edges of u_2 in the context of u_1. However, if an integration fault is related to the definition of x at node 3 and its subsequent use at node $i.3$ (path highlighted in the graph), the test set would not raise the related error. In this case, a data-flow criterion – such as the all-pairwise-integrated-uses presented later in this section – would necessarily cover the highlighted path, driving the tester to find the integration fault.

Therefore, with respect to the data-flow criteria we decided to revisit the known all-uses criterion. We used the approach by Linnenkugel and Müller-berg [21] as a basis to define the data-flow interactions between two units. Since the data-flow information is very much dependent on the language and representation used, all definitions in this part of the paper are based on the Java and AspectJ languages, and on the Java bytecode specification.

Before defining formally our data-flow criterion, a data-flow model for the Java/AspectJ languages was defined, based on the model proposed by Vincenzi et al [36,37]. It is important to notice that because we use bytecode, AspectJ does not introduce any new data type in Java: we consider an aspect as a

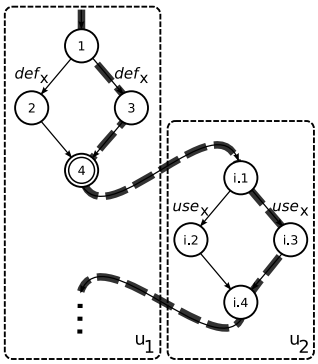


Fig. 8. Example of data-flow integration error.

singleton class and advices are treated as methods. The following types of variables are considered: local variables, elements of array, static attributes, instance attributes, and formal parameters.

Let us use the following definitions to show generically some examples of the decisions taken:

- c: a literal;
- n: a primitive value of type integer;
- p: a primitive type variable;
- a: a variable that refers to an array;
- $a[]$ an array element with primitive or reference type;
- C: a class with: an instance field f, a static field s, an instance method mi, and a static method ms.
- $C.s$: a static field of class C with type primitive or reference;
- r: a variable referring to an instance (object) of class C;
- $r.f$: an instance field of r with type primitive or reference;
- $null$: reference to a non existing object or array;
- v: a variable of type p, a, $a[]$, r, $r.f$, or $C.s$;
- t: a parameter of type c or v;
- e: a simple or complex expression;

We then established the following conservative rules to classify definitions and uses (the examples refer to lines of Table 2):

(1) A literal c is never considered as used in this context because the related data never changes (Example: line 1).
(2) The definition or use of a primitive variable p is considered as being only a definition or use of p, because there is no other data aggregated to it (Examples: lines 2 and 3).
(3) The definition of a reference variable r can be a null reference or a reference to an object (being constructed or already existing), which can be an array or an instance of a class, because those are the possible data-flow

implications for reference variables. Therefore, the definition of a reference variable r involving a null reference is considered as being only a definition of r. The definition of a reference variable r involving a reference to an object is considered as the definition of r and, for an array, the definition of array variables $r[]$, and, for an object of class C with f as one of its instance variables the definition of the instance field $r.f$. This is a more conservative approach for data-flow implications in these cases (Examples: lines 4 to 6 and 10 to 13).

(4) Array variables are considered as occupying a single memory position. Therefore, the definition of an array variable $a[]$, which is an element of an array referred to by the reference variable a, is considered as being the definition of $a[]$ and the definition of the array referred to by a (represented as a definition of a). The use of an array variable $a[]$ is considered as being the use of the reference variable a, which gives access to the element, and the use of the array variable $a[]$. This is also a more conservative approach for data-flow implications in these cases (Example: lines 7 to 9).

(5) The definition of an instance field f of a reference variable r of type class C is considered as being a use of the reference variable r that allows access to the field, the definition of the instance field (represented by $r.f$), and the definition of the object referred by the reference variable r (represented as the definition of r). The use of an instance field f is considered as being the use of a reference variable r (to access the field) and the use of an instance field $r.f$. These are the most straightforward data-flow implications for these cases (Examples: lines 14 to 16).

(6) Access to static attributes (or class attributes) is done without the need of a reference variable. Thus, the definition or use of any static field s of a class C is considered as being only a definition or use of the static field represented by $C.s$ (Examples: lines 17 to 19).

(7) When an instance method mi is called, such as $r.mi(t_1; t_2; \ldots ; t_n)$, where t_i is a parameter of type literal or variable, we consider that variable r and parameters $t_1, t_2, \ldots, t_n$ are used, according to the rules described on items 1 to 6. When a static method ms is called, such as $C.ms(t_1; t_2; \ldots; t_n)$, we consider that the parameters $t_1, t_2, \ldots, t_n$ are used, according to the rules described on items 1 to 6. These are the most straightforward data-flow implications for these cases (Examples: lines 20 to 25).

(8) When an expression is assigned to a variable v in the form $v = e_1\ op\ e_2\ op\ \ldots\ e_n$, where e_i is an item of the expression that can be a literal or a variable, and op is an operator, we consider that $e_1, e_2, \ldots, e_n$ are used, according to the rules described on items 1 to 6, and that v is defined. In the case of lazy operators 'or' and 'and', we also consider the use of each operand, since a fault might be related to the use of an operand closer to the rightmost side of the operation (Example: line 26).

Data-flow based integration testing consists in testing the variables that affect

Table 2
Examples of definitions and uses of data ('definition' is abbreviated as 'def')

Type	**Sentence**	**Variables Uses/Definitions**
1	$p = c$	def of p
2	$p = v$	use of v and def of p
3	$v = p$	use of p and def of v
4	$a = new\ type[n]$	def of the array referred by a (represented as def of a) and def of $a[]$
5	$a = null$	def of a
6	$a_1 = a_2$	use of a_2, def of a_1 and def of $a_1[]$
7	$a[n] = c$	use of a, def of $a[]$, def of the array referred by a (represented as def of a)
8	$a[n] = v$	use of v, use of a, def of $a[]$ and def of the array referred by a (represented as def of a)
9	$v = a[n]$	use of a, use of $a[]$ and def of v
10	$r = null$	def of r
11	$r = new\ C()$	def of the object referred by r (represented as the def of r) and def of $r.f$
12	$r = new\ C(t_1,\ldots,t_n)$	use of t_1, ..., use of t_n, def of the object referred by r (represented as the def of r) and def of $r.f$
13	$r_1 = r_2$	use of r_2, def of r_1, and def of $r_1.f$
14	$r.f = c$	use of r, def of $r.f$, and def of the object referred by r (represented as the def of r)
15	$r.f = v$	use of v, use of r, def of $r.f$, and def of the object referred by r (represented as the def of r)
16	$v = r.f$	use of r, use of $r.f$, and def of v
17	$C.s = c$	def of $C.s$
18	$C.s = v$	use of v and def of $C.s$
19	$v = C.s$	use of $C.s$ and def of v
20	$v = C.ms(c)$	def of v
21	$v_1 = C.ms(v_2)$	use of v_2 and def of v_1
22	$v = C.ms(t_1,\ldots,t_n)$	use of t_1, ..., use of t_n and def of v
23	$v = r.mi(c)$	use of r and def of v
24	$v_1 = r.mi(v_2)$	use of r, use of v_2, and def of v_1
25	$v = r.mi(t_1,\ldots,t_n)$	use of r, use of t_1, ..., use of t_n, and def of v
26	$v = e_1\ op\ \ldots\ e_n$	use of the factor e_1, ..., use of the factor e_n, and def of v

the communication between base and integration units. These variables are called communication variables. They can be of any Java type, that is, both primitive and reference. In OO and AO programs the following communication variables types can be identified: Formal Parameters – FP; Actual Parameters – AP; and Static fields of the module(s) of the base or integrated units or from other modules of the program – SF. Instance Fields – IF – can also be

considered communication variables when the integrated unit is an instance method, however, they are treated as actual parameters (AP) and formal parameters (FP). An instance field is a field whose value is object-specific and not class-specific. In this case, the object reference from which the method is being called is considered as a parameter being passed to the integrated unit.

Our pairwise structural testing approach considers only paths (or def-use relations) that directly affect the communication between units, that is:

- wrt the communication variables x used as inputs, we consider the paths composed by the sub-paths that go from the last definition of x prior to the call to the call inside the base unit and the sub-paths that go from the integrated unit entry to where x is used inside the integrated unit.
- wrt the communication variables x used as outputs, we consider the paths composed by the sub-paths that go from the last definition of x inside the integrated unit to the exit of the integrated unit and the sub-paths that go from the return of the integrated unit to the use of x inside the base unit.

OO and AO programs consist of units U_n. For each U_n we define the following sets:

- $FP\text{-}IN(U_n)$ is the set of formal parameters of U_n used as inputs.
- $FP\text{-}OUT(U_n)$ is the set of formal parameters of U_n used as outputs.
- $SF\text{-}IN(U_n)$ is the set of static fields used inside U_n.
- $SF\text{-}OUT(U_n)$ is the set of static fields defined inside U_n.

Let U_a be the base unit and U_b be the integrated unit. The point where the flow of control is transferred from U_a to U_b is represented by U_{ba}. For this point the following sets are defined:

- $AP\text{-}IN(U_{ba})$ is the set of actual parameters used as inputs in U_{ba}.
- $AP\text{-}OUT(U_{ba})$ is the set of actual parameters used as outputs in U_{ba}.

To describe the relations between actual and formal parameters and between static fields used by the units two mappings are defined: I_{ba} and O_{ba}. Note that while doing the mappings for the parameters and static fields, we also map fields (for object references) and aggregated variables (for array references) related to these references. Another side note is related to static fields: they have the same name both in the base unit and in the integration unit.

The I_{ba} mapping relates each input actual parameter used in U_{ba} with the corresponding input formal parameter in U_b and each input static field with itself:

- $I_{ba} : AP\text{-}IN(U_{ba}) \cup SF\text{-}IN(U_b) \rightarrow FP\text{-}IN(U_b) \cup SF\text{-}IN(U_b)$, where $AP\text{-}IN(U_{ba}) \rightarrow FP\text{-}IN(U_b)$ and $SF\text{-}IN(U_b) \rightarrow SF\text{-}IN(U_b)$

The O_{ba} mapping relates each output actual parameter used in U_{ba} with the corresponding output formal parameter in U_b and each output static field with itself:

- $O_{ba} : AP\text{-}OUT(U_{ba}) \cup SF\text{-}OUT(U_b) \rightarrow FP\text{-}OUT(U_b) \cup SF\text{-}OUT(U_b)$, where $AP\text{-}OUT(U_{ba}) \rightarrow FP\text{-}OUT(U_b)$ and $SF\text{-}OUT(U_b) \rightarrow SF\text{-}OUT(U_b)$

Based on these definitions and on the $\mathcal{PWDU}$s of the units, additional sets must be defined. Let $def(i)$ be the set of variables defined in the node i; $c\text{-}use(i)$ be the set of variables for which there are computational uses in i; and $p\text{-}use(j,k)$ be the set of variables for which there are predicate uses in edge (j,k) [28]. Thus, for each integrated unit U_b we define the following sets:

- $C\text{-}USE\text{-}INTEGRATED(U_b, x)$ is the set of nodes i in U_b such that $x \in c\text{-}use(i)$ and there is a def-clear path wrt x from the entry node of U_b to the node i, and $x \in FP\text{-}IN(U_b)$ or $x \in SF\text{-}IN(U_b)$.
- $P\text{-}USE\text{-}INTEGRATED(U_b, x)$ is the set of edges (j,k) in U_b such that $x \in p\text{-}use(j,k)$ and there is a def-clear path wrt x from the entry node of U_b to the edge (j,k), and $x \in FP\text{-}IN(U_b)$ or $x \in SF\text{-}IN(U_b)$.
- $DEF\text{-}INTEGRATED(U_b, x)$ is the set of nodes i in U_b such that $x \in def(x)$ and there is a def-clear path wrt x from the node i to the exit node of U_b, and $x \in FP\text{-}OUT(U_b)$ or $x \in SF\text{-}OUT(U_b)$.

For the U_{ba} we define the following sets:

- $DEF\text{-}BASE(U_{ba}, x)$ is the set of nodes i in U_a such that $x \in def(i)$ and there is a def-clear path wrt x from i to the interaction node, and $x \in AP\text{-}IN(U_{ba})$ or $x \in SF\text{-}IN(U_b)$.
- $C\text{-}USE\text{-}BASE(U_{ba}, x)$ is a set of nodes i in U_a such that $x \in c\text{-}use(i)$ and there is a def-clear path wrt x from the return nodes to i, and $x \in AP\text{-}OUT(U_{ba})$ or $x \in SF\text{-}OUT(U_b)$.
- $P\text{-}USE\text{-}BASE(U_{ba}, x)$ is the set of edges (j,k) in U_a such that $x \in p\text{-}use(i)$ and there is a def-clear path wrt x from the return nodes to (j,k), and $x \in AP\text{-}OUT(U_{ba})$ or $x \in SF\text{-}OUT(U_b)$.

From those definitions, we define the **all-pairwise-integrated-uses** criterion, used to derive testing requirements based on the interface variables of pairs of units.

- **all-pairwise-integrated-uses** (All-PW-Uses$_i$): Π is adequate wrt the all-pairwise-integrated-uses if:

(1) for each $x \in AP\text{-}IN(U_{ba})$ and each $x \in SF\text{-}IN(U_b)$, Π includes a def-clear path wrt x that goes from each node $i \in DEF\text{-}BASE(U_{ba}, x)$ to each node $j \in C\text{-}USE\text{-}INTEGRATED(U_b, I_{ba}(x))$ and each edge $(j,k) \in P\text{-}USE\text{-}INTEGRATED(U_b, I_{ba}(x))$. In other words, this criterion requires the execution of a def-clear path wrt each communication variable from each

relevant definition in the base unit to each computational and predicative use in the integrated unit.

(2) for each $x \in AP\text{-}OUT(U_{ba})$ and each $x \in SF\text{-}OUT(U_b)$, Π includes a def-clear path wrt x from each node $i \in DEF\text{-}INTEGRATED(U_b, O_{ba}(x))$ to each node $j \in C\text{-}USE\text{-}BASE(U_{ba}, x)$ and each edge $(j,k) \in P\text{-}USE\text{-}BASE(U_{ba}, x)$. In other words, this criterion requires the execution of a def-clear path wrt each communication variable from each relevant definition in the integrated unit to each computational and predicate use in the base unit.

An exception to clause (2) has to be addressed, wrt the definition of formal parameters inside the integrated unit and their following uses after returning to the base unit. Variables in Java hold only primitive type values or object references and both are passed by value. When the actual parameter is of a reference type, the corresponding formal parameter receives and loads the address of the object in memory referred to by the actual parameter. We can say that the formal parameter is a copy of the actual parameter. Thus, any modification of the value of the copy of an actual parameter is not going to affect a later use of it, regardless of the type of the actual parameter (reference or primitive). Therefore, when there are later uses of the actual parameter after the interaction, def-use pairs are not generated for them.

The same does not occur when the actual parameter is of a reference type and its copy modifies through the reference address the fields of the object referred to by the actual parameter. In this case the definition will affect the actual parameter, since the object that it references was modified. Therefore, if there is a use of the actual parameter after the interaction, a def-use pair is generated for it.

4.3.3 Example

Table 3 shows the requirements derived for the All-PW-Nodes$_i$, the All-PW-Edges$_i$, and the All-PW-Uses$_i$ criteria for the pairs of units *pmf – combination* ($R_{x,pmf-combination}$) and *pmf – printOperands*$_2$, second execution of *printOperands* ($R_{x,pmf-printOperands_2}$). 'R' refers to Requirements; 'n', 'e', and 'u' refer to nodes, edges, and uses. The related source code, $\mathcal{AODU}$ graphs, and $\mathcal{PWDU}$ graphs were shown in Figures 2, 3, 6, and 7.

Notations (x,i,j) and $(x,i,(j,k))$ used to represent the R_u requirements (the latter not present in this example) indicate that a variable x is defined in node i and there is a computational use of x in node j (with a def-clear path wrt x going from one node to another) or a predicate use of x in edge (j,k) (also with a def-clear path wrt x going from one node to the edge). This notation uses the name of the variable as defined in the base unit. For instance, the

Table 3
Set of testing requirements derived by the pairwise integration testing criteria for the *pmf – combination* and the *pmf – printOperands* interfaces.

Criterion	**Requirements**
All-PW-Nodes$_i$	$R_{n,pmf-combination}$ = { $i.0$, $i.5$, $i.10$, $i.15$, $i.25$, $i.30$, $i.32$, $i.36$, $i.42$, $i.51$ } $R_{n,pmf-printOperands_2}$ = { $i.0$, $i.19$, $i.44$, $i.66$ }
All-PW-Edges$_i$	$R_{e,pmf-combination}$ = { $(i.0, i.5)$, $(i.0, i.15)$, $(i.5, i.10)$, $(i.5, i.15)$, $(i.10, i.25)$, $(i.25, i.30)$, $(i.25, i.32)$, $(i.32, i.36)$, $(i.36, i.42)$, $(i.42, i.51)$ } $R_{e,pmf-printOperands_2}$ = { $(i.0, i.44)$, $(i.0, i.19)$, $(i.44, i.66)$, $(i.19, i.66)$ }
All-PW-Uses$_i$	$R_{u,pmf-combination} = \emptyset$ $R_{u,pmf-printOperands_2}$ = { $(thirdTermBase, 30, i.0)$, $(thirdTermBase, 30, i.19)$, $(thirdTermBase, 30, i.44)$, $(thirdTermExp, 30, i.66)$ }

requirement $(thirdTermBase, 30, i.0)$ indicates that variable $thirdTermBase$ is defined in node 30 and computationally used in node $i.0$. Note that the use in node $(i.9)$ is in fact related to variable b in the integrated unit, which corresponds to the communication variable $thirdTermBase$ in the base unit.

The requirements generated for the *pmf – combination* pair for the All-PW-Nodes$_i$, shown in the screenshot of the JaBUTi/PW-AJ tool in Figure 5(b), are listed in $R_{n,pmf-combination}$ (these requirements refer to the white nodes in the $\mathcal{PWDU}$ presented in Figure 5(a)). Also, the All-PW-Uses$_i$ requirements for the $\mathcal{PWDU}$ shown in Figure 7(b) are listed in $R_{u,pmf-printOperands_2}$. $R_{u,pmf-combination}$ is empty because there are no communication variables for that interface.

4.4 Strategy

As pointed out in the beginning of this section, most testing processes divide the testing activity in three levels: (1) Unit testing, (2) Integration testing, and (3) System testing [3]. Following this strategy, our pairwise criteria should be more effectively applied after unit testing the program. Thus, the natural testing strategy to be followed in this context would be: (1) focus on each unit by testing each method and advice in isolation (by using, for instance, the criteria proposed before [18,36,37]); (2) focus on the integration of units inside each module by testing the intra-module pairs; and (3) focus on the integration of modules by testing the inter-module pairs. Moreover, the test set used in the precedent level can be a starting point set for the next level, and it can be enhanced as needed. In this way we can have a more efficient testing activity.

For instance, consider the example presented in Figures 2 and 3. A natural approach to test the program would be the following: (1) unit test each method and advice; (2) test the intra-module pairs inside class *BinomialDistribution*

by using the available unit test set as a starting point; and (3) test the inter-module pairs related to the interfaces between *pmf* and the *printOperands* advice by using the available test set as a starting point. Section 3 showed a walk through this example using the tool we have implemented and this strategy.

Another concern is the order in which the criteria themselves are applied at each level. Using the same mindset, they should be applied in order of strength, starting with the easiest to cover. For instance, in unit testing, the order in which the criteria should be applied is the following: all-nodes, all-edges and all-uses (because the last subsumes the second which subsumes the first [44]). The same idea could be applied for intra-module and inter-module testing, the order should be the following: all-pairwise-integration-nodes, all-pairwise-integration-edges and all-pairwise-integration-uses.

5 Tool support

The JaBUTi family of tools [18,36,37] was extended to support the pairwise testing of Java OO programs and AspectJ AO programs. The implementation of the extended tool, called JaBUTi/PW-AJ (for PairWise AspectJ) – whose usage was exemplified in Section 3 –, was divided in four parts. Each part handled the following features: (1) identification of pairs of units that interact with each other in an application; (2) generation of the $\mathcal{PWDU}$ graphs; (3) implementation of the proposed criteria; and (4) implementation of the intra and inter-module testing environment (a more complete description of the JaBUTi/PW-AJ tool implementation can be found elsewhere [7]).

The identification of the pairs of units is made by scanning the Java bytecode of each unit, searching for the following instructions: *invokevirtual*, *invokespecial*, *invokestatic*, and *invokeinterface*. These instructions identify interactions between units, both through method calls and advice executions (see Section 2). Moreover it is also possible to check which unit is being called and to which module it belongs to, thus making it possible to determine whether the interaction is intra-module or inter-module. The name given to the interaction pairs follows a naming pattern composed of two parts, one representing the base unit and the other representing the integrated unit. Each part is formed by the full qualified name of the module within parentheses and the signature of the unit at bytecode level [20]. A '-' is used to separate the parts. Figure 9 shows an example of naming for a pair of units that interact in the previously shown application. Note that both classes are implemented under the *br.math* package.

When a unit interacts with another unit more than once in its body (*e.g*,

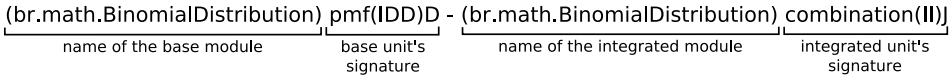


Fig. 9. Naming convention for a pair of interacting units.

two calls to the same method), we enumerate them. Also, since there may be polymorphic calls for which called methods cannot be determined at compile time, we generate pairs for each method that can be possibly called. For this case we also use a "<P>" before the called unit, to indicate that it refers to a polymorphic call.

Figure 10 shows the intra-module pairs identified by JaBUTi/PW-AJ for the example presented in Figure 3. The top part shows which classes (and possibly, aspects) present intra-module interactions, and the bottom shows the pairs of interacting units for those classes. In this case there is a single class with intra-module interactions. Note that, since *exponentiation* is called twice in *pmf*, there are two pairs relating these methods.

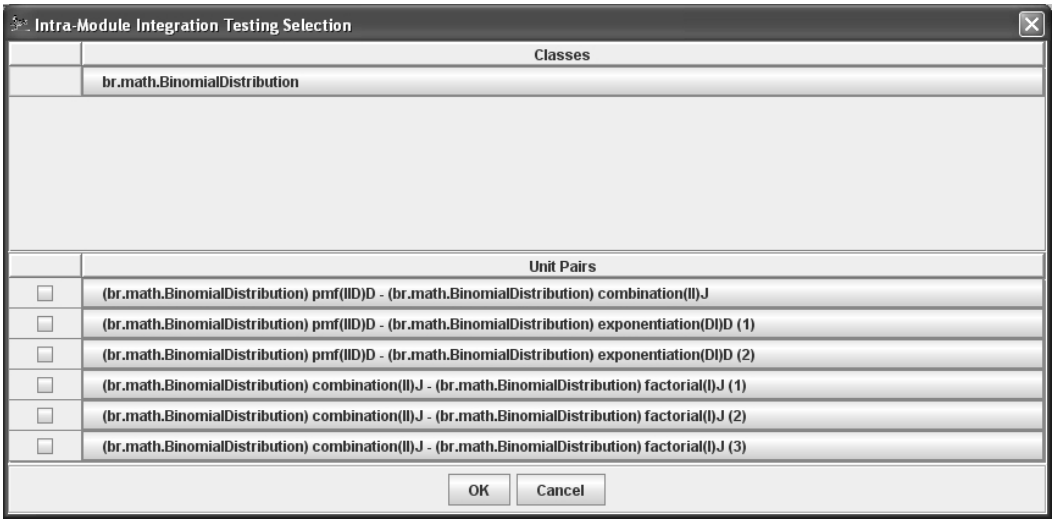


Fig. 10. Pairs of intra-module units identified for the *BinomialDistribution* class.

6 Exploratory evaluation

We conducted an experiment to evaluate two questions related to the approach proposed in this paper: (1) in what situations applications need additional test cases to cover the pairwise criteria, considering that they had already been unit tested (*i.e.*, 100% of unit coverage wrt the all-nodes, all-edges, and all-uses criteria)?; and (2) when additional test cases are indeed required, what is the effort to adequate these unit test sets to the pairwise criteria? The first question is related to the usefulness of our approach: if unit testing is enough to cover the pairwise criteria, there is no point in using them. The

second question is related to cost and feasibility: testing criteria that require an enormous number of additional test cases are impractical.

The hypothesis of our experiment was that, with respect to question (1), most applications would indeed require additional test cases for the pairwise testing phase, thus enforcing the construction of more robust test sets. However, with respect to question (2), the hypothesis was that the number of additional test cases would be small, when indeed required. This is because of two intuitions: (1) since we are covering all statements, conditionals, and def-use pairs inside each unit, we also cover all method calls and join points present in the body of each unit; and (2) since, according to an empirical study conducted by Souter and Pollock [30], OO programs usually contain units with a small number of statements and simple intraprocedural control-flow (the static branch count within a method is often zero, and on average between 0 and 3), we end up covering much of the integrated units with the unit test cases. If we reach every interface present in the program with the unit test set, there usually remains only a few paths to be covered at the pairwise integration testing level. More importantly, these are the paths that might contain unrevealed faults related to interface problems (such as in the motivation example presented in this paper).

Inspired by a study conducted by Xu and Rountev [42] to evaluate a regression testing technique for AO programs, we collected 7 subjects to conduct our experiment. Some of them have also been used in other experiments [27,29,42]. The subjects were collected from different sources; the following are applications taken from the AspectJ distribution [1] and also used as a benchmark by Hendren et al. [27]: a subject-observer pattern implementation using aspects (*subj-obs*); a Java Beans component application using aspects (*bean*); a geometrical point application with inter-type declarations (*point*); a simulation of a telephony system using aspects (*telecom*); and a two-dimension graphics application that models some shapes and presents a tracing aspect (*shape*). The other two applications were taken from a paper by Rinard et al. [29] (a Stack implementation with an aspect to prevent negative numbers to be passed as parameters – *stack*) and from an Enterpriser AspectJ tutorial presented by Bodkin and Laddad [4] (an online music service – *music*).

For each application we collected the following information: number of classes (*#Cs*); number of aspects (*#As*); number of units, *i.e.*, methods and pieces of advice (*#u.*); number of intra-module pairs (*#intra p.*); number of inter-module pairs (*#inter p.*); number of lines of code (*#LOC*); number of test cases required to cover 100% of all-nodes, all-edges, and all-uses of each unit (*#u. TCs*); number of test cases added to the initial unit test set to cover the pairwise criteria (*#ad. TCs*); and percentage of additional test cases in relation to the number of initial test cases (*%ad. TCs*). We used the number and percentage of additional test cases as a measure for the effort to cover the

requirements of the pairwise criteria.

Table 4
Data collected from testing 7 AO applications using the pairwise structural testing criteria.

Subject	**#Cs**	**#As**	**#u.**	**#intra p.**	**#inter p.**	**#LOC**	**#u. TCs**	**#ad. TCs**	**%ad. TCs**
stack	4	2	13	0	9	56	5	0	0%
subj-obs	5	2	14	0	17	106	6	0	0%
bean	1	1	15	5	11	153	5	0	0%
point	1	3	15	10	12	254	14	1	7%
telecom	6	3	46	9	37	321	22	2	9%
music	10	2	45	4	29	132	19	3	16%
shape	5	1	52	32	110	191	25	6	24%
Average	4.57	2	28.57	8.57	32.14	173.28	13.71	1.71	12.5%

Analyzing Table 4 we can see that the larger tested applications (in number of units) required additional test cases for the pairwise criteria, thus confirming our hypothesis that most applications require more robust test sets for these criteria. Moreover, we can also see the number of additional test cases tend to increase according to the number of units of the application. With respect to effort, we can see that only a small number of test cases were required to cover the additional requirements of the pairwise criteria (12.5% – 1.71 – additional test cases in average), when indeed required. This is an evidence for our hypothesis that a small effort is required to adequate an initial unit test set – already adequate for the all-nodes, all-edges, and all-uses unit testing criteria – to the pairwise testing criteria.

6.1 Threats to Validity

Empirical studies present limitations that must be considered when interpreting the results. In our case, the primary threats are related to subject representativeness, affecting the ability of our results to generalize. The applications considered are of small size and commercial applications with different characteristics may be subject to different cost-effectiveness trade-offs.

For instance, in applications with units that interact with others and have several conditional statements, it is more likely that they will require several additional test cases to cover the pairwise criteria. However, we must also consider that, as commented earlier, OO programs have been reported as having simple intraprocedural control-flow with small number of conditional statements [30,41]. In fact, in a recent study conducted within our group [17] to evaluate the cost of application of different integration criteria, results showed that the number of additional test cases are highly dependent on the complexity, number, and interactions of units (that is the reason why some applications in Table 4 with more #LOC than others ended up requiring less additional test

cases). Again, since units in OO programs usually have simple intraprocedural control-flow, the pairwise criteria can be considered applicable in general.

With respect to AO programs, a problematic case is when pieces of advice affect the program at several join points and also present conditional statements. In this case, it is again more likely that several additional test cases may be required at the pairwise integration testing phase. However, since pieces of advice are structurally similar to methods, we can use the same argument that very few applications present advice with complex control-flow (in fact, in our experience, we have seen that advice logic is usually simpler than method logic). For instance, the *shape* application presents several inter-module pairs (110) mainly because of two tracing pieces of advice that affect every unit in the program. But since they do not present conditional statements, the unit test cases were enough to cover all interactions. Moreover, as commented by Lesiecki [19], advice should be made more testable by moving its actual logic to methods, and calling them from the advice. In this case, we have only a single intra-module pair with more 'complex' control-flow and several inter-module pairs with simple control-flow, such that the latter would necessarily be covered while unit testing the program. Additionally, in the same study mentioned in the last paragraph [17], results showed that advice is usually applied to few join points in the systems, which implies a more practical pairwise testing activity for AO programs.

7 Conclusion and Future work

In this paper we have presented an approach for pairwise structural testing of OO and AO Java programs. The approach includes a model to represent the structure of pairs of interacting units and three testing criteria to enhance the confidence in those interfaces. Since we consider each pair of units separately, the practicality of the testing activity is also considered. Even though we applied the idea to Java programs, we also believe that it could be generalized to other languages (such as C++).

According to an empirical study conducted on a set of large Java applications, Souter et al. [32] found that OO design commonly results in systems with many methods, each with a small number of statements and simple intraprocedural control-flow. Based on the same study, they claim that computation is achieved primarily through manipulation of instance variables of objects via method calls. This suggests that unit testing is not enough to thoroughly test OO programs (and, by transitivity, AO programs as well). Moreover, since computation is generally achieved via method calls, testing method interfaces with the pairwise criteria presented in this paper is an important way to enhance the confidence in these types of programs.

The infeasibility issue, related to paths required by criteria which cannot be covered, poses an undecidable problem that can also occur in our context [38,44]. For instance, there can be conditions in the integrated unit that can never be satisfied through inputs issued on the base unit, generating infeasible requirements. In the example tested through Section 3, there are 5 infeasible requirements; however, for the 7 experimental subjects mentioned in Section 6, no additional infeasible requirements were found while pairwise testing the applications. In any case, this problem is minimized by the support JaBUTi/PW-AJ provides to the tester to indicate which requirements are infeasible, which discards them. Additionally, note that the number of infeasible requirements is always smaller than the number of requirements of the pairwise criteria. Considering that the application has already been unit tested – such as the subjects of our experiment –, we provided evidence that the number of additional requirements is usually small. Therefore, in this common situation, we can say that the possible number of infeasible requirements will also be relatively small.

Future work includes studying whether it is possible to enlarge the integration of units considering deeper call chains, without making the integration testing activity too expensive. An idea is to make the depth configurable and defining criteria based on a n-depth integration strategy. Moreover we also want to extend the idea presented in this paper to test *clusters* of units. For instance, instead of considering only pairs of interacting units, we would consider a unit with all the units that interact with it in a single level of depth or in a configurable level of depth. With respect to AO programs and their specific types of faults, we are also investigating a way of collecting sets of interacting pairs of units related to each pointcut, to detect faulty interfaces added by aspects [15–17].

References

[1] AspectJ Team. The AspectJ programming guide. Online, 2003. Available from: `http://www.eclipse.org/aspectj/doc/released/progguide/index.html` (accessed 11/27/2007).

[2] V. R. Basili and B. T. Perricone. Software errors and complexity: an empirical investigation. *Commun. ACM*, 27(1):42–52, 1984.

[3] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[4] R. Bodkin and R. Laddad. Enterprise AspectJ tutorial using eclipse. Online, 2005. EclipseCon 2005. Available from: `http://www.eclipsecon.org/`

2005/presentations/EclipseCon2005_
EnterpriseAspectJTutorial9.pdf (accessed 12/3/2007).

[5] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: An approach for integration testing. *IEEE Trans. Softw. Eng.*, 27(3):228–247, 2001.

[6] T. Elrad, G. Kiczales, M. Akşit, K. Lieberher, and H. Ossher. Discussing aspects of AOP. *Communications of the ACM*, 44(10):33–38, 2001.

[7] I. G. Franchin, O. A. L. Lemos, and P. C. Masiero. Pairwise structural testing of object and aspect-oriented Java programs. In *Proceedings of the 21st Brazilian Symposium on Software Engineering*, pages 377–393, Porto Alegre, RS, Brasil, 2007. SBC Press.

[8] A. Haley and S. Zweben. Development and application of a white box approach to integration testing. *J. Syst. Softw.*, 4(4):309–315, 1984.

[9] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 312–326, New York, NY, USA, 2001. ACM Press.

[10] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick. Incremental testing of object-oriented class structures. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 68–80, New York, NY, USA, 1992. ACM Press.

[11] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 154–163, New York, NY, USA, 1994. ACM Press.

[12] E. Hilsdale and J. Hugunin. Advice Weaving in AspectJ. In *Proceedings of the 4th AOSD 2004*, pages 26–35, Lancaster, UK, 2004.

[13] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. Lopes, C. Maeda, and A. Menhdhekar. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the ECOOP*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[14] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the* 27^{th} *International Conference on Software Engineering (ICSE'2005)*, pages 49–58. ACM Press, 2005.

[15] O. A. L. Lemos, F. C. Ferrari, P. C. Masiero, and C. V. Lopes. Testing aspect-oriented programming pointcut descriptors. In *WTAOP '06: Proceedings of the 2nd workshop on Testing aspect-oriented programs*, pages 33–38, New York, NY, USA, 2006. ACM Press.

[16] O. A. L. Lemos and P. C. Masiero. Using structural testing to identify unintended join points selected by pointcuts in aspect-oriented programs. In

Proc. of the 32nd Annual IEEE Software Engineering Workshop (SEW 2008). IEEE Press, October 2008.

[17] O. A. L. Lemos and P. C. Masiero. A pointcut-based coverage analysis approach for aspect-oriented programs (submitted for publication). 2009.

[18] O. A. L. Lemos, A. Vincenzi, J. C. Maldonado, and P. C. Masiero. Control and data-flow structural testing criteria for aspect-oriented programs. *Journal of Systems and Software*, 80(6):862–882, 2007.

[19] N. Lesiecki. Unit test your aspects. Online, 2005. IBM DeveloperWorks. Available from: `http://www-128.ibm.com/developerworks/java/library/j-aopwork11/` (accessed 12/3/2007).

[20] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Prentice Hall PTR, 2 edition, 1999.

[21] U. Linnenkugel and M. Müllerburg. Test data selection criteria for (software) integration testing. In *ISCI '90: Proceedings of the first international conference on systems integration '90*, pages 709–717, Piscataway, NJ, USA, 1990. IEEE Press.

[22] J. C. Maldonado. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*. PhD thesis, DCA/FEE/UNICAMP, Campinas, SP, July 1991.

[23] M. Mortensen and R. T. Alexander. An approach for adequate testing of AspectJ programs. In *Proceedings of the* 1^{st} *Workshop on Testing Aspect Oriented Programs – in conjunction with AOSD'2005*, Chicago/IL, USA, 2005.

[24] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas. *The Art of Software Testing*. John Wiley & Sons, 2nd. edition, 2004.

[25] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 241–251, New York, NY, USA, 2004. ACM Press.

[26] A. Paradkar. Inter-Class Testing of O-O Software in the Presence of Polymorphism. In *Proceedings of the 1996 Conference of the Centre For Advanced Studies on Collaborative Research*, page 30, Toronto, Ontario, Canada, 1996. IBM Press.

[27] Programming Tools Group and Sable Research Group. abc: The aspectbench compiler for AspectJ. Online, 2007. Available from: `http://abc.comlab.ox.ac.uk/benchmarks` (accessed 11/27/2007).

[28] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367–375, 1985.

[29] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of the* 12^{th} *ACM*

SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT'2004/FSE-12), pages 147–158, Newport Beach/CA - USA, 2004. ACM Press.

[30] A. L. Souter and L. L. Pollock. Omen: A strategy for testing object-oriented software. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 49–59, New York, NY, USA, 2000. ACM Press.

[31] A. L. Souter and L. L. Pollock. The construction of contextual def-use associations for object-oriented systems. *IEEE Trans. Softw. Eng.*, 29(11):1005–1018, 2003.

[32] A. L. Souter, L. L. Pollock, and D. Hisley. Inter-class def-use analysis with partial class representations. In *PASTE '99: Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 47–56, New York, NY, USA, 1999. ACM Press.

[33] K. Stobie. Too darned big to test. *Queue*, 3(1):30–37, 2005.

[34] K. C. Tai and Y. Lie. A test generation strategy for pairwise testing. *IEEE Trans. Softw. Eng.*, 28(1):109–111, 2002.

[35] P. R. S. Vilela, J. C. Maldonado, and M. Jino. Data Flow Based Integration Testing. In *Anais do 13ž Simpósio Brasileiro de Engenharia de Software*, pages 393–409, Florianópolis, SC, Brasil, 1999.

[36] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado, and W. E. Wong. Establishing structural testing criteria for java bytecode. *Softw. Pract. Exper.*, 36(14):1513–1541, 2006.

[37] A. M. R. Vincenzi, J. C. Maldonado, W. E. Wong, and M. E. Delamaro. Coverage testing of java programs and components. *Science of Computer Programming*, 56(1–2):211–230, 2005.

[38] E. J. Weyuker. The applicability of program schema results to programs. *Int. J. Comput. Inf. Sci.*, 8(5):387–403, 1979.

[39] E. J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Commun. ACM*, 31(6):668–675, 1988.

[40] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 187–206, New York, NY, USA, 1999. ACM Press.

[41] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Trans. Softw. Eng.*, 18(12):1038–1044, 1992.

[42] G. Xu and A. Rountev. Regression test selection for AspectJ software. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 65–74, Washington, DC, USA, 2007. IEEE Computer Society.

[43] J. Zhao. Data-flow-based unit testing of aspect-oriented programs. In *Proceedings of the* 27^{th} *Annual IEEE International Computer Software and Applications Conference (COMPSAC'2003)*, pages 188–197, Dallas/Texas - USA, 2003. IEEE Computer Society.

[44] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.