Pairwise structural testing of object and aspect-oriented Java programs*

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

Departamento de Sistemas de Computação,
ICMC/USP - São Carlos - Caixa Postal 668
13560-970 São Carlos-SP-Brasil

{ivan;oall;masiero}@icmc.usp.br

Resumo. *Em um trabalho recente, explorou-se o teste estrutural de unidade – tanto de métodos quanto de adendos – de programas Java orientados a objetos (OO) e a aspectos (OA). Um problema não tratado pelo teste de unidade é a interação entre as unidades, no que diz respeito à corretude das interfaces. Neste artigo é apresentada uma abordagem de teste de integração que estende a abordagem de teste de unidade apresentada anteriormente. Para que a atividade de teste seja factível, em vez de considerar níveis arbitrários de chamadas de uma só vez, trata-se do teste de cada par de unidades. Um modelo para representar o fluxo de controle e de dados de pares de unidades de programas Java OO e OA chamado grafo $\mathcal{PWDU}$ (PairWise Def-Use) é proposto juntamente com três critérios de teste. Uma implementação da abordagem utilizando como base a família de ferramentas de teste JaBUTi (Java Bytecode Understanding and Testing) juntamente com um exemplo de uso também são apresentados.*

Abstract. *Most structural testing approaches are targeted at units of implementation (i.e., unit testing). A problem that is not addressed by unit testing is the interaction among units, with respect to the correctness of their interfaces. We present a structural integration testing approach for object-oriented (OO) and aspect-oriented (AO) Java programs as an extension of a unit testing approach we have developed before. To make the activity feasible, instead of considering arbitrary call depths, we address the testing of pairs of units. We propose a model called $\mathcal{PWDU}$ (PairWise Def-Use) graph to represent the control and data-flow of pairs of units. Based on the $\mathcal{PWDU}$, three testing criteria are defined: all-pairwise-integrated-nodes, all-pairwise-integrated-edges (control-flow based criteria), and all-pairwise-integrated-uses (a data-flow based criterion). We also present the implementation of our approach as an extension to the Java Bytecode Understanding and Testing (JaBUTi) family of testing tools along with an example of usage.*

1. Introduction

In a recent work [Lemos et al. 2007], we explored the structural testing of units – both methods and pieces of advice – of object-oriented (OO) and aspect-oriented (AO) Java programs in isolation (*i.e.*, unit testing). A problem that is not addressed by unit testing

*The authors are financially supported by CNPq and FAPESP.

is the interaction among units, with respect to (wrt) the correctness of their interfaces. In such context, unit testing is not enough. In this paper we propose a step ahead to provide more confidence in OO and AO Java programs: integration structural testing of units that interact with each other.

Since even for small systems there might be a great number of interactions among units, it is usually infeasible to test the integration of units in arbitrary call depths. Moreover, for large systems the problem can be exponentially worse [Stobie 2005]. Therefore to keep the integration testing activity more feasible, we propose the testing of pairs of units, both intra-module (units that interact with each other inside classes and aspects) and inter-module (units of different classes and aspects that interact with each other). We thus also address a problem that has not been fully addressed yet: the case of interactions between advice and methods, which is an important issue related to testing AO programs.

Based on a Java bytecode data-flow and control-flow model, we defined three specific testing criteria to test both OO and AO Java programs. These model and criteria were implemented in a testing tool, extended from a family of tools named JaBUTi (Java Bytecode Understanding and Testing) [Vincenzi et al. 2006].

The rest of the paper is structured as follows. Section 2 presents basic knowledge about AO programming (since OOP can be considered common knowledge), to provide a basis to understand our approach; and Section 3 presents the proposed model and criteria for testing pairs of OO and AO Java programs. Section 4 presents our prototype testing tool that implements our model and criteria and Section 5 presents an example and explanation of the tool usage. Finally, Section 6 concludes and discusses future work.

2. AOP

The AOP main idea is that while OO programming, procedural and other programming techniques by themselves help separating out the different concerns implemented in a software system, there are still some requirements that cannot be clearly mapped to isolated units of implementation [Kiczales et al. 1997]. Examples of those concerns are 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 [Elrad et al. 2001].

AOP supports the construction of separate units – called aspects – that have the ability to cut across the system units, defining behavior that would otherwise be spread throughout the base code. A generic AOP language should define: a model to describe hooks in the base code where additional behavior may be defined (these hooks are called *join points* which, for our purposes, are well-defined points in the execution of a program); a mechanism of identification of these join points; units that encapsulate both join point specifications and behavior enhancements; and a process to combine both base code and aspects (which is called the *weaving* process) [Elrad et al. 2001].

2.1. The AspectJ Language

AspectJ is an extension of the Java language to support AOP. In AspectJ, aspects are units that combine: join point specifications (pointcuts), pieces of advice, which are the desired

```
01. public aspect Logging {
02.   pointcut loggedOp(int a1, int a2):
03.     execution(* src.Calculus.calculate(..))
          && args(a1,a2);
04.
05.   before(int a1, int a2) : loggedOp(a1,a2) {
06.     System.out.println("Numbers: " +
          a1 + " and " + a2);
07.   }
08. }

09. public class Calculus {
10.   public int resSum;
11.   public int resSub;
12.
13.   public void calculate(int p1, int p2) {
14.     this.resSum = p1 + p2;
15.     if ( p1 > p2 )
16.       this.resSub = p1 - p2;
17.     else
18.       this.resSub = p2 - p1;
19.   }
20. }

21. public class Main {
22.   public static void doCalculation(
        int d1, int d2) {
23.     int num1 = d1;
24.     int num2 = d2;
25.     Calculus calc = new Calculus();
26.     calc.calculate(num1, num2);
27.     System.out.println("Sum = " +
          calc.resSum);
28.     System.out.println("Subtraction = " +
          calc.resSub);
29.   }
30. }
```

Figure 1. Source code of a simple sum and subtraction application with a logging aspect.

behavior to be added at the join points and methods, fields and inner classes. Also, aspects can declare members (fields and methods) to be owned by other types, what is called inter-type declarations. The current version of AspectJ also support declarations of warnings and errors that arise when join points are identified or reached [AspectJ Team 2003]. 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, respectively. These constructs can also pick context information from the join point that has caused them to execute. Figure 1 lists the source code of a simple sum and subtract aspect-oriented program that will be used along this paper.

In any AOP language implementation, aspect and non-aspect code must run in a properly coordinated fashion. In order to do so an important issue is to ensure that pieces of advice run at the appropriate join points as specified by the program. Previous versions of AspectJ used the strategy of inlining the advice code directly into the join points, which resulted in `.class` files that were a mixture of aspect and non-aspect code [Hilsdale and Hugunin 2004]. In fact, older versions of AspectJ did the weaving process based on source code: first files were pre-processed into standard Java and then these new files were compiled with a standard compiler. The recent AspectJ advice

weaver is based on bytecode, so this process is made by bytecode transformation rather than on source code files.

The AspectJ advice weaver statically transforms the program so that at 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 the aspect and advice declarations into standard Java classes and methods (at bytecode level). Parameters of the pieces of advice are now parameters of these new methods (with some special treatment when reflexive information is needed). In order to coordinate aspects and non-aspects the system code is instrumented and calls to the "advice methods" are inserted considering that certain regions of the bytecode represent possible join points (these are called join point *static shadows*). Furthermore if the join point cannot be completely determined at compile time, these calls are guarded by dynamic tests to make sure that the pieces of advice run only at appropriate time (these tests are called *residues*) [Hilsdale and Hugunin 2004].

3. Structural testing of OO and AO programs

Testing is the execution of a piece of software with the intent of finding faults [Myers et al. 2004]. The different testing techniques can be classified by the artifact used to derive the testing requirements. *Functional testing* derives its requirements from the specification of the system, without regarding 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 software development process.

Software testing is usually performed in three phases:

1. Unit testing, where the smallest pieces of the software 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;
3. and system testing, which consists in verifying the integration of all elements of the software to assure that the system and these other elements (for instance, hardware and data bases) combine adequately and that expected global functioning/performance is obtained.

Harrold and Rothermel [Harrold and Rothermel 1994] 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 a class (which is not considered in our approach). In their case, no restrictions were made with respect to the depth of calls that were considered, which can make the implementation of the approach infeasible. In fact, no implementations of the approach were provided by the authors. The same leading author together with other colleagues explored some other problems related to OO program testing such as regression testing and incremental testing of OO program structures [Harrold et al. 1992, Harrold et al. 2001, Orso et al. 2004].

Vilela et al. have also proposed a pairwise integration testing approach, but in their case targeted at procedural programs [Vilela et al. 1999]. Based on the work of Linnenkugel and Müllerberg [Linnenkugel and Müllerburg 1990] (also used for definitions

related to data-flow testing in this paper), they extended the family of Potential-Uses data-flow criteria [Maldonado 1991] to the pairwise integration testing of procedural programs. Paradkar [Paradkar 1996] uses the idea of pairwise testing to the integration of classes.

In this paper we focus on integration testing, building on top of unit testing approaches described in other papers [Lemos et al. 2007, Vincenzi et al. 2006]. 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 purpose 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 [Rapps and Weyuker 1985].

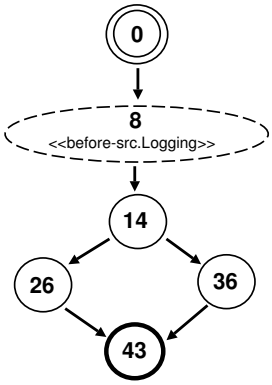
For our purposes the occurrence of a variable is either classified as a definition or use. As to the use occurrences it is called a *predicate* use (or p-use) a use of a variable in a conditional statement – for instance: `if (i == 5)` – and a *computational* use a use of a variable that directly affects 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 wrt some variable is then a pair of definition and subsequent use locations such that there is a definition clear path wrt that same variable from the definition to the use location [Rapps and Weyuker 1985].

The basic unit testing model for OO and AO Java programs is the aspect-oriented def-use ($\mathcal{AODU}$) graph [Lemos et al. 2007], that builds on top of Vincenzi et al.'s work 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*). The element C was added to the original def-use model defined by Vincenzi et al. [Vincenzi et al. 2006] to represent the basic interaction that occur in AO programs. Examples of $\mathcal{AODU}$ graphs are presented in Figure 2. The units refer to methods of the example presented in Figure 1. Note that the crosscutting nodes are represented by dashed elipses containing additional information of what kind of advice is affecting that point and to which aspect it belongs.

3.1. Pairwise integration testing

The anticomposition axiom defined by Weyuker [Weyuker 1988] states that testing each piece of a program in isolation is not necessarily sufficient to deem the entire program

(a) $\mathcal{AODU}$ of the `doCalculation` method



(b) $\mathcal{AODU}$ of the `calculate` method

Figure 2. Examples of $\mathcal{AODU}$s.

adequately tested. Thus, additional testing is required when units are combined or integrated. For integration testing, the interface between the units is the focus. Interface problems include errors in input-output format, incorrect sequencing of subroutine calls, and misunderstood entry or exit parameter values. Therefore, after each unit is individually tested, it is interesting to test the interactions among them [Harrold et al. 1992].

Concerned with this problem, we propose the extension of our unit testing approach to integration testing of pairs of units. To keep the activity more feasible, we propose the testing of each pair of unit at a time, instead of addressing arbitrary call depths at once.

With that purpose in mind, we propose a structural model called $\mathcal{PWDU}$, to represent the structure of a pair of units. For OO programs we have only one type of pair 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 another method and advice-advice – when an advice is affected by another advice.

Based on the $\mathcal{PWDU}$ we derive three testing criteria, to make sure that the structure of the integrated units are being thoroughly covered, both from the control-flow and from the data-flow perspectives.

3.2. $\mathcal{PWDU}$: a control/data-flow model for Java Bytecode

To adequately represent the execution flow that occurs inside a pair of units, we need to define a graph which integrates the units' def-use graphs. With that intention we define the $\mathcal{PWDU}$, which integrates the $\mathcal{AODU}$s of the pairs of units that interact with each other.

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 the crosscutting nodes – which represent the unit-advice interaction – plus the call nodes that

represent method calls – the unit-method interactions. With these types of nodes we are able to identify all interactions among units of OO and AO programs.

We call the unit in the pair that is either calling a method or being affected by an advice as the *base* unit and the unit to which the control flow can be passed as 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 units, we define the *integrated* nodes which represent the nodes of the integrated unit and two kinds of edges: the *integrated* edges which are the edges that connect two integrated nodes, and the *integration* edges which represent the flow of control between a node of the base unit and a node of the integrated unit, and vice-versa.

Figure 3 presents an example of a $\mathcal{PWDU}$ for the `doCalculation` and `calculate` methods whose source code and $\mathcal{AODU}$ graphs were presented in Figures 1 and 2. The table in the upper left corner shows the mapping of the communication variables used for the data-flow criterion (see Section 3.3) and the notes coming from the nodes and edges show which variables are being defined (def), computationally-used (cu), or predicatively-used (pu) at those places.

3.3. A family of pairwise structural testing criteria

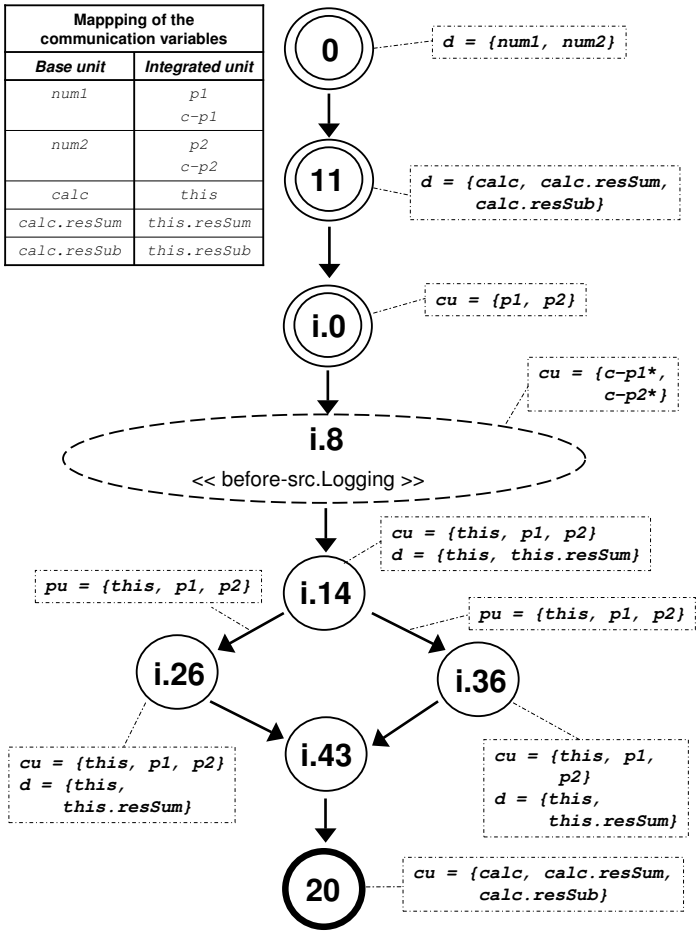
Testing criteria are a very important way to provide systematic selection and evaluation of test sets. To enhance the confidence that two units are combined in a correct way, we propose three structural testing criteria: two control-flow based and one data-flow based. The main idea is to make sure that the integrated unit is thoroughly covered by test cases issued 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, $1 \leq j \leq m - 1$.

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 $\mathcal{PWDU}$



*Variables `c-p1` and `c-p2` are copies of `p1` and `p2`. They are generated by the AspectJ compiler during weaving.

Figure 3. The $\mathcal{PWDU}$ for the pair of methods doCalculation and calculate.

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.

Data-flow criterion

With respect to the data-flow criteria we decided to revisit the known all-uses criterion. We took the work of Linnenkugel and Müllerberg [Linnenkugel and Müllerburg 1990] 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.

Data-flow based integration testing is about testing the variables that affect 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 an OO and AO program the following communication variables types can be identified:

Formal Parameters – FP; Actual Parameters – AP; and Static field 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.

An OO and AO program consists 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 we define two mappings: I_{ba} and O_{ba}. It is important to note that while doing the parameters and static fields mappings for reference types, 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 units' $\mathcal{PWDU}$, other 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) [Rapps and Weyuker 1985]. 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.

Table 1. Requirements' set derived by the pairwise integration testing criteria.

Criterion	Requirements
All-PW-Nodes$_i$	R$_n$ = { $i.0$, $i.8$, $i.14$, $i.26$, $i.36$, $i.43$ }
All-PW-Edges$_i$	R$_e$ = { $(i.0, i.8)$, $(i.8, i.14)$, $(i.14, i.26)$, $(i.14, i.36)$, $(i.26, i.43)$, $(i.36, i.43)$}
All-PW-Uses$_i$	R$_u$ = { $(num1, 0, i.0)$, $(num2, 0, i.0)$, $(num1, 0, i.8)$, $(num2, 0, i.8)$, $(calc, 11, i.14)$, $(calc, 11, (i.14, i.26))$, $(calc, 11, (i.14, i.36))$, $(num1, 0, i.14)$, $(num1, 0, (i.14, i.26))$, $(num1, 0, (i.14, i.36))$, $(num2, 0, i.14)$, $(num2, 0, (i.14, i.26))$, $(num2, 0, (i.14, i.36))$, $(num1, 0, i.26)$, $(num2, 0, i.26)$, $(num2, 0, i.36)$, $(num1, 0, i.36)$, $(calc.resSum, i.14, 20)$, $(calc.resSub, i.26, 20)$, $(calc, i.26, 20)$, $(calc.resSub, i.36, 20)$, $(calc, i.36, 20)$ }

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. Java only has variables that hold primitives 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 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, if there is a later use of the actual parameter after the interaction, a def-use pair is not created for it.

The same does not occur if the actual parameter is a reference type and its copy modifies, using the reference address, the fields of the object referred 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 will be created for it. The complete data model used to caracterize the definitions and use of variables for Java/AspectJ used in our approach can be found elsewhere [Franchin 2007].

Table 1 shows the requirements derived for the All-PW-Nodes$_i$, All-PW-Edges$_i$ and All-PW-Uses$_i$ criteria for the pair of units `(Main)doCalculate(DD)V - (Calculus)calculate(DD)V` whose source code and $\mathcal{PWDU}$ graph were shown before (Figures 1 and 2).

Notations (x,i,j) and (x,i,(j, k)) used to represent the R$_u$ requirements 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 w.r.t. x going from one to another) or a predicate use of x in edge (j, k) (also with a def-clear path w.r.t. x going from one to another). This notation uses the name of the variable as defined in the base unit. For instance, requirements $(num1, 0, (i.14, i.26))$ and $(calc.resSum, i.14, 20)$ indicate that the variable $num1$ is defined in the node 0 and there is a predicate use in the edge $(i.14, i.26)$. Note that the use in edge $(i.14, i.26)$ is not exactly related to variable $num1$ but $p1$, which is the $num1$ corresponding communication variable in the base unit. The second requirement indicates that variable $calc.resSum$ is defined in node $i.14$ and there is a computational use in node 20. Note that, in this case, the definition that occurs in node $i.14$ is related to $this.resSum$, which is the $calc.resSum$ corresponding communication variable in the base unit.

4. Tool support

The JaBUTi family of tools [Vincenzi et al. 2006, Lemos et al. 2007] was extended to make it possible the use of our pairwise testing approach for Java OO programs and AspectJ AO programs. The extension implementation, that we call JaBUTi/PW-AJ (for pairwise AspectJ), was divided in four parts: the first part was concerned with the identification of the pairs of units that interact with each other in a program; the second part was concerned with the generation of the $\mathcal{PWDU}$ graph; the third part was concerned with the implementation of the criteria; and the forth part was concerned with the implementation of the intra and inter-module testing environment.

4.1. Identification of interacting units

The identification of the units' pairs is made by scanning the Java bytecode of each unit, searching for the `invokevirtual`, `invokespecial`, `invokestatic` and `invokeinterface` instructions. These instructions identify interactions between units, both with method calls and advice interceptions (see Section 2.1). Moreover it is also possible to have knowledge about 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 follow a naming pattern: it is composed of two parts – one for the base unit and another for the integrated unit. Each part is formed by the name of the full qualified name of the module inside parenthesis and the signature of the unit at bytecode level [Lindholm and Yellin 1999]. The '-' is used as a separator between the parts. Figure 4 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 `src` package.

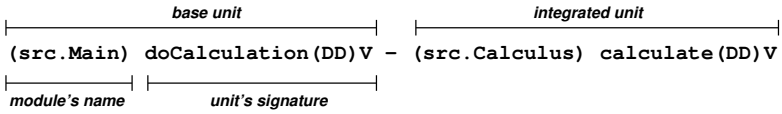


Figure 4. Naming convention for a pair of interacting units.

When a unit interact with another unit in more than one place in its body, we also give a number to the pair between parenthesis, to differentiate each interaction. 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 5 shows the inter-module pairs identified by JaBUTi/PW-AJ for the example presented in Figure 1. The top part shows which classes (and possibly, aspects) present inter-module relations, and the bottom shows the pairs of interacting units for those classes.

4.2. Generation of the $\mathcal{PWDU}$ graph

To construct the $\mathcal{PWDU}$, we extended the part of JaBUTi that constructed the unit's $\mathcal{ODU}$ graph. While constructing the $\mathcal{ODU}$s the interaction nodes are identified, so that the integration between graphs can happen later.

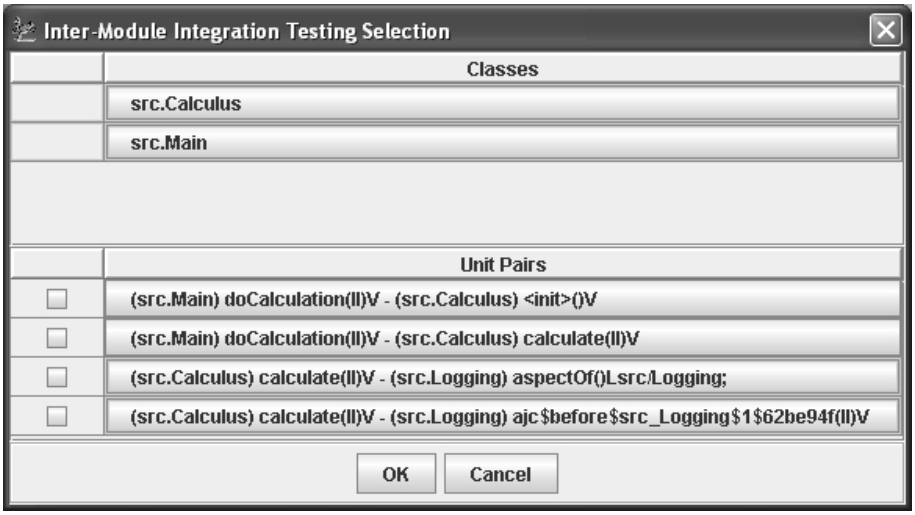


Figure 5. Pairs of inter-module units identified for classes `Calculus` and `Main`.

To a $\mathcal{PWDU}$ graph for a particular pair of units, we generate the $\mathcal{AODU}$ of the base unit first. Then, the $\mathcal{AODU}$ of the integrated unit is generated, and all of its nodes and edges are marked as integrated nodes and edges. Before integrating the two graphs, we remove the edges that connect the interaction nodes (crosscutting and call nodes) and the following nodes in the base unit. This is to create a gap to integrate the integration unit. Then, we create an integration edge from the interaction node in the base $\mathcal{AODU}$ to the entry node of the integrated $\mathcal{AODU}$. We also create additional integration edges from each exit node in the integrated $\mathcal{AODU}$ to the nodes that follow the interaction nodes in the base unit. With these steps the graphical and internal representation of the $\mathcal{PWDU}$ is ready to be used to derive the requirements.

4.3. Implementation of the pairwise integration testing criteria

To implement the pairwise integration criteria, we created classes to represent the new types of nodes and edges: integration nodes and edges. The implementation of the criteria was straightforward with the previous unit all-nodes, all-edges and all-uses criteria that were already implemented. The only difference was that we provided a $\mathcal{PWDU}$ instead of a $\mathcal{AODU}$ to the methods, and filtered the testing requirements to consider only the integrated nodes and edges for the all-pairwise-integrated-nodes and all-pairwise-integrated-edges criteria, and to consider only the communication variables for the all-pairwise-integrated-uses criterion.

The all-pairwise-integrated-uses criterion is the one that required the most effort to be implemented. The most important implementation detail is related to the mapping of the communication variables. For that purpose we used two structures, to do the mapping and to identify the definition and use of variables along the interaction paths. Thus we could keep track of the definition and use of variables in both the base and the integrated unit, and also map variables in the integrated unit to variables in the base unit and vice-versa.

4.4. Implementation of the intra and inter-module testing environment

To support the pairwise integration testing approach, we added two environments to the original JaBUTi/AJ tool: one related to the intra-module testing – testing of unit pairs that interact inside the same class or aspect – and inter-module testing – testing of unit pairs inside different classes and aspects that interact with each other. These two environments use the same modules selected to be tested and instrumented in the unit testing environment – the initial environment of the tool – and, from these modules, we identify the pairs of interacting units as discussed before.

Each testing environment has to support their specific testing data. Thus, all that happens in one environment does not affect the others. An example is the execution of test cases. While executing a particular test case in the unit testing environment, for instance, we don't want such execution to affect the integration testing information. The tester can also save the testing project and the separation among the environments' information will be kept for later usage of the same project.

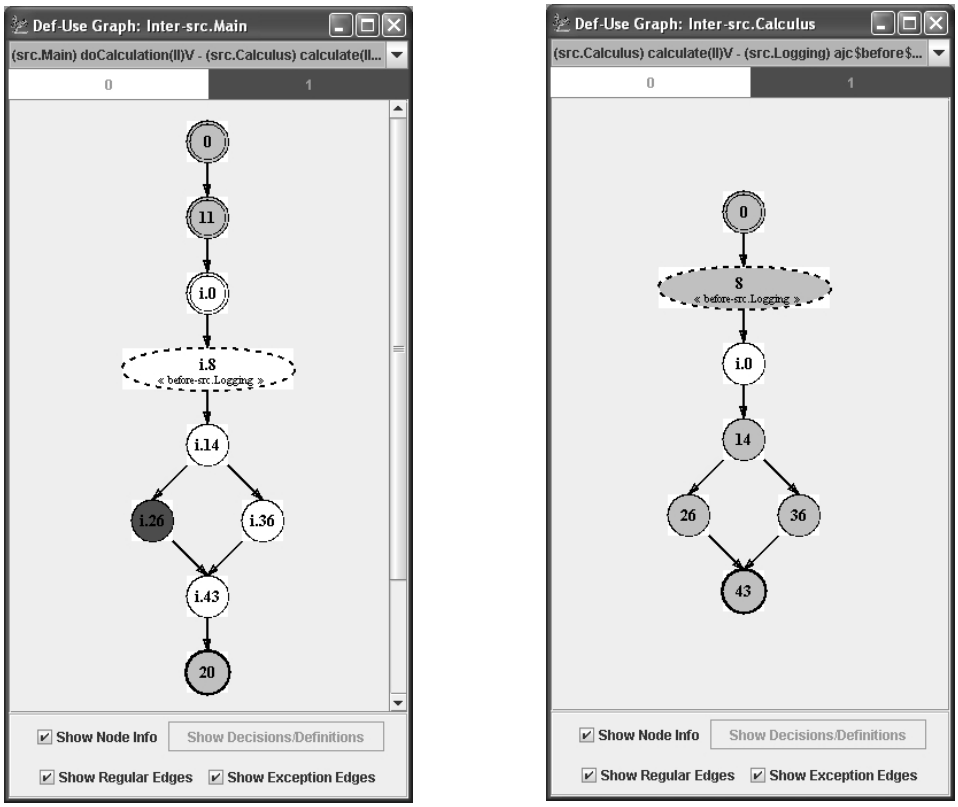
5. Tool usage

To test an application using JaBUTi/PW-AJ we need to first create a testing project. In this step, the tester selects which classes and aspects he wishes to test. For instance, to test the simple calculation application showed before, we need to select the classes and the aspect. After the selection, the tool generates an $\mathcal{AODU}$ graph and derives the 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, can enhance the coverage compared to the other requirements wrt the criteria that are being considered.

Since the focus of this paper is on integration testing, we will demonstrate a typical pairwise testing scenario. Assuming that the units have been tested, we can select the intra-module pairwise testing environment. For our example, no interesting pairs of intra-module units are present, so we can go on to the inter-module testing environment. When this environment is selected, the tool shows all the pairs of interacting units so that the tester can select the ones he wants to test (Figure 5 shows the inter-module pairs for our example). The first pair represents the instantiation of the `Calculus` class on line 25 of the source code listed in Figure 1. The second pair represents the call to the `calculate` method on the next line (26). The third pair is related to the call to the `aspectOf` method, an internal AspectJ method to get the instance of the `Logging` aspect. The fourth pair represents the execution of the advice of the `Logging` aspect before the call to the `calculate` method on line 26.

The tester could select for testing, for instance, pairs 2 and 4, since the integration between the method and the constructor is not so much relevant (because the constructor is a default one) and the call to `aspectOf` is not so much relevant either (because it is more related to the AOP infrastructure than to the application itself). After selecting pairs 2 and 4, the tool generates the two $\mathcal{PWDU}$s, and also gathers the requirements for each pairwise criterion.

Now, if the tester already had test cases for testing the `doCalculation` method (the base unit), he could import those to the tool and check the coverage wrt the integration



(a) $\mathcal{PWDU}$ for the method-method pair `Main.doCalculation-Calculus.calculate`

(b) $\mathcal{PWDU}$ for the method-advice pair `Main.doCalculation-Logging.before`

Figure 6. $\mathcal{PWDU}$s of the second and forth pairs of inter-module units after the execution a one test case.

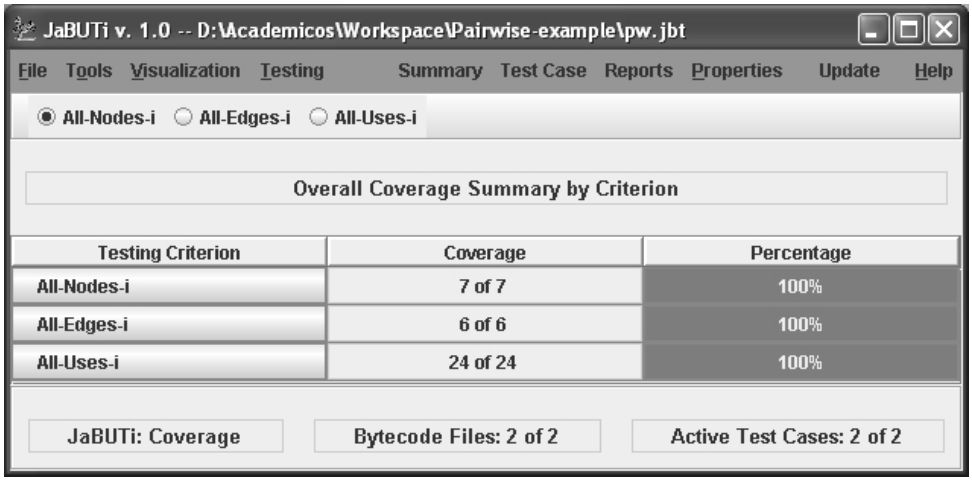


Figure 7. Summary of the pairwise criteria after importing two test cases.

pairwise testing criteria. If there are no test cases, we must create them to cover the parts of the integrated unit through the base unit. For pair 2, test cases must be constructed to cover both sides of the `if` on line 15 of the source code (Figure 1 and node $i.14$ of the $\mathcal{PWDU}$ graph in Figure 3). Analyzing the logic of both methods, two test cases,

one with a higher number as the first parameter and another with a higher number as the second parameter, would be sufficient to cover all the integrated nodes, edges and uses. For pair 4, any test case would be sufficient to execute the before advice and all the integrated nodes, edges and uses. Figure 6 shows the two $\mathcal{PWDU}$s after the execution of one of the test cases with the input as described above. The white nodes represent the executed integrated nodes. Figure 7 shows the summary of the testing requirements for all the pairwise integration criteria after two test cases with inputs as described above are imported to the tool.

6. Conclusion and Future work

In this paper we 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 at those interfaces. Since we consider each pair of units separately, the practicality of the testing activity is also considered.

The infeasibility issue, related to paths required by the criteria which cannot be covered, poses an undecidable problem that can also occur in our context. For instance, there can be conditions in the integrated unit that can never be satisfied through inputs issued on the base unit, generating requirements with infeasible paths. This problem is minimized by the tester setting the infeasible requirements through the JaBUTi/PW-AJ tool, which make the related requirements to be discarded (since they could never be covered).

Future work includes studying whether it is possible to enlarge the integration of units considering deeper call chains, without making the integration testing activity infeasible. An idea is to make the depth configurable and defining criteria based on an n-depth integration strategy. 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 faults related to faulty pointcuts [Lemos et al. 2006].

References

AspectJ Team (2003). The AspectJ programming guide. Online. Available from: `http://www.eclipse.org/aspectj/doc/released/progguide/index.html` (accessed 20/01/2006).

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

Franchin, I. G. (2007). Teste estrutural de integração par-a-par de programas orientados a objetos e a aspectos: Critérios e automatização. Master's thesis, ICMC-USP, São Carlos, SP.

Harrold, M. J., Jones, J. A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S. A., and Gujarathi, A. (2001). 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. ACM Press.

Harrold, M. J., McGregor, J. D., and Fitzpatrick, K. J. (1992). 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. ACM Press.

Harrold, M. J. and Rothermel, G. (1994). 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. ACM Press.

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

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

Lemos, O. A. L., Ferrari, F. C., Masiero, P. C., and Lopes, C. V. (2006). 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. ACM Press.

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

Lindholm, T. and Yellin, F. (1999). *The Java Virtual Machine Specification*. Prentice Hall PTR, 2 edition.

Linnenkugel, U. and Müllerburg, M. (1990). 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. IEEE Press.

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

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

Orso, A., Shi, N., and Harrold, M. J. (2004). 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. ACM Press.

Paradkar, A. (1996). 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. IBM Press.

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

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

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

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

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