Static Analysis of Java Bytecode for Domain-specific Software Testing*

Márcio E. Delamaro[1], Paulo A. Nardi[1], Otávio Lemos[2], Paulo C. Masiero[2], Edmundo S. Spoto[1], José C. Maldonado[2], Auri M. R. Vincenzi[3]

[1]Centro Universitário Eurípides de Marília (UNIVEM)
Marília, SP, Brazil

[2]Universidade de São Paulo (USP)
São Carlos, SP, Brazil

[3]Universidade Católica de Santos (UNISANTOS)
Santos, SP, Brazil

`{delamaro,nardi,dino}@univem.edu.br`
`{oall,masiero,jcmaldon}@icmc.usp.br`
`auri@unisantos.br`

Resumo. *Embora seja de alto custo, a atividade de teste é de fundamental importância no processo de desenvolvimento de software. Técnicas e ferramentas são essenciais para a melhoria da qualidade e da produtividade na atividade de teste. A técnica de teste estrutural usa estruturas de fluxo de controle e de dados para derivar requisitos de testes. Buscando exercitar tais requisitos o testador supostamente fornece casos de teste que melhoram a qualidade do software. O teste estrutural requer a execução de várias atividades que exigem a análise de código que, em geral, é realizada no código fonte do produto em teste. Com o advento da linguagem Java tornou-se usual a realização da análise diretamente no código objeto (bytecode) o que traz algumas vantagens. Neste artigo, é discutido como usar as características da análise de bytecode Java e como estendê-la para a implementação de critérios de teste estruturais para dois domínios específicos: programas orientados a aspectos e aplicações de banco de dados.*

Abstract. *Software testing is an important and expensive activity of the software development process. Techniques and tools are essential to improve test quality and productivity. Structural testing is a technique that uses characteristics such as control-flow and data-flow structures to derive testing requirements. By exercising such testing requirements the tester supposedly provides test cases that improve the software quality. Structural testing requires several activities that make use of code analysis, in general performed on the program source code. With the advent of Java it has become usual to perform such analysis directly on the bytecode instead of the source code, with a number of advantages. In this paper we discuss how to use the characteristics of Java bytecode analysis and how to extend it to implement structural criteria for two specific domains: aspect oriented programs and database applications.*

*This project is supported by the Brazilian Funding Agencies – CNPq, FAPESP and Capes.

1. Introduction

Software testing is one of the most important validation and verification activities, complementing other techniques as technical revision and formal methods. A critical factor in the testing process is the quality of the test set used to execute the software. Its selection should be done based on testing requirements that establish which aspects of the software must be exercised.

Structural testing is a technique that determines testing requirements from an implementation code. Requirements can be statements, branches or data flow associations, for instance. The application of structural criteria requires program analysis in order to abstract the program structure and to compute testing requirements. In general, structural testing uses a program representation known as def-use graph that abstracts the flow of control and variable definition and use.

Several approaches have explored the use of structural testing criteria and a solid theoretical basis has been built [Boujarwah et al. 2000, Herman 1976, Maldonado 1991, Rapps and Weyuker 1985, Zhao 2000]. Some of them explore the use of structural criteria in specific application domains. In [Spoto et al. 2000] it is proposed a technique to identify definition and use of persistent variables (tables) in relational database applications and defined a set of structural criteria for unit and integration testing of such applications. In [Lemos et al. 2007] it is defined structural criteria to deal with specific characteristics of aspect oriented programs.

Traditionally, static analysis and testing requirement computation are performed on the source code. More recently, the use of object code (bytecode) analysis has become popular, since bytecode analysis can, in some cases, present advantages over source code analysis. For example, it allows the analysis and use of structural coverage information of third part components when the source code is not available and allows to test programs with code mobility features in which source code is not available as well [Delamaro and Vincenzi 2003]. In addition, it is language independent, i.e., it can be used to test any program, in any source language.

A number of papers can be found in the literature that discuss static bytecode analysis [Boujarwah et al. 2000, Haddox et al. 2002, Zhao 2000]. In [Vincenzi et al. 2005] the authors discuss its use in the implementation of a tool that supports control and dataflow testing on Java programs and components.

In this paper we present the techniques used to analyze Java bytecode in order to implement structural testing support in the two mentioned domains: relational database applications and aspect oriented programs. The main operational characteristics of our testing tool are shown in Section 2. Section 3 discusses the implementation of the control and data flow models used in the tool, based on bytecode analysis. Sections 4 and 5 discuss the fundamentals and the static analysis techniques used to support structural testing to aspect-oriented and database applications, respectively. Section 6 presents the final remarks.

2. The JaBUTi testing tool

The use of structural criteria for software testing requires the execution of a few tasks like static analysis of the program under test, computation of the required elements, program instrumentation, execution of the instrumented program and coverage analysis. Obviously, the complexity of those tasks prevents their application manually, requiring supporting tools to execute them automatically.

In the **JaBUTi** [Vincenzi et al. 2005] architecture two distinct parts can be identified. One of them is responsible to perform the static analysis of the program under test, the generation of the required structural elements and the test case coverage analysis. The other dynamic part takes care of the program instrumentation, test case execution and trace data collection.

The first step on **JaBUTi** utilization is the creation of a hierarchical abstraction of the program being tested. It is known that a Java program is composed by a set of classes organized in a tree hierarchy, having the class `java.lang.Object` as its root. The model used by **JaBUTi** delimits the program to a restrict set of classes. From a "base class" – the one containing the program entry point – the tool computes a set of needed classes that excludes: 1) the Java API; 2) classes which the class file is not found in the class path; and 3) packages which the tester decides to ignore.

Inside **JaBUTi**, such a model is used in the creation of a testing project. The tester chooses the base class and the `classpath` to locate the rest of the classes and the tool computes and exhibits the set of required classes to execute the program. The tool shows the packages and classes that compose the program and allows the tester to select those which should really be tested and those which should be ignored i.e., excluded from the program structure. In addition, the tester defines a project file name to store the project data.

Once the testing project is created, the tester has eight structural criteria to work with. These criteria are summarized in Table 1. More details can be found elsewhere [Vincenzi et al. 2005]. By selecting a criterion, the tester visualizes information about the program, concerning the selected criterion. For example, using criterion $all\text{-}nodes_{ei}$ one can see the source code (if available), the bytecode or the def-use graph. In either case, the tester is provided with hints about which testing requirement should be covered to achieve higher coverage.

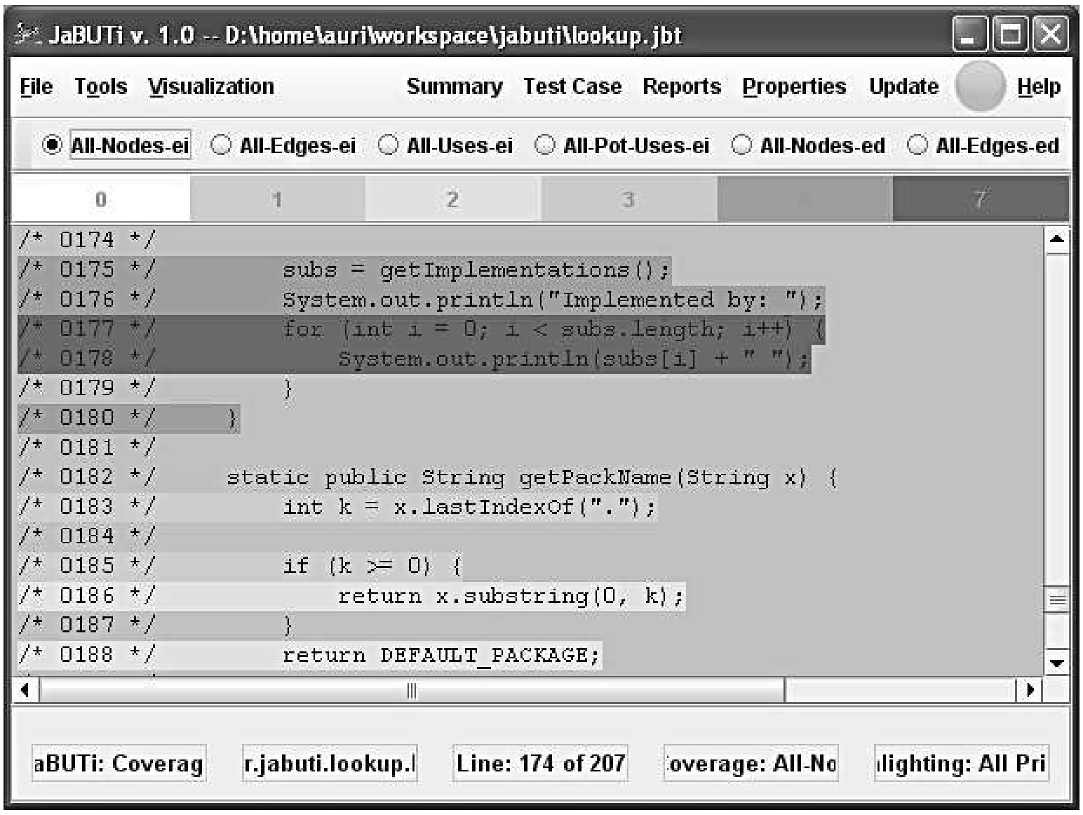
JaBUTi introduces a control flow graph with two different types of edges. One represents the normal control flow between blocks of commands and the other represents the flow created by the handling of an exception, inside a given method. Using these concepts, the criteria require normal test cases, in which no abnormal condition or exception occurs and test cases in which such conditions occur.

Figure 1 shows the source code of a program with information about $all\text{-}nodes_{ei}$. The colored pieces of code (in gray scale in this article) indicate the requirements (in this case, statements) that have not yet been covered. Each color is an indicative of the number of nodes that would be covered when that part of the code is executed. With this information, the tester can prioritize the code with higher weight.

The "native" form to view a program in **JaBUTi** is its bytecode, since the analysis is totally performed at the bytecode level, not at source code level. The same kind of hints about requirement coverage is shown when the tester chooses the source code view. In fact, that information is computed at bytecode level and mapped back to source code level using the bytecode-to-source code mapping, present in the class file.

Table 1. Testing criteria implemented in JaBUTi.

Name	Meaning	Explanation
$all\text{-}nodes_{ei}$	all nodes, independent of exceptions	requires the execution of each node in the graph that can be executed without occurring an exception
$all\text{-}edges_{ei}$	all edges, independent of exceptions	requires the execution of each edge in the graph that can be executed without occurring an exception
$all\text{-}uses_{ei}$	all uses, independent of exceptions	requires the coverage of each def-use pair that can be executed without occurring an exception
$all\text{-}pot\text{-}uses_{ei}$	all potential-uses, independent of exceptions	requires the coverage of each def-potential-use [Maldonado 1991] pair that can be executed without occurring an exception
$all\text{-}nodes_{ed}$ $all\text{-}edges_{ed}$ $all\text{-}uses_{ed}$ $all\text{-}pot\text{-}uses_{ed}$	same criteria, dependent of exceptions	require, respectively, the coverage of nodes, edges, def-use pairs and, the coverage of nodes, edges, def-use pairs and def-potential-use pairs that only can be executed with the occurrence of an exception



```
/* 0174 */
/* 0175 */         subs = getImplementations();
/* 0176 */         System.out.println("Implemented by: ");
/* 0177 */         for (int i = 0; i < subs.length; i++) {
/* 0178 */             System.out.println(subs[i] + " ");
/* 0179 */         }
/* 0180 */     }
/* 0181 */
/* 0182 */     static public String getPackName(String x) {
/* 0183 */         int k = x.lastIndexOf(".");
/* 0184 */
/* 0185 */         if (k >= 0) {
/* 0186 */             return x.substring(0, k);
/* 0187 */         }
/* 0188 */         return DEFAULT_PACKAGE;
```

aBUTi: Coverag | r.jabuti.lookup.l | Line: 174 of 207 | overage: All-No | lighting: All Pri

Figure 1. Source code visualization, with information about $all\text{-}nodes_{ei}$**.**

The same information is also provided in the graph visualization. Figure 2 shows how the nodes are displayed in different colors, representing the weight in relation to the total method coverage. Node 114 corresponds to a handler for exceptions that may be raised in nodes 24 or 57. Then, the outgoing edges from those nodes to node 114 represent flow of control that happens when an exception occurs and is treated by the exception handler. They are drawn as dashed arrows in the graph visualization. Node 114 is not colored because it is not part of the criterion being displayed, i.e., $all\text{-}nodes_{ei}$. Placing the mouse pointer on a graph node, additional information is provided to the tester as, for instance, the set of variables used or defined in the node.

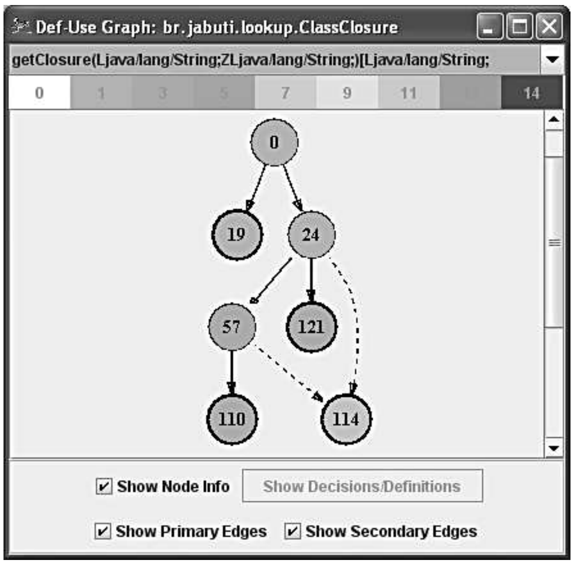

Def-Use Graph: br.jabuti.lookup.ClassClosure
getClosure(Ljava/lang/String;ZLjava/lang/String;)[Ljava/lang/String;
0
1
3
5
7
9
11
14
0
19
24
57
121
110
114
Show Node Info
Show Decisions/Definitions
Show Primary Edges
Show Secondary Edges

Figure 2. Def-use graph visualization, with information about $all\text{-}nodes_{ei}$.

The tester can manage testing requirements, for example, marking a requirement as infeasible. A testing requirement can be covered by the execution of a test case. This is done “outside” the tool, by a test driver that instruments the program under test and then starts the instrumented program. Ended the execution of the test case, the tool collects the trace data from the trace file written by the instrumented program and updates the information about coverage of each criterion.

The **JaBUTi** tool has a large variety of reports that allow the tester to check testing requirement coverage. For example, the tester can create a summary of the coverage by criterion, by class or by method. It can also create `HTML` files with static or dynamic data obtained until that point in the project. In addition to the testing criteria, the tool implements a slicing tool to help the tester to locate possible existing defects in the code and a set of static complexity metrics.

Next section presents a few considerations about the implementation of the tool. In particular, the computation of the control-flow graph and the definition/use of information associated to the nodes of the graph are discussed.

3. Implementation aspects

All the analysis performed by the **JaBUTi** tool is done directly at the bytecode. A third part library named BCEL [Dahm 2001] is used to manipulate the bytecode. More precisely, **JaBUTi** does not implement low-level structures to represent classes and bytecode or to access class files. It constructs, on top of the resources furnished by BCEL, its own abstractions, like a program hierarchy and a method control-flow graph and implements its own functionality, as def-use computation.

This section discuss an important aspect in the analysis performed by **JaBUTi**: the computation of the control and data flow models extracted from the bytecode. **JaBUTi** package `graph` contains the classes to represent a method in the form of a graph and to associate data flow information to it.

Class `Graph` is the base class for constructing and manipulating graphs inside **JaBUTi**. Associated to it, there is a `GraphNode` class, which represents a vertex (node) of the graph. Each `GraphNode` object stores a list of successors and predecessors. The `Graph` class stores general information about the graph, for instance, the initial node and the list of final nodes.

Classes `Graph` end `GraphNode` implement the basic functionality to graph construction and manipulation, such as node and edge insertion and deletion, computation of strongly connected components, computation of the depth-first tree, computation of the dominator nodes, etc.

The analysis of the program code is conducted at the method level. Initially, each method is parsed, extracting from it a graph representation called **instruction graph** (*IG*). In this graph, each node corresponds to a single bytecode instruction.

An *IG* is represented by a `InstructionGraph` object, a subclass of `Graph`. One important factor in the construction of the *IG* is exception handling. In the bytecode, exception handling is defined by a table that relates exception handlers to contiguous blocks of code. In the implementation of **JaBUTi** the list of successors and predecessors are split in two, in order to represent the two different types of control-flow. Thus, each node in the graph has a list of successors (predecessors) independent of exceptions and a list of successors (predecessors) dependent of exceptions. This characteristic is important to the definition of the structural criteria described in Table 1.

A second characteristic of the Java bytecode that influences the construction of the *IG* is the use of subroutines. The JVM has instructions **JSR** and **RET** that allow pieces of code inside the method to be used as subroutines, being called from several points of the same method. This feature is commonly used to translate `finally` blocks. One such example is show in Figure 3.

The solution adopted to deal with subroutines was to expand each subroutine call. In this way, each subroutine call is represented by an independent set of nodes that refer to the same instruction in the bytecode. Figure 4 gives the graph for the code in Figure 3, according to the adopted model. The exception edges are not shown in the figure.

After the *IG* is built, it is processed with the objective of collecting information about the state of the JVM, i.e., the vector of local variables and the operand stack for each node of the graph. For that, each node of the *IG* stores the following information:

- The set of possible “configurations” that the operand stack can have when the instruction in the node is executed. A configuration is an array in which each element describes the type of the value in the stack, the instruction that pushed the value into the stack and the type of definition/use the instruction does on that stack position;

```
void finallyExample([Ljava/lang/String;)V
0:    iconst_0
1:    istore_2
2:    new       <java.io.FileInputStream> (2)
5:    dup
6:    aload_1
7:    iconst_0
8:    aaload
9:    invokespecial java.io.FileInputStream.<init>
                     (Ljava/lang/String;)V (3)
12:   astore_3
13:   jsr       #36
16:   goto      #42
19:   astore_3
20:   iconst_1
21:   istore_2
22:   jsr       #36
25:   goto      #42
28:   astore    %4
30:   jsr       #36
33:   aload     %4
35:   athrow
36:   astore    %5
38:   iconst_2
39:   istore_2
40:   ret       %5
42:   iinc      %2  1
45:   return

Exception table:
  from  to  target type
     2   13     19  <Class java.io.IOException>
     2   16     28  <Class all>
    19   25     28  <Class all>
    28   33     28  <Class all>
```

Figure 3. Code that uses subroutine call.

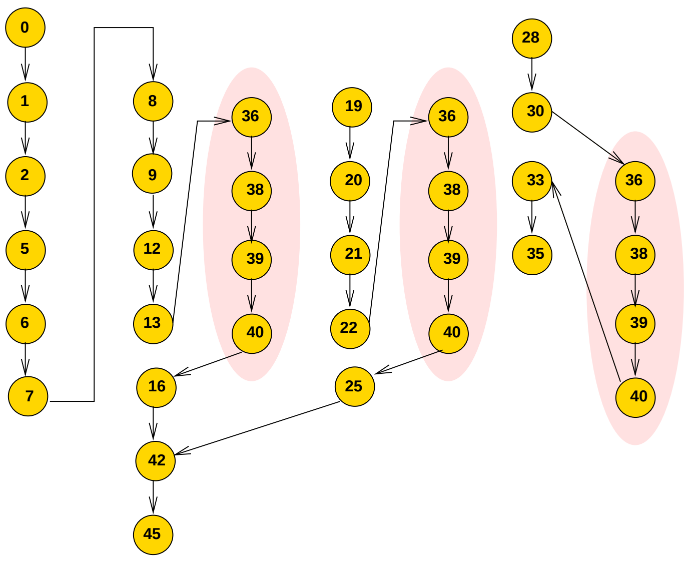


Figure 4. *IG* for a method with subroutine.

- The set of possible "configurations" that the local variable vector can have when the instruction in the node is executed. A configuration is an array in which each element represents the type of an element in the local variable array.

For example, the following piece of bytecode would generate the set of configurations described in Figure 5.

```
      ILOAD_1
      JNE         L0
      ALOAD_2
      GOTO        L1
L0:   GETSTATIC   ClassA.f1
L1:   LDC    0
      PUTFIELD    ClassB.f2
```

With that information, and knowing the semantic of the instruction in a given node, it is possible to compute data-flow events in the node. For some nodes, it is a trivial computation that depends only on the instruction semantics. For example:

ILOAD_1: use of local variable #1 (represented as `L@1`);
JNE: no use or definition;
ALOAD_2: use of local variable #2 (represented as `L@2`);
GOTO: no use or definition;
GETSTATIC: use of class variable `ClassA.f1` (represented as `S@ClassA.f1`);
LDC: no use or definition.

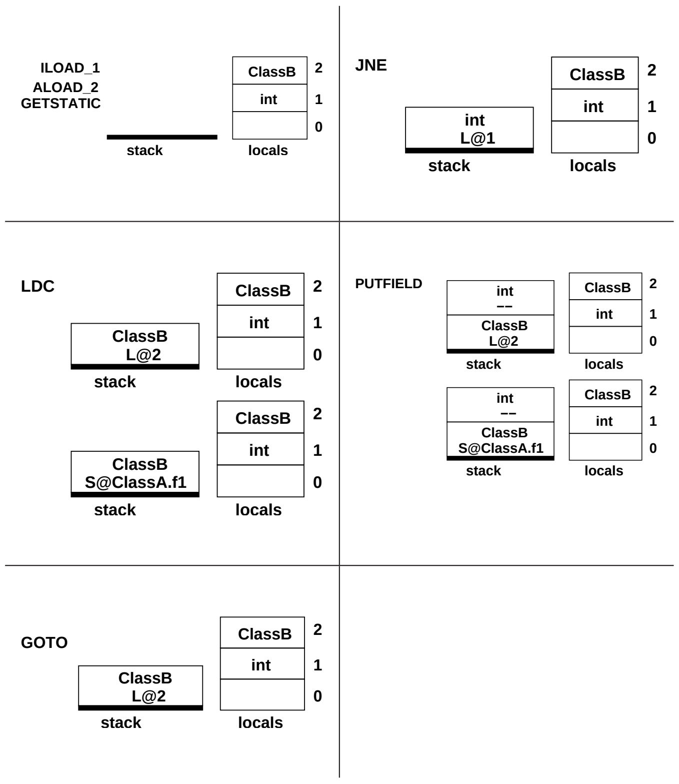


Figure 5. Example of stack and local variable configurations associated to the *IG* nodes.

Other nodes may require the use of the stack configurations of the node. It is the case of instruction **PUTFIELD** in Figure 5. For such an instruction there should be considered the definition of the field `f2` in the object whose reference is stored in the top of the stack. Since two configurations exist in that node, two definitions should be associated to that node: `L@2.f2` (field `f2` of object whose reference is in local variable #2) and `S@ClassA.f1.f2` (field `f2` of object whose reference is in static variable `ClassA.f1`).

The *IG* is too complex to be used directly by the tester. A trivial method can originate a graph with dozens of nodes. This is too much for a model that supposedly should be an abstraction of the method. For this reason, the **JaBUTi** tool exhibits a method in the form of a block graph, i.e., each node represents an indivisible block of instructions. Such a graph, called def-use graph (*DUG*) if obtained from the *IG* by the application of a reduction algorithm that joins several *IG* nodes into a single *DUG* node. The object created by such an algorithm is a `CFG`, a subclass of `Graph`. It stores control and data flow information, preserving the characteristics obtained from the *IG* concerning exception handling and subroutine flows. Figure 6 shows the *IG* of Figure 4 reduced to its *DUG* form.

Labels are assigned to the nodes of the *DUG*. They are, in general, the offset of the first instruction in the block. The exceptions are the nodes inside a subroutine. Each call to the subroutine replicates the instructions in the *IG* and consequently, the nodes in the *DUG*. Such nodes cannot have the same label. Thus, the rules for labeling the nodes are more precisely described as: 1) a node that is not part of a subroutine is labeled with the offset of its first instruction; 2) a node in a subroutine is labeled with the label of the node that has the **JSR** associated to the subroutine call followed by a ".", followed by the offset of its first instruction. In Figure 6 the three calls to subroutine at offset 36 gave origin to the nodes 13.36, 19.36, and 28.36.

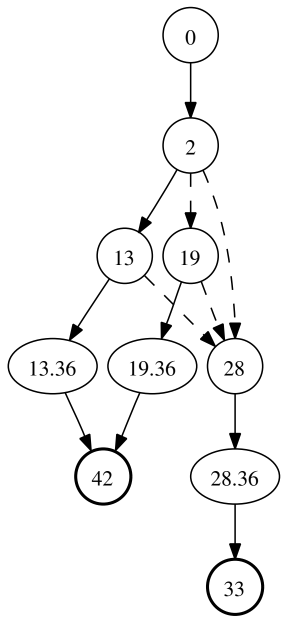


Figure 6. Example of a *DUG*.

4. Testing aspect-oriented programs

Most programming languages do not support the clear separation of some types of requirements that tend to be spread throughout (crosscut) several modules of implementation (for instance the implementation of logging). Aspect-oriented programming (AOP) supports the modularization of such concerns by a mechanism that can add behavior to selected elements of the programming language semantics (the *join points*), and thus isolating implementations that would otherwise be spread. In a number of presently available AOP languages these join points constitute regions in the dynamic control flow of an application. A join point can, for instance, represent a call to a method, the execution of a method, the setting of a field or the handling of an exception.

Advice is the behavior implemented in a block of code (similar to a method) that can be executed before, after or instead of (around) a join point. Also, there are *pointcuts* which are used to select sets of join points in a program where advice will be added to. Aspects are units that combine pieces of advice, pointcuts and other basic elements. After classes and aspects are coded, a process named *weaving* must take place together with the compilation so that aspect and non-aspect units are combined into an executable program.

One of the most prominent AOP languages is AspectJ, an extension of Java to support AOP. The basic new constructs are the aspect itself; before, after and around advice, that are used to define crosscutting behavior at the join points; and the pointcuts which are used to define sets of join points in the program. In AspectJ, aspects are units that combine: join point specifications, pieces of advice, methods, fields and inner classes. Also, aspects can declare members (fields and methods) to be owned by other types, what is called inter-type declaration. Latest versions of AspectJ also support declarations of warnings and errors that arise when join points are identified at compile time or reached at runtime, respectively [The AspectJ Team 2006]. Before, after and around advice are method-like constructs that can be executed before, after and in place of the join points, respectively. These constructs can also pick context information from the join points that caused them to execute. Figure 7 lists part of the source code of an aspect-oriented program that simulates a telephony system. The `Billing` aspect implements the billing concern and declares a payer to each connection and also makes sure that local, long distance and mobile calls are charged accordingly. The rest of the system models connections, calls, customers and other elements involved in a telephony system.

With respect to the implementation of AspectJ, its 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 become parameters of these new methods (with some special treatment when reflexive information is needed) [Hilsdale and Hugunin 2004]. These methods have special names to be identified as pieces of advice in the bytecode (they start with '`ajc$`'). In order to coordinate aspects and non-aspects, the code of the system is instrumented and calls to the "advice-methods" are inserted considering that certain regions of bytecode represent possible join points (also called *static shadows* [Hilsdale and Hugunin 2004]).

With the AspectJ implementation strategy one can identify the places where pieces of advice are adding behavior in the bytecode resulted from the compilation/weaving process. That is, every call made to an advice-method found in the bytecode represents an execution of the corresponding advice of some aspect in the affected join point. Figure 8 shows part of the resulting bytecode of the constructor of the `Call` class. Advice executions are identified by calls to methods starting with `ajc$`*`adviceType`*`$` generated by the AspectJ compiler.

```
public class Call {
    private Customer caller, receiver;
    private Vector connections = new Vector();

    public Call(Customer caller, Customer receiver, boolean iM) {
        this.caller = caller;
        this.receiver = receiver;
        Connection c;
        if (receiver.localTo(caller)) {
          c = new Local(caller, receiver, iM);
        } else {
          c = new LongDistance(caller, receiver, iM);
        }
        connections.addElement(c);
    }
    ...
}
public aspect Billing {
    declare precedence: Billing, Timing;
    ...
    after(Customer cust) returning
      (Connection conn): args(cust, ..)
      && call(Connection+.new(..))
    { conn.payer = cust; }
    ...
}
```

Figure 7. Part of the example of an AO program written in AspectJ.

In order to detect the places where advice runs, only the source code is not sufficient without analyzing the whole system, because before the compilation/weaving processes there are no explicit references to the join points identified by aspects. Therefore, together with the fact that the AspectJ compiler produces pure Java bytecode, **JaBUTi** presents itself as a suitable tool to be extended in order to correctly represent control/data flow of aspect-oriented programs implemented with AspectJ. Moreover, extra aspect-oriented testing criteria can also be defined based on such extended control/data flow representation, to exercise the special aspect-oriented elements.

The AODU (aspect-oriented definition-use) graph is an extension of the *DUG* to represent the unit control and data flow of the woven methods, pieces of advice and regular methods of an AspectJ program. The graph was extended with dashed nodes to represent advice enhancements (*crosscutting nodes* [Lemos et al. 2007]) with additional information about the type of advice that affects that point, and the name of the aspect it belongs to (for instance ≪`before-AnAspect`≫ corresponds to a before advice of the `AnAspect` aspect). These nodes are identified when the block of bytecode which it represents contains a call to an advice-method. Figure 9 shows the AODU of the constructor of the `Call` woven class previously presented, based on its bytecode.

Based on the AODU, three aspect-oriented criteria were defined: all-crosscutting-nodes, all-crosscutting-edges and all-crosscutting-uses. These criteria can give an idea of how many of the advice executions have been exercised by the test cases, and by which ways (with respect to the edges that start or end in crosscutting nodes or def-use pairs which uses are in crosscutting nodes) [Lemos et al. 2007].

```
0 aload_0
1 invokespecial #15 <Method Object()>
4 aload_0

  ...

69 invokevirtual #110 <Method void
   ajc$afterReturning$telecom_Billing$1$8a338795(
   telecom.Customer, telecom.Customer,
   boolean, telecom.Connection)>
72 nop
73 astore 4
75 goto 120
78 aload_1
79 aload_2
80 iload_3
81 istore 9
83 astore 10
85 astore 11
87 new #36 <Class telecom.LongDistance>
90 dup

  ...

112 aload 12
114 invokevirtual #110 <Method void
    ajc$afterReturning$telecom_Billing$1$8a338795(
    telecom.Customer, telecom.Customer,
    boolean, telecom.Connection)>
117 nop
118 ...
```

Figure 8. Part of the bytecode of the `Call` class constructor.

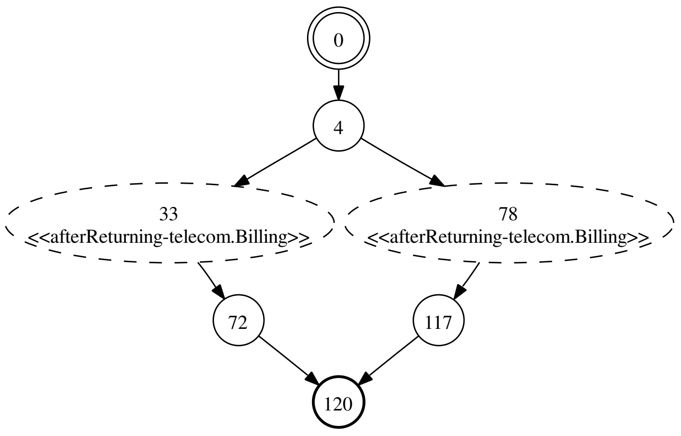

0
4
33
<<afterReturning-telecom.Billing>>
78
<<afterReturning-telecom.Billing>>
72
117
120

Figure 9. AODU of the `Call` class constructor.

In order to implement the AODU graph, the **JaBUTi** `graph` package (see Section 3) was extended. In particular a class named `CFGCCNode` that extends the `CFGNode` class was created to represent the crosscutting nodes with extra information about the type of advice and the name of the aspect it belongs to. These nodes are identified analyzing each bytecode instruction to check whether it represents a call to an advice-method. If such a call is found in the corresponding block, a crosscutting node is used in the graph.

To gather information about the kind of advice and the name of the aspect it belongs to, the advice-method name is analyzed. The three new criteria were implemented with the strategy of gathering all requirements of the all-nodes, all-edges and all-uses criteria and then selecting only the ones related to crosscutting nodes.

5. Testing database applications

In this section we discuss the use of dataflow testing criteria defined by Spoto et al. [Spoto et al. 2000] specifically for applications that manipulate persistent data in the form of relational databases (Relational Database Applications – RDA). Initially, the main concepts related to the RDA criteria are presented, them we discuss how the support to such criteria has been implemented, based on Java bytecode analysis, in the **JaBUTi** tool.

We consider a RDA as a program, written in a "host language" (C, Pascal, Java, etc) with support to the use of a database manipulation language embedded in its code. The means by which this is done may differ from one language to another. In the context of this paper, a RDA is a Java program that uses Java's native API to manipulate relational databases. Such API, known as Java Database Connectivity (JDBC), provides a series of classes and methods that allow the programmer to connect to and to interact with relational database managers using SQL. Figure 10(a) shows an example of the use of JDBC.

The main concepts related to the use of dataflow criteria in RDA are: *table-variable* and *t-use*. A table-variable corresponds to a database table used in the RDA program. For example, taken the following SQL statement, two table-variables can be identified: *team* and *player*.

```
SELECT teamName, playerName FROM team t inner
        join player p ON t.codteam=p.codteam
```

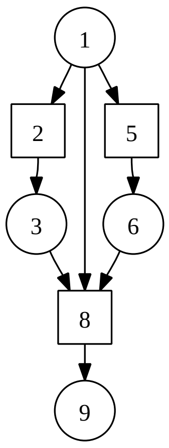
A definition of a table-variable (a persistent definition) occurs when data is inserted, updated or removed from the database table. A t-use (use of the table-variable) there exists when a query is executed to recover a value from the database table. We also consider a t-use on statements that change the value of a table-variable because the variable is first read from the persistent storage and then changed. Thus, updating, removing and inserting also represent a t-use.

The traditional program graph has been adapted by Spoto et al. [Spoto et al. 2000] in order to represent persistent definitions and t-uses. Each SQL statement, embedded in the program code is represented by a different kind of node in the graph. Such nodes highlight the access to table-variables, as shown in Figure 10, where SQL nodes are represented as squares and regular nodes as circles.

The definitions and uses of a table-variable are extracted from the code by scanning the SQL node to discover: 1) the type of statement used, which characterizes definition or use of the variables; and 2) the names of the table-variables present in the statement. A t-use is associated to the edge leaving a node where a use of a table-variable has been accessed. In this way, normal control flow and the flow caused by an exception are covered by the dataflow criteria. Thus, the testing criteria can be defined based on dataflow interactions as for regular programs. For example, the *all-t-uses* criterion requires every association between a persistent definition and a t-use to be covered by a test case.

```
1 switch(Opt)
  {
     case 1:
2      stmt.execute(
       "insert into client values(1,’Joice’)");
3      System.out.println("Insert selected");
4      break;
    case 2:
5     stmt.execute(
      "delete from client where codclient=1");
6     System.out.println("Delete selected");
7     break;
  }
8 rs=stmt.executeQuery("select * from client c" +
          " inner join sale s on c.codc=v.codc");
9 System.out.println("End.");
```

(a) Java code



(b) Program graph

Figure 10. Graph adapted to a RDA

In the example of Figure 10, the embedded SQL statements in lines 2, 5 and 8 are represented as squares and t-uses are identified on edges (2,3), (5,6) and (8,9). On the first and second there are t-uses of variable `client` and on the last there is a t-use of variable `sale`. Persistent definitions of variable `client` are present in nodes 2 and 5.

The *all-t-uses* criterion has been implemented in **JaBUTi**. This criterion requires each association between a persistent definition and a t-use, for every table-variable in the same method to be covered at least once. In this way, a required element for *all-t-uses* is a persistent-def/t-use association, with respect to a table-variable.

In the example of Figure 10, two such elements can be identified: the definition of table-variable `client` in node 2 and its t-use in edge (8,9) and definition of `client` in node 5 and use in edge (8,9). These two required elements are denoted $\langle client, 2, (8,9)\rangle$ and $\langle client, 5, (8,9)\rangle$. Table-variable `sale` also has a t-use on edge (8,9) but no persistent definition in the method. For those cases, we assume a fake definition in the first node of the graph, for each t-use without a corresponding definition, originating required elements $\langle sale, 1, (8,9)\rangle$, $\langle client, 1, (2,3)\rangle$ and $\langle client, 1, (5,6)\rangle$.

With the implementation of *all-t-uses* criterion, **JaBUTi** identifies the nodes corresponding to the SQL statements and represents them as squared nodes. In addition, determines which table-variables are accessed in each node. The tool computes the list of required elements following the same reasoning explained in Section 2, i.e., the requirements are split in two groups: the ones independent of exceptions and those dependent of exceptions.

All the static analysis performed to implement the *all-t-uses* criterion is done on the Java bytecode, as explained in Section 3. The algorithms are basically the same used to implement the *all-uses* criterion, but with some particularities to identify the persistent definitions and the t-uses. Unlike the variables of the host language that are explicitly present in the program code, the table-variables must be extracted from the SQL statement that is passed as an argument in a method invocation. For example, let's consider the following line of Java code:

```
stmt.execute("delete from client where codclient=1");
```

Such statement would be translated to Java bytecode as something like:

```
ldc "delete from client where codclient=1"
invokeinterface java.sql.Statement.execute
                         (Ljava/lang/String;)Z
```

Instruction `ldc` sends a constant to the top of the JVM stack. Instruction `invokeinterface` makes a call to method `execute` using as argument the element on the top position of the stack, i.e., the string just pushed by the previous instruction. In Section 3 we explained that each node in the *IG* stores information about the state of the JVM that allows computing the dataflow through the graph. Such information includes the type of the values in the stack, the instruction which pushed each of such values and the variable being used or defined by the respective instruction.

The verification of definitions and uses is accomplished by an algorithm that analyzes each instruction during the construction of the *IG*. Because it is necessary to know the SQL expression to decide the type of statement it represents and the table-variables involved, it is also necessary to store the SQL expression itself. Thus, in the case of the example above, when the `ldc` is analyzed by **JaBUTi**, the value of the constant pushed into the stack is registered. In this way, when instruction `invokeinterface` is analyzed, it is possible to verify which method is being called, since this information is part of the instruction itself, and also what are the arguments passed by analyzing the expected content of the stack.

The example below shows the case in which the SQL statement is not directly pushed into the stack. It is first stored in a local variable and then moved to the stack. The situation highlights the fact that every movement of values to the stack should be analyzed and the value pushed should be recorded, if possible.

```
String sqlCommand= "delete from client where codcli=1";
stmt.execute(sqlCommand);

--------------------------------------------------

ldc      "delete from client where codcli=1"
astore    %5
aload_2
aload     %5
invokeinterface   java.sql.Statement.execute
                          (Ljava/lang/String;)Z
```

In this case, the type and the value of the constant are pushed into the stack then; its value is popped and stored in the local variable vector. Next, the `Statement` object is pushed to the stack and then the local variable number 5 is pushed to the stack, placing the SQL statement back on the top of the stack.

In both examples, when the `invokeinterface` instruction is analyzed, the SQL statement is available as a string constant on the top of the stack. At this point, it is first necessary to check whether the method invoked is one of interest as `Statement.execute`, `Statement.executeQuery` or `Statement.executeUpdate`. Next, the analysis focus on the string passed as argument, placed on the top of the stack. It is scanned and the tool collects the name of the tables being accessed and the type of access, determined by the SQL statement, which characterizes a persistent definition or a t-use.

The technique is restricted to the analysis of SQL statements provided directly or indirectly by a string constant. In the cases where it is not possible to statically identify the argument passed to the JDBC method, the tool still identifies the SQL node in the graph but does not assign any definition or t-use to it.

6. Conclusions

Structural testing has been widely studied and constitutes one of the most important techniques both as research subject and as support for industrial software development.

The use of structural testing requires static analysis that traditionally has been performed at the source code. With the advent of the Java language, it has become more popular to perform static analysis at Java bytecode level. This approach has a number of advantages, in particular it hides the peculiarities of the source language that can be Java, extensions of it or even a completely different language from which Java bytecode can be generated. In addition, it allows the implementation of testing criteria to deal with particular aspects of the specific domains.

In this paper we presented two cases of domain specific structural testing criteria and discussed how to implement them from the bytecode static analysis. The first domain is aspect oriented programs using the AspectJ language, an extension that allows adding aspect programming features to the Java language. In this case, there are interactions between classes and advices that are not explicitly present in the class source code but can be identified in the generated bytecode. In this scenario we discuss how to define structural criteria to exercise such interactions.

The second domain we presented is database applications using JDBC. Java programs can have access to databases by using an API that allows the programmer to embedded SQL statements as arguments to method calls. From the SQL statements it is possible to extract persistent data-flow associations, i.e., identify points where table variables are defined and points where they are used, as with regular program variables.

In both cases we also discuss how the domain specific criteria can be implemented by analyzing only the bytecode and how such analysis differs from conventional control-flow and data-flow bytecode analysis.

References

Boujarwah, A., Saleh, K., and Al-Dallal, J. (2000). Dynamic data flow analysis for Java programs. *Journal of Information and Software Technology*, 42(11):765–775.

Dahm, M. (2001). Byte code engineering with the BCEL API. Technical Report B-17-98, Freie Universität Berlin – Institut für Informatik, Berlin – German. Available at: `http://bcel.sourceforge.net/`. Accessed on: 01/03/2004.

Delamaro, M. E. and Vincenzi, A. M. R. (2003). Structural Testing of Mobile Agents. In Nicolas Guelfi, E. A. and Reggio, G., editors, *III International Workshop on Scientific Engineering of Java Distributed Applications (FIDJI'2003)*, Lecture Notes on Computer Science, pages 73–85. Springer.

Haddox, J. M., Kapfhammer, G. M., and Michael, C. C. (2002). An approach for understanding and testing third party software components. In *Reliability and Maintainability Symposium – RAMS'2002*, pages 293–299, Seattle, WA, USA. IEEE Press.

Herman, P. M. (1976). A data flow analysis approach to program testing. *Australian Computer Journal*, 8(3):92–96.

Hilsdale, E. and Hugunin, J. (2004). Advice weaving in aspectj. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35. ACM Press.

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

Maldonado, J. C. (1991). *Potential-Uses Criteria: A Contribution to the Structural Testing of Software*. PhD thesis, DCA/FEE/UNICAMP, Campinas, SP, Brazil. (in Portuguese).

Rapps, S. and Weyuker, E. J. (1985). Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375.

Spoto, E. S., Jino, M., and Maldonado, J. C. (2000). Structural software testing: An approach to relational database applications. In *XIV Brazilian Symposium on Software Engineering*, João Pessoa, PA, Brazil. (in Portuguese).

The AspectJ Team (2006). Aspectj programming guide. Available at: `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`. Accessed on: 04/03/2007.

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

Zhao, J. (2000). Dependence analysis of Java bytecode. In *24th IEEE Annual International Computer Software and Applications Conference (COMPSAC'2000)*, pages 486–491, Taipei, Taiwan. IEEE Computer Society Press.