Coverage testing of Java programs and components

A.M.R. Vincenzi\textsuperscript{a,c}, J.C. Maldonado\textsuperscript{a,*}, W.E. Wong\textsuperscript{b}, M.E. Delamaro\textsuperscript{c}

\textsuperscript{a} Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, Av. Trab. São Carlense, 400, Cx. Postal 668, 13560-970 São Carlos, SP, Brazil

\textsuperscript{b} Department of Computer Science, University of Texas at Dallas, 2601 North Floyd Road, P.O. Box 830688, Richardson, TX 75080, USA

\textsuperscript{c} Centro Universitário Eurípides de Marília, Av. Hygino Muzzi Filho, 529, Cx. Postal 2041, 17525-901 Marília, SP, Brazil

Received 8 November 2003; received in revised form 30 August 2004; accepted 6 September 2004
Available online 13 December 2004

Abstract

Although software testing is a crucial activity in the software development process, only recently have more sound and consistent testing tools become available for testing Java programs and their components. These tools support mostly functional and control-flow based structural criteria. In this paper we explore control-flow and data-flow based coverage criteria to support the testing of Java programs and/or components. We also describe a testing tool, named JaBUTi, which can be used by both the component developer and the component user to test Java-based components and/or systems. To achieve this goal, the tool works at the bytecode level such that no source code is required during the testing activity. We illustrate these ideas and concepts with an example extracted from the literature.

© 2004 Elsevier B.V. All rights reserved.

Keywords: Java program testing; Java component testing; Java bytecode; Structural testing; Control-flow testing; Data-flow testing

* Corresponding author. Tel.: +55 16 273 9669; fax: +55 16 273 9751.
E-mail address: jc.maldon@icmc.usp.br (J.C. Maldonado).
1. Introduction

Software testing is a crucial activity in the software development process. In order to perform software testing in a systematic way and on a sound theoretical basis, testing techniques and criteria have been investigated. Testing techniques are classified as functional, structural, fault based, and state based, depending on the source used to determine the required elements to be exercised (the test requirements). These testing techniques are complementary and the question to be answered is not “Which one to use” but “How to use them in a coordinated way, taking advantage of each one.”

Each of the above mentioned techniques includes several criteria that define specific requirements that should be satisfied by the test set. In this way, requirements determined by a testing criterion may be used either for test set evaluation or test set generation. The adequacy of the testing activity is related to the determination of the effectiveness of a test criterion. Test effectiveness is related to the task of creating the smallest test set for which the output indicates the largest set of failures.

With the advance of the Component-Based Software Development (CBSD), the component user needs to have some way to evaluate the quality of the software components being integrated. Different testing strategies have been developed to test software components and component-based systems [13,27,14,22,4,33].

Many initiatives have been taken to support OO program testing; some can be instantiated or directly applied in the context of Java programs/components: how to perform static analysis [35,31]; definition of testing criteria and strategies [27,22,15,21,9,6,28]; and empirical evaluation of testing criteria [21,9]. With respect to (w.r.t.) the automation, there are some tools to support functional [2,7] and structural testing criteria (statement and decision coverage) [5,26,19,10,12].

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only [29]. A software component can be deployed independently and is subject to composition by third parties. It can be observed that software components inherit many characteristics of the OO paradigm, but the notion of components transcends the notion of objects. Software components may range from in-house components to Commercial Off-The-Shelf (COTS) components. The former allow users to have complete access and control over the code, including the development and testing processes. For the latter, on the other hand, knowledge about the development or testing processes may not be accessible.

This paper explores the use of control-flow and data-flow criteria to support the testing of Java programs (Java bytecode) aimed at establishing intra-method structural testing criteria to test Java programs and components. Six intra-method structural testing criteria, based on the traditional all-nodes, all-edges and all-uses criteria, are defined. The underlying data-flow model, characterizing the definitions and uses of variables, is discussed. A testing tool, named JaBUTi\(^1\) (Java Bytecode Understanding and Testing), which supports the application of such criteria for testing Java programs and components, is also described. Since the majority of the testing criteria for component-based software

\(^1\) Ja buti—Brazilian for turtle.
are classified as functional, our tool provides a set of intra-method structural testing criteria that can be applied to test Java programs and components.

The remainder of this paper is organized as follows. An overview of related studies is presented in Section 2. In Section 3, we describe our approach to collecting control-flow and data-flow information from Java bytecode to construct the underlying model required to derive structural testing requirements. The definition of four control-flow based and two data-flow based testing criteria is also presented. A simple vending machine example, adapted from [22], is used to illustrate the concepts and how to use JaBUTi. In Section 4, the JaBUTi coverage analysis tool is described. Finally, in Section 5 we present our conclusions and future research.

2. Related studies: Java program and component testing

For testing Java programs and components there are different testing criteria and strategies which can be used. Functional criteria (black-box testing), including statistical software testing, can be applied indiscriminately to procedural and OO programs [3] since the testing requirements are obtained from the product specification. The major problem with the functional testing is that since it is based only on the specification there is no evidence that essential parts of the implementation have been covered. On the other hand, structural testing [15] (white-box testing), which allows measuring the coverage of a given program/component, requires the source code to derive the testing requirements that may not be available to the tester/user.

Black-box and white-box testing are complementary to each other in the sense that they are likely to uncover different classes of faults. Black-box testing focuses on the functional requirements of the software. It aims at faults related to incorrect or missing functions, interface errors, behavior or performance errors and initialization and termination errors. White-box testing focuses on the internal structure of the program, to guarantee that all independent paths within a module have been exercised at least once, exercise all logical decisions on their true and false sides, execute all loops at their boundaries and within their operational bounds and exercise internal data structures [20].

When considering the testing of Java components, it is possible to analyze the problem of testing from two perspectives: the component developer and the component user [16]. Component developers test their components without knowing all possible contexts of their applications. Component users in their turn test their applications possibly without access to the component source code or data about the testing performed on them by the developers [16].

This implies that, since the component developer has complete access to the component source code, he/she can use functional, structural, fault-based and/or state-based criteria to perform the component verification and testing. However, from the user’s perspective, when no source code is available and the component does not have built-in testing capabilities, only functional and/or state-based criteria have been used.

Since structural criteria, in general, require the Java source code to derive the testing requirements, alternative criteria, which do not require the source code, have been proposed based on reflection [13], polymorphism [27], metadata [14] and metacontent [22],
state-based testing [4] and built-in testing [33,13]. They are adequate to test Java components. These approaches try to minimize the problem from the user side when no source code is available. The user, except when the source code is available or when working with components that have built-in test or metacoment capabilities, cannot obtain any information about the coverage of individual components or about the component-based system being implemented. In the case of Java-based components, such as JavaBeans, this situation can be overcome by carrying out structural testing, such as control-flow and data-flow criteria, directly on the Java bytecode.

Bytecode instructions resemble an assembly-like language, but the so-called class file (.class) retains very high-level information about a program such that it is possible to derive structural testing requirements direct from Java bytecode. Working at the bytecode level, both component developer and component user can use the same underlying representation to perform the testing activity. Moreover, the user can evaluate how much of the bytecode of a given component has been covered by a given test set, i.e., he/she can evaluate the quality of the functional test set for covering structural elements of the component.

Both control-flow and data-flow based testing criteria were originally defined to test procedural programs [25,18]. Considering the data-flow testing criteria, they were also extended to object oriented programs [15].

Many times, when defining data-flow criteria, it is not clear what should be taken as the definition and use of variables. However, when such criteria are implemented, they require the definition of a data-flow model that precisely establishes how the definitions, uses and data-flow associations are identified, considering the common structures in the target language. Mainly for the C programming language, which makes extensive use of pointers, different data-flow models have been defined and implemented to deal with aggregated variables (arrays), structured variables (records), and dereferenced variables [17,23,30].

Zhao [35] and Chambers et al. [8] describe two different techniques to identify dependency analysis in Java bytecode. Our work is similar to the one developed by Zhao [35]. In Zhao’s work, control-flow and data-flow dependence analysis is also carried out directly on the Java bytecode. The main difference is the data-flow model used to identify the set of defined and used variables. Zhao’s data-flow model considers only the set of local variables defined in a method to identify the data dependence. In our data-flow model, in addition to the set of local variables we also consider the set of instance variables and class variables. We believe that with these extensions a more precise set of data-flow associations can be collected and used, for example, to implement a coverage testing tool. Moreover, Zhao [35] did not define any testing criteria. Based on our underlying model to represent the intra-method control-flow and data-flow of a given method, briefly described in Section 3.2, six different structural testing criteria are defined (Sections 3.3 and 3.4). In future work we intend to perform some empirical studies comparing our approach with the one developed by Zhao [35].

---

1 In fact, we are assuming in this paper that the Java bytecode is obtained from a Java source code, but the origin of the class file can be other than a Java source code, which makes the approach of analyzing the object code automatically extensible to other languages that generate Java bytecode.
Software component testing, as testing in general, requires several types of supporting tools: test case generators, drivers and stubs generators, coverage analysis tools, and so on. From the developer’s perspective, most testing tools for test case generation can be used to support component testing. The main problems in component testing automation are also related to the lack of information embedded into the component by the component developers. The lack of a standard to define how and what information should be embedded is a drawback since the development of a generic testing environment, in the sense that it would be able to deal with components from different developers and implemented in different languages, would be more difficult. Nevertheless, there are some tools in the literature to automate the testing of a component-based system, such as PiSCES [5], TCAT/Java [26], JCover [19], JUnit [2], CTB [7], JTest [11], Glass JAR Toolkit (GJTK) [12].

Therefore, the framework JUnit [2] can be used to test Java-based components and systems, but it only enables performing black-box testing; it does not report coverage information. PiSCES [5], TCAT/Java [26], JCover [19] and JTest [10] report coverage information (statements and decision coverage) but all of them require the source code to perform their activities such that they are not adequate for component testing at the user side. The only tool that the authors are aware of that operates at bytecode is GlassJAR [12], which provides coverage information with respect to control-flow testing criteria (statements and decisions coverage) for Java programs and components. But GlassJAR does not support data-flow coverage testing criteria. JaBUTi, described in Section 4, provides coverage information considering both control-flow and data-flow testing criteria for Java programs and components. The testing criteria supported by JaBUTi are described in the next section.

3. Definition of control-flow and data-flow coverage criteria for Java programs and components

In this section we present the definition of six intra-method structural testing criteria that can be used for testing Java programs and components. First, in Section 3.1, a simple example used in this paper is described. In Section 3.2 the underlying representation from where control-flow and data-flow testing requirements can be derived is presented. Then, in Sections 3.3 and 3.4 we present the definitions of the control-flow and data-flow based testing criteria implemented in JaBUTi, respectively.

3.1. The vending machine example

To illustrate the ideas, concepts and the JaBUTi’s functionalities we will use the vending machine example, adapted from [22], consisting of a component and an application that uses it. The Dispenser component keeps information about the price of each item and which items are valid and available. The most important method of Dispenser class is the Dispenser.dispense() method, which receives as parameters the amount of money (credit) and the number of the selected item (sel) and decides whether the item can be dispensed, returning its value (val), or not, issuing an error message. Another method is the Dispenser.available(), which checks whether a given item is available or not.
3.2. The underlying bytecode data-flow model

The most common model to establish control-flow testing criteria [20] is the control-flow graph (CFG), from which the testing requirements are derived. Data-flow testing criteria [25] use the def-use graph (DUG), which is an extension of the CFG with information about the set of variables defined and used in each node/edge of the CFG. In this paper, the DUG graph is the underlying model used to derive both control-flow and data-flow testing requirements. The generation of the DUG graph from the bytecode is described elsewhere [31].

Fig. 1(b) illustrates the DUG graph of the Dispenser.available() method. Observe that on this DUG we have two different types of edges: (1) regular edges (continuous lines) representing the regular control-flow, i.e., that defined by the language statements; and (2) exception edges (dashed lines) representing the control-flow when an exception is raised.

Let $E$ be the set of edges of a DUG graph. A path $\pi$ is a finite sequence of nodes $(n_1, n_2, \ldots, n_k), k \geq 2$, such that there exists an edge $(n_i, n_{i+1}) \in E$ for $i = 1, 2, \ldots, k - 1$. An exception-free path is a path $\pi \mid \forall(n_i, n_j) \in \pi \Rightarrow (n_i, n_j)$ that is reachable through a path which does not contain any exception edge.
Fig. 1(a) illustrates the DUG graph of another method (Dispenser.dispense()). Observe that we also make a distinction between different types of nodes in the DUG graph: termination nodes represented by bold line circles (as block 148 of Fig. 1(a)); and regular nodes represented by single line circles (as all the other nodes in Fig. 1(a)).

In order to augment the DUG graph with data-flow information it is necessary to define the data-flow model for Java bytecode. We defined a data-flow model which extends Zhao’s data-flow model to consider not only the set of local variables defined in a method, but also the set of instance variables and class variables. The data-flow model indicates what kinds of bytecode instructions lead to variable definition/use and how reference and array variables are considered. Observe that some of the data-flow associations, e.g., \((valSet, 0, (4, 41))\) (Fig. 1(b)), would not be required in Zhao’s data-flow model,
such that faults related to such a variable might not be detected. In the following we present a summary of the data-flow model for Java bytecode extracted from [31].

In Java and Java bytecode, the variables can be classified in two types: basic or reference types. Fields can be of a basic or reference type and classified as instance or class fields, depending on whether they are unique to each object of the class or unique to the entire class (static fields), respectively. Local variables can also be of basic or reference type. An aggregated variable (array) is of reference type. To deal with aggregated variables we are following the approach proposed by Horgan and London [17], which considers a unidimensional aggregated variable as a unique storage location such that when a definition (use) of any element of the aggregated variable occurs, what is considered to be defined (used) is the aggregated variable and not a particular element. So, in our data-flow model the following assumptions apply to identify definitions and uses of variables:

1. Aggregated variables are considered as a unique storage location and the definition/use of any element of the aggregated variable $a[i]$ is considered to be a definition/use of $a[i]$. So, in the statement "$a[i] = a[i] + 1$" there is a definition and a use of the aggregated variable $a[i]$.

2. If an aggregate variable $a[][i]$ is declared, an access to its elements is considered a definition or use of $a[][i]$. Then, in the statement "$a[0][0] = 10$" there is a definition of $a[][i]$, and in the statement "$a[0] = \text{new int[10]}$", a definition of $a[]$.

3. Every time an instance field (or an array element) is used (defined) there is a use of the reference variable used to access the field and a use (definition) of the field itself. Considering $\text{ref}_1$ and $\text{ref}_2$ as reference variables of a class $C$ which contains two instance fields $x$ and $y$ of type int, in the statement "$\text{ref}_1.x = \text{ref}_2.y$" there are uses of the reference variables $\text{ref}_1$ and $\text{ref}_2$, a use of the instance field $\text{ref}_2.y$, and a definition of the instance field $\text{ref}_1.x$. Since instance fields are valid in the entire scope of a given class, each non-defined instance field used in the context of a given method is considered to have a definition in the first node of the $DUG$ graph of such a method.

4. Class fields (static fields) can be considered global variables and do not require a reference variable to be accessed. Considering a class $C$ with static fields $w$ and $z$, in the statement "$C.z = C.w + y$" there is a use of $C.w$ and a definition of $C.z$. Even if a static field is accessed using a reference variable $\text{ref}_1$ of class $C$, such that "$\text{ref}_1.w = 10$", at the bytecode level, such a reference is converted to the class name and there is no use of the reference variable in this statement. Since static fields are in the entire scope of a given class, each non-defined static field used in the context of a given method is considered to have a definition in the first node of the $DUG$ graph of such a method.

5. A method invocation such as $\text{ref}_1.\text{foo}(e_1, e_2, \ldots, e_n)$ indicates a use of the reference variable, $\text{ref}_1$. The rules for definition and use identification inside expressions $e_1, e_2, \ldots, e_n$ are the same as described in items 1 to 4.

6. For instance methods, a definition of this is assigned to the first node of the $DUG$, and also the definition of each local variable corresponding to the formal parameters of such a method, if any. For class methods, only the local variables corresponding to the parameters are considered; no instance is required to invoke such a method.
Based on these assumptions the $DUG$ of a given method is traversed and to each node of such a graph a set $D_i$ of defined variables and a set $U_i$ of used variables are assigned, as illustrated in the $DUG$s of Fig. 1. One point that should be highlighted is that, due to the stack oriented structure of the Java Virtual Machine, it would be hard to distinguish between $p$-$use$ (Predicate Use) and $c$-$use$ (Computational Use) by analyzing the bytecode. All bytecode instructions related to control transfer only check the value into the top of the operand stack to decide which path to follow. Therefore, to identify which variables have predicative or computational uses would be expensive, since it would be necessary to identify all variables which influence the current value into the top of the operand stack.

In our implementation of the all-uses criterion we consider each use in a predicative node (a node with more than one out-going regular edge) as a $p$-$use$ associated with each out-going regular edge. Only $p$-$use$ associations are created, since the related $c$-$use$ associations are covered once the $p$-$use$ associations are covered. This simplifies the algorithm to identify $p$-$uses$ and $c$-$uses$, although additional def-$use$ associations are required. Observe that no $p$-$use$ association is created with respect to the out-going exception edges. Moreover, our implementation does not care of transitive definition.

Therefore, considering a given $DUG$ graph node $i$, three sets can be derived, considering an adaptation of Rapps & Weyuker’s work [25] at the bytecode level:

$$\text{def}(i) = \{\text{variables with definition at node } i\}$$
$$\text{c-use}(i) = \begin{cases} \{\text{variables use at node } i, \text{if } i \text{ is a computational node}\} \\ \emptyset, \text{otherwise (} i \text{ is a predicative node)} \end{cases}$$
$$\text{p-use}(i) = \begin{cases} \{\text{variables use at node } i, \text{if } i \text{ is a predicative node}\} \\ \emptyset, \text{otherwise (} i \text{ is a computational node)} \end{cases}.$$  

From the $c$-$use$ and $p$-$use$ sets, two other sets are defined: $\text{dcu}(x, i) = \{\text{DUG graph nodes } j \text{ such that } x \in \text{c-use}(j) \text{ and there exists a def-clear path w.r.t. } x \text{ from node } i \text{ to the node } j\}$ and $\text{dpu}(x, i) = \{\text{edges } (j, k) \text{ of a DUG graph such that } x \in \text{p-use}(j) \text{ and there exists a def-clear path w.r.t. } x \text{ from node } i \text{ to the edge } (j, k)\}$.

A definition-$c$-$use$ association is a triple $(i, j, x)$ where $i$ is a node containing a definition of a variable $x$ and $j \in \text{dcu}(x, i)$. A definition-$p$-$use$ association is a triple $(i, (j, k), x)$ where $i$ is a node containing a definition of $x$ and $(j, k) \in \text{dpu}(x, i)$. An association is a definition-$c$-$use$ association or a definition-$p$-$use$ association.

### 3.3. Control-flow criteria

The $DUG$ graph is an abstract representation of a given method in a program/component. Control-flow testing criteria can be derived based on such a program representation to provide a theoretical and systematic mechanism to select and assess the quality of a given test set. We are using two well known control-flow testing criteria to derive testing requirements from the $DUG$ graph: all-nodes and all-edges [20].

Based on the concept of exception-free path, the set of requirements of the All-Nodes and All-Edges criteria can be subdivided in two non-overlapping subsets such that the tester can concentrate on different aspects of a program at a time.

Let $N$ be the set of nodes of a $DUG$ graph. $N_{ed} \subseteq N$ is the subset of exception-dependent nodes defined as $N_{ed} = \{n \in N \mid \exists \pi \text{ a complete exception-free path } \pi \text{ such that} \}$. 


that \( n \in \pi \), i.e., \( N_{ed} \) is the subset of \( \mathcal{DUG} \) graph nodes which can only be reached after an exception has been raised. The set of exception-independent nodes \( N_{ei} \) is defined as: \( N_{ei} = N - N_{ed} \), i.e., the subset of all \( \mathcal{DUG} \) graph nodes which can be reached without an exception to be thrown. Observe that these subsets of nodes are disjoint, i.e., \( N_{ei} \cap N_{ed} = \emptyset \), and together they represent the complete set of \( \mathcal{DUG} \) graph nodes, i.e., \( N_{ei} \cup N_{ed} = N \).

Let \( E \) be the set of edges of a \( \mathcal{DUG} \) graph. \( E_{ed} \subseteq E \) is the subset of exception-dependent edges defined as \( E_{ed} = \{ e \in E \mid \exists \pi \text{ a complete exception free path } \pi \text{ such that } e \in \pi \} \). The set of exception-independent edges \( E_{ei} \) is defined as: \( E_{ei} = E - E_{ed} \). Observe that these subsets are disjoints, i.e., \( E_{ei} \cap E_{ed} = \emptyset \), and together they represent the complete set of edges of a \( \mathcal{DUG} \) graph, i.e., \( E_{ei} \cup E_{ed} = E \).

Let \( T \) be a test set for a program \( P \) (\( \mathcal{DUG} \) is the corresponding def-use graph of \( P \)), and let \( \Pi \) be the set of complete paths executed by \( T \). It is said that a node \( i \) is included in \( \Pi \) if \( \Pi \) contains a path \((n_1, \ldots, n_m)\) such that \( i = n_j \) for some \( j, 1 \leq j \leq m \). Similarly, an edge \((i_1, i_2)\) is included in \( \Pi \) if \( \Pi \) contains a path \((n_1, \ldots, n_m)\) such that \( i_1 = n_j \) and \( i_2 = n_{j+1} \) for some \( j, 1 \leq j \leq m-1 \). A path \((i_1, \ldots, i_k)\) is included in \( \Pi \) if \( \Pi \) contains a path \((n_1, \ldots, n_m)\) and \( i_1 = n_j, i_2 = n_{j+1}, \ldots, i_k = n_{j+k-1} \), for some \( j, 1 \leq j \leq m-k+1 \).

- **all-nodes criterion** (referred as All-Nodes): \( \Pi \) satisfies the All-Nodes criterion if every node \( n \in N \) of a \( \mathcal{DUG} \) graph is included in \( \Pi \). In other words, this criterion requires that each bytecode instruction of a given method is executed at least once.

To distinguish between bytecode instructions that are executed under the normal execution of the program from the others that require an exception to be executed, we subdivide such a criterion in two non-overlapping testing criteria:

- **all-nodes-exception-independent (All-Nodes\textsubscript{ei})**: \( \Pi \) satisfies the All-Nodes\textsubscript{ei} criterion if every node \( n \in N_{ei} \) is included in \( \Pi \). In other words, this criterion requires that every node of the \( \mathcal{DUG} \) graph, reachable through an exception-free path, is executed at least once. Considering Fig. 1(b), nodes 0, 2, 4, 13, 17, 29 and 41 are exception-independent nodes.

- **all-nodes-exception-dependent (All-Nodes\textsubscript{ed})**: \( \Pi \) satisfies the All-Nodes\textsubscript{ed} criterion if every node \( n \in N_{ed} \) is included in \( \Pi \). In other words, this criterion requires that every node of the \( \mathcal{DUG} \) graph, not reachable through an exception-free path, is executed at least once. Considering Fig. 1(b), node 38 is an exception-dependent node.

Analogously, we also defined the criterion **All-Edges** and the corresponding criteria **all-edges-exception-independent** (All-Edges\textsubscript{ei}) and **all-edges-exception-dependent** (All-Edges\textsubscript{ed}).

### 3.4. Data-flow criteria

With respect to data-flow testing, we are using the well known **all-uses** criterion [25] that is composed of the **all-c-uses** and **all-p-uses** criteria.

In order to allow the subdivision of the set def-use associations in two disjoint sets considering whether there exists a def-clear path which is also an exception-free path, the following subsets, derived from \( dcu(x, i) \) and \( dpu(x, i) \), are defined:

- \( dcu_{ed}(x, i) = \{ \mathcal{DUG} \text{ graph nodes } j \in dcu(x, i) \text{ and there does not exist an exception-free path w.r.t. } x \text{ from node } i \text{ to node } j \} \);
• \( dcu_{ei}(x, i) = dcu(x, i) - dcu_{ed}(x, i) \);
• \( dpu_{ed}(x, i) = \{ DG \) graph edges \( (j, k) \in dpu(x, i) \) and there does not exist an exception-free path w.r.t. \( x \) from node \( i \) to the edge \( (j, k) \} \);
• \( dpu_{ei}(x, i) = dpu(x, i) - dpu_{ed}(x, i) \).

Considering these definitions, an exception-independent def-c-use association is a triple \( (i, j, x) \) where \( i \) is a node containing a definition of a variable \( x \) and \( j \in dcu_{ei}(x, i) \), and an exception-dependent def-c-use association is a triple \( (i, j, x) \) where \( i \) is a node containing a definition of a variable \( x \) and \( j \in dcu_{ed}(x, i) \). Similarly, an exception-independent def-p-use association is a triple \( (i, (j, k), x) \) where \( i \) is a node containing a definition of \( x \) and \( (j, k) \in dpu_{ei}(x, i) \), and an exception-dependent def-p-use association is a triple \( (i, (j, k), x) \) where \( i \) is a node containing a definition of \( x \) and \( (j, k) \in dpu_{ed}(x, i) \).

- **all-uses criterion (All-Uses)**: \( \Pi \) satisfies the All-Uses criterion if for every node \( i \in N \) and for every variable \( x \in \text{def}(i) \), \( \Pi \) includes a def-clear path w.r.t. \( x \) from node \( i \) to every element of \( dcu_{ei}(x, i) \) and \( dpu_{ei}(x, i) \). In other words, this criterion requires that every def-c-use association \( (i, j, x) \mid j \in dcu_{ei}(x, i) \) and every def-p-use association \( (i, (j, k), x) \mid (j, k) \in dpu_{ei}(x, i) \) is exercised at least once for some test case \( t \in T \).

As with the All-Nodes and All-Edges criteria, we divided the All-Uses criterion such that two sets of non-overlapping testing requirements are obtained. We named such criteria all-uses-exception-independent and all-uses-exception-dependent, respectively.

- **all-uses-exception-independent (All-Uses_{ei})**: \( \Pi \) satisfies the All-Uses_{ei} criterion if for every node \( i \in N \) and for every variable \( x \in \text{def}(i) \), \( \Pi \) includes a def-clear path w.r.t. \( x \) from node \( i \) to every element of \( dcu_{ei}(x, i) \) and \( dpu_{ei}(x, i) \). In other words, this criterion requires that every exception-independent def-c-use association \( (i, j, x) \mid j \in dcu_{ei}(x, i) \) and every exception-independent def-p-use association \( (i, (j, k), x) \mid (j, k) \in dpu_{ei}(x, i) \) is exercised at least once for some test case \( t \) of \( T \).

Considering Fig. 1(b), some examples of exception-independent def-use associations are: \( \{L@2, 0, 41\} \), \( \{L@0.valSel, 0, (4, 13)\} \) and \( \{L@0.valSel, 0, (4, 41)\} \).

- **all-uses-exception-dependent (All-Uses_{ed})**: \( \Pi \) satisfies the All-Uses_{ed} criterion if for every node \( i \in N \) and for every variable \( x \in \text{def}(i) \), \( \Pi \) includes a def-clear path w.r.t. \( x \) from node \( i \) to every element of \( dcu_{ed}(x, i) \) and \( dpu_{ed}(x, i) \). In other words, this criterion requires that every exception-dependent def-c-use association \( (i, j, x) \mid j \in dcu_{ed}(x, i) \) and every exception-dependent def-p-use association \( (i, (j, k), x) \mid (j, k) \in dpu_{ed}(x, i) \) is exercised at least once for some test case \( t \) of \( T \).

Considering Fig. 1(b), there is only one exception-dependent def-use association: \( \{L@2, 38, 41\} \).

### 4. JaBUTi: a coverage analysis tool for Java

The use of structural testing requires a series of tasks like program analysis for testing requirement computation and coverage analysis that are not trivial. Testing tools to support such criteria are essential to assure quality of the testing activity. JaBUTi \cite{32} is a tool to support the structural testing of Java programs/components.

The tasks performed by JaBUTi are similar to other testing tools which support structural testing criteria. What differentiates JaBUTi from other tools is that its static
analysis and instrumentation processes are not performed in the Java source code but in the object code (Java bytecode). In fact, the tester does not even need the source code to test a program, although, because the Java bytecode retains information about source code, if the source is available, the tool maps the data about the testing requirements and coverage back to the source program. For the testing of Java-based components it is very important that analysis is performed at the bytecode level, since the source code of the component is not always available.

JaBUTi has two main parts. The first makes all the analysis, requirement computation and test case evaluation, and corresponds to the top part of Fig. 2. The second (bottom part of Fig. 2) is a test driver that instruments the classes, and loads and executes the instrumented program.

The test driver has an appropriate class loader that instruments the original classes by placing at the beginning of each \texttt{DUG} node a call to a static method \texttt{probe.DefaultProber.probe} that stores the information about the piece of code (the node) about to be executed. When the program ends, another method \texttt{probe.DefaultProber.dump} is called and all the data collected by the calls to \texttt{probe.DefaultProber.probe} are written to a trace file. Those data constitute a new test case.

The analysis tool constantly polls the trace file. When a new test case is appended in the trace file, the tool updates the test information, recomputing the coverage. The analysis part also has several facilities to perform code visualization, \texttt{DUG} graph visualization, report generation and test case manipulation. A program/component test is organized in a test project, indicating the set of classes under test that can be saved and re-loaded by the tester.
4.1. JaBUTi testing project

First of all the tester needs to create a testing project indicating the set of classes under test (CUTs) (VendingMachine and Dispenser in our example). Once such classes are selected, the set of testing requirements for each criterion is obtained.

In addition to what kind of coverage information should be collected and analyzed, it is also important to study how such data should be presented. For each method of each class under test, JaBUTi provides three views: two for its textual view representation as bytecode (e.g. Fig. 3(a)) and source code (if it is available) and one for its DUG graph (e.g. Fig. 3(b)). Note that if the cursor is moved onto a DUG graph node, general information about the node is displayed. It is important to observe that even if no source code is available, the tester can use the two other views to perform the testing activity.

4.2. Improving the coverage using JaBUTi

Once the set of testing requirements w.r.t. each testing criterion has been determined, such requirements can be used to access the quality of an existent test set and/or to develop
additional test cases to improve/create a test set. For example, the tester can decide to generate a test set at random and to evaluate the coverage of such a test set w.r.t. all testing criteria supported by JaBUTi. On the other hand, the tester can visualize the set of testing requirements of each criterion w.r.t. each method, check which of them have already been covered, and develop additional test cases to cover each of the non-covered testing requirements. The tool also allows the tester to activate/deactivate different combinations of testing requirements and to mark a given testing requirement as infeasible when it is not possible to develop a test case which causes its execution.

Studies have shown that it is important to generate a test set with high coverage so that more faults are likely to be detected [24,34]. Since, in general, there is a large number of testing requirements to be covered, JaBUTi provides hints to the tester to help him/her to increase the coverage as quickly as possible w.r.t. a given test criterion. The main idea is to assign weights to each testing requirement based on dominator and super-block analysis [1]. The objective is to generate a test to cover one of the requirements with the highest weight first, if possible, before other areas with a lower weight are covered so that the maximum coverage, w.r.t. control-flow (such as All-Nodes$_{ei}$ and All-Edges$_{ei}$) and data-flow testing requirements, can be added in each single execution.

JaBUTi provides the visualization of the testing requirements at the bytecode, source code and $DUG$ graph of each method, for all the supported criteria. Colors are associated with the testing requirements according to their weights. Referring to Fig. 3(a) and (b) and considering that the All-Nodes$_{ei}$ criterion is selected, two nodes (the nodes 105 and 140) are identified as “hot spots” highlighted in red as they have the highest weight, 7.
This implies that a test that covers any one of these nodes will increase the number of covered requirements by at least seven. Since such a weight is computed by a conservative approach, the actual improvement on nodes coverage can be more than seven nodes. Indeed, this is the case. After executing Test Case 0001 which covers node 105 (one of the “hot spots”), the All-Nodes_{ei} coverage is increased to 56% (refer to Fig. 5(a)). The updated display of the Dispenser.dispense() method w.r.t. the All-Nodes_{ei} criterion is in Fig. 4(a). In this figure, we can clearly see that many blocks are in white because they are covered by Test Case 0001. Also, the hot spots, i.e., the remaining uncovered nodes with the highest weights, move to different locations to provide different hints on how to generate the next effective test after Test Case 0001. The difference between the weights in the initial display and those in the updated display after executing Test Case 0001 can be clearly identified by comparing Figs. 3(a) and 4(a). The highest weight in our case is reduced from 7 to 1. This is consistent with the common understanding in coverage-based testing that it becomes more and more difficult to further improve the coverage after a few tests have been executed.
For the purpose of comparison, suppose instead of taking advantage of the hints provided by JaBUTi to generate a test to cover the hot spot, a tester generates another test (Test Case 0003 in Fig. 5(a)) by which the highest weight of all the nodes covered is only 7. We notice a clear difference between the coverage (All-Nodes$_{ei}$ in our case) achieved by executing Test Case 0001 (which is 56%) and that by Test Case 0003 (which is 23%). A similar difference is also found w.r.t. All-Edges$_{ei}$, All-Uses$_{ei}$, All-Nodes$_{ed}$, All-Edges$_{ed}$ and All-Uses$_{ed}$ criteria.

4.3. Customizing a coverage report

To evaluate the progress of the testing activity, it is possible to decompose the overall cumulative coverage obtained by executing a set of tests on a JaBUTi project into coverage w.r.t. a subset of test cases on certain parts of the project. For example, one might be interested in the All-Nodes$_{ei}$ coverage w.r.t. each test case to discover which one is the more effective. JaBUTi provides the tester with an option to report coverage w.r.t. each test case, as illustrated in Fig. 5(a).

It is also important to know the coverage measurement with respect to each criterion. This information can help the tester decide whether a “saturation effect” has been reached w.r.t. a given criterion. If so, the coverage should be measured w.r.t. another stronger criterion (e.g. moving from the All-Nodes$_{ei}$ to the All-Edges$_{ei}$ criterion). JaBUTi can also report the total coverage w.r.t. six different criteria: All-Nodes$_{ei}$, All-Edges$_{ei}$, All-Uses$_{ei}$, All-Nodes$_{ed}$, All-Edges$_{ed}$ and All-Uses$_{ed}$. Fig. 5(b) gives such a report. Additionally, it is also possible to obtain testing reports regarding each class/method to discover how much of the class/method has been tested w.r.t. a given criterion.
5. Conclusion and future research

We propose a white-box-based coverage testing for Java-based programs and components. The coverage is measured directly on Java bytecode instead of the Java source code. Six different coverage criteria are used. Four of them are control-flow based (All-Nodes_{ei}, All-Nodes_{ed}, All-Edges_{ei} and All-Edges_{ed}) and two are data-flow based (All-Uses_{ei} and All-Uses_{ed}). We emphasize that the focus of coverage testing should not only be limited to measuring the coverage. We should also think one step ahead in how coverage data collected during testing can help practitioners in subsequent activities such as debugging and understanding Java programs/components.

A coverage analysis tool, JaBUTi, was implemented to aid in testing programs/components written in Java. In addition to reporting the current coverage w.r.t. a set of tests on a Java program/component, some of the special facilities of JaBUTi also provide the following important features:

- detailed coverage reports w.r.t. each test case, each coverage criterion, each class, and each method;
• customized reports by decomposing an overall cumulative coverage w.r.t. selected tests on certain parts of a given class under test;

• user-friendly interfaces for visualizing control-flow and data-flow based coverage in Java code as well as in its def-use graph; and

• hints through a dominator and super-block analysis to guide testers in generating tests to increase coverage in an effective way.

Finally, we outline the direction for our future research. An interesting and important study is to apply JaBUTi to industry projects to evaluate the cost and benefits of control-flow and data-flow based criteria in testing Java programs and components. It would also be interesting to examine how much time testers can save by using the hints provided by our tool in generating tests to achieve high coverage in comparison to the approach without using JaBUTi. It is also important to incorporate such hints into a process for automatic test generation. In addition, we want to extend the intra-method testing criteria to inter-method level such that after the unit testing, integration testing can also be performed both by component developers and component uses, improving the confidence in the system/component under test.

Acknowledgments

This work is a joint project supported by the Brazilian Funding Agencies—CNPq (under grant 474890/01-5), CAPES and FAPESP (under grant 98/16294-1)—and by The University of Texas at Dallas (USA). The authors want to thank the XiSuds research team at Telcordia Technologies (formerly Bellcore). The first author was supported by FAPESP Process Number 98/16492-8 as well as The University of Texas at Dallas. Mr. Vincenzi was a visiting student under Professor Eric Wong’s supervision from February 2002 to January 2003.

References


