JaBUTi: A Coverage Analysis Tool for Java Programs

A. M. R. Vincenzi[1], W. E. Wong[2], M. E. Delamaro[3], J. C. Maldonado[1]

[1]Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo
Av. Trabalhador São Carlense, 400, Caixa Postal 668 – 13560-970 São Carlos, SP, Brazil

[2]Department of Computer Science – University of Texas at Dallas
2601 North Floyd Road, P.O. Box 830688 – Richardson, Texas, USA, 75080

[3]Faculdade de Informática – Centro Universitário Eurípides de Marília
Av. Hygino Muzzi Filho, 529, Caixa Postal 2041 – 17525-901 Marília, SP, Brazil

{auri, jcmaldon}@icmc.usp.br, ewong@utdallas.edu, delamaro@fundanet.br

Abstract. *This paper describes a coverage testing tool, named JaBUTi, designed to test Java programs and Java-based components. From the Java bytecode, the tool extracts intra-method control-flow and data-flow testing requirements that can be used to generate or to assess the quality of a given test set. Many existing control-flow and data-flow testing tools require the source code to perform their activities. However, this approach may not be feasible in some situations, e.g., for component-based software development as some of the components can be commercial off-the-shelf products or developed by a third party, and therefore the corresponding source code is not always available. For programs written in Java, using the object code (bytecode) one can overcome this constraint.*

1. Introduction

This paper describes a coverage testing tool for Java programs and Java-based components, named JaBUTi (Java Bytecode Understanding and Testing). Differently from other testing tools that require the Java source code to carry out the program analysis, instrumentation and coverage assessment, our tool requires only the Java bytecode. Bytecode can be viewed as an assembly-like language that retains high-level information about a program [Lindholm and Yellin, 1999]. Therefore, JaBUTi enables the user to test Java programs and Java-based components even when no source code is available. There are others testing tools that work at the bytecode level. The framework JUnit [Beck and Gamma, 2002], for instance, can be used to test Java bytecode, but it only enables performing black-box testing; it does not report any coverage information. JTest [Corporation, 2002] and GlassJAR [Edge, 2002] provide coverage information with respect to control-flow testing criteria at the bytecode level, but none of them supports data-flow coverage testing criteria.

By working at the bytecode level we want to provide a tool that can be used for testing not only Java applications that have the corresponding source code available, but also, the ones that have only the bytecode, like software components.

2. The JaBUTi Tool

JaBUTi intends to be a complete toolsuite for understanding and testing Java programs and Java-based components. JaBUTi can be used to perform coverage analysis using

different testing criteria, and to localize faults using slicing heuristics. It also collects complexity metrics information based on the bytecode. The focus of this paper is to describe how to use JaBUTi as a coverage analysis tool.

JaBUTi implements four intra-method control-flow based testing criteria (all-pri-nodes, all-sec-nodes, all-pri-edges, and all-sec-edges) and two intra-method data-flow based testing criteria (all-pri-uses and all-sec-uses). Observe that the pair all-pri-nodes and all-sec-nodes and the pair all-pri-edges and all-sec-edges compose the traditional all-nodes and all-edges criteria, respectively [Roper, 1994], likewise the all-pri-uses and all-sec-uses compose the all-uses criterion [Rapps and Weyuker, 1985]. We decided to distinguish between two different kinds of edges to represent the exception-handling mechanism of Java. **Primary-edges** (pri-edges) correspond to the "normal" control flow of the program, while **secondary-edges** (sec-edges) represent the control flow when an exception is raised. Primary nodes are those reachable through a path that does not include any secondary edge. Primary def-use associations are those that can be covered through a path that does not included any secondary edge. This distinction permits the tester to concentrate on different aspects of the program at a time, performing the testing activity in an incremental way. More details about the criteria can be found elsewhere [Vincenzi et al., 2003].

The main activities executed by JaBUTi to perform the coverage analysis are: to instrument class files, to collect the dynamic execution trace during the classes execution, and to determine how well the classes have been tested with respect to a given testing criterion. Using the concepts of dominator and super-block [Agrawal, 1994], the tool assigns weights to the testing requirements providing hints on which requirement should be covered first to increase the coverage as much as possible. Different testing reports can also be generated to analyze the coverage with respect to each criterion, each class, each method, and each test case. In Sections 2.1 and 2.2 we describe the operational and implementation aspects of JaBUTi, respectively. To illustrate the operational aspects of JaBUTi, we use a simple example, adapted from [Orso et al., 2001], which simulates the behavior of a typical vending machine. The two main classes are the Dispenser component and the VendingMachine application which uses the Dispenser component. Since these classes have no main method, we implement an additional class, named TestDriver, responsible to guide the execution of the VendingMachine and the Dispenser.

2.1. Operational Aspects

Fist of all the tester needs to call JaBUTi and provides the name of the base class file that he/she wants to test. A testing project indicating the set of classes under test (CUTs) needs to be created. This is done in the JaBUTi's Project Manager (Figure 1). On the left are the user-defined classes, identified from the base class TestDriver by the Lookup module (described in the next section). From the user-defined classes, two (VendingMachine and Dispenser) are selected to be tested (instrumented).

By clicking on the Ok button, JaBUTi creates a new project (vending.jbt in our example), constructs the def-use graph (DUG) for each method of the CUTs, derives the set of testing requirements for each criterion, calculates the weight of each testing requirement, and presents the bytecode of a given CUT to the tester, as illustrated in Figure 2(a). In addition to the bytecode view, the tool can show the DUG of each method (Figure 2(b)), and also the source code of the corresponding bytecode (if it is available).

JaBUTi uses different colors to provide hints to the tester to ease the test case

generation. The different colors represent requirements with different weights, and the weights are calculated based on dominator and super-block analysis [Agrawal, 1994]. Informally, the weight of a testing requirement corresponds to the number of testing requirements that will be covered if this particular requirement is covered. Therefore, covering the requirement with the highest weight will increase the coverage faster[1].

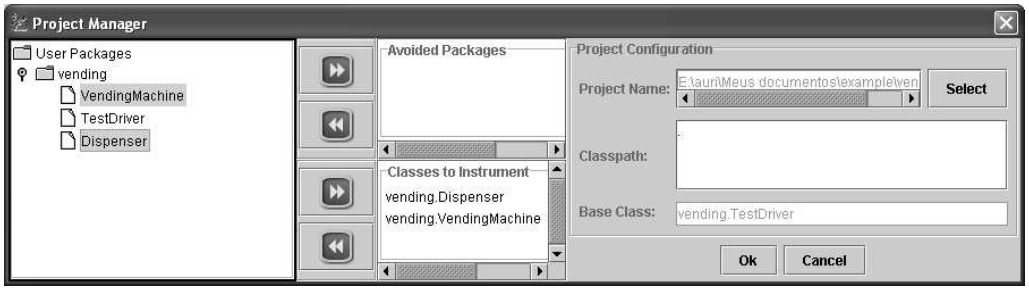
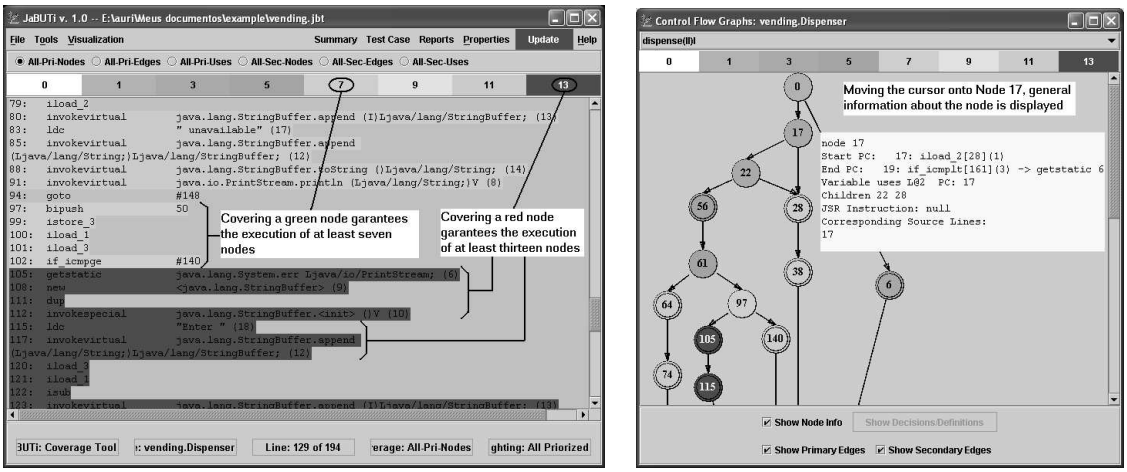


Figure 1: Project Manager dialog.



(a) Initial Bytecode (b) Initial DUG

Figure 2: The initial display of JaBUTi with respect to the All-Pri-Nodes criterion.

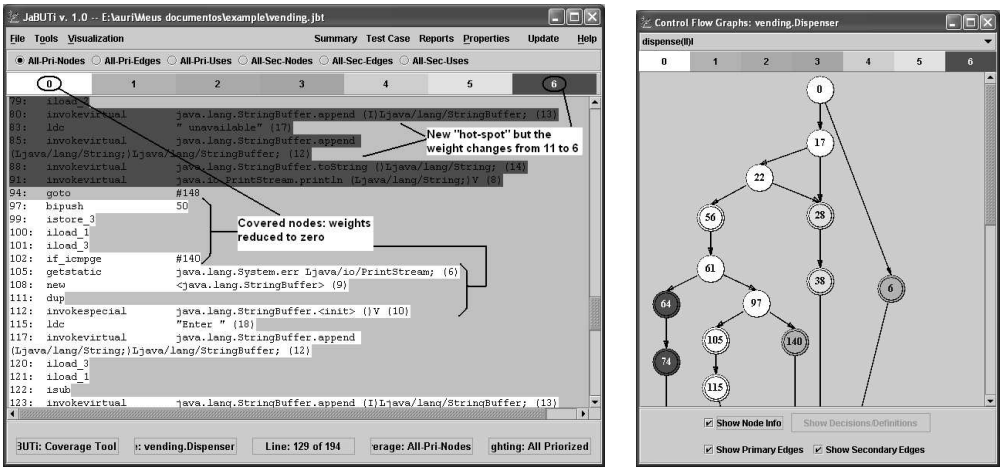
Figure 2 shows the bytecode and the DUG of the method Dispenser.dispense() before the execution of any test case. The different colors correspond to the weight of each node, considering the all-pri-nodes criterion. Observe that the color bar (in grey scale in this article) goes from white (weight 0) to red (weight 13). Node 105 in Figure 2(b), composed by bytecote instructions from offset 105 to 112 (Figure 2(a)), is one with the highest weight. This represents that a test case that exercises node 105 increases the coverage in at least 13. A requirement with weight zero is a covered requirement and it is painted in white.

A test case to be analyzed by JaBUTi should be generated by executing the CUTs using the JaBUTi's class loader such that the execution trace can be collected. Each

[1]Observe that the weight is calculated considering only the coverage information and should be seen as hints. It does not take into account, for instance, the complexity or the criticality of a given part of the program. The tester, based on his/her experience may desire to cover first a node with a lower weight but that has a higher complexity or criticality and then, after recomputing the weights, uses the hints provided by JaBUTi to increase the coverage faster.

JaBUTi's class loader execution corresponds to a new test case, and the collected execution trace is appended in the end of a trace file. Every time the size of the trace file increases, the Update button in JaBUTi's graphical interface becomes red, indicating that the coverage information can be updated considering the new test case(s). Figure 2(a) illustrates this event.

By clicking on the Update button, JaBUTi imports the new test case(s) and updates both the coverage information with respect to each testing criterion and the weights. For example, after importing a test case that exercises node 105 of Figure 2(b), Figure 3 shows the resultant colors and weights. Observe that node 105 is covered and now one of the nodes with the highest weight is node 74, which had a weight of 11 (Figure 2(a)) and now has a weight of 6.



(a) Updated bytecode (b) Updated DUG

Figure 3: Updated display of JaBUTi after executing Test Case 0001.

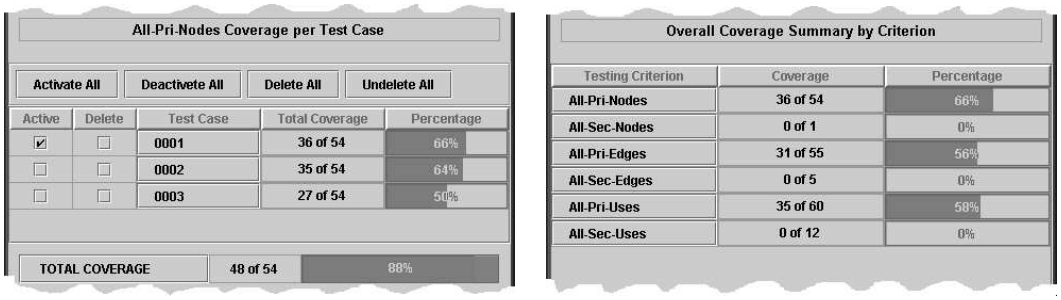
Figure 4 illustrates two kinds of testing reports that can be generated. Figure 4(a) shows the coverage by test case, considering the all-pri-nodes criterion, with respect to the entire project. The tester can change the criterion by clicking on the radio buttons below the JaBUTi's main menu (Figure 2(a)). Test Case 0001 is responsible to cover 36 out of 54 nodes (66%) required by the criterion all-pri-nodes. Figure 4(b) shows the testing report by criterion, considering the entire project. Observe that all-pri-nodes criterion has the highest coverage (66%) followed by all-pri-uses (58%), all-pri-edges(56%). Observe that since the current test set does not execute any exception-handling construction, all-sec-nodes, all-sec-edges, and all-sec-uses criteria have no requirement covered. Additional summary reports are also available such that the tester can have a more detailed coverage information not only at project level but also at class and method levels.

2.2. Implementation Aspects

Figure 5 illustrates the general architecture of JaBUTi to perform the coverage analysis. The tool is developed in Java and uses a third party package, named BCEL (Bytecode Engineering Library) [Dahm, 2001], to manipulate and display the Java bytecode.

The first step in JaBUTi is to create a testing project (a .jbt file, e.g. vending.jbt), which identifies the set of classes under test (CUTs). The Lookup module is used to

identify the complete class hierarchy. It receives as input a class file (CF), which corresponds to the base class file, and produces as output the complete class hierarchy (CH) necessary to run the base class, including both system and user-defined classes. From the user-defined classes, any subset can be selected, characterizing the CUTs. CH and CUT are stored in a project database. Selected the CUTs, the tool uses the Graph module to construct the def-use graph (DUG) of each method. Based on the DUG, testing requirements (TRs) with respect to the six testing criteria implemented in JaBUTi are derived and stored in the project database.



(a) By test case: all-pri-nodes (b) By criterion

Figure 4: Summary reports considering only Test Case 0001 as active.

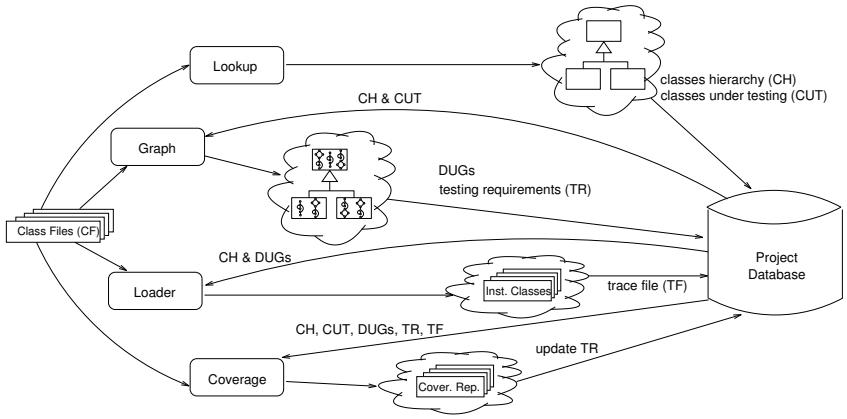


Figure 5: JaBUTi main modules.

The Loader module is the JaBUTi's class loader. Such a module instruments the CUTs, loads the instrumented classes to be executed, and records the execution trace information in a trace file (TF) (a .trc file, e.g. vending.trc). Each execution of the JaBUTi's class loader corresponds to a new test case.

The Coverage module uses the information produced by the other modules, including the set of testing requirements (TR) and the generated trace file (TF), to identify the set of covered requirements considering the execution trace stored in TF. It is also responsible to generating different testing reports, which are used by the tester to evaluate the quality of the current test set.

3. Conclusion and Future Work

This paper described how to use JaBUTi as a coverage analysis tool. The greatest advantage of JaBUTi is that it does not require the Java source code to perform its activities

since all information is collected from the Java bytecode. This characteristic allows using the tool not only on testing Java applications, but also Java-based components, which, in general, are deployed without the source code. Moreover, considering the component user, the majority of the testing criteria available is classified as black-box, which does not measure the coverage of the implementation aspects of the component. In case of Java-based components, JaBUTi provides a way to evaluate the coverage with respect to both, control and data flow testing criteria.

Our next step will be to evaluate the JaBUTi's functionality, usability, scalability, and robustness by using the tool on empirical studies and on testing real programs, making available a complete visualization and analysis toolsuite for Java applications to help programmers and testers work more effectively and efficiently.

Acknowledgments

The authors would like to thank the Brazilian Funding Agencies – FAPESP, CNPq, and CAPES – and the University of Texas at Dallas (USA) for their partial support to this research. The authors would also like to thank the anonymous referees for their comments.

References

Agrawal, H. (1994). Dominators, super block, and program coverage. In *SIGPLAN – SIGACT Symposium on Principles of Programming Languages – POPL'94*, pages 25–34, Portland, Oregon. ACM Press.

Beck, K. and Gamma, E. (2002). JUnit cookbook. web page. Available on-line: `http://www.junit.org/` [01-20-2003].

Corporation, P. (2002). Automatic Java software and component testing: Using JTest to automate unit testing and coding standard enforcement. web page. Available on-line: `http://www.parasoft.com/` [01-04-2002].

Dahm, M. (2001). Byte code engineering with the BCEL API. Technical Report B-17-98, Freie Universität Berlin – Institut für Informatik, Berlin – German. Available on-line at: `http://bcel.sourceforge.net/` [04-13-2002].

Edge, T. (2002). Glass JAR toolkit. web page. Available on-line at: `http://www.testersedge.com/gjtk/` [01-04-2003].

Lindholm, T. and Yellin, F. (1999). *The Java Virtual Machine Specification*. Addison-Wesley, 2 edition.

Orso, A., Harrold, M. J., Rosenblum, D., Rothermel, G., Do, H., and Soffa, M. L. (2001). Using component metacontent to support the regression testing of component-based software. In *IEEE International Conference on Software Maintenance (ICSM'01)*, pages 716–725, Florence, Italy. IEEE Computer Society Press.

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

Roper, M. (1994). *Software Testing*. McGrall Hill.

Vincenzi, A. M. R., Delamaro, M. E., Maldonado, J. C., and Wong, W. E. (2003). Java bytecode static analysis: Deriving structural testing requirements. In *2nd UK Software Testing Workshop – UK-Softest'2003*, pages –, Department of Computer Science, University of York, York, England. University of York Press. (accepted for publication).