JaBÁ: A Java Bytecode Analyzer

A. M. R. Vincenzi[†], M. E. Delamaro[‡], A. S. Simão[†], W. E. Wong[§] and J. C. Maldonado[†]

[†]Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo
São Carlos, São Paulo, Brazil
{auri, jcmaldon, adenilso}@icmc.sc.usp.br

[‡]Faculdade de Informática
Fundao Eurípedes Soares da Rocha
Marília, São Paulo, Brazil
delamaro@fundanet.br

[§]Department of Computer Science
University of Texas at Dallas
Richardson, Texas, USA
ewong@utdallas.edu

Abstract

Many existing control-flow and data-flow based techniques for software testing and program analysis require source code instrumentation. However, this approach may not be feasible for component-based software as some of its 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, this problem can be solved by conducting instrumentation directly on bytecode, instead of on the Java source code. In this paper, we present a Java Bytecode Analyzer (JaBÁ), developed to help programmers and testers collect control-flow and data-flow based information through bytecode instrumentation.

Keywords: *bytecode instrumentation, Java Virtual Machine (JVM), software testing, program analysis.*

1 Introduction

Control-flow and data-flow information has been widely used to help software practitioners with testing, slicing, debugging, program comprehension and performance analysis [2, 6, 10, 14, 15]. However, most techniques require instrumentation on the source code to collect the required information before any analysis can be accomplished. The use of component-based software development is steadily increasing, following the advancement and widespread use of object-oriented system design and web-based development. Software components can be commercially available off-the-shelf, developed in-house, or developed contractually. As a result, the source code of certain components may not be available all the time because the whole application is developed in a heterogeneous fashion (multiple teams in different environments). For programs written in Java, this problem can be solved by conducting instrumentation directly on bytecode instead of the Java source code.

Since Java bytecode can be viewed as an assembly-like language, tools such as Bytecode Engineering Library (BCEL) [3], Coffi [13] and Soot [12] have been developed to provide an object representation of a Java class file. Although these tools are useful for bytecode manipulation and optimization, they are designed for a general purpose which is not appropriate, without specific tailoring, for instrumenting bytecode to collect control-flow and data-flow based information. On the other hand, BIT (Bytecode Instrumenting Tool) [8] is a tool for bytecode instrumentation by

providing a set of classes which allows a probe to be inserted at any point in the bytecode. The problem with BIT is that it does not handle exceptions. After the instrumentation, information about the exception handlers is lost and any subsequent execution of the instrumented bytecode can result in runtime errors. In this paper we present a Java Bytecode Analyzer (JaBÁ) to overcome some problems of existing tools and to allow collection of control-flow and data-flow information directly from Java Virtual Machine (JVM) instructions (i.e., Java bytecode).

The rest of this paper is organized as follows. Section 2 gives an overview of some basic concepts and terminology. Section 3 presents a description of our tool, JaBÁ. Conclusions and future directions are in Section 4.

2 Basic Concepts and Terminology

In this section, we present an overview of some basic concepts and terminology required for understanding the rest of the paper.

Let `C` be a class under test and `m` one of its methods (denoted by `C.m`). A control-flow graph (CFG) of `C.m` shows the possible flow of control when `C.m` is executed. Each node in the graph represents an indivisible *block* of code and, each edge, a possible flow of control, i.e, a *decision*, between nodes. A block, also known as a basic block, is a sequence of consecutive statements or expressions containing no transfers of control except at the end, so that if one element of it is executed, all are. As a result, not every line of code (i.e., each statement in `C.m`) needs to be represented by a different node. An edge from node `a` to node `b` implies that `b` can be executed after `a`.

A def-use graph, which represents the flow of data in a method, is an extension of the corresponding CFG of the same method with additional information about variable definitions and uses (*def-use* information). A *def* of a variable represents the definition of this variable.

A def-use graph can be very useful for data-flow based testing criteria [4, 10].

Two other kind of graphs are the interprocedural control-flow graph (ICFG) and the call graph (CG), which are useful to show how each method is related with each other. More details about these kind of graphs can be found in [7, 11].

In Java, each class is encoded into a `class` file containing information on inheritance, fields, methods and other relevant attributes of the class. A Java Virtual Machine (JVM) is used to interpret/execute the Java bytecode. During the execution, the JVM creates a local stack frame for every method invocation.

The `class` file is divided into several parts. In particular, the *constant pool* and the *code* attribute are of much interest to this work. A constant pool has a structure similar to a symbol table. It contains various string constants, class and interface names, field names, and other constants that are referred to within the `class` file [9]. A code attribute contains the JVM instructions and auxiliary information for a single method, instance initialization method, or class/interface initialization method [9].

JVM instructions (bytecodes) can be seen as a typical assembly-like language. Streams of bytecodes can be represented by their mnemonics followed by operand values (if any). A complete description of each JVM instruction can be found in [9].

The JVM starts up by loading and creating an initial class, which is specified in an implementation-dependent manner, by using the bootstrap class loader. The JVM then links the initial class, initializes it, and invokes its public class method `void main(String[])`. The invocation of this method drives all further execution. Execution of the JVM instructions constituting the `main` method may cause linking (and consequently creation) of additional classes and interfaces, as well as invocation of additional methods [9]. Another important characteristic of JVM is that it enables users to define their own class loader. Every user-defined class loader

is an instance of the abstract class `ClassLoader`. Applications employ class loaders in order to extend or replace the way in which the JVM dynamically loads and, thereby, creates classes.

3 The packages in JaBÁ

To support the instrumentation, understanding and testing of Java programs at bytecode level, we have worked on the development of a tool named JaBÁ. Currently JaBÁ has five packages: `lookup`, `graph`, `instrumenter`, `util` and `verifier`. They are useful mainly to deal with bytecode instrumentation, aiming at to collect dynamic control-flow and data-flow information when the bytecode is being executed by the JVM. Moreover, they allow to collect static information, such as, the CFG of each method or the ICFG of a given class. The idea is to evolve this set of packages to provide a complete tool suite for testing Java bytecodes.

Other packages in Java API [5] and BCEL (Bytecode Engineering Library) API [3] are also used to implement JaBÁ. Below each one of these packages are described shortly and some applications are used to illustrate what kind of information can be collected from and inserted into Java bytecodes.

`lookup` package

This package provides useful classes to retrieve information from a given Java class file and other class files which are related to this class. It can also show the inheritance among these classes and categorize them as "system classes" or "non-system classes."

The `lookup.Lookup` provides information of all the classes that have to be loaded in order to execute a given class file. This set of classes includes both system and non-system classes.

`graph` package

This package is used to collect information related to nodes, edges, CFG, ICFG, CG, definitions and uses of variables and live variables at each node, as well as the domination relation [1] between nodes in a CFG. The `graph` package depends on `verifier` and `util` packages to implement its functionality.

`instrumenter` package

This package is used to instrument the bytecode, aiming at collecting information during run time. It provides a easier way to insert probes into Java bytecode.

`util` package

It has one class to help debugging the tool (`Debug`) and another that implements some utility methods concerning the use of the JVM instructions (`InstructCtrl`).

`verifier` package

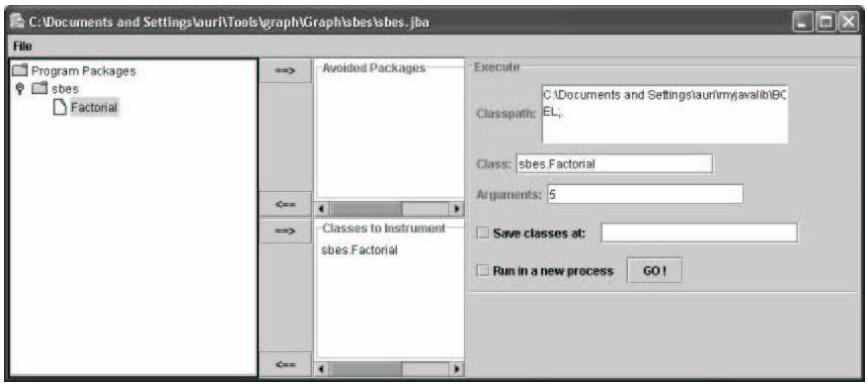
It has a set of class to collect information about a method. In this package the JVM method is read into a graph (a `Graph` object) where each node is a single instruction. On such a graph, data on definition and use of variables, method invocation, etc are collected. The `CFG` class uses this kind of object to analyze the code and then transform it to a block graph.

3.1 Examples of Applications

To show part of the functionalities provided by these packages, consider the graphical interface depicted in Figure 1(a). This GUI integrates all packages described above. First, after the user choose the main `class` file to be tested (in our case, `Factorial.class`), the `lookup` package provide the information shown on the left side of the GUI. From this information the user can select which class will be instrumented during the execution.

By providing the required argument (a positive integer) and clicking on "GO!", our class loader calls the main class file with the respective argument. During the loading process it detects which class(es) should be instrumented and uses `graph` and `instrumenter` packages for both to

find each node should be instrumented and inserting the probes on each node, respectively. As the result, the instrumented bytecode is executed, the normal output is generated ("Factorial of 5 is 120") and the trace information (as illustrated in Figure 1(b)) is saved to be further evaluated.



(a) (b)

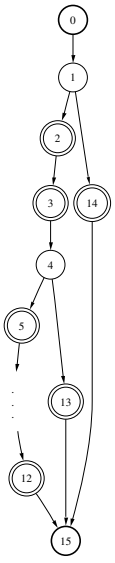
Figure 1: JaBÁ: (a) Simple GUI and (b) Sample of trace file.

```
Method void main(java.lang.String[])
   0 new #2 <Class sbes.Factorial>
   3 dup
   4 invokespecial #3 <Method sbes.Factorial()>
   7 astore_1
   8 aload_0
   9 arraylength
  10 ifle 83
  ...
  55 aload_1
  56 iload_2
  57 invokevirtual #14 <Method long compute(int)>
  60 invokevirtual #15 <Method append(long)>
  63 invokevirtual #16 <Method toString()>
  66 invokevirtual #17 <Method println(String)>
  69 goto 91
  72 getstatic #7 <Field PrintStream out>
  75 ldc #18 <"Error: positive integer required!!!">
  77 invokevirtual #17 <Method println(String)>
  80 goto 91
  83 getstatic #7 <Field PrintStream out>
  86 ldc #18 <"Error: positive integer required!!!">
  88 invokevirtual #17 <Method println(String)>
  91 return
```

(a)

```
$ java graph.ClassSummary -all -i sbes.Factorial
Number of Nodes: 16
  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Number of Decisions: 2
  1 4
...
Node 1
  Start PC:    7: astore_1[76](1)
  End PC:   10: ifle[158](3) -> getstatic 7
  Variable uses L@0  PC: 8
  Variable definitions L@1  PC: 7
  Children 2 14
  Pre-Dominators: 0 1
  Post-Dominators: 1 15
  Alive definitions:   L@0 BK: 0
...
Node 9
  Start PC:   55: aload_1[43](1)
  End PC:   57: invokevirtual[182](3) 14
  Variable uses L@2  PC: 56 L@1  PC: 55
  Children 10
  Call to Lsbes/Factorial;.compute(I)
  Variable uses:  L@2 PC: 56  L@1 PC: 55
  Children: 10
  Pre-Dominators: 2 5 1 3 0 7 4 6 8 9
  Post-Dominators: 12 9 10 11 15
  Alive definitions:   L@2 BK: 4  L@1 BK: 1  L@0 BK: 0
...
```

(b)



(c)

Figure 2: Sample information provided by `graph.CFG`

Although not yet integrated in the GUI, the `graph` package provided more information, useful for understanding and testing of Java bytecodes. For example, considering the bytecode of `Factorial.main` method illustrated in Figure 2(a), Figure 2(b) shows a summary of the total number of nodes and decisions[1] for `Factorial.main` method (16 and 2, respectively) and also detailed information with respect to each node in the CFG. For example, considering node 1 (a

[1]We consider a node to be a decision node if it has more than one child on the CFG.

decision node with children 2 and 14), the set of instructions in this node starts at offset 7 and ends at offset 10. Local variable number one – referred as `L@1` for short – is defined at offset 7 (in the bytecode). The definition of `L@0` at node 0 is alive because there is a definition-clear path with respect to `L@0` from node 0 to node 1, and at offset 8 (node 1) there is a use of `L@0`. Information about pre- and post-dominators, as defined in [1], are also collected. In our example, Nodes 0 and 1 pre-dominate node 1, whereas nodes 1 and 15 post-dominate it. All such information is useful for the the development of control-flow and data-flow based testing adequacy criteria.

Figure 2(c) presents the graphical representation of Figure 2(b). Each method call generates one node in the CFG (call node), represented by double circles. To save space, nodes 6 to 11 (call nodes) are not shown in Figure 2(c). Considering a call node (say, node 9), besides the same information previously mentioned, it is also possible to determine which is the method being called (`compute(I)`) and also the class(es) of the objects(s) used to make the call, in the case of non-static methods. Such information is very useful for building ICFG's.

4 Conclusion

This paper describes a Java Bytecode Analyzer (JaBÁ) which provides a set of Java packages to extract control-flow and data-flow based information directly from Java bytecode. Also provided is the inheritance relationships among all the classes of a given Java application. Information so collected can be very useful for program testing, debugging, comprehension and performance analysis.

For example, some of our on-going projects which require a detailed analysis of JVM instructions can significantly benefit from using a tool such as JaBÁ. One project is to construct a framework for Java mobile agent testing based on techniques for structure testing and program instrumentation. It has two approaches. A "client-based" approach instruments the agent code before it is launched, and for this part the instrumentation can be conducted on either Java source code or bytecode; whereas a "server-based" approach needs to do an instrumentation of code arriving in a given host. Such instrumentation can only be accomplished on bytecode since the corresponding Java source code is not available. Another project is to test component-based software by measuring the code coverage (using a control-flow or data-flow based testing adequacy criterion) of each component before it is integrated into a software system. Since such a component can be a commercial off-the-shelf product or developed by a third party, we once again face the same problem as discussed above, namely, the source code is not always available. As a result, the instrumentation needs to be performed directly on Java bytecode. From these two projects, we can clearly see the importance of developing a tool such as JaBÁ to extract control-flow and and data-flow based information directly from bytecode.

Our next step is to further improve JaBÁ to make the data collection more robust as well as develop a friendly graphical user interface. In addition, more features will be included into JaBÁ to make it a complete visualization and analysis toolsuite for Java applications to help programmers and testers work more effectively and efficiently.

References

[1] H. Agrawal. "Dominators, super blocks, and program coverage". In *Proceedings of the 21st Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'94)*, pages 25–34, Portland, OR, January 1994.

[2] The χSuds Team. "Mining system tests to aid software maintenance". *IEEE Computer*, 31(7):64–73, July 1998.

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

[4] J. R. Horgan and S. A. London. "Data flow coverage and the C language". In *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, pages 87–97, Victoria, Canada, October 1991.

[5] C. S. Horstmann and G. Cornell. *Core Java 2*, volume I - Fundamentals. Prentice Hall, 2001.

[6] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *XXIV International Conference on Software Engineering – ICSE'2002*, pages 467–477, Orlando, FL, May 2002. ACM Press.

[7] W. Landi and B. .G Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

[8] H. B. Lee and B. G. Zorn. BIT: A tool for instrumenting Java btecodes. In *USENIX Symposium on Internet Technologies and Systems (USITS'97)*, pages 73–82, Monterey, CA, December 1997. USENIX Association.

[9] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2 edition, 1999.

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

[11] G. Rothermel, M. J. Harrold, and J. Dedhia. Regression testing for C++ software. *Software Testing, Verification and Reliability*, 10(2):77–109, June 2000.

[12] R. Vallèe-Rai. Soot overview/disassembling classfiles. Soot Web Page, March 2000. Avaliable on-line at: `http://www.sable.mcgill.ca/soot/tutorial/` [04/13/2002].

[13] C. Verbrugge. *Using Coffi*. School of Computer Science – McGill University, Montréal, Québec, Canada, October 1996. Avaliable on-line at: `http://www-acaps.cs.mcgill.ca/~clump/research.html` [04/13/2002].

[14] W. E. Wong, S. S. Gokhale, J. R. Horgan, and K. S. Trivedi. "Locating program features using execution slices". In *Proceedings of the Second IEEE Symposium on Application-Specific Systems and Software Engineering Technology*, pages 194–203, Richardson, TX, March 1999.

[15] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. "Effect of test set size and block coverage on fault detection effectiveness". In *Proceedings of the Fifth IEEE International Symposium on Software Reliability Engineering*, pages 230–238, Monterey, CA, November 1994.