Analyzing Control Flow in Java Bytecode

Jianjun Zhao
Department of Computer Science and Engineering
Fukuoka Institute of Technology
3-10-1 Wajiro-Higashi, Higashi-ku, Fukuoka 811-02, Japan
zhao@cs.fit.ac.jp

Abstract

Understanding control flows in a computer program is essential for many software engineering tasks such as testing, debugging, reverse engineering, and maintenance. In this paper, we present a control flow analysis technique to analyze the control flow in Java bytecode. To perform the analysis, we construct a *control flow graph* (CFG) for Java bytecode not only at the intraprocedural level but also at the interprocedural level. We also discuss some applications of a CFG in a maintenance environment for Java bytecode. some primary experimental results.

Keywords

Control flow analysis, Java virtual machine, software testing, software maintenance.

1 Introduction

Many compiler optimizations and program analysis techniques rely on control flow information, which is topically represented by a *control flow graph* (CFG) [4]. The CFG, although originally proposed for compiler optimizations, has been used for various software engineering tasks such as program slicing, debugging, testing, maintenance, and complexity measurements [7, 8, 9].

Control flow analysis was originally considered for procedural programs. Recently, as object-oriented software become popular, researchers have applied control flow analysis to object-oriented programs [10].

This paper presents a control flow analysis technique for Java bytecode. In Java, programs are being compiled into a portable binary format call *bytecode*. Every class is represented by a single class file containing class related data and bytecode instructions. These files are loaded dynamically into an interpreter, i.e., the Java Virtual Machine (JVM) and executed [6]. Although there are several control flow analysis techniques for binaries on different operating systems and machine architectures [2, 5], the existing control flow analysis techniques can not be applied to Java bytecode due to the specific features of JVM. In order to perform control flow analysis on Java bytecode, we must extend the existing control flow analysis techniques for adapting Java bytecode. To perform analysis, we construct control flow graphs for Java bytecode not only at the intraprocedural level but also at the interprocedural level. The constructed graphs can be used to perform maintenance activities such as structural testing and complexity measurement, and also can be used by other analysis techniques such as data flow analysis and dependence analysis, which are also useful for Java bytecode understanding and maintenance.

The rest of the paper is organized as follows. Section 2 briefly introduces the Java virtual machine. Section 4 considers the control flow analysis of Java bytecode, and presents control flow graph for Java bytecode. Section 6 discusses some applications of the control flow graph. Concluding remarks are given in Section 7.

2 The Java Virtual Machine

The Java Virtual Machine (JVM) is a stack-based virtual machine that has been designed to support the Java programming language [1]. The input of the JVM consists of platform-independent class files. Each class file is binary file that contains information about the fields and methods of one particular class, a constant pool (a kind of symbol-table), as well as the actual bytecodes for each method.

Each JVM instruction consists of a one-byte **opcode** that defines a particular operation, followed by zero or more type **operands** that define the data for the operation. For example, instruction '`sipush 500`' (push constant 500 on the stack) is represented by the bytes 17,1,and 244.

For most JVM opcodes, the number of corresponding operands is fixed, whereas for the other opcodes (`lookupswitch`, `tableswitch`, and `wide`) this number can be easily determined from the bytecode context. Consequently, once the offset into a class file and the length for the bytecode of a certain method have been determined, it is straightforward to parse the bytecode instructions of this method.

At runtime, the JVM fetches an opcode and corresponding operands, executes the corresponding action, and

```
class Test
   int array[];
   int Test() {
     int i = 0, j = 0;
     try {
       while (i < 100) {
          i = i + 1;
          j = j + array[i];
       }
     }
     catch(Exception e) { return 0; }
     finally            { a = null; }
     return j;
   }
}
```

javac

```
0:  iconst_0
1:  istore_1
2:  iconst_0
3:  istore_2
4:  goto [label_20]
-------------------------
7:  iload_1
8:  iconst_1
9:  iadd
10: istore_1
11: iload_2
12: aload_0
13: getfield [Test.array]
-------------------------
16: iload_1
17: iaload
-------------------------
18: iadd
19: istore_2
20: iload_1
21: bipush 100
23: if_icmplt [label_7]
-------------------------
26: goto [label_37]
-------------------------
```

```
29: pop
30: iconst_0
31: istore_3
32: jsr [label_51]
----------------------
35: iload_3
36: ireturn
----------------------
37: jsr [label_51]
----------------------
40: goto [label_60]
----------------------
43: astore temp_4
45: jsr [label_51]
----------------------
48: aload temp_4
50: athrow
----------------------
51: astore temp_5
53: aload_0
54: aconst_null
55: putfield [Test.ar
----------------------
58: ret temp_5
----------------------
60: iload_2
61: ireturn
```

Figure 1: A Java Bytecode program.

then continues with the next instruction. At the JVM-level, operations are performed on the abstract notion of **words** [6] : words have a platform-specific size, but two words can contain values of type `long` and `double`, whereas one word can contain values of all other types.

3 Preliminaries

We give some preliminaries which are necessary for defining a control flow graph from a graphical viewpoint.

Definition 1 *A digraph is an ordered pair* (V, A)*, where* V *is a finite set of elements called* vertices*, and* A *is a finite set of elements of the Cartesian product* $V \times V$*, called* arcs*, i.e.,* $A \subseteq V \times V$ *is a binary relation on* V*. For any arc* $(v_1, v_2) \in A$*,* v_1 *is called the initial vertex of the arc and said to be adjacent to* v_2*, and* v_2 *is called terminal vertex of the arc and said to be adjacent from* v_1*. A predecessor of a vertex* v *is a vertex adjacent to* v*, and a successor of* v *is a vertex adjacent from* v*. The in-degree of vertex* v*, denoted by in-degree(v), is the number of predecessors of* v*, and the out-degree of a vertex* v*, denoted by out-degree(v), is the number of successors of* v*. A simple digraph is a digraph*(V, A) *such that no* $(v, v) \in A$ *for any* $v \in V$*.*

Definition 2 *A path in a digraph* (V, A) *is a sequence of arcs* $(a_1, a_2, \ldots, a_l)$ *such that the terminal vertex of* a_i *is the initial vertex of* a_{i+1} *for* $1 \leq i \leq l-1$*, where* $a_i \in A (1 \leq i \leq l)$ *or* $a_i \in A_1 \cup A_2 \cup \ldots \cup A_{n-1} (1 \leq i \leq l)$*, and* $l (l \geq 1)$ *is called the length of the path. If the initial vertex of* a_1 *is* v_I *and the terminal vertex of* a_l *is* v_T*, then the path is called a path from* v_I *to* v_T*.*

4 Intraprocedural Control Flow Analysis

In this section, we discuss the issues related to control flow analysis for the bytecode of individual Java methods, and describe how to construct the control flow graph for such methods. We also consider the exception-handling issues in this case.

We first give the definition of a basic black.

A *basic block* is a sequence of instructions with a single entry point and single exit point: execution of a basic block can start only at its entry point, and control can leave a basic block only at its exit point. Thus, if control enters a basic block, each instruction in that block will be executed.

4.1 Definition of the Control Flow Graph

Now we give a formal definition of a control flow graph for a Java method.

Definition 3 *A control flow graph (CFG) of a Java method* M *is a 4-tuple* (V, A, s, t)*, where* (V, A) *is a simple digraph such that* V *is a set of vertices which represent basic blocks in* M*, and* $A \subseteq V \times V$ *is a set of arcs which represent possible flow of control between basic blocks in* M*.* $s \in V$ *is a unique vertex, called start vertex which represents the entry point of* M*, such that in-degree(s) = 0, and* t *is a set of vertices, called termination vertex which represents the exit point of* M*, such that out-degree(t) = 0 and* $t \neq s$*, and for any* $v \in V$ *(*$v \neq s$ *and* $v \neq t$*), such that there exists at least one*

path from s to v and at least one path from v to t.

Traditional control flow analysis treats an individual statement of a program as a vertex in the CFG. However, when analyzing Java bytecode, using a basic block as a vertex in the CFG is considered more appropriate as the number of instructions is very large when compared to their high-level language counterpart. As a result, in our CFG for Java bytecode, vertices represent basic blocks, and arcs represent the control of flow between the basic blocks. Moreover, our CFG contains one unique vertex s to represent the entry point of a method, and one unique vertex t to represent the termination of the method.

4.2 Constructing the CFGs

To construct a CFG for the bytecode of a method, we should first divide the instructions of a method into basic blocks at places where the control flow may change, such as after `if` or `goto`, or before instructions that are the target of such.

4.2.1 Determining Basic Blocks

We can use the following rules to determine the basic blocks, that is, by finding the set of leaders.

- The first instruction of the method and each first instruction of every handler for the method are leaders.
- Each instruction that is the target of an unconditional branch (`goto`, `goto_w`, `jsr`, `jsr_w`, and `ret`) is a leader.
- Each instruction that is the target of a conditional branch (`ifeq`, `iflt`, `ifle`, `ifne`, `ifgt`, `ifge`, `ifnull`, `ifnonnull`, `if_icmpeq`, `if_icmpne`, `if_icmplt`, `if_icmpgt`, `if_icmple`, `if_icmpge`, `if_acmpeq`, `if_acmpne`, `lcmp`, `fcmpl`, `fcmpg`, `dcmpl`, `dcmpg`) is a leader.
- Each instruction that is one of the target of a compound conditional branch (`tableswitch` or `lookupswitch`) is a leader.
- Each instruction that immediately follows a conditional or unconditional branch, or a `<T>return` (`ireturn`, `lreturn`, `freturn`, `dreturn`, `areturn`, and `return`), or a compound conditional branch instruction is a leader.

Now, each individual leader give rise to a basic block, consisting of all instructions up to the next leader or the end of the bytecode. Furthermore, we enclose each method invocation (`invokevirtual`, `invokespecial`, `invokestatic`, and `invokeinterface`) in a basic block of its own.

4.2.2 Building the CFGs

Once we divide instructions of a method into basic blocks, we can use the following rules to construct the CFG for the method. Let $u \in V$ and $v \in V$ be two basic blocks:

- An arc (u, v) is added to A if v follows u in the bytecode and u does not terminate in a unconditional branch.
- An arc (u, v) is added to A if the last instruction of u is a conditional or unconditional branch to the first instruction in v.
- An arc id added an arc (u, v) to A from the basic block of each `tableswitch` or `lookupswitch` to the basic block of each instruction that is defined as the target in the switch.
- For each subroutine call, two arcs are added to A: one from the basic block of the `jsr` or `jsr_w` to the basic block of target, and one from the basic block containing the corresponding `ret` to the basic block of the instruction that immediately follows the call.

Example. Figure 1 shows a simple Java class `test` and its corresponding bytecode instructions.

4.3 Exception Handling

In Java, during execution of bytecode, three exceptional situations may arise:

- The JVM throws an instance of a subclass of `VirtualMachineError` in case an internal error or resource limitation prevents further execution.
- An exception is thrown *explicitly* by the instruction `athrow`.
- An exception is thrown *implicitly* by a JVM instruction.

JVM contains 255 bytecode instructions which can be used to implement a Java program at the bytecode level. Among these instructions, there is only one instruction, i.e., `athrow` that may throw an exception explicitly, and there are 40 instructions, i.e., `aaload`, `aastore`, `anewarray`, `arraylength`, `baload`, `bastore`, `caload`, `castore`, `checkcast`, `daload`, `dastore`, `faload`, `fastore`, `getfield`, `getstatic`, `iaload`, `iastore`, `idiv`, `instanceof`, `invokeinterface`, `invokespecial`, `invokestatic`, `invokevirtual`, `irem`, `laload`, `lastore`, `ldc`, `ldc_w`, `ldc2_w`, `ldiv`, `lrem`, `monitorenter`, `monitorexit`, `multianewarray`, `new`, `newarray`, `putfield`, `putstatic`, `saload`, and `sastore`, that may implicitly throw an exception. These

instructions complicates the control flow analysis task as we have to specially consider the exception handler when perform control flow analysis. In the following, we define an extended CFG to represent a Java method which contains exception handler.

Definition 4 *An extended control flow graph (eCFG) of a Java method M is a 4-tuple (V, A, s, T), where (V, A) is a simple digraph such that V is a set of vertices which represent basic blocks in M, and $A \subseteq V \times V$ is a set of arcs which represent possible flow of control between basic blocks in M. $s \in V$ is a unique vertex, called start vertex which represents the entry point of M, such that in-degree(s) = 0, and $T \subset V$ is a set of vertices, called termination vertices which represent the exit points of M, such that for any $t \in T$ out-degree(t) = 0 and $t \neq s$, and for any $v \in V$ ($v \neq s$ and v does not belong to T), $\exists t \in T$ such that there exists at least one path from s to v and at least one path from v to t.*

Please notice that we use a set of terminate vertices T to represent the termination of the execution of a method because the control flow may terminate at multiple program points due to handling exceptions during the execution for the method.

In order to represent exception-handling in the CFG of a method, two aspects should be extended, that are, determining the leaders and construction rules for handling exceptions.

If an exception handler is present, instructions that may throw an exception also end a basic block. So we can use the following rules to determine the basic blocks, that is, by finding the set of leaders related to exception handling.

- Each instruction that immediately follows an instruction that may explicitly throw an exception is a leader.
- Each instruction that immediately follows an instruction that may implicitly throw an exception (ignoring JVM errors) is a leader.

Also, we can use the following rules to construct the CFG for the method that contains exception handling issues. Let $u \in V$ and $v \in V$ be two basic blocks:

- An arc (u, v) is added to A from the basic block of each instruction that may throw an exception to the entry basic block of every exception handler that covers the region of bytecode in which the instruction appears.
- An arc (u, v) is added to A from the basic block of each instruction that may throw an exception to a dummy basic block that represents abnormal completion of the method, i.e., the situation where a thrown exception is not caught by any handler of the method (in the prototype, the types of exceptions are not accounted for during CFG construction, so that some redundant arcs may arise).

After the complete CFG has been constructed, any vertex that can not be reached from the entry vertex s is discarded from the CFG. This may reduce the number of redundant arcs in the CFG.

Example. Figure 2 shows the CFG of the bytecode instructions in Figure 1.

5 Interprocedural Control Flow Analysis

In this section, we discuss the issues related to interprocedural control flow analysis of a partial or whole Java program. We also consider the exception-handling issues in this case.

Definition 5 *An interprocedural control flow graph (ICFG) for a partial or complete Java program is a tuple $(G_1, G_2, ..., G_k, C, R)$, where G_1, G_2, ..., G_k are control flow graphs representing the Java methods in the program, C is a set of call arcs, and R is a set of return arcs. An interprocedural control flow graph for a partial program satisfies the following conditions:*

- *There is a one-to-one onto mapping between C and R. Each call arc is of the form $(u, v_{IG_i}) \in C$ and the corresponding return arc is of the form $(v_{IG_i} \in C, u) \in R$, where $u \in V_{G_i}$ for some $G_i \in g$ and v_{IG_i} and v_{IG_j} are the initial and final vertices, respectively, of some $G_j \in g$.*
- *g contains two distinguished vertices: an initial vertex $v_{IG} = v_{IG_i}$ and a final vertex $v_{FG} = v_{FG_i}$, $G_i \in g$.*

An ICFG for a partial or complete Java program consists of a set of CFGs each representing the control flow of a method in the program, and some call and return arcs linked together. Each call arc is an arc from a vertex representing a method call to the entry vertex of the CFG for the called method. There is a corresponding return arc for each call arc from the final vertex of the called method's CFG back to the vertex representing the method call. For simplicity, we assume that there is a designated final vertex for the ICFG and that there are no unstructured halts within methods.

The difference between an ICFG for a partial program and a complete program is that for a partial program's ICFG, the start vertex may be the entry vertex of any

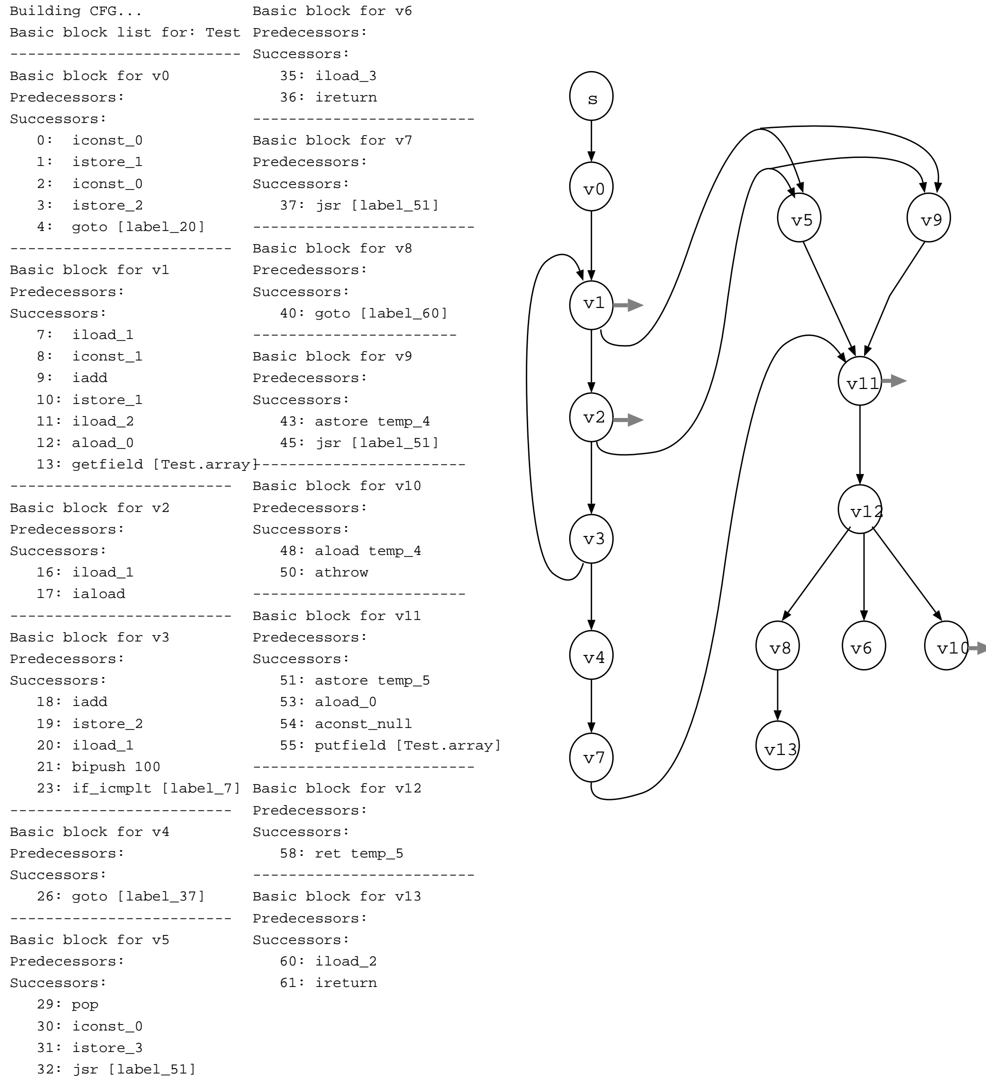

Figure 2: The CFG of the bytecode instructions in Figure 1.

method in the partial program, whereas for a complete program's ICFG, the start vertex must be the start point of the `main()` method of the program. The ICFG for a partial program provides only partial interprocedural control flow information of the complete program; for example, we can get such information by starting to analyze some method of interest in the program in the case that we need not to know the overall information of the program. On the other hand, the ICFG for a complete program provides the overall interprocedural control flow information for the program being analyzed.

6 Applications of the CFG

The CFG and ICFG presented in previous sections can be used in many software engineering tasks for Java bytecode. Here we briefly describe two tasks: structural testing, structural complexity measurement.

6.1 Structural Testing for Java Bytecode

Structural testing techniques use a program's structure to guide the selection of test cases. Since our CFG represents execution paths in Java bytecode, they can be used to define control-flow based coverage criteria, i.e., test data selection rules based on covering execution paths, for testing Java bytecode. Therefore using our representation for structural testing provides an accurate measure of the adequacy of a test suite.

6.2 Structural Metrics for Java Bytecode

Software metrics are useful in software engineering tasks such as program understanding, debugging, testing, analysis, and maintenance, and project management. One could imagine that once some metrics could be proposed for Java bytecode, they should be helpful in

the development of Java bytecode. Because the CFG of Java bytecode represents control flows properties in the program, based on the CFG, we can define some structural metrics for measuring the complexity of Java bytecode from different viewpoints. For example, we can extend the McCabe's control-flow based metrics [7] to the case of Java bytecode.

7 Concluding Remarks

In this paper we presented a control flow analysis technique for Java bytecode. To perform the analysis, we constructed a *control flow graph* (CFG) for Java bytecode not only at the intraprocedural level but also at the interprocedural level. We also discussed some applications of a CFG in a maintenance environment for Java bytecode, and give some primary experimental results. The constructed graphs can be used to perform maintenance activities such as structural testing, and also can be used by other analysis techniques such as data flow analysis and dependence analysis, which are also useful for Java bytecode understanding, testing, debugging, reverse engineering, and reengineering.

REFERENCES

[1] K. Arnold and J. Gosling, "The Java Programming Language," Addison-Wesley, 1996.

[2] C. Cifuentes, "Reverse Compilation Techniques," Ph.D. Thesis, School of Computer Science, Queensland University of Technology, 1994.

[3] C. Cifuentes and A. Fraboulet, "Intraprocedural Static Slicing of Binary Executables," *Proc. International Conference on Software Maintenance*, pp.188-195, October 1997.

[4] J.Ferrante, K.J.Ottenstein, J.D.Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transaction on Programming Language and System*, Vol.9, No.3, pp.319-349, 1987.

[5] J. R. Larus and E. Schnarr, "EEL: Machine-independent Executable Editing," *SIGPLAN Conference on Programming Languages, Design and Implementation*, pp.291-300, June 1995.

[6] T. Lindholm and F. Yellin, "The Java Virtual Machine Specification," Addison-Wesley, 1997.

[7] T. McCabe, "A Complexity Measure," *IEEE Transaction on Software Engineering*, Vol.2, No.4, pp.308-320, 1978.

[8] K. J. Ottenstein and L. M. Ottenstein, "The Program Dependence Graph in a Software Development Environment," *ACM Software Engineering Notes*, Vol.9, No.3, pp.177-184, 1984.

[9] A. Podgurski and L. A. Clarke, "A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance," *IEEE Transaction on Software Engineering*, Vol.16, No.9, pp.965-979, 1990.

[10] S. Sinha and M. J. Harrold, "Analysis of Programs with Exception-Handling Constructs," *Proc. International Conference on Software Maintenance*, October 1998.

[11] M. Weiser, "Program Slicing," *IEEE Transaction on Software Engineering*, Vol.10, No.4, pp.352-357, 1984.