RESIDUAL COVERAGE MONITORING OF JAVA PROGRAMS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Christina Pavlopoulou

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

August 1997

ACKNOWLEDGMENTS

I would like to thank prof. Michal Young for his valuable help and guidance during the building of the system and the writing of this report.

DISCARD THIS PAGE

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Pavlopoulou, Christina. MS, Purdue University, August 1997. Residual Coverage Monitoring of Java Programs. Major Professor: Michal Young.

It is common for a product to be released without 100% coverage, since exhaustive testing is impossible. The assumptions made for the remaining test obligations (residue) is that it is either infeasible or occurs rarely. The purpose of residual testing it to provide a user-adjustable monitoring of deployed software, and thereby to provide feedback that can be used to help validation and refinement of quality assurance activities by developers. This dissertation focuses on an important step towards the above goal: residual coverage monitoring. Instrumentation is inserted in the user program, so that its output indicates whether any test obligations not covered in previous executions are covered during the current use.

1. INTRODUCTION

Quality assurance activities in the development environment, including systematic dynamic testing, cannot be performed exhaustively, therefore they always depend on models. Static analysis depends on the fidelity of models extracted for analysis. Statistical testing for reliability estimation depends on models of program usage. Partition testing depends on the models used to divide program behaviors into classes that should be "covered." Discrepancies between these models and actual program behavior are valuable information, even when they don't result in observed program failures, because they indicate how quality assurance activities in the development environment can be improved. For example, having some way of judging when "enough" testing has been done can be valuable in a negative sense. Test adequacy criteria indicate, not when testing is definitely adequate, but when there is evidence that a set of tests is inadequate because some significant class of program behaviors has never been tested.

The family of structural coverage criteria (statement coverage, branch coverage, dataflow coverage, etc.) are based on syntactic models of program control and data flow. These syntactic models are conservative in the sense that they include not only all control and data flows that will occur in any execution, but also many infeasible paths that can never occur. It is (provably) impossible to determine exactly which paths are infeasible. Thus even exhaustive testing would often fail to satisfy structural coverage criteria. When a software product is released without 100% coverage, testers are explicitly or implicitly assuming that the remaining test obligations (the residue) is either infeasible, or occurs in a vanishingly small set of possible executions.

In critical software systems such as avionics, these assumptions may be explicit. For example, developers or testers may be asked to explain, for each block of code which has not been executed under test, why such execution is infeasible (e.g., because it is a handler for an error that should never occur). In applications with less stringent reliability requirements, the assumptions may be implicit. For example, it is not uncommon to set a target of less than 100% coverage. A target of, for example, 90% satisfaction of a test coverage criterion is implicitly an assumption that the remaining 10% of test obligations are either infeasible or so rarely executed that they have negligible impact on quality.

We cannot completely avoid models and assumptions. What we can do is validate them. If we have implicitly or explicitly assumed that a particular path or region in code is never, or almost never executed, then knowing that an execution of that path or region has occurred in the deployed use is valuable information, even if the software performed correctly in that case. However, in current practice this is not possible since there is a sharp divide between unit, integration, and system testing on the one hand, and feedback from deployed software on the other. While developers have access to a variety of monitoring tools in the development environment, monitoring in the deployed environment is typically limited to error and sanity checks, and the channel from users back to developers is just a list of trouble reports.

With ubiquitous networking this is no longer necessary. The internet provides a new opportunity for extending monitoring capabilities from the development environment into the fielded environment.

1.1 Residual Testing

The goal is to provide deployed software with monitoring of adjustable level and focus to address varying performance requirements with user control to address concerns of security, and confidentiality. Feedback of the monitored software will help validation and refinement of models and assumptions used in quality assurance activities based on actual usage.

An important step towards the above goal is very low-cost monitoring of the "residue" of coverage testing, i.e., checking for test obligations which were not fulfilled in the development environment but correspond to execution behaviors that occur in actual use. Residual coverage monitoring should use the output of permanent instrumentation of the developed programs to indicate whether any test obligations not covered in the development phase are covered during the current use. This could occur, for example, if an execution path that has been presumed infeasible by developers is in fact feasible.

1.2 Related Work

Residual monitoring coverage is a new idea and therefore has different goals and complications from other software verification and validation techniques. The most closely related area is assertion-based testing which also uses the run-time information of instrumented programs to decide whether they conform to the desired behavior as described with a specification language. Systems implementing this technique include Anna [LvH85], TSL [Ros91], APP [Ros95], Nana, [Mak], ADL [SH94].

Anna (ANNotated Ada) is a language extension to Ada to include facilities for formal specification of intended program behavior. The user inserts annotations in the source program which are just comments for Ada but obey the syntactic and semantic rules of Anna. Anna can be used in many stages of software development among which is construction of self-checking production-quality software ([SL86]). In this case,the annotations are transformed into runtime checks that can be left permanently in the user program. A transformation tool produces a program that is self-checking and reports inconsistencies during program operation between the actual Ada code and the Anna specifications. APP, Nana, and ADL are essentially reimplementations of parts of ANNA functionality for the C programming language.

Similarly, TSL (Task Sequencing Language) ([Ros91]) is a language that supports writing specifications for concurrent Ada systems at a high level of abstraction. The TSL compiler transforms these specifications into Ada code and the TSL runtime system automatically checks an Ada tasking program's execution behavior for consistency with its TSL specifications by comparing the events generated by an execution program with those expressed in the specifications.

Furthermore, Bytecode Instrumenting Tool (BIT) ([Lee97a], [Lee97b]) is a tool for instrumenting Java bytecodes which was developed independently of this system. However, since it is more closely related to the implementation issues and not the concept of this work more details about it will be referred in a later chapter.

1.3 Current Work

The primary question addressed by the current work is whether residual monitoring can have sufficiently low impact on execution characteristics, particularly efficiency and responsiveness, to be acceptable to users. I have constructed a proof-of-concept prototype system, which selectively monitors execution of Java programs for basic blocks. Initially, all basic blocks[1] are monitored, but subsequent to a few test runs the program can be instrumented again, removing monitoring of basic blocks that have already been covered and leaving only the probes needed to recognize execution of the "residue" of unexecuted code. Since the high-frequency program paths tend to be executed on almost every program run, the cost of selective reinstrumentation quickly decreases and for the programs I have examined it approaches zero.

Figure 1.1 illustrates the whole process. Initially, instrumentation is inserted to all the basic blocks in the Java class files. The modified class files are executed through the Java interpreter and as a side effect the instrumentation creates a file containing the basic blocks executed. Next, this file is used to determine the set of basic blocks that were not executed previously and will be instrumented in the next running of the instrumenter. Every time the process is repeated, the input is the set of basic blocks the were not executed in previous executions. The basic blocks that have been executed can be viewed using a graphical user environment like Emacs.

[1] The precise meaning of a basic block is explained in chapter 2.

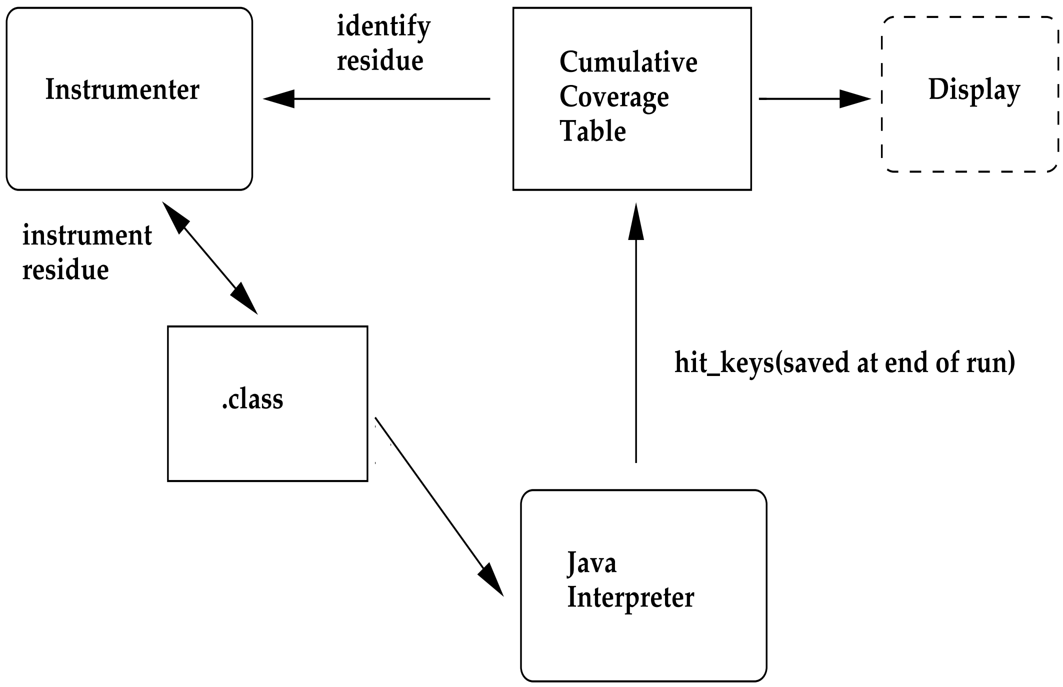

Figure 1.1 Instrumentation process: in every iteration the basic blocks that were not covered in previous executions are instrumented and the new class file is executed in order to collect coverage information.

2. RESIDUAL TESTING WITH SELECTIVE MONITORING

For a first proof of concept we limit the problem to selective monitoring of block coverage, i.e. which basic blocks of the user application have been executed under a test case. A *basic block* [SU86] is a sequence of instructions in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. First, we will describe the main approach independently of the language used and then discuss in more detail an implementation on the Java Virtual Machine.

2.1 Basic Design

Our design must accommodate the following:

- instrumentation of basic blocks that have not been executed in previous runs (or all the basic blocks upon request of the user)
- association of basic blocks with the corresponding line numbers of the source file so that coverage results can be viewed
- simple run-time processing so that the overhead of the instrumentation is as small as possible

We divide the design into compile-time processing and run-time processing. During the compile-time processing the instrumentation in the user code is inserted in each basic block that has not been previously executed. During the run-time processing the user program is executed under test. After the run-time processing, information about which basic blocks were covered during this execution need to be gathered.

2.1.1 Compile-time structures

Several compile-time structures are needed to keep track of what to instrument each time:

- A table (*IdTable*) which associates a unique identifier (*block id*) with each basic block. The IdTable should be stable in the sense that, if the same program is compiled twice without changes, the same unique identifiers are associated with each basic block.
- A table (*CoverageTable*) consisting of the basic block id's that have been covered in previous executions. This table does not maintain any information other than presence or absence of a basic block id. It can be a table of booleans with `CoverageTable[i]` set to true when the basic block with block id `i` has been executed. The testing tool reads this table to determine whether to insert monitoring code for a particular basic block.
- A table (*CorrespondenceTable*) that associates integers (*hit keys*) with block ids. This table may change with every compilation, because it associates integers only with those block id's that are not in the coverage table. The purpose of this table is to provide a simple, efficient key for the run-time table.

For example, for a program with 6 basic blocks the above compile-time structures might have the contents illustrated in Figure 2.1.

2.1.2 Run-time structures

During the run-time processing we need one table *HitTable* indexed by the hit keys, that will contain information about what has been executed. It can be just an array of booleans, initially all false. `HitTable[i]` will be set to true when the i-th basic block has been executed. Furthermore, we need a procedure that will initialize the above array at the beginning of the user application and a procedure that will dump the array at the end of the user application.

IdTable

block	block_id
block 0	0
block 1	1
block 2	2
block 3	3
block 4	4
block 5	5

Coverage Table

block_id	covered
0	true
1	false
2	true
3	true
4	false
5	true

CorrespondenceTable

block_id	hit_key
1	0
4	1

Figure 2.1 Compile-time structures: *IdTable* assigns a unique integer to every basic block in the program, *CoverageTable* keeps track of which basic blocks have been executed so far, and *CorrespondenceTable* assigns new unique integers to the basic blocks that have not yet been covered.

Table Name	Associates
IdTable	basic block $\rightarrow$ block id
CoverageTable	block id $\rightarrow$ {true, false}
CorrespondenceTable	block id $\rightarrow$ hitkey
HitTable	hit key $\rightarrow$ {true, false}

Table 2.1 Tables for maintaining the information needed for selective monitoring

The relation among the various tables appears in Table 2.1

2.1.3 Processing

During the compile-time processing, the IdTable for the basic blocks is constructed. The CoverageTable is loaded and for all its false entries (i.e. basic blocks not previously covered), an entry in the CorrespondenceTable is created and a unique integer (hit key) is associated to it. For each of the basic blocks that have been assigned hit key `i`, code is inserted that sets `HitTable[i]` to true. At the beginning of the user program code to initialize HitTable and code to dump it is inserted.

When the user executes his or her application the HitTable is modified as a side-effect of running the program. When the program terminates and the HitTable is dumped to a file, we will have information about which basic blocks were covered during this particular execution.

What remains to be done is the processing of the dumped file. The post-processor reads the dumped table, the IdTable, the CorrespondenceTable and the CoverageTable. Each hit key is translated to a basic block id through the CorrespondenceTable. If the block id is not already present in the coverage table, it is added, i.e. the corresponding entry is set to true. The modified CorrespondenceTable is then written back to the disk, to be used in the next compilation. During every post processing phase the HitTable needs to be processed, since it contains the basic blocks covered in the last execution.

Furthermore, using line information that associates the basic blocks with line numbers in the source code, the user source code listing is marked using colors in a visual display.

2.2 Implementation Issues for Java

We implemented the above ideas for Java. The first approach tried was to modify the front-end of the Java compiler. The idea was to insert the necessary instrumentation in the abstract syntax tree and then let the compiler generate the code. This was not overly difficult but would not be successful regarding the clean separation of the instrumentation processing from other aspects of the compiler. Any changes to the source code of the compiler or the language itself would require corresponding substantial programming effort to update the test instrumentation tool.

The idea for directly instrumenting the byte code representation of the Java program (the .class file) appeared more attractive for several reasons. Assembly language has a small and simple repertoire of instructions and its structures are much simpler than those of the original language. The Java Virtual Machine is well specified and changes to it can have a small impact on our tool. A compiler (AppletMagic by Intermetrics) that translates Ada to Java byte codes has appeared, so with small changes and little effort we could have the same tool for Ada as well. The programming effort and complication could be reduced by using source code by Sun to load classes. Furthermore, debugging information can be used for relating the assembly code with the source file.

The main steps that we follow to selectively monitor class files are:

- Instrumentation Phase

  The user class files are instrumented.

  - Load the class files.
  - Construct or read the tables that keep track of the basic blocks previously executed.

  - Identify basic blocks.
  - For each basic block, insert monitoring code.

- Post-processing phase

  Information from the execution of instrumented class files is collected.

  - Load the class files.
  - Read the tables that keep track of the basic blocks executed.
  - Update these arrays.
  - Output line information.

2.3 Class File Format

This section to describes briefly those aspects of the class file format that are critical for understanding the main implementation issues. For more details the reader can refer to [LY96].

A class file, which is produced after compiling a Java source file, can be viewed as a Java class (or C structure) containing the following fields:

```
BinaryClass {
  int magic;
  short minorVersion;
  short majorVersion;
  short constantPoolCount;
  cpInfo constantPool[constantPoolCount-1];
  short accessFlags;
  short thisClass;
  short superClass;
  short interfacesCount;
  short interfaces[interfacesCount];
```

```
  short fieldsCount;
  fieldInfo fields[fieldsCount];
  short methodsCount;
  methodInfo methods[methodsCount];
  short attributesCount;
  attributeInfo attributes[attributesCount];
}
```

where int is represented by 4 bytes and short by 2 bytes.

`constantPool` is a table of variable length structures representing various string constants, class names, field names, and other constants that are referred to within the `BinaryClass` structure and its substructures. Each entry is of the form:

```
cpInfo {
  byte tag;
  byte info[];
}
```

The format and length of the array `info[]` is specified from the value of the `tag` field. Currently, there are tags for constant arithmetic values (integer, float, long, double), as well as for constant strings, names of classes, interfaces, fields, methods and signatures.

`methods` is a table of variable length `methodInfo` structures, which provide a complete description of the methods explicitly declared in this class. In the case of interfaces, there is only the interface initialization method. The format of the `methodInfo` structure is the following:

```
methodInfo {
  short accessFlags;
  short nameIndex;
  short descriptorIndex;
  short attributesCount;
```

```
  attributeInfo attributes[attributesCount];
}
```

`attributes` is a table of variable length `attributeInfo` structures used to provide information about the various structures and substructures of a class file. There are several attributes and all have the following general format:

```
attributeInfo {
  short attributeNameIndex;
  int attributeLength;
  byte info[attributeLength];
}
```

`attributeNameIndex` points to an entry of the constant pool which is a constant string representing the name of the attribute. There are several predefined attributes in JDK (Java Development Kit) release 1.1, and some of them are important for the correct loading of a class file by the interpreter. Any additional attributes are not allowed to influence the semantics of the Virtual Machine. In the case of the `methodInfo` structure, the predefined attribute that is used for the implementation of the tool is the `Code` attribute.

The `attributeInfo` representing the `Code` attribute has the following format:

```
codeAttribute {
  short attributeNameIndex;
  int attributeLength;
  short maxStack;
  short maxLocals;
  int codeLength;
  byte code[codeLength];
  short exceptionTableLength;
  {  short startPC;
     short endPC;
```

```
    short handlerPC;
    short catchType;
  } exceptionTable[excpetionTableLength];
  short attributesCount;
  attributeInfo attributes[attributesCound];
}
```

`code` is the array of the actual Virtual Machine byte codes that implement this method. The instrumentation process modifies this array. The `exceptionTable` consists of entries describing an exception handler in the code array. Each exception handler is described by four values: `startPC` and `endPC` indicate the ranges in the code array at which the exception handler is active, `handlerPC` indicates the start of the exception handler and `catchType` is an index to the constant pool representing the name of the class of exceptions. `attributes` is an array of `attributeInfo`, from which the `LineNumberTable` Attribute is used. The latter has the following format:

```
lineNumberTableAttribute {
  short attributeNameIndex;
  int attributeLength;
  short lineNumberTableLength;
  { short startPC;
    short lineNumber;
  } lineNumbertable[lineNumberTableLength];
}
```

Each entry in the `lineNumberTable` contains the items `startPC` which indicates the index into the `code` array at which the code for a new line in the original Java source file begins, and `lineNumber` which gives the corresponding line number in the original Java source file. The `LineNumberTable` attribute is present when debugging information is included, as when Sun's Java compiler is invoked with the -g option.

This information is used by the testing tool to provide the user with information about which lines of the source code have been executed.

2.4 Identifying Basic Blocks

Partitioning the byte codes generated for a Java method into basic blocks is straightforward. The main idea is sketched below and is adapted from [SU86]:

1. Scan the byte code array to determine which byte codes can be the *leaders*, i.e. the first byte codes of the basic blocks. The rules for distinguishing leaders are:
   (a) The first bytecode is a leader.
   (b) Any bytecode that is a target of a control transfer instruction or the target of the `lookupswitch` or `tableswitch` instructions is a leader.
   (c) Any instruction that immediately follows a control transfer instruction is a leader.
2. For each leader, each basic block consists of the leader and all instructions up to but not including the next leader.

For example consider the following program fragment (the numbers at the left are the line numbers at which the method can be found in the source file):

```
14    void BubbleSort() {
15      for (int i = a.length; --i>=0; )
16        for (int j = 0; j<i; j++) {
17          if (a[j] > a[j+1]) {
18            swap(j, j + 1);
19          }
20        }
21    }
```

Its translation to byte codes will be the following:

```
Method void BubbleSort()
   0 aload_0
   1 getfield #16 <Field sorting.a [I>
   4 arraylength
   5 istore_1
   6 goto 47
   9 iconst_0
  10 istore_2
  11 goto 42
  14 aload_0
  15 getfield #16 <Field sorting.a [I>
  18 iload_2
  19 iaload
  20 aload_0
  21 getfield #16 <Field sorting.a [I>
  24 iload_2
  25 iconst_1
  26 iadd
  27 iaload
  28 if_icmple 39
  31 aload_0
  32 iload_2
  33 iload_2
  34 iconst_1
  35 iadd
  36 invokevirtual #32 <Method sorting.swap(II)V>
  39 iinc 2 1
  42 iload_2
  43 iload_1
```

```
44 if_icmplt 14
47 iinc 1 -1
50 iload_1
51 ifge 9
54 return
```

The basic blocks identified by the previous procedure are:

block	start byte code index	end byte code index
0	0	8
1	9	13
2	14	30
3	31	38
4	39	41
5	42	46
6	47	53
7	54	55

2.5 Instrumentation

The instrumentation statements must be inserted in such a way that the the Java Virtual Machine Specification is not violated, since before executing any class file the Java interpreter will ensure that the specific class file is legal, that is:

1. it conforms to the format dictated by the Java Virtual Machine Specification
2. the interpreter will be able to execute it, i.e. instructions have the correct format, the appropriate arguments are on the top of the stack when needed, etc.

The instrumentation statements are calls to methods of the Java class *Monitor*, whose methods and fields are listed below:

```
public class Monitor {
  static boolean hitTable[];
  public static void init(int length);
  public static void dump();
  public static void hit(int index);
}
```

The method `init` initializes the array `hitTable` and the method `dump` dumps the array in a file after the execution of the user program. `hit` is invoked every time a basic block is executed.

For each basic block of each method of each class file which has not been previously executed, we find its hit_id and before its corresponding byte codes we insert the necessary code for the call `Monitor.hit(hit_id)`. The translation to byte codes of the above statement depends on the value of hit_id. If it is less than 256, then we can use the following code:

```
bipush hit_id
invokestatic #index
```

`bipush` puts the value hit_id on top of the stack, index points to the entry of `Monitor.hit(hit_id)` in the constant pool. If hit_id is greater than or equal to 256, then the following assembly instructions can be used:

```
ldc #hit_index
invokestatic #index
```

Here `ldc` puts on top of the stack the integer constant found in the entry of the constant pool pointed to by hit_index.

After inserting all the necessary calls to `Monitor.hit(int index)` we must insert calls to `Monitor.init(length)` and `Monitor.dump()`. This is done in every method:

```
public static void main(String argv[])
```

since the execution of a Java program starts from and terminates to the method with the above header of the class that serves as the main class. (In particular, every class is allowed to have one main method but since it is not known in advance from which class of the program the user wants to start the execution, we instrument the main methods of all the classes.) The call to `Monitor.init(length)` is inserted at the beginning of the method and the call to `dump` before each `return`.

The constant pool contains only the constants used by the specific class and since the `Monitor` class is not known to the user there are no entries in the constant pool for its methods. Therefore, we need to modify the constant pool of each class file suitably.

Finally, we must increment the size of the stack of each of the methods by 4 in order to support the execution of the added byte codes.

However, adding code changes the address of instructions, and thus we need to update the target addresses of the control transfer instructions, the instructions `lookupswitch` and `tableswitch`, as well as the exception table.

For this reason we construct a table, that for each byte code it associates its old address with its new address. After the monitoring code has been inserted, the byte codes are examined and all the target addresses are updated. The new target address should not simply point to the same instruction as previously. In case that instrumentation has been inserted at the beginning of the basic block the target address should point at the beginning of the instrumentation. Moreover, the `lookupswitch` and `tableswitch` instructions require some 0-bytes to ensure alignment of data. Since the address of these instructions change, the number of 0-bytes need to be inserted may have to change.

For example the instrumented method from the previous section will be:

```
Method void BubbleSort()
   0 bipush 1
   2 invokestatic #134 <Method test.Monitor.hit(I)V>
   5 aload_0
   6 getfield #16 <Field sorting.a [I>
   9 arraylength
  10 istore_1
  11 goto 77
  14 bipush 2
  16 invokestatic #134 <Method test.Monitor.hit(I)V>
```

```
19 iconst_0
20 istore_2
21 goto 67
24 bipush 3
26 invokestatic #134 <Method test.Monitor.hit(I)V>
29 aload_0
30 getfield #16 <Field sorting.a [I>
33 iload_2
34 iaload
35 aload_0
36 getfield #16 <Field sorting.a [I>
39 iload_2
40 iconst_1
41 iadd
42 iaload
43 if_icmple 59
46 bipush 4
48 invokestatic #134 <Method test.Monitor.hit(I)V>
51 aload_0
52 iload_2
53 iload_2
54 iconst_1
55 iadd
56 invokevirtual #32 <Method sorting.swap(II)V>
59 bipush 5
61 invokestatic #134 <Method test.Monitor.hit(I)V>
64 iinc 2 1
67 bipush 6
69 invokestatic #134 <Method test.Monitor.hit(I)V>
```

```
72 iload_2
73 iload_1
74 if_icmplt 24
77 bipush 7
79 invokestatic #134 <Method test.Monitor.hit(I)V>
82 iinc 1 -1
85 iload_1
86 ifge 14
89 bipush 8
91 invokestatic #134 <Method test.Monitor.hit(I)V>
94 return
```

Notice the insertion of calls to `Monitor.hit(hit_key)` as well as the change of the target addresses of the control transfer instructions.

2.6 Identifying Line Information

In order to be able to produce some visible and comprehensive output we must find the correspondence between the basic blocks executed and the source code. For that we use the `LineNumberTable` attribute of each class file. As mentioned earlier, the line information for a method is contained in an array of the form:

index	start byte code index	line number
0	s_0	ℓ_0
1	s_1	ℓ_1
2	s_2	ℓ_2
$\vdots$	$\vdots$	$\vdots$

The s_i values appear in an increasing order and only once, whereas the l_i values may appear more than once and generally not in sorted order. Note that a line number points to the first of the byte codes to which this line has been translated to.

Suppose that for a basic block b_i which starts from byte code index s_i and ends at byte code index e_i , we want to find the line numbers of the source file to which it corresponds. Assuming that the class file is unoptimized, the bytecodes of a basic block are contained in consecutive lines. The way to proceed is the following:

1. Find the line number that corresponds to an entry j such that $s_i \geq s_j$.

2. Find the line number that corresponds to an entry k such that $e_i \leq s_j$.

   The reason for the above inequalities comes from the way the line number information is generated: not all byte codes to which a method is translated to are contained in the line number table.

3. The line numbers to which the basic block b_i corresponds to are all l_i for $j \leq i \leq k$.

For example, the line information generated for the `BubbleSort` method (stated earlier) is:

```
Line numbers for method void BubbleSort()
   line 15: 0
   line 16: 9
   line 17: 14
   line 18: 31
   line 16: 39
   line 15: 47
   line 14: 54
```

Using the above idea, line 15 corresponds to basic block 0, line 16 to basic block 1, etc.

2.7 Related Work

As mentioned earlier, BIT is also a system that instruments Java bytecodes and it was developed independently and contemporaneously with the residual testing tool.

The purpose of BIT is to provide a general, easy to use tool to insert method calls to the user class files, whereas the residual testing tool inserts specific method calls to the user program and at the same time provides data structures to maintain runtime information. Because of the diverse goals, the systems present many other differences. BIT allows the user to specify the instrumentation statements, whereas it does not provide the capability of removing monitoring code automatically and of using the debugging information to relate source code and bytecodes. Moreover, the addition of statements leads to changes in the instruction addresses and affects exception handling code, something that is not taken care of by BIT.

3. EXPERIENCE

The experiments were conducted in a SPARC 5 processor at 70 MHz, using four Java applications: ArcTest, Sorting, Elevator, and the instrumentation program itself.

The general observation is that the execution of a fully instrumented program may have significant overhead but after successive reinstrumentations the overhead reduces dramatically. The additional execution time required for the instrumentation statements of a program depends mainly on the size and number of loops as well as on the size of the input data.

The execution times for the various applications are contained in the tables that follow. In each table, the first row contains the execution times of the uninstrumented application for different data inputs. The execution times regarding the instrumented program (second row in the tables) were measured as follows: initially instrumentation is inserted in every basic block and the program is executed with the first test case (time in first column); then instrumentation is inserted in those basic blocks that did not execute previously and the program is run with the second test case (time in second column); the process is repeated for all test cases. Finally the last row contains the number of basic blocks that were instrumented for each test case.

The first application ArcTest is a simple program that draws on the screen a number of arcs with random beginning and end. In this case the overhead from the instrumentation is small, because the time to execute a monitoring statement is small compared to the time to execute the original code.

The Sorting program sorts an array of randomly generated numbers using either binary or quick sort. The first test case is sorting numbers with binary sort, the second test case is sorting numbers with quicksort and the third is sorting numbers with quicksort again. In the first two tests the instrumented code needs more than

	test 1
original	4.5
instrumented	5.0
blocks	29

Table 3.1 ArcTest execution times in sec

	test 1	test2	test3
original	24	2	2
instrumented	55	4.5	2
blocks	43	25	6

Table 3.2 Sorting execution times in sec

double than the time that the original needs, since these cases force the execution of different parts of the program. However, the third test case is executed in the same time with the original program, as no instrumentation code is executed.

Elevator is a simulation program for the operation of two elevators that can go up or down a number of floors. In this case the overhead is not significant since loops containing monitoring code are not executed many times.

	test 1	test 2	test 3
original	5.5	6.1	6.5
instrumented	6.3	6.8	6.6
blocks	323	240	119

Table 3.3 Elevator execution times in sec

	test 1	test 2	test 3
original	4.3	1.7	4.4
instrumented	4.7	1.8	4.4
blocks	1000	614	547

Table 3.4 Instrumentation program execution times in sec

Finally, the instrumentation system itself has been instrumented and has been used to instrument the program Sorting. Again the execution time of the instrumented version is not much more than the execution time of the original version.

4. EXTENSIONS AND FUTURE WORK

Point-based coverage, which has been investigated so far, gives us information only about which basic blocks have been executed.

However, one usual question regarding testing is which definition-use pairs have been covered. The answer to that involves path-oriented testing, which provides information about the paths executed at each run. However, some test coverage criteria involve sub-paths rather than individual points. The best known of these is data flow coverage testing, in which execution of particular "definition use" pairs (what compiler writers know as "reaching definitions") is monitored. The interested reader may refer to [RW85] for definitions and an in-depth discussion of data flow testing.

We consider again whether the run-time performance impact of test coverage monitoring can be made insignificant. Whereas in point-based monitoring it is easy to reduce the instrumentation overhead, since once a basic block has been executed we can remove the monitoring code, in path-based monitoring it is not clear how and when the overhead can be reduced.

In the worst case we might have code like the following code:

```
if (cond1)
   x = ...; // 1
else
   y = ...; // 2

if (cond2)
   z = x;   // 3
else
```

```
z = y;    // 4
```

with the assumptions that

- the code occurs in a high frequency loop,
- in each of the two if statements, the "then" branch is taken 50% of the time and the "else" branch is taken 50% of time,
- when the "then" branch is taken in the first "if" statement, the "else" branch is always taken in the second, and when the "else" branch is taken in the first "if" statement, the "then" branch is always taken in the second.

We observe that the definition-use pairs (1,3) and (2,4) are never executed, even though every point in the path is executed 50% of the time. In this case, unless we transform the code, we cannot avoid monitoring at a point that is executing so often, unlike the case of point-oriented monitoring. The hope is that such cases do not occur very often and thus the instrumentation of every point in such paths does not lead in significant overhead. However, experimental evidence is needed to determine the frequency with which such cases occur in real programs.

A simple heuristic is to use point-oriented monitoring wherever it is sufficient. For example, if a definition pre-dominates [1] a reference and all the paths from the definition to the reference are definition-free, then it suffices to instrument only the reference. Similarly, one can instrument only the definition if the reference post-dominates the definition and all the paths between them are definition-free. Again experimentation is needed to determine the usefulness of the approach.

Another more complicated approach can be based on the path profiling algorithm described by Ball and Larus [BL96]. Essentially, the algorithm identifies distinct paths with distinct states and encodes them as integers from 0 to $n-1$. Minimal

[1]In a rooted, directed graph G , node p pre-dominates node q if every path from the root of G to q passes through p. If the graph has a set of "end" nodes e1, e2, etc. (e.g., the exit nodes from a procedure flow graph), then node q is said to post-dominate node p if every path from p to an exit node passes through q.

instrumentation is then placed in the user program so that transitions are computed by simple arithmetic operations.

More specifically, first the simple case of a directed acyclic graph (DAG) with a unique source vertex $ENTRY$ and a sink exit $EXIT$ is considered. This DAG is obtained from the control flow graph (CFG) of a program after removal of the loop backedges. The main steps followed are:

- A non-negative constant value is assigned to each edge in the DAG such that no two paths compute the same path sum.
- Given an edge value assignment, a minimal cost set of edges is found along which these values will be computed. This minimal set is the set of edges that do not belong in a maximal cost spanning tree of the graph. The weighting of the graph can be specified by the frequencies of execution of various edges.
- The edges found in the previous stage are instrumented appropriately. The instrumentation code contains additions to find the index of the path currently being executed.
- After collecting the run time information, the executed paths are derived.

In case of CFGs that include loop backedges, the following can be done:

- For each vertex v that is the target of one or more backedges, we add a dummy edge $ENTRY \rightarrow v$. For each vertex w that is the source of one (or more) backedges, we add a dummy edge $w \rightarrow EXIT$.
- We eliminate the backedges from the graph and the steps of the algorithm described earlier are applied.

The good performance of the above algorithm depends on how much the cost of the instrumentation code can be reduced. If we have access to many registers, then the additions needed to find the index of the path become cheap. This is not however the case with the Java Virtual Machine, since it is a stack-oriented machine where we cannot control the registers.

Cheap path-oriented monitoring is harder than the corresponding point-oriented problem. Experiments to give an idea about the frequency of the different cases mentioned above would help decide about a suitable course of action. Furthermore, combination of simple heuristics as the one described above that exploit specific features might lead to small monitoring overhead.

5. CONCLUSIONS

A proof of concept system that implements residual testing has been presented. The system inserts instrumentation code in each basic block of the user program that has not been executed in previous runs. When the user program is executed under some test cases, information about which monitored basic blocks have been covered is produced.

The instrumentation takes place in Java class files. The simple repertoire of the instructions and the platform independence make JVM a reasonable choice for the implementation of the system. Although JVM performs various checks prior to the execution of a class file, it is possible to insert code directly into a class file without violating any of the JVM specifications.

The main issue addressed in this thesis is the run-time impact of residual testing, since users would be unlikely to sacrifice much performance to provide more information to testers. The results however seem encouraging for point-based monitoring. Although the instrumentation overhead initially can be large especially for programs with frequently executed loops, after a few reinstrumentations it decreases fast and approaches zero for the experiment programs.

Interesting but more difficult is the problem of path-based monitoring. No solution that guarantees small overhead seems obvious, since there are cases where certain paths are never executed, whereas every point of them is executed frequently. Experiments must be conducted to see how often such cases occur in programs. If these are rare, then with the help of simple heuristics path-oriented residual testing might also lead to good results.

In general, with the widespread use of networking facilities and internet, residual testing might prove a useful way of testing software with the help of users.

BIBLIOGRAPHY

BIBLIOGRAPHY

[BL96] T. Ball and J. Larus. Efficient path profiling. In *Proc. MICRO-29*. IEEE, 1996.

[Lee97a] H. Lee. Bit: A tool for instrumenting java bytecodes. In *Proc. USITS*, 1997.

[Lee97b] H. Lee. Bit: Bytecode instrumenting tool. Master's thesis, University of Colorado, 1997.

[LvH85] D. Luckham and F. von Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2(2):9–22, 1985.

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[Mak] P. J. Maker. Gnu Nana: improving support for assertions and logging in c and c++ (web page). http://www.cs.ntu.edu.au/homepages/pjm/nana-home/.

[Ros91] D. Rosenblum. Specifying concurrent systems with TSL. *IEEE Software*, 8(3):52–61, 1991.

[Ros95] D. Rosenblum. Towards a method of programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, 1995.

[RW85] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, 1985.

[SH94] S. Sankar and R. Hayes. Specifying and testing software components using ADL. Technical Report SMLI TR-94-23, Sun Microsystems Laboratories, Inc., April 1994.

[SL86] D. Rosenblum S. Sankar and D. Luckham. Concurrent runtime checking of annotated Ada programs. In *Proceedings of the 6th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 10–35. Springer-Verlag-Lecture Notes in Computer Science, 241, 1986.

[SU86] A. Aho R. Sethi and J. Ullman. *Compilers: Principles Techniques and Tools*. Addison-Wesley, 1986.

APPENDICES

Appendix A: Identifying Statements in Byte Codes

Given a class file, we want to find which byte codes correspond to which statements of the source program. An algorithm for that would in fact be the inverse procedure of the code generation phase. Thus by observing the way the code is generated, we can associate byte codes of a class file to java statements of the corresponding source file.

Most of the Virtual Machine instructions involve manipulation of the stack, i.e. they expect to find the necessary arguments on the top of the stack. For example the instruction `fadd` removes the top two elements of the stack, adds them, and puts the result back in the stack. A statement of a source program is translated into a sequence of assembly instructions that first manipulate the stack and then perform the necessary operations on the arguments placed on top of the stack. Based on that, it is possible to separate the Java Virtual Machine instructions into groups according to whether the code generation of a statement can end to such an instruction and under which conditions.

The groups that the VM instructions can be separated are:

- *EndOfStatement*: set of instructions which are definitely found at the end of the sequence of instructions to which a statement is translated. Such instructions are: `return` instructions, some control transfer instructions, instructions to which the switch statement is translated (`lookupswitch` and `tableswitch`).
- *AlmostEndOfStatement*: set of instructions which can be found at the end of the sequence of instructions to which a statement is translated, if the next instruction does not belong to the *EndOfStatement* group. Such instructions are: `store` instructions, instructions for method invocation, etc.
- *NotEndOfStatement*: Contains all the instructions that do not belong in the previous groups, that is instructions that cannot be found at the end of a sequence of assembly code corresponding to a Java statement. For example, stack

manipulation instructions, `load` instructions, casting instructions, arithmetic instructions, etc.

Based on the above categorization, the general algorithm that identifies to which byte codes a source statement translates, follows:

```
get_first_instructionCode
for all instructionCode of the method {
   if (instructionCode is in isNotEndInstr) then
      statements[instructionCode] = false
   else
      if (instructionCode is in isEndInstr)
        statements[instructionCode] = true
      else {
        if (next InstructionCode is in isEndInstr)
         statements[instructionCode] = false
        else
         statements[instructionCode] = true
      }
   get_next_instructionCode
}
```

For each byte in the byte code array there is an associated flag which is set to `true` if this byte code corresponds to the last of the byte codes to which a statement is translated, and is set to `false` otherwise. To see more clearly the algorithm, consider the following simple method (taken from [LY96]):

```
MyObj example() {
  MyObj o = newMyobj();
  return silly(o);
}
```

The byte codes generated for the above source code, using their mnemonic names, are:

```
0  new #2              //Class MyObj
3  dup
4  invokespecial #5    //Method MyObj.<init>()V
7  astore_1
8  aload_0
9  aload_1
10 invokevirtual #4
             // Method Example.silly(LMyObj;)LMyObj;
13 areturn
```

The indices at the beginning of each of the assembly instructions may be thought as a byte offset from the beginning of the method. Applying the above algorithm, we get the following correspondence between byte codes and source statements:

statement	start byte code index	end byte code index
0	0	7
1	8	13

In fact there are more complications than the above sketch of algorithm indicates. In the case where the source code contains `finally` some of the code generated does not correspond to a source statement. For example:

```
0  void tryFinally() {
1     try{
2       tryItOut();
3     } finally {
4        wrapItUp();
5     }
6  }
```

is translated to:

```
0  aload_0            //Beginning of try block
1  invokevirtual #6   //Method Example.tryItOut()V
4  jsr 14             //Call finally block
7  return             //End of try block
8  astore_1           //Beginning og handler for any throw
9  jsr 14             //Call finally block
12 aload_1            //Push thrown value
13 athrow             //and rethrow the value to the invokers
14 astore_2           //Beginning of finally block
15 aload_0            //Push this onto stack
16 invokevirtual #5   //Method Example.wrapItUp()V
19 ret2               //Return from finally block
Exception Table:
From To Target Type
0    4  8      Any
```

Observe that the previous algorithm would associate the byte codes 8 till 13 some statements of the source files whereas they are generated to handle exceptions. The solutions adopted for the above are:

- Ignore the byte codes that are generated to implement the `finally` statement. We can do that by looking at the exceptions array to find if there are any exception handlers of type `any`.
- Ensure that a control transfer instruction always branches to an instruction that has been identified as corresponding to the beginning of a statement.