INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

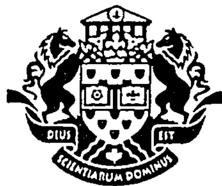
The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®



Université d'Ottawa · University of Ottawa

High-Yield (Fault-Directed) Approach to Test Suite Evaluation

by

Khalid Khidhir

A thesis
submitted to the School of Graduate Studies and Research
in partial fulfillment of the degree of

Masters in Computer Science

School of Information Technology and Engineering
University of Ottawa
(Ottawa - Carleton Institute for Computer Science)

© Khalid Khidhir, Ottawa, Ontario, Canada, 2001

National Library of Canada
Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services
Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-72773-4

Canada

Abstract:

Given a test suite *T* designed to test a program *P*, there are at least three attributes of *T*, which can be used to assess *T*'s effectiveness. The first attribute is its *Completeness* measured by *Code Coverage* when *P* is executed on all the tests in *T*. The second attribute is *Adequacy* or *Fault Coverage* of the test suite *T*, which is the ability of the test suite *T* to exercise *P* so that for every fault (according to some fault model) in *P* there is at least one test case in *T* that will detect it. The third attribute is *Requirements Coverage*, which checks for every requirement *R* whether there is at least one test case in *T* that will demonstrate the conformance of the program *P* to the function described by *R*.

In this thesis we will evaluate the effectiveness of an industrial test suite by measuring the first attribute and estimating the second. In this case *T* is a *Java* conformance test suite, and *P* is a Java compiler. We show that basic coverage measurement techniques and a simple fault model can be used to obtain useful preliminary data early in the testing phase of the software development life cycle. This was the case in our case study. The technique is low-cost and allows the enhancement of conformance test suite for languages, protocols, or any standard specification.

Acknowledgements:

I wish to thank all the people who have influenced my work and life in many different ways during my studies in the University of Ottawa. First of all, I thank my supervisor Prof. Dr. Robert Probert, who provided financial and research support. I thank him for encouraging me to finish, and for the time he has spend reviewing my work especially the past few months. I thank IBM, CAS and OTI for their support in allowing me access to their labs to perform the experiments. My Mother Henna Issac, sister Teresa and two brothers Hanni and Max for their support and patience, you guys are the best. I would also like to thank Prof. Sylvia Boyd for her interesting classes and prompt replies to my questions, and Prof. Luigi Logrippo for his support in administrative matters. Thanks to Louise Desrocher, there are not enough words to thank her. Special thanks go also to Prof. Klaus Kanneman; it was always a pleasure to TA for him. My friends have always been there for me, sometimes even when I didn't want them :-). We played pool, Ping-Pong and volleyball together, went on picnics and bike rides, and competed in spicy food cooking and eating. Jelber Sayyad, Alan Williams, Rossana Andrade, Stephane Some, Nicolas Anquetil, Francisco and Gina Herrera, Daniela Savin, Marta and Juan Pablo Zamorra, Ana Negrete, Fernanda Caropreso, and Felipe Contreras they were all good reasons why I didn't want to leave the university too soon. And finally a special thank you goes to my fiancée, Vivi Nastase. I am grateful for your patience, support, and friendship, they were very much appreciated.

1.0 Edition

Published 2002

Copyright © 2001, 2000 by Khalid Khidhir., Ottawa, Ontario Canada.

To my mother Henna Issac.

Table of Contents

Abstract: i

Acknowledgements: ii

Glossary 3

1. Introduction 1

1.1. Motivation for Evaluating Test Suites 1

1.2. Problem Under Study 3

1.3. Objective and Contributions of the Thesis 3

1.4. Organization of the Thesis 4

2. Definition and Current Use of Coverage Measurements 6

2.1. Motivation 6

2.2. Types of Coverage Goals 7

2.2.1. Code Coverage Fundamentals 7

2.2.1.1. Overview 8

2.2.1.2. Common Code Coverage Measurements 13

2.2.1.2.1. Statement Coverage (Basic block coverage): 13

2.2.1.2.2. Branch Coverage (Decision Coverage): 15

2.2.1.2.3. Condition Coverage: 16

2.2.1.2.4. Multiple Condition Coverage: 18

2.2.1.2.5. Path Coverage (Predicate Coverage): 19

2.2.1.3. Other Code Coverage Measurements 21

2.2.1.3.1. Function/Method Coverage 21

2.2.1.3.2. Call Coverage (Call Pair Coverage) 22

2.2.1.3.3. Linear Code Sequence and Jump (LCSAJ) Coverage .. 24

2.2.1.3.4. Data Flow Coverage ... 26

2.2.1.3.5. Loop Coverage ... 29

2.2.1.3.6. Relational Operator Coverage ... 29

2.2.1.3.7. Exception Coverage ... 30

2.2.2. Fault Coverage (Adequacy) ... 31

2.2.2.1. Fault model (informal definition) ... 32

2.2.3. Function (Requirements) Coverage ... 34

2.3. Discussions and Conclusions ... 34

3. The High Yield Approach to Test Management ... 36

3.1. The Problem ... 37

3.2. The High Yield Code Coverage Strategy ... 38

3.3. Hypotheses: ... 41

3.4. The Instrumentation Tool ... 42

4. Case Study and Assessment ... 43

4.1. The compiler (PBJ) ... 43

4.2. Fault Model ... 48

4.3. The Test Suite ... 48

4.4. Results ... 51

4.5. Determining when the PBJ passed/Failed a test case ... 53

4.6. Summary and Discussion ... 55

5. Coverage Tool Design and implementation issues ... 58

5.1. Requirements for an Experimental Code Coverage Tool (JCov) ... 58

5.2. JCov Architecture ... 61

5.3. Invoking JCov ... 68

5.4. Testing the instrumentation tool 69

5.5. Summary 70

6. Conclusions and Suggestions for Futher Work 72

6.1. Conclusion 72

6.2. Suggestions for Further Work 74

References 76

A. JCov Program Listing 80

A.1. Tree Nodes 80

A.2. The Visitor Classes 85

B. The JavaCC grammar 100

B.1. JavaCC Options 101

B.2. Productions 102

List of Tables

2-1. LCSAJs for the program in Example 2-11 25

4-1. PBJ source code metrics 45

4-2. More PBJ source code metrics 45

4-3. More PBJ source code metrics 46

4-4. Test cases 49

4-5. PBJ coverage metrics using vm test cases 51

4-6. coverage metrics using lang test cases with a Java parser 52

5-1. Coverage criteria implemented by JCov and probe information 59

List of Figures

2-1. How to Insert Probes Automatically 8

2-2. DD-Path graph for the triangle.java program (numbers refer to source lines) 11

2-3. DD-Path for structured programming constructs 12

2-4. (a) Source-level construct exits (2), versus (b) Object-level construct exits 16

4-1. Compiling a Java source file or a class bytecode file with the PBJ compiler 44

4-2. Using an oracle to obtain actual output for every test case 54

5-1. Using the coverage/instrumentation tool 60

5-2. Using JCov to instrument the compiler under test (CUT) (Figure 5-1 in dotted area) 61

5-3. JCov components 62

List of Examples

2-1. Triangle Program 9

2-2. Statement coverage 13

2-3. Branch Coverage 15

2-4. Condition coverage 17

2-6. Multiple Condition coverage 18

2-8. Path Coverage 20

2-9. Function/Method Coverage 22

2-10. Call Coverage example 23

2-11. LCSAJ Coverage 24

2-12. Data flow coverage 26

2-14. Loop coverage example 29

2-15. Relational operator example 30

2-16. Example: 30

4-1. Positive Test 49

4-2. Negative Test 50

A-1. KKIfStatement class 80

A-2. KKIfNode class 81

A-3. KKElseNode 83

A-4. LoopVisitor class 86

A-5. MethodVisitor class 88

A-6. BranchVisitor class 91

A-7. ConditionVisitor class 94

Glossary

branch condition coverage

The percentage of branch condition outcomes in every decision that have been exercised by a test case suite.

code coverage

An analysis method that determines which parts of the software have been executed (covered) by the test case suite and which parts have not been executed and therefore may require additional attention.

decision coverage

The percentage of decision outcomes that have been exercised by a test case suite.

defect

An incorrect or missing software component that results in a failure to meet a functional or performance requirement. Any implementation deviation from requirements and specification.

error

A human action that produces an incorrect result or a fault.

failure

Deviation of the software from its expected delivery or service.

fault

A manifestation of an error in software. A fault, if encountered may cause a failure.

fault model

A set of hypothesis that consider areas of the specification and/or code where error have higher probability of occurring or been made.

feasible path

A path for which there exists a set of input values and execution conditions that causes it to be executed.

infeasible path

A path that cannot be exercised by any set of possible input values.

negative test

A test case that provide incorrect input. The program should reject it with appropriate action/error message.

null else

The added *else* part to an *if* statement that was programmed without one.

null switch default

The added *default* branch to a *switch* statement that was programmed without one.

path

A sequence of executable statements of a component, from an entry point to an exit point.

positive test case

A test case that provides valid input. The program must accept it and provide correct output.

test suite adequacy

Is the ability of the test suite to exercise the program under test so that for every fault (according to some fault model) in the program there is at least one test case in the test suite that will detect it.

yield

Is the number of high and moderate-risk defects detected by a particular testing strategy [Humphrey99], [Humphrey95]

Chapter 1. Introduction

1.1. Motivation for Evaluating Test Suites

Software systems touch our lives in many different ways, ranging from simple systems such as a drawing program to high-risk applications such as a control system for a nuclear reactor. System developers must provide assurance to the users that these systems are reliable (they contain no serious errors). This can be achieved by proving that the software system is correct. However such methods are usually limited in scope and impractical to use in a cost-effective manner for large systems. Rigorous software testing provides the only known practical solution to the problem by systematically exercising the system in a controlled environment. Software testing is one of the most important and critical phases of the software development life-cycle. Testing is also the most time consuming and costly phase, requiring much creativity and patience on the part of the software testers [Lamarche98]. The main issues or weaknesses of software testing are:

1. The impossibility of ensuring the absence of errors in the implementation under test [Myers79].
2. The size and cost to develop the test suite required to be executed to uncover existing errors, often due to the large size of the input domain of non-trivial programs. This is referred to as the *Economics of Testing* [Myers79].

Since we cannot achieve exhaustive software testing, important questions must be asked, particularly:

- When can we stop testing?
- Can we measure the completeness and the validity of our testing strategy?
- Can we measure the coverage and use such measurements to estimate the number of faults remaining in the system?

Higher coverage increases the confidence of users, designers and testers in the system.

Compilers are examples of software systems that are used by a large base of users that rely on them for the development of other software systems. They are large and complex to develop and test. The huge time and effort put in developing a compiler is evident in the FORTRAN compiler, which took 18 man-years to implement [Saleh94]. Today with the market demand and the development of tools that automatically implement components of these compilers, development and testing time has been reduced considerably. A compiler's main task is to generate object code from a program written in a certain language according to some language specification with the added requirements of *high reliability*. In the case of a compiler, this mean that the compiler has not introduced any defects in the compiled object code. Traditionally, compilers are mainly tested for *conformance* to the language specification by using black-box (specification based) testing techniques. [Myers79]. We feel that additional quality effectiveness and completeness will result from measuring code coverage of the compiler by the test suite. This is a white-box testing technique that was not used in the industrial development project we studied, and we believe it is not generally used for compiler testing. In addition black-box testing is not sufficient for compiler testing for the following reason [Poston96]: A conformance test suite can produce two drastically different code coverage results when used with two compilers designed for the same language (implementation differences, for example data structures). These implementation details are ignored by the black-box/conformance test suite. In this

thesis, an examination and assessment of such a test suite is undertaken. This test suite was designed to certify the conformance of an implementation to the Java standard.

1.2. Problem Under Study

Here we state a generic form of the problem we address in this thesis:

Given a conformance test suite *T* designed to test the conformance of an implementation to a standard specification *S*, and a program *P* that implements *S*, can white-box test code coverage measurements be used to assess the quality of the test suite *T*?

In this thesis, *S* will be the language standard, *T* will be the Java conformance test suite, and *P* will be a Java compiler under test.

1.3. Objective and Contributions of the Thesis

Our main objective is to develop a method to evaluate a test suite given for an implementation. Our method is based on a *High-Yield* approach, defined in the third chapter. Using this method we can help the designers to identify weaknesses of their testing strategy, make decisions about improving it, and determine the suitability of a given test suite to their implementation.

In this thesis we propose a set of coverage metrics and techniques, develop a prototype tool to instrument the source code, then run the test suite and measure the coverage and report or draw conclusions.

We will see later that, with the application of these techniques to compiler test suites, we will be able to show the lack of coverage of high-yield areas (where faults can

hide), and the well covered low-yield area (normal behavior). We will also draw some conclusions and propose suggestions on how to improve the effectiveness of the test-suite in the light of our findings.

We hope that this work will contribute to evaluating the testing strategies and test suites in the work place, firstly compiler test suites, and secondly to robustness test suites in other areas.

Our specific contributions are:

1. Method of designing robustness-oriented coverage measurements tools.
2. We show how to generate a coverage tool given a specific structural fault mode.
3. The specific tool (prototype only) *JCov* for assessing adequacy of a Java test suite.
4. The methodology of evaluating a test suite using a structural fault model.
5. An application of this method to an industrial test suite (case study).

1.4. Organization of the Thesis

In chapter 2 we will review testing techniques and coverage techniques, explaining the advantages and disadvantages of each technique, possibly giving an example to demonstrate each technique. In chapter 3 we will discuss our current work in detail, the coverage strategy chosen and justifying our motivation and approach. In chapter 4 we will show the results of the that we collected from an industrial case study, showing our findings and giving recommendation to improve the test suite. Chapter 5 will conclude our work, giving some future work that can be done. A glossary is added to help the reader with some terms and acronyms, an appendix describing the structure of

the grammar used, and finally an appendix showing the source code of our prototype tool.

Chapter 2. Definition and Current Use of Coverage Measurements

2.1. Motivation

Effective software testers start the testing process with three purposes in mind: [Poston96]

1. To find the failures in the software (fault coverage),
2. To exercise all the parts of the software (code coverage),
3. To demonstrate that the software functions as specified (requirement coverage).

Software testers will end the testing process by evaluating how thoroughly the three purposes were satisfied. In other words they need to evaluate their testing process, by examining the following:

1. What percentage of failures was found in the software? [Li98]
2. What percentage of statements, branches, and paths were exercised?
3. What percentage of required functional behavior was demonstrated?

To answer the above questions, testers need to transform them into questions involving metrics. Corresponding measurements show to management and users *how much testing was done*, and *whether it is time to stop testing*. In this chapter, we give formal and informal definitions and techniques that can be used to obtain such measurements.

In this chapter, we will review the three types of coverage, and give detailed explanation to *Code Coverage* with examples showing the different techniques. In this thesis we only tackled the problem of evaluating a test suite for completeness from the code coverage aspect, and estimating adequacy (fault coverage, robustness) by linking code coverage to fault coverage by our fault model, discussed in section 3 below.

2.2. Types of Coverage Goals

In this section we give formal and informal definitions for three coverage goals. We also provide a program model that is the basis of the code coverage used in this thesis.

2.2.1. Code Coverage Fundamentals

Code coverage is the process of adding *Probes* or instrumenting a program for the purpose of measuring one or more of the following coverage criteria. Figure 2-1 shows how to automatically insert probes in an implementation under test.

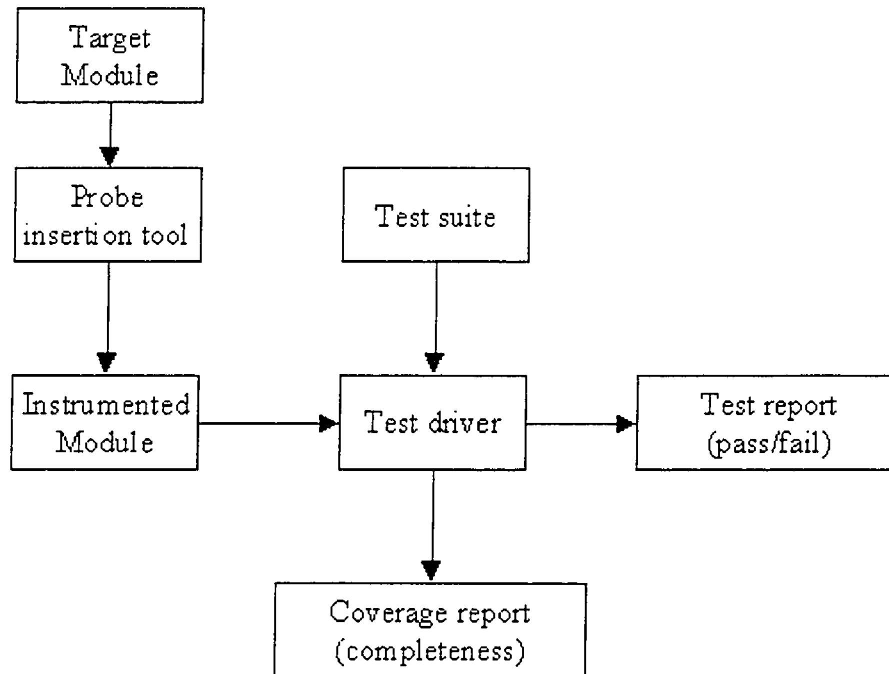


Figure 2-1. How to Insert Probes Automatically

2.2.1.1. Overview

Here we give some definitions that will help in setting up a structural model for a program. This program model is used to define the different coverage techniques discussed in this thesis. [Jorgensen95]. The definitions are illustrated below via Example 2-1 (code) and Figure 2-2 (program graph).

Program graph: Given a program written in an imperative language, its *program graph* is a directed graph in which nodes are either entire statements or fragments of a statement, and edges represent flow of control.

If `i` and `j` are nodes in a program graph, there is an edge from node `i` to node `j` iff the statement corresponding to node `j` can be executed immediately after the statement corresponding to node `i`.

Paths: A *path* is a *chain* (a sequence) of edges in a graph such that any consecutive edges e_i and e_{i+1} in the chain, are adjacent to a common node.

DD-Paths: A *decision to decision path (DD-Path)* is a known construct of structural testing [Jorgensen95]. It refers to a chain of statements that begins with a decision statement and ends with the next decision statement. There are no internal branches in such sequence. In terms of nodes in a directed graph (program graph), DD-Paths is a sequance of nodes such that:

1. It consists of a single node with `indeg = 0`
2. It consists of a single node with `outdeg = 0`
3. It consists of a single node with `indeg >= 2` or `outdeg >= 2`
4. It consists of a single node with `indeg = 1` and `outdeg = 1`
5. It is a maximal sequence of length >= 1
6. All nodes in athe chain have outdegree <=1

Here is an example of a simple program to illustrate the idea of DD-Paths, taken from [Jorgensen95], rewritten in Java:

```
1 /* A simple triangle program, it accepts three integers and
2    answers whether they represent the three edges of an Isosceles,
3    Scalene, Equilateral or an invalid Triangle.
4  */
5 public class triangle
6 {
```

```
 7    public static void main(String[] argv)
 8    {
 9       if(argv.length != 3)
10       {
11         System.err.println("error: not correct parameters");
12         System.exit(1);
13       }
14
15       int s1 = Integer.parseInt(argv[0]),
16           s2 = Integer.parseInt(argv[1]),
17           s3 = Integer.parseInt(argv[2]);

18       System.out.println("Using values: " + s1 + ", " + s2 + ", " + s3);

19       boolean isTriangle;

20       if( (s1 < s2 + s3) && (s2 < s1 + s3) && (s3 < s1 + s2) )
21          isTriangle = true;
22       else
23          isTriangle = false;

24       if (isTriangle)
25       {
26          if( ( s1 == s2 || s2 == s3 || s3 == s1 ) &&
27              ! ( s1 == s2 && s2 == s3 ) )
28             System.out.println("Isosceles");
29          if( s1 != s2 && s2 != s3 && s3 != s1 )
30             System.out.println("Scalene");
31          if( s1 == s2 && s2 == s3 )
32             System.out.println("Equilateral");
33       }
34       else
35       {
36          System.err.println("Not a valid triangle");
37          System.exit(2);
38       }

39       System.exit(0);
40    }
41 }
```

Example 2-1. Triangle Program

The program graph for the above program is shown below:

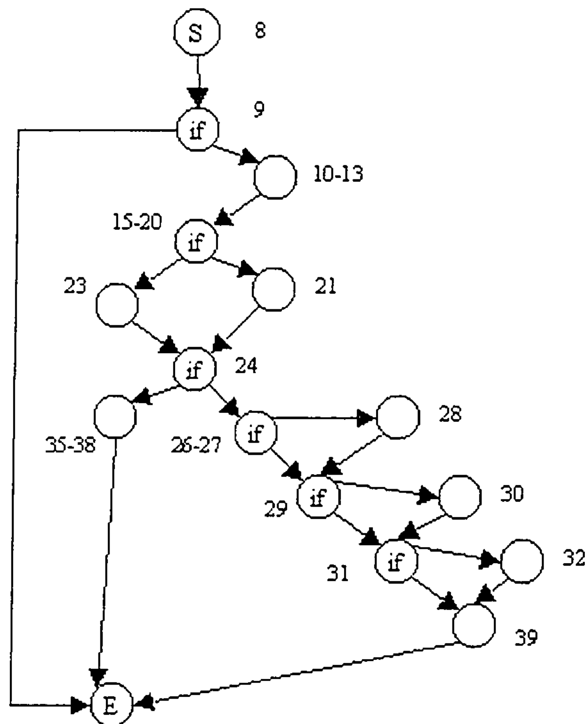


Figure 2-2. DD-Path graph for the triangle.java program (numbers refer to source lines)

In the above example we see the DD-Path representation for an `if` statement. In Figure 2-3 we give the graph representation for all the other constructs of a structured programming language such as Java. This representation is very useful for planning and implementing software tools, such as *JCov* discussed in chapters 4 and 5, which inserts coverage measurement probes into strategic locations in the program.

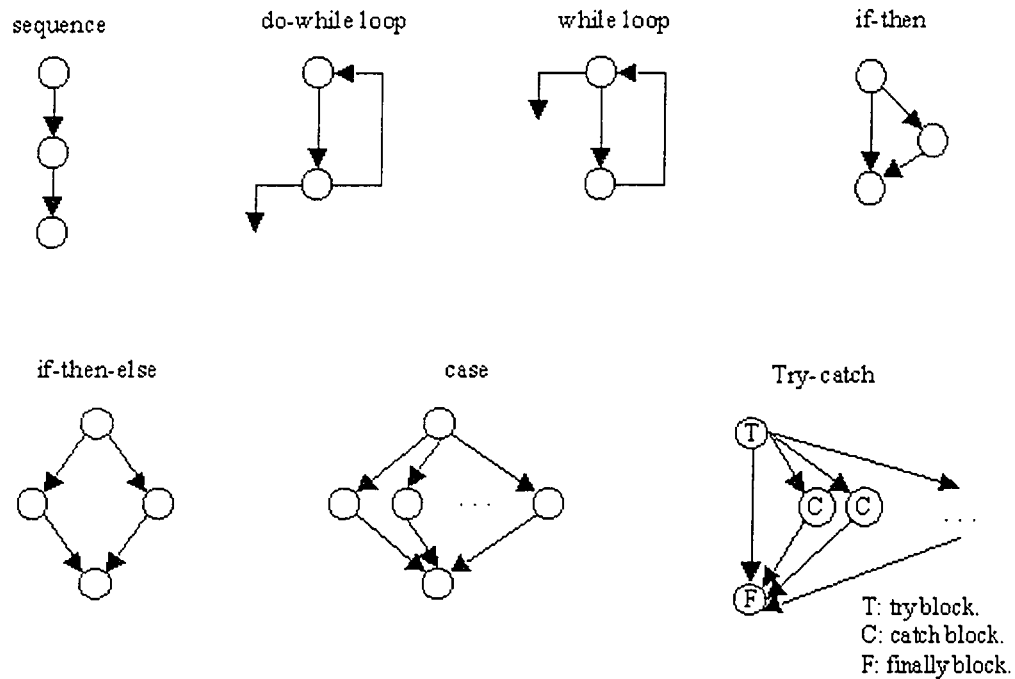


Figure 2-3. DD-Path for structured programming constructs

In our work we used a complexity metric to differentiate the blocks of code from each other, we applied this on only the methods of the classes, but it can be applied to the individual blocks too. This is closely related to the code coverage techniques (nearly all code coverage techniques generate this metric [Binder00]) we used, and it will allow us to highlight the *high-risk* areas of the code. The following is a formal definition of this metric using the program graph.

Cyclomatic Complexity: The *Cyclomatic complexity of a graph* [McCabe76] G is given by $C(G) = e - n + p$, where:

1. e is the number of edges in G

2. n is the number of nodes in G

3. p is the number of components in G

Informally C(G) is the number of distinct regions in the graph. One approach of structural testing postulates the notion of basis paths in a program, from which all other paths can be derived. Cyclomatic number is actually the number of these basis paths.

Now, we review the best known code coverage criteria.

2.2.1.2. Common Code Coverage Measurements

In this section we list the simple or basic code coverage measurements [Binder00], and we illustrate each technique with an example. We also give its advantages and disadvantages, and we also rate each technique with a *Value* factor. This factor is derived mainly from our assessment of the probability of each technique of detecting faults in the program (more on this in the following chapters).

2.2.1.2.1. Statement Coverage (Basic block coverage):

Reports the percentage of executable statements in the source code which have been executed. Basic block coverage is the same as statement coverage except the unit of code measured is each sequence of non-branching statements.

```
int b = 20;
int a = 0;
if (some_condition)
  a = 10;
b = b/a;
```

Example 2-2. Statement coverage

In the above example, statement coverage will report 100% if the test case causes the condition to evaluate to `true`. Thus one test case is sufficient to achieve 100% coverage. However if the condition evaluates to `false`, then this code will fail. Thus statement coverage is a very low degree of coverage, and is inferior for detecting faults. Another disadvantage of this technique, is that it reports only whether loops are executed or not, not how many times they were executed. Also it is insensitive to logical operators like *OR* and *AND*.

Advantages: The main advantage of this technique is that it can be applied to the object code and does not need the parsing of the source code. Thus it is easier to implement an instrumentation tool for statement coverage alone (discussed in chapter 4).

Disadvantages: The main disadvantage is that it does not consider optional parts of control structures like *if* and *case* statements, which [Binder00] has noted the fall-throughs (NULL Else, NULL defaults, ...) "invite bugs".

One argument in favor of statement coverage over other measures is that faults are evenly distributed through code; therefore the percentage of executable statements covered reflects the percentage of faults discovered. However, one of our fundamental assumptions is that faults are related to control flow, and not computations. Additionally, we could reasonably expect that programmers strive for a relatively constant ratio of branches to statements.

Value: This technique is not sensitive to flow control, and it treats all the statements the same way, without any regard to the statement importance or complexity.

2.2.1.2.2. Branch Coverage (Decision Coverage):

Reports the percentage of exercised exits (true and false) for boolean expressions in control structures (such as the if-statement and while-statement). The entire boolean expression is considered as one predicate regardless of whether it is a compound boolean expression (contains *AND* or *OR* operators). This measure also includes the coverage of branch statements (case statement), and exceptions (try, catch, throw).

```
if (cond1 && (cond2 || fun1()))
  statement1;
else
  statement2;
```

Example 2-3. Branch Coverage

In the above example, branch coverage will report 100% without the call to `fun1()`. The expression evaluates to true if `cond1` and `cond2` are true, and false when `cond1` is false. This is due to the nature of the short-circuit operators that will avoid executing the function.

Advantages: Simplicity without the problems associated with Statement Coverage.

Disadvantages: One disadvantage with this coverage criteria is that due to the short-circuit operators, this measure ignores some parts of the compound boolean expression, as illustrated in the example above. We can achieve 100% statement and 100% branch coverage without the execution of `fun1()`. As well, this criteron requires access to the source code, and cannot be done by instrumenting the object code.

Value: Medium (better than statement coverage), it is sensitive to the flow control of the program.

2.2.1.2.3. Condition Coverage:

Reports the percentage of true and false evaluations over all each boolean sub-expression. Sub-expressions are separated by logical-*OR* and logical-*AND*. This technique measures the sub-expressions independently. It is similar to Branch Coverage but has better sensitivity to the control flow. For example the source level graph and the object level graph are shown in Figure 2-4. More exits are visible at the object level due to the compound nature of the boolean expressions.

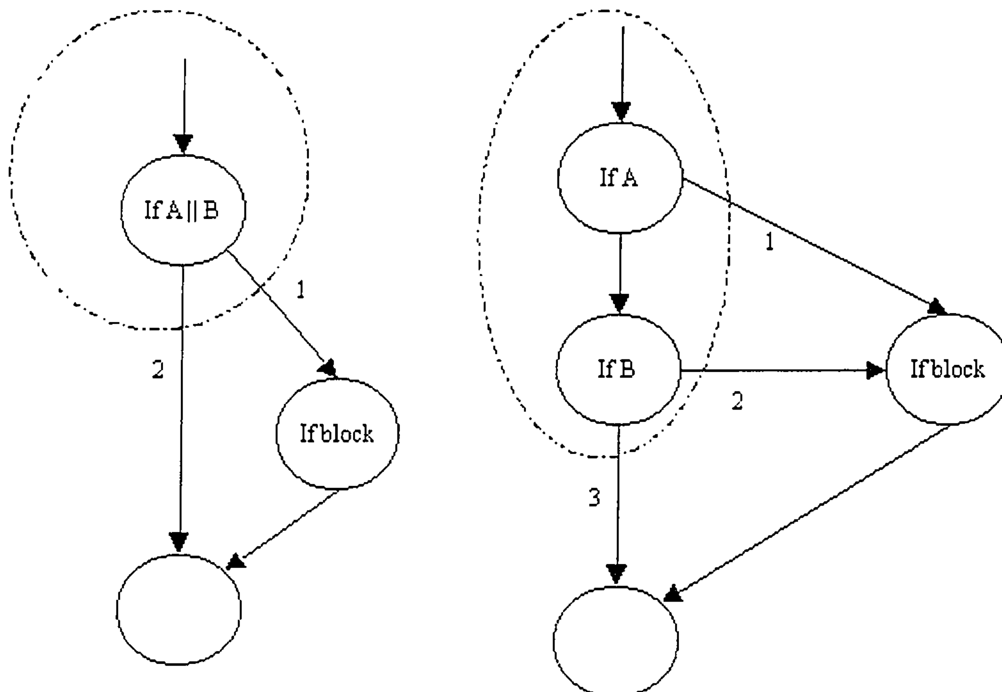


Figure 2-4. (a) Source-level construct exits (2), versus (b) Object-level construct exits

As we see in the above diagram, the evaluation of each sub-expression yields to the traversal of all the exits of the if construct in object code.

```
if (a && (b || c))
{
}
else
{
}
```

Example 2-4. Condition coverage

To satisfy this criterion `a`, `b` and `c` must evaluate to true and false.

Advantages: Similar advantages to Branch Coverage, plus the advantage of evaluating all the boolean sub-expressions. Hence overcoming the problem with the short-circuit operators that the previous coverage suffered from.

Disadvantages: It may require more test cases to be executed than in Branch Coverage. Also this measure does not guarantee full Branch Coverage, for example:

```
if (a || b)
{
  // ...
}
else
{
  // ...
}
```

Example 2-5. Example of 100% condition coverage, but 50% branch coverage

In the above example, if `a` and `b` evaluate to true and true, true and false, and false and true, the `else` part of the program will not execute.

Value: Medium, does not include Branch Coverage. High if combined with Branch Coverage.

2.2.1.2.4. Multiple Condition Coverage:

Reports percentage of possible combination of boolean sub-expression is evaluated. This is very similar to Condition Coverage, but it will consider not only the sub-expressions but also the combinations of these sub-expressions. Test cases required for a 100% coverage are given by the truth table of the logical operators of the condition.

```
if(a && b && (c || (d && e)))
{
  ...
}
else

  ...{
}
```

Example 2-6. Multiple Condition coverage

In the above example 6 (not 32) test cases will be required (due to the short circuit operators in Java and C). For example when `a==false` the whole expression will evaluate to false, and there is no need to evaluate the other operands. This will reduce the test cases in half, and we can see that the number will be reduced to 8 when operand `b==false`, and so on.

Advantages: Has the same advantages as Condition Coverage. It requires very thorough testing since it also considers the combination of the sub-conditions.

Disadvantages: Can be tedious to find the minimum number of test cases required for complex expressions, more difficult to implement than the Condition Coverage, and the number of test cases can vary for conditions with similar complexity. For example

the following condition would require 11 test cases, but it has the same number of operands and operators as the condition in the above example:

```
((a || b) && (c || d)) && e
```

Example 2-7. Multiple condition coverage

As with Condition Coverage, this technique does not include Branch Coverage.

Note: For languages without short circuit operators such as Pascal, multiple condition coverage is similar to Path Coverage for logical expressions, with the same advantages and disadvantages. For example, given the following Pascal code segment:

```
If a And b Then
Begin
  ...
End
```

Multiple condition coverage requires four test cases, one for each of the combinations of logical values for `a` and `b`. As with Path Coverage each additional logical operator doubles the number of test cases required.

Value: High, but so is the cost, complex to implement specially for languages with short circuit operators.

2.2.1.2.5. Path Coverage (Predicate Coverage):

Reports the percentage each of possible paths in each function which have been followed or executed. A *path* is a unique sequence of branches from the function entry to the exit. Predicate coverage views paths as all possible combinations of logical conditions in a function.

Loops introduce a problem for this measure, since they introduce an unbounded number of paths. This has the effect of greatly increasing the number of paths, and consequently the number of test cases required to exercise all these paths. To cope with this problem, a limited number of looping possibilities are considered as proposed by *boundary-interior path testing* which considers two possibilities for loops: for `while` loops, zero repetitions and more than zero repetition. For `do-while` loops, the two possibilities are one iteration and more than one iteration, while for `for` loops, it may include zero, one and more than one iterations.

```
if(a)
{
  ...
}
...
if(b)
{
  ...
}
```

Example 2-8. Path Coverage

In the above example, there are 4 paths: `a=true` and `b=true`, `a=false` and `b=true`, `a=true` and `b=false`, `a=false` and `b=false`.

Advantages: Path coverage requires very thorough testing, which means it requires more test cases to be executed. It includes all other coverage criteria.

Disadvantages: One disadvantage is that the number of paths is exponential in the number of branches. For example, a function containing 10 `if`-statements has 1024 paths to test. Adding one more `if`-statement doubles the count to 2048. Another disadvantage is that many paths may be impossible to exercise due to certain conditions in the code. For example, consider the following:

```
if(condition1)
   statement1

if(condition1)
   statement2
```

In the above example Path Coverage requires 4 paths. However only two are feasible, namely `condition=true` and `condition=false`. This can be considered as an advantage as this technique, it may points out situations like the above to the designers, which can be a result of a coding mistake (`condition2` instead of `condition1` in the second `if`-statement).

There is a variation on this technique, the *n-length sub path coverage*. This technique reports whether each path of length `n` branches is been executed.

Value: High in theory, but too time consuming and costly to be practical. Note that 100% coverage cant not usually be achieved.

In summary, there are a great many code coverage criteria. In order to be pragmatic, we will focus on Branch/decision coverage, method coverage, and condition coverage. Next, we briefly consider other coverage criteria. In particular, we consider the following two object oriented criteria.

2.2.1.3. Other Code Coverage Measurements

2.2.1.3.1. Function/Method Coverage

Reports the percentage of functions/methods which have been invoked. Although this sounds very simple or trivial, it is useful during preliminary testing (sanity testing) to make sure that at least some coverage in all areas of the software product has been provided by the test suite.

```
f1(int a)
{
  if(condition)
  {
    f2(...)
  }
}

f2(int b)
{
}

f3()
{
  f1(...)
  f2(...)
}
```

Example 2-9. Function/Method Coverage

100% Function/method Coverage is satisfied if `condition` evaluates to `false`, but it will not satisfy 100% Call Coverage (see the next criterion).

Advantages: Broad, shallow testing finds gross deficiencies in a test suite quickly. Simple to implement either in the source code, or in Object code.

Disadvantages: It is insensitive to the internal structure of the code, Best to be used as a preliminary measure, or with other measures.

This measure is included in Statement Coverage

Value: Low. Statement Coverage has higher value, since it includes this criterion. Function/method Coverage.

2.2.1.3.2. Call Coverage (Call Pair Coverage)

Reports whether each function call have been executed. The hypothesis is that *faults commonly occur in interfaces between modules/classes*. This plays an important role in

Object Oriented Programming, since polymorphism and inheritance introduce new problems. Example 2-10 illustrates this coverage criterion.

```
class A
{
  int methodx()
}

class B extends A
{
  int methodx()
}

class C
{
  fun(A a)
  {
     a.methodx();
  }

  /* test driver for A, B, and C
   */
  main()
  {
     fun(new A);
     B b = new B();
     b.methodx();
  }
}
```

Example 2-10. Call Coverage example

The above example provides 100% Function/method Coverage, but it does not provide 100% Call Coverage. It is missing a call such as `fun(new B)` which pairs class `C` with class `B`. This is due to the fact that `A` is a super class of `B` and it is allowed to pass an object of a subclass to a method that has a parameter type of a super class.

Advantages: Essential in testing interfaces between classes or modules, in integration testing, and class libraries.

Disadvantages: Not simple to implement, requires the knowledge of the inheritance structure of the code which can depend on other code/libraries.

Value: High. It can potentially uncover problems with module interactions specially with object oriented code.

2.2.1.3.3. Linear Code Sequence and Jump (LCSAJ) Coverage

This coverage criterion is a modified Path Coverage. It considers only sub-paths that can be easily represented in the program source code without requiring a flow graph. An *LCSAJ* is a sequence of source code lines executed in sequence. This *linear* sequence can contain decisions as long as the control flow actually continues from one line to the next at run-time. Sub-paths are constructed by concatenating LCSAJs. [Roach89]

```
1   public class LCSAJ_Example
2   {
3      public static void main(String[] args)
4      {
5         int x = Integer.parseInt(args[0]);
6         int y = Integer.parseInt(args[1]);
7         float z = 0;
8         int pow = 0;
9
10        System.out.println("processing : " + x + ", " + y);
11
12        if(y < 0)
13           pow = (int) -y;
14        else
15           pow = (int) y;
16
17        z=1;
18
19        while(pow != 0)
20        {
21           z = z * x;
22           pow --;
23        }
24
```

```
25          if(y < 0)
26             z = 1/z;
27
28          System.out.println("The answer is: " + z);
29          System.exit(0);
30       }
31  }
```

Example 2-11. LCSAJ Coverage

Here are some LCSAJs:

LCSAJ	start line	end line	target line
1	5	12	15
2	16	19	24
3	24	25	28

Table 2-1. LCSAJs for the program in Example 2-11

Where *start line* is the first line of the program or a target line. *target line* is any line to which execution control flow can be given (other than the previous line). *end line* is any line from which execution control flow can be given to a target line.

Advantages: The advantage of this measure is that it is more thorough than Branch Coverage yet it avoids the exponential difficulty of Path Coverage.

Disadvantages: As it is the case in Path Coverage, this technique does not avoid infeasible paths.

Value: High, but costly.

2.2.1.3.4. Data Flow Coverage

This is also a variation of Path Coverage. It reports on sub-paths from variable assignments to subsequent references to the variables. The data flow model recognizes three different basic variable access actions [Binder00]:

- *Define* action occur in a statement that changes the state of a variable, for example: an initialization method, a constructor, a modifier method, and an assignment statement. It is usually abbreviated to *D*.
- *Use* action occur in a statement that gets the value of the variable. It is abbreviated to *U*. There are two types of uses: *Computation use (c-use)* applies when the variable is used in the right hand side of an assignment statement. *Predicate use (p-use)* occurs when the variable is used in a predicate expression.
- *Kill* action occur when the object referenced by the variable is deallocated or released, and is no longer visible within its scope. It is abbreviated to *K*.

Some common access related defects include the following:

1. Using an uninitialized variable.
2. Assigning a value to a variable more than once without an intermediate reference.
3. Sending a message to an object more than once without an intermediate accessing message.

Example 2-12 shows an example of a simple Java code to illustrate the data flow actions:

```
1   void example(int param, String filename)
2   {
3      if (param > 10)
4      {
5          File aFile = new File("");
```

```
 6
 7          // some code that dosn't assess the variable x
 8          ...
 9
10          aFile.setFilename(filename);
11
12          // some more code here
13          ...
14
15         String s = aFile.getFilename();
16
17         // some more code here
18         ...
19      }
20      ...
21   }
```

Example 2-12. Data flow coverage

The following flow paths for the `aFile` object are identified:

- Lines 5-10 is a *DD* path. At line 5, `aFile` is initialized, and at line 10 `aFile` is changed.
- Lines 10-15 a *DU* path. The definition in line 10 is used in line 15.
- Lines 10-20 a *DK* path. The variable `aFile` goes out of scope in line 20, and it will be released to be garbage collected.

Advantages: The advantage of this measure is that the paths considered have direct relevance to the way the program handles data. Another advantage is that some compilers including *javac* report on some variation of this measure during the compilation process. Unfortunately the reporting is restricted to local (automatic) variables only. Here is an example:

```
public class Test1
{
   private static String s;

   public static void main(String[] argv)
   {
      if(s.equals("test"))
      {
      }

      String ss;

      System.out.println("some text ...");

      if(ss.equals("...."))
      {
      }
   }
}
```

Example 2-13. Javac compiler and the detectiion of unassigned variables.

When compiling the above example, *javac* will report on the local variable `ss`, but not on the class variable `s` (which is global for all instances of this class). Here is the output of the javac compiler:

```
C:\test>javac test1.java
test1.java:16: variable ss might not have been initialized
      if(ss.equals("...."))
          ^
1 error
```

Disadvantages: One disadvantage is that this measure does not include Branch Coverage. Another disadvantage is complexity. Although researchers have proposed numerous variations, all of which increase the complexity of this measure. For example, variations distinguish between the use of a variable in a computation versus a

use in a predicate, and between local and global (static) variables. Pointers in C++ present some problems.

Value: High specially for computation oriented programs.

2.2.1.3.5. Loop Coverage

Reports whether loop's body was executed zero times, exactly once, and more than once consecutively. Zero times is not possible for `do-while` loops, in this case it just reports whether it has been executed exactly once, and more than once.

```
while(condition) { ...  }
for(int i=0; i<10; i++){ ... }
do { ... } while (condition);
```

Example 2-14. Loop coverage example

In the case of the `FOR` statement it is not possible to execute the loop body once or zero times, but it is possible to test. Similarly if `condition` in the `while` loop evaluates always to `true`.

Advantages: Information reported in this measure, are not reported by other measures, plus it tests for boundaries for the loops, which is a very high-yield area.

Disadvantages: It is not always possible to execute a loop zero times, or exactly once. As shown in the above example.

Value: High, can uncover potential problems with loops.

2.2.1.3.6. Relational Operator Coverage

Reports whether boundary problems occur with relational operators (<,<=, >, >=). The hypothesis is that boundary test cases find off-by-one errors and mistaken uses of

wrong relational operators such as < instead of <=. Relational operator coverage reports whether the situation a==b occurs. If a==b occurs and the program behaves correctly, you can assume the relational operator is not suppose to be <=.

```
if (a < b)
    ...
```

Example 2-15. Relational operator example

The coverage will report whether `a==b` occurs, if it does and the program behaves correctly, then the assumption is that the relational operator is correct and it is not supposed to be <=

Advantages: It examines a very problematic area of the code. Very effective in finding errors due to using the wrong operator in a condition.

Disadvantages: It requires more tests, hence it can be costly to implement.

Value: High.

2.2.1.3.7. Exception Coverage

Reports whether each exception has been generated in the code.

```
try
{
  ...
}
catch(FileNotFoundException e)
{
}
catch(IOException e)
{
}
....
```

Example 2-16. Example:

Here the test suite is required to execute the body of each `catch`, or in other way, generate the exception in the `try` body, so that it can be caught.

Advantages: Easy to implement for languages like Java or C++, they support the `try-catch` and the `throw` constructs and enforces it at compilation time.

Disadvantages: Difficult to implement for languages without language support for `try-catch` like C.

Value: High, exceptions are usually not well exercised by the test suite.

2.2.2. Fault Coverage (Adequacy)

Fault-based adequacy criteria focus on the faults that could possibly be contained in the software. The adequacy of a test suite is measured according to its ability to detect such faults. Since it is not possible to know the total number of faults in the program, there are techniques to help in measuring these adequacy criteria[Zhu97]:

1. *Error seeding* is based on the assumption that the artificial faults inserted in the program are as difficult to detect as the normal errors found in the program. This assumption is not generally correct.

2. *Mutation analysis* generates systematically and automatically a larger number of mutants of the program under test. A mutant represents a possible fault. It will be detected by a test case if it produces a behavior different than the original program's behavior. The idea of mutation-adequacy analysis can be extended to specification-based adequacy analysis. There is a high cost associated with this technique. It may require large computation resources to store and execute a large

number of mutants. It may also require human effort to determine if live mutants are equivalent to the original program. Reduction of the testing effort is an acceptable level has been an active research topic. There are variants of this technique such as *Weak mutation testing*, *firm mutation testing* and *ordered mutation testing* have been proposed and researched. In summary although progress has been made to reduce the cost of this technique, there remain open problems.

3. *Perturbation testing* is concerned with the possible functional differences between the program under test and the hypothetical correct program. The adequacy of the test suite is decided by its ability to limit the error space defined in terms of a set of functions.

4. *The RELAY model* analyzes the condition under which a test case reveals a fault. Given a fault, test cases can be selected to satisfy the conditions and hence guarantee the detection of the fault.[Richardson93].

2.2.2.1. Fault model (informal definition)

While we can be certain there are unknown faults in any non-trivial program, we can not know the number or where are these faults in the program [Myers79]. The number of places to look for these faults is a very large number. Any high-yield testing strategy must be guided by a *fault model*[Lamarche98]. A fault model is a set of hypotheses and assumptions on types and locations where faults can occur. It describes potential faults based on common sense, experience, analysis and experiments. If the fault model is complete and realistic then the tests derived from it can detect all the faults in the fault model and provide the appropriate coverage criteria for the testing strategy.

Two general testing strategies and their fault models exist [Binder00]:

1. *Conformance-directed testing* seeks to establish conformance to requirements or specification. Tests are designed to represent the functionality and the features of the system as they are described in the specification. This testing strategy relies on a *non-specific fault model*: any faults found will prevent the system from conforming to the specification. Conformance-directed testing need not consider potential implementation faults in details, it is a way to establish coverage against the requirements of the system. Moreover, language conformance tests are not designed to target exception-handling (high-yield areas) since each compiler may handle them differently. Thus, exception handling tests are left up to the implementor. Thus, by design, language conformance tests are low-yield, i.e., are not intended to detect many defects.
2. *Fault-directed (high-yield) testing* seeks (as described above) to reveal implementation faults. It is motivated by the observation that conformance can be satisfied for an implementation that contain faults. Because the combination of inputs, state, output and paths is astronomically large, efficient probing of the software requires a *specific fault model* to guide the testing effort.

The choice of a fault model suggests a testing strategy. Conformance oriented techniques should be *feature efficient*: they should exercise all specified features. Fault oriented techniques should be *fault efficient*: they should have a high probability of detecting faults.

Finally we give a definition to a *robust* program, it is a failure and fault tolerant, a program that handles unexpected input in a way to minimize performance degradation, data corruption, incorrect output, and abnormal termination.

2.2.3. Function (Requirements) Coverage

To measure how adequately a test suite covers (demonstrates) requirements, software testers often complete four actions. First, they count how many requirements must be covered. This count can be labeled *TR*, it stands for *total requirements* to be covered. Next software testers trace each requirement to all the test cases that exercise that requirement. Then they count all the test cases that the software has processed correctly (that is passed the test cases). When all the test cases that exercise one requirement pass, the requirement is declared *covered*. The count of all the *covered* requirements is called *CR*. Finally, to report on how well requirements were covered, software testers may calculate a requirement coverage factor by dividing *CR* by *TR*. This simple procedure yields an objective measurement of how thoroughly requirements are covered or demonstrated (i.e. test suite quality in terms of requirements coverage) by the test suite under evaluation. This simple metric can be expanded to include parameters of *input domain* and *output range* coverage (Input domain is a set of all possible values that the program can have as input; output range is the set of all possible values that the program can output.) [Poston96]. Again, functional requirements are not completely covered by conformance (black-box) tests. This is already done in practice and is not discussed further in this thesis.

2.3. Discussions and Conclusions

In this chapter we reviewed the basics of coverage in general and code coverage in particular. Although there is a lot of research showing the benefits of applying code coverage measurements, specially code coverage [Mei01], [Frate95] for improving the reliability of software, in practice, code coverage measurements are not used.

Specifically, much commercial software is released with too little testing [Miller01]. Actually [Miller01] goes beyond this fact and proposes a minimal standard level of testing *MSTS* (minimal standard for software testing) below which no software should be released. This standard focuses on source code rather than the specification, in which certain coverage criteria must be met before the software can be released. In our experiment, we meet and exceed the requirements for MSTS decision coverage, since we feel that more rigorous testing is required for compilers. Having said this, we have set realistic and achievable goals and have used measurements that are practical and cost effective to collect.

In the next chapter we will discuss the high-yield strategy that guided our approach to evaluate a test suite using code coverage.

Chapter 3. The High Yield Approach to Test Management

A *high-yield* testing strategy aims at optimizing the testing effort by providing efficient fault detection techniques, thus reducing the cost and increasing the overall quality of the software product. This strategy is *fault-directed*, that is, it is based on hypothesis about where faults are likely to be found. The *Yield* [Humphrey99], [Humphrey95] is the number of high and medium-risk defects detected in a test suite that was generated by a certain testing strategy. *Risk* is the cost associated with a failure caused by a defect. A *high-yield* test case is one that has a high probability of causing a failure, and hence detecting a fault. For example, exception-handling scenarios deal with error-handling behavior in which the software has to deal with incorrect input. It is likely that the designer did not carefully consider all the possibilities. In fact it is generally accepted that exception-handling code is not well documented and not well understood or not very clear in the specification. By comparison a *low-yield* test case, is one that have little or no probability for detecting defects. Tests exploring normal behavior are low-yield, since it only investigates the well-understood and well-documented functionality in the requirements and is likely to be thoroughly considered in design.

The reader might ask the question *doesn't all testing focus on exceptional behavior?*. The answer is *Yes* or at least we hope to be Yes. However not all testing is focused on exceptional behavior, for example conformance testing focuses on showing the correct behavior of the system, not the exceptional behavior, and that is the purpose of such

testing technique. Beside emphasizing on the exceptional behavior, high-yield testing is about increasing the effectiveness of the testing process throughout using proven effective methods.

In this chapter we give a detailed view of the problem we tried to solve, what strategies we used to evaluate the test suite from the high-yield point of view. We give an overview of the test suite used and the system under test (SUT) that motivated this research, plus some high yield areas of the Java source code.

3.1. The Problem

Given a Java compiler (native code compiler, not a bytecode compiler), and a Java conformance test suite which certify the conformity to the following standards:

1. The Java language specification.
2. The Java virtual machine specification.
3. The Java API specification.

Determine:

1. Yield of the test suite, or how effective it is in finding problems in the SUT.
2. Since the implementation is slightly different the one the test suite was designed to test, how suitable it is to test the SUT?
3. Can we use a subset of the test suite to perform a *Sanity* testing? Designers need to execute a related subset of the test suite to the modules they change to check quickly for problems they might introduce by the changes.

Although the first and second item may look as if they deal with the same issue, the first deals with coverage provided to high-yield areas of the code, while the second it considers all the code. We will attempt to give an idea on how to deal with the problem in the third item; we did not have the time to explore it in more details.

3.2. The High Yield Code Coverage Strategy

In our approach each project chooses a minimum percentage of code coverage based on the testing resources available and the reliability requirements for the product. Using statement coverage, decision coverage, and condition/decision coverage, we may want to attain 90%, 80%, and 60% coverage respectively, or more before the releasing. The rational behind selecting such percentages is that condition/decision coverage implies decision coverage, and decision coverage implies statement coverage, the objective for the statement coverage is usually set higher than decision coverage, which in turn is set higher than decision/condition coverage. In the high-yield strategy, we will keep a similar goal for coverage, but we will emphasize [Binder00], [Glass81] certain control constructs listed below because:

1. Designing safe and effective exception handling is difficult, it requires good technical knowledge of the target environment and programming language.
2. Even well-designed and carefully crafted exception handling is more error-prone than application code. Several general kinds of exceptions can occur at any time.

According to the high yield testing strategy, we want to focus on the exceptional behavior of the program. So we chose the areas of the code where this exceptional behavior is handled (or is not handled or missing). Here is the list of the code constructs we chose:

1. null `else` parts of `if` statements. We will add these `null elses` to the `if` statements that were programmed without the *else* part. Then we will look for the once that were not covered/executed. Here is an example of such added else:

```
if(...)                 if(...)
{                       {
  ...                     ...
}                       }
                        else
                        {
                           // null else added.
                        }
```

2. null `default` of `switch` statements. Same here, we add a null default to each `switch` statement that was programmed without having one. We look for the once that were not covered/executed.

```
switch(...)             switch(...)
{                       {
   case ...                case ...
   case ...                case ...
   ...                     ...
   case ...                case ...
}                          default:
                              // null default added.
                        }
```

3. Loop boundaries (none, 1, and more than 1). For each loop we look for whether it has been executed once, non, or twice.

4. Exceptions not caught by the `catch` part of a `try` statement. This means that the test suite did not provide the correct condition for these exceptions. Adding the null `finally` part to `try` to bring to attention whether more processing is required

after an exception was handled (mainly releasing resources that were acquired in the *try* block).

```
try
{
  ...
}
catch(...) { ... }
catch(...) { ... }
...
catch(...) { ... }
```

```
try
{
   ...
}
catch(...) { ... }
catch(...) { ... }
...
catch(...) { ... }
finally
{
   // added null finally
}
```

5. Methods that throw exceptions. In Java, if an exception is not caught or handled in the methods code, the method can be declared to throw ("throws") that exception, this way the caller will handle it. We are looking at this kind of methods that were not executed or called.

```
public void read() throws IOException
{
   ...
}
```

6. Method call pairs. We look for the method calls that were not executed or messages that were not sent. Refer to Call Coverage for an example.

7. Conditions and specially sub-conditions. For each compound condition, we looked at whether all the sub conditions were all evaluated to true and false. Refer to Condition Coverage for an example.

8. The outcome of compound conditions that is harder to get, for example:

```
(A || B) && (C && D)
```

From the truth table of this expression, it is easier to get a false outcome (13 times), and hard to get true outcome (3 times), in this case the later outcome we consider as high-yield.

Plus we gave every block a degree of *importance*, i.e. we differentiate between blocks according to their content, for example a block with an assignment is less likely to fail than a block that is creating an object or starting a thread. We consider such blocks high-yield blocks and we are more interested in measuring their coverage. We implemented this idea in our tool based on keywords, the coverage tool has a table of keywords, each associated with an integer between 1 and 5, if the tool, finds any of the keywords, it marks the block with that integer. Then when we parse the output of the coverage, we can identify these high-yield or important blocks. This idea can be implemented differently based on the Cyclomatic Complexity [McCabe76] of the block, or as a user defined, this can be done by simply entering a comment with some specific syntax, for example:

```
/*** priority=3 */
```

Which can be parsed and used to identify the blocks with that priority.

3.3. Hypotheses:

We will state some hypotheses about a high-yield test suite, and later we will show for the test suite in our project how each is verified.

1. The ratio of positive test cases over negative test cases in a low yield test suite is more than 1.
2. High yield test cases are the negative test cases in a test suite; they uncover or have the potential of uncovering defects in the code. They exercise the exceptional behavior of the system. This behavior is usually not well documented or specified, and not well understood by the designer.
3. Conformance test suites are low yield test suites delete this item? .
4. A test suite designed for one SUT, does not necessarily mean that it would test with the same effectiveness another similar but slightly different SUT.

3.4. The Instrumentation Tool

We built a prototype of an instrumentation tool, using the Java compiler compiler (JavaCC), and a grammar of the Java language, we created a parser, that builds a syntax tree of the given program, then we traverse the tree and insert or modify the nodes so that they satisfy one or more of the coverage criteria mentioned above.

Chapter 4. Case Study and Assessment

In this chapter we will give the results that we collected from the project we worked on with IBM. The implementation (a Java compiler) and the test suite (Java conformance test suite) were provided to us for a limited period of time to perform our analysis. We start by giving a short description and give some basic metrics for the compiler and the test suite.

4.1. The compiler (PBJ)

The Performance Based Java (PBJ) compiler under test is a bytecode to object code compiler. It takes a bytecode file or a *.class* (a machine independent executable) file and it generates a machine dependent object file or a *.obj*, which can be linked to the appropriate libraries to generate an executable file a *.exe* file for the windows operating system. This sounds like a violation to the *machine independent* property of Java since the result of the compilation will be executable only on one operating system/architecture. The motivation behind creating such compiler was performance. Performance is a very important requirement for applications like servers. There is still a need to speed up these applications and one way to do that is to generate a machine object code. Servers usually are built to run on a specific machine, unlike applets, they do not need to be machine independent programs. The reader might ask why Java, why not implement the servers in a different language like C++. Well, here the objective of this project was to allow the user to take advantage of the Java environment with all its

benefits and flexibility, give the user a uniform development environment for both clients and servers and compile existing server programs to speed their execution.

Here is a diagram to show this in more detail:

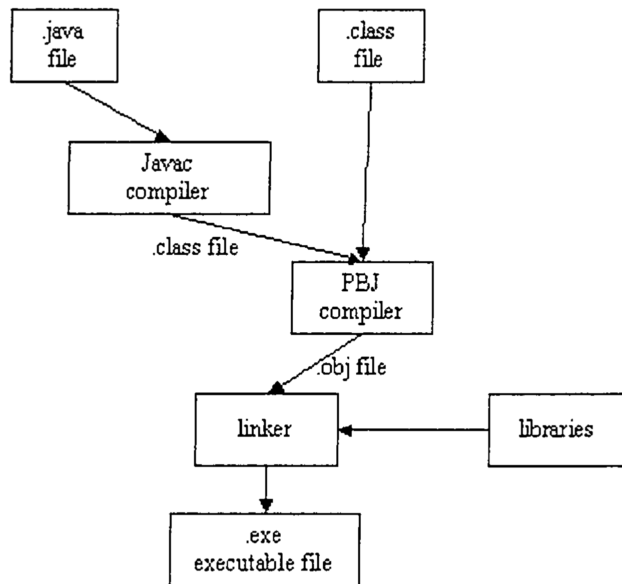


Figure 4-1. Compiling a Java source file or a class bytecode file with the PBJ compiler

As we see in the diagram, *javac* is used to compile a Java source file into a bytecode class file. The PBJ compiler is strictly bytecode to object code compiler; it reuses the functionality already implemented in *javac* to perform the task of compiling the Java file.

In terms of the implementation of the PBJ compiler, it is written in a combination of three languages: Java, C++, and assembly. Here are some metrics for the whole project:

file type	num. of files	LOC

file type	**num. of files**	**LOC**
.java	239	71173
.c	348	156733
.cpp	2	733
.h	313	103831
.hpp	8	2315
.asm	34	3603
total	944	

Table 4-1. PBJ source code metrics

For simplicity we considered the Java part of the project, more efforts are required to examine the entire project. Here are more metrics about the Java part of this project:

package	**num. of classes**	**num. of functions**	**num. of uncommented source statements**
com.pbj.driver	53	531	4704
com.pbj.runtime.common	7	35	318
com.pbj.tools	8	53	684
com.pbj.trans	173	1719	27940
com.pbj.util	23	57	509

package	num. of classes	num. of functions	num. of uncommented source statements
total	264	2395	34155

Table 4-2. More PBJ source code metrics

More data was collected, mainly the Cyclomatic number for each method, but this list is very long to include in here. Here only the methods with a Cyclomatic number greater that 50 are listed:

method	num. of uncommented lines of code	Cyclomatic number	covered
com.pbj.driver.ClassFile.classToObject(String,boolean)	188	75	no
com.pbj.trans.AttrCode.genFlowGraph(int,int[])	705	212	yes
com.pbj.trans.AttrCode.processCode(JavaInfo,int)	903	255	yes
com.pbj.trans.AttrCode.opcEnd(BasicBlock,int,int)	117	61	yes
com.pbj.trans.NativeHeader.genPrototypes()	272	74	yes
com.pbj.trans.Options.processOptions(String[])	291	93	yes
com.pbj.trans.PropArray.applyOpcode(int,int,int)	548	238	no

method	num. of un-commented lines of code	Cyclomatic number	covered
com.pbj.trans.PropBasic.applyOpcode(int,int,int)	479	226	no
com.pbj.trans.PropNull.applyOpcode(int,int,int)	481	236	no
com.pbj.trans.PropNum.applyOpcode(int,int,int)	995	280	yes
com.pbj.trans.VCPool.verifyCPool(ConstantPool)	163	91	yes
com.pbj.trans.Verify.verifyMethod(JavaInfo,byte[],int[],Instruction,int,int,int,java.util.Hashtable,OpLengths,ExceptionInfo[])	176	73	yes
com.pbj.trans.Verify.VerifyByteCode(VBasicBlock,LocalsSignature,VStackSignature,byte[],JavaInfo,JsrControl,CFGraph,java.util.Hashtable,OpLengths)	1512	660	no

Table 4-3. More PBJ source code metrics

As shown above, some of these methods which have high cyclomatic number, are not covered by the test suite. This should sound an alarm to the software testers, a method or a block with high cyclomatic number, has a higher probability of containing faults [Binder00].

4.2. Fault Model

We used the objectives stated in Section 3.2 as our structural fault model for evaluating the test suite. Since we consider them as *high-yield* areas, we expect that a *high-yield* test suite will provide adequate coverage of them. We argue that the lack of coverage of these areas of the code will indicate that the test suite is a *low-yield* test suite. Obviously this is a simple approach, and needs more refinement, ideally by involving the designers, where their experience in this project and previous projects would have helped in predicting what other areas of the program that have a higher probability of containing errors should be added to this fault model. Unfortunately we did not have the chance to interact with the designers, as project constraints did not allow them freedom to take part in such interactions.

4.3. The Test Suite

The test suite is a conformance test suite for the *Java* language. The Java specification is divided into three parts:

1. The Java Language specification.
2. The Java Virtual Machine specification.
3. The Java API specification.

The test suite is structured similarly, containing three main directories for tests belonging to each of the above specifications categories. Table 4-4 shows the number of test cases for each category:

specification	**positive tests**	**negative tests**

specification	positive tests	negative tests
lang	2340	1639 (60% not enough)
vm	1587	864 (33% poor)
api	2576	0 (0% BAD)
total	6503	2503

Table 4-4. Test cases

In the above table, the *api* section does not contain any *negative* test case. This is *BAD*, since it means that all the test cases are normal or expected behavior, and there is not a single test case to exercise the exceptional behavior. From our discussion on the high-yield testing, we know that this part of the test suite is a low-yield, and has a low probability of detecting faults.

Note that *negative* test cases are test cases that the program must fail, i.e. they provide incorrect input and the program should reject them by producing an exception or an error message or both. *Positive* test cases provide correct and valid input which the program must accept, and produce correct output. The following examples illustrate this:

```
/** Runtime, API test case:
    This test case attempts to delete a file from the current directory.
    There is a check to see if the deletion was successful.
    Any exception thrown, will render the test result to inconclusive.
*/

import java.io.*;
import testsuite.Test;

public class positive
{
   File f;
   public int run()
   {
      try
      {
```

```
            f = new File("./test1.txt");
            if(f.exists())
            {
               f.delete();
            }

            if(f.exists())
            {
              return Test.FAIL;
            }
            return Test.SUCCESS;
        }
        catch(Exception e)
        {
           return Test.INC;
        }
    }
}
```

Example 4-1. Positive Test

```
/** Compiler test case:
    This is a negative test case, the compiler must reject this code,
    rationale: you can not have a function without a return type.
*/

import java.io.*;

class negative extends MyClass
{
   public static main()
   {
   }
}
```

Example 4-2. Negative Test

4.4. Results

We run all the tests in the *vm* and *lang* categories on the PBJ compiler. We found that the test cases in the *lang* directory added very little coverage to the one obtained from running the tests in the *vm* directory. This is due to the fact that the PBJ compiler is a bytecode to object code compiler, and does not compile Java source files. Table 4-5 summarizes the results:

coverage criteria	**percent covered**
branch	73
loop	58
method call	81
method pair	78
exception blocks	41
null else/all elses	56
null else/all null elses	26
null default/all defaults	40
null default/all null defaults	23
condition	52
null finally	96

Table 4-5. PBJ coverage metrics using vm test cases

Note that the percentage coverage of null elses, null defaults and exception blocks is very low (this is of concern to us, as we consider these a high-yield areas of the code), we believe that these metrics are most important for *fault* coverage (missing code, design omissions, ...), this is where the faults can hide [Glass81]. Also we believe that tests which exercise such null constructs are *high-yield* test cases, i.e. they are more likely to detect the presence of faults (see chapter 3).

To test for the language specification, we used the Java language grammar, created a parser (described in the next chapter) and instrumented it, and ran all the tests in the *lang* directory. Of course this is only a language syntax test, not a whole language test, since the parser generated from the grammar deals with the syntax of the language only, not the semantics.

Here are the results from the case study:

coverage criteria	**percent covered**
branch	48
loop	52
method call	56
method pair	55
exception blocks	38
null else/all elses	42
null else/all null elses	26
null default/all defaults	48
null default/all null defaults	18
condition	52

coverage criteria	percent covered
null finally	85

Table 4-6. coverage metrics using lang test cases with a Java parser

It is interesting to note that in evaluating the coverage for the same metrics but from the syntax view (we are testing the parser that was generated from the Java grammar, which has a mapping of one to one), we find almost the same distribution and values as we found with the tests for the *vm* category. Here it is obvious why we are getting such low coverage. It is because the *lang* category did not provide many test cases that violated the syntax of the Java language grammar. We think there is a need for such test cases that can be generated systematically and automatically from mutating the Java grammar [Sirer99].

4.5. Determining when the PBJ passed/Failed a test case

One of the test efficiencies which is possible during the process of testing compilers is the use of an established, trusted compiler as a test *oracle*. In this case study we used the *javac* and *java* as our *oracle* to determine the expected output for each test case in the test suite. An oracle in general is a program that performs the same functions as the software under test, but is trusted to return conforming values most of the time. An oracle may be a simulator that mimics the software under test. An ideal oracle would always produce the correct value, but such oracles do not exist in practice. Two practical alternatives: one to use a human (slow, and can make error too), second use a trusted implementation (very cost effective) and investigate the reported fails carefully

(check the test case, check the oracle, and if both are OK then check the compiler for the defect). Figure 4-2 shows the process that we used to determine the actual output for each test case using the available *javac* and *java* software:

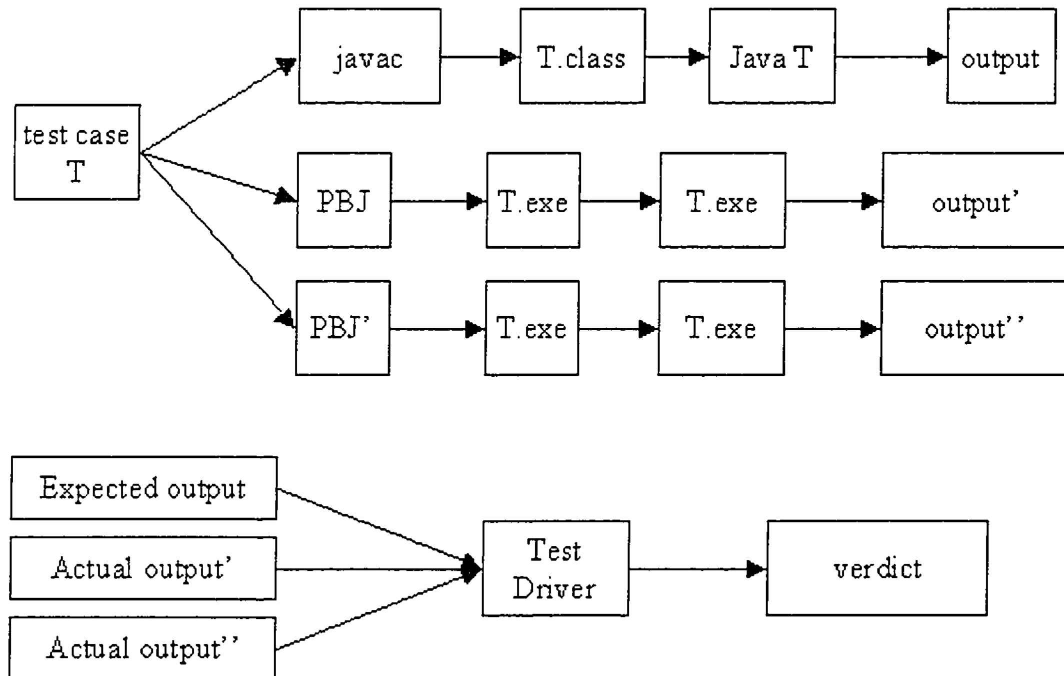


Figure 4-2. Using an oracle to obtain actual output for every test case

By comparing expected *output* and actual *output'* the test driver can declare if the test case has passed or failed. The same process can be used to obtain the expected output for the compiler when it is ported onto different platforms, as shown in Figure 4-2 *PBJ'* will be run on a different platform and *output'* and *output"* will be compared by the test harness to declare the pass/fail verdict.

4.6. Summary and Discussion

From our results we see that the commercial test suite is not a substitute for a fault-directed test suite. Or we at least demonstrated that the commercial test suite can be modified (adding new test cases, and removing tests that don't add any coverage) to adapt it to a fault-directed testing strategy. Although more investigation and more empirical data is needed, we can at least state that our choice of simple coverage techniques was able to detect that the test suite is not high-yield. Thus, we conclude that the case study test suite is not complete and not adequate for testing the PBJ compiler. In fact, we can extend this argument to any other Java conformance test suite and other Java compilers. We believe this statement is true, because of the nature of conformance test suites, which cover the functions specified, and is not a fault-oriented test suite as we fond after our analysis. Also, compatibility between the ported PBJ compiler and the initial release of PBJ can be checked using an oracle and compare the output of the ported compiler and the oracle (the initial release). This may be used to control and manage multiple releases. But further discussions and investigations of this is beyond the scope of this thesis.

Here are some specific observations about the case study:

1. The case study supports our methodology: the compiler test suite was a low-yield test suite as was expected from the beginning. It was designed to be a language conformance test suite, and was not designed to be a high-yield (fault-oriented) test suite. We draw this conclusion based on the actual coverage we measured in this experiment.
2. We showed that the test suite under investigation which is widely accepted in industry as a *conformance test suite*, is virtually *useless* for effective (fault-

oriented) testing. This is expected by the author, but not widely understood in the industrial compiler design community.

3. The low coverage for the *elses*, *defaults*, and specially the *null* parts, should raise a question: *why are these test cases missing?*. These measurements in addition to the coverage measurements for the *null finally* can be used in the *Code Inspection* phase, to raise concerns about these areas and maybe focus attention on what is missing in the code. As [Glass81] and others have pointed out, missing code may have been derived from missing parts in the specification or in the requirements.

4. The published (conformance) test suite is not suitable for robustness testing of the compiler. According to our fault model, it cannot uncover many potential faults in the code since it does not exercise the *high-yield* areas of the code. This test suite is not suitable as is for development (in which developers strive to eliminate most of the faults and errors from the code). However some of its test cases may be useful for building a good regression test suite. <--but it can contribute to a good regression test suite for verification of the essential functionality of the compiler pending review of the coverage results. --> Coverage measurements may be useful in the selection of customized regression tests. This has been studied for general software [Rosenblum97], but so far has not been studied specifically for compilers.

5. The coverage results can be used in *Code Inspection*. For example, a missing `finally` should raise a flag for justification. Perhaps it indicates a potential resource leak (even in Java), and should be present to release these resources used in the `try` block. In the same way, the missing `else` and `default` should be drawn to the attention of software inspectors, and then corrected at an early stage of the system development. In other words, having a large number of missing *elses*,

defaults and finallys may indicate out that the system is not fault-tolerant, for example.

Chapter 5. Coverage Tool Design and implementation issues

In this chapter we give a description of the tool (JCov) we implemented and used to instrument the compiler under test (CUT). We will give a description of the tool's usage, its design and the requirements that guided the design. We wrote the tool in Java and we used JavaCC (Java compiler compiler) tool to generate the parser and token analyzer components of JCov from the Java Language grammar.

5.1. Requirements for an Experimental Code Coverage Tool (JCov)

As described in chapter 2 and shown in Figure 2-1, code coverage measurements are gathered automatically during test execution. This is accomplished by instrumenting the source code of the compiler under test (CUT) with *probes*. *Probes* are method or procedure calls which are strategically inserted (according to the coverage criteria we want to measure) into the code constructs to be covered by the tests in the compiler test suite. When the construct of interest (say the `else` part of an `if` statement) containing the probe is executed, the probe method call is made. The probe method may update a counter, display a message, or write to a log file (also called an execution profile).

Table 5-1 lists the code coverage criteria needed to meet the requirements of our structural fault model listed in the Section 3.2, and the information required to be written to the log file by the probe method:

coverage criteria	**info. written by the probe method to the log file**
branch/decision coverage	probe id
loop coverage	probe id, loop index (0, 1, 2)
condition coverage	probe id, condition index, value of condition (true, false)
method call coverage	probe id, id(entered, exited)
exception coverage	probe id

Table 5-1. Coverage criteria implemented by JCov and probe information

Please note that *exception coverage* is included in *branch/decision coverage* and *methods which throw exception* in the fault model are included in the *method coverage*. In addition to the above requirements the tool expects as input a valid Java file, and by valid we mean it is syntactically correct Java file, usually it must have been compiled without errors by `javac` compiler or a similar tool. The output will be the instrumented Java file: if *JF* is the input Java file, then *JF'* is the output file, where the difference between *JF* and *JF'* is the probes inserted by JCov. Beside the instrumented output file, JCov writes all the probe descriptions (probe id, probe type, ...) to a probe data file. We will explain this in a later section in this chapter. Figure 5-1 shows the process of using this tool to instrument a Java file:

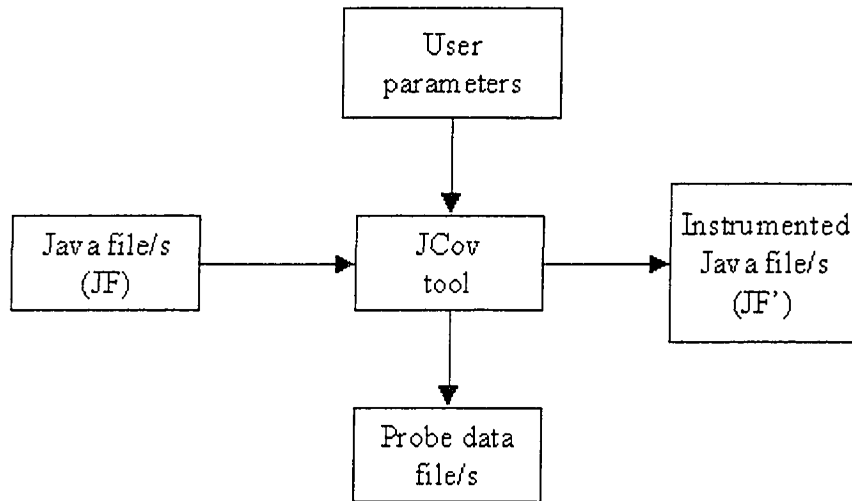


Figure 5-1. Using the coverage/instrumentation tool

Figure 5-2 shows specifically how the JCov tool is used to instrument the compiler under test (CUT).

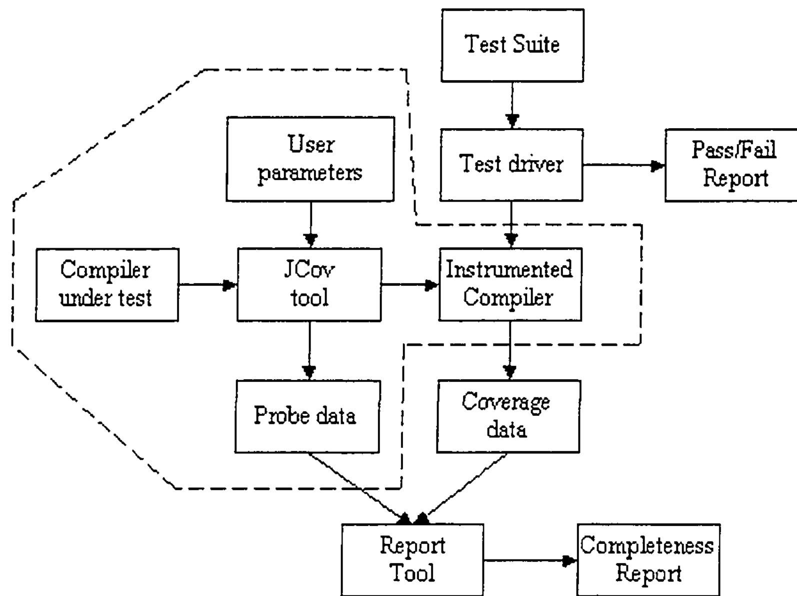


Figure 5-2. Using JCov to instrument the compiler under test (CUT) (Figure 5-1 in dotted area)

5.2. JCov Architecture

The tool consists of several components. Figure 5-3 shows these components. In the following sections, we describe each component and its function briefly.

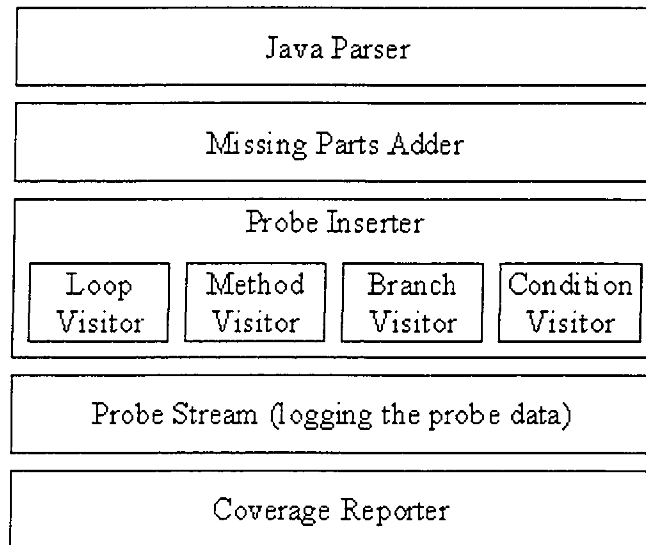


Figure 5-3. JCov components

- *Java parser*: builds a syntax tree from the Java file given as its input. Figure 5-4 shows this component, (A) is how to generate the Parser using JavaCC and the Java language grammar. (B) shows how to generate the syntax Tree using the resulting parser when applied to a Java source file.

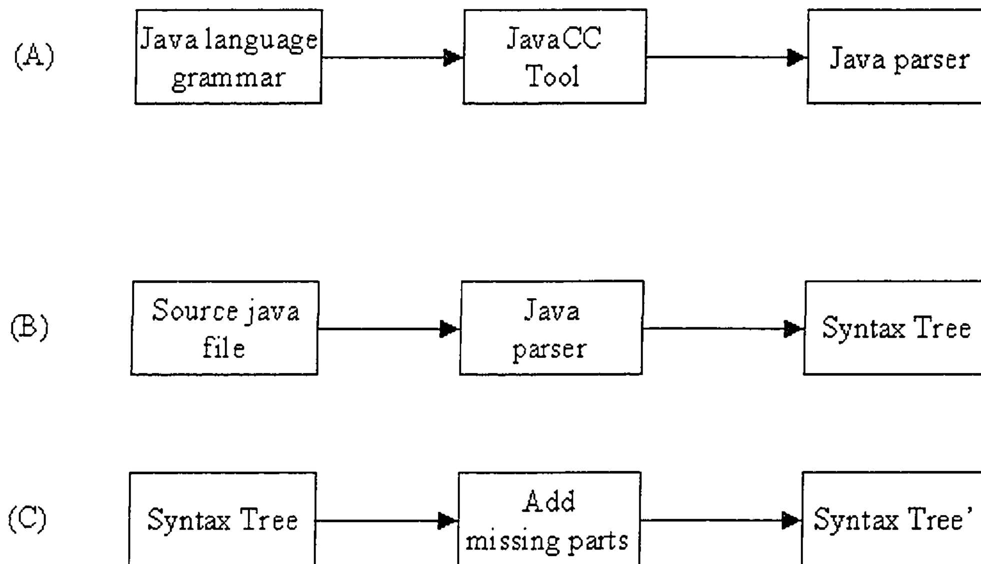


Figure 5-4. The Java parser component of the JCov tool

- *Missing parts Adder*: in this component, we add the null elses, null defaults, null finallys and convert statements in *if*, *while*, *for* and *do-while* to blocks. Table 5-2 illustrates that we will insert `{}` around the statement, this will make all the above statement homogeneous and make the implementation of the coverage component easier. Plus it will add the missing `else` part of the `if` statement.

original code	missing parts added
`if(...)`	`if(...)`

original code	missing parts added

`statement`	`{`
	`    statement`
	`}`
	`else`
	`{`
	`}`

Table 5-2. Add missing parts to an `if` statement

- *Probe Inserter*: This is the code that traverses the syntax tree and inserts the probes according to some criteria. This is accomplished using the *Visitor* design pattern. Four Visitors were implemented, LoopVisitor, MethodVisitor, BranchVisitor and ConditionVisitor which will insert loop, method, branch and condition probes. Appendix A lists the source code for these *Visitors*. Example 5-1, Example 5-2, and Example 5-3 show a BNF grammar template for how to instrument an if statement using the BranchVisitor:

```
IfStatement() :
{ }
{
      "if" "(" Expression() ")"
      Statement()
      [
          LOOKAHEAD(1)
          "else"
          Statement()
      ]
}
```

Example 5-1. BNF grammar for an `if` statement

after performing the "addMissingPart" and the branch coverage, the result will be one of the following:

```
IfStatement() :
{ }
{
      "if" "(" Expression() ")"
      "{"
         Probe()
         Statement()
      "{"
       "else"
       "{"
          Probe()
          Statement()
       "}"
}
```

Example 5-2. `if` with probe inserted, `else` was present

```
IfStatement() :
{ }
{
      "if" "(" Expression() ")"
      "{"
         Probe()
         Statement()
      "{"
       "else"
       "{"
          Probe()
       "}"
}
```

Example 5-3. `if` with probe inserted, `else` was not present

Note that in Example 5-3 the `else` part was added with the `{}` brackets by the `addMissingParts()` and the `Probe()` is inserted by the `BranchVisitor` component.

`Probe()` represents a call to one of the following depends on which visitor was used (what coverage criteria used):

```
kk.jcov.ProbeStream.writeBranch(int file, int probe)
kk.jcov.ProbeStream.writeMethod(int file, int probe, int index)
kk.jcov.ProbeStream.writeLoop  (int file, int probe, int index)
kk.jcov.ProbeStream.writeCondition(int file, int probe, int index, boolean
cond)
```

These are implemented in the `ProbeStream` class. This class has to be available on the `CLASSPATH` for the JVM to find it, or to be compiled into the PBJ executable. The ProbeStream class writes the probe data either to a socket if there is a server listening to the TCP port, or to a compressed file in the `gzip` format. Appendix A lists the source code for the `ProbeStream` class. Each time one of the above methods are called from within the instrumented program, one of the following records will be written to the *Coverage Data* or the log file:

```
"b" file probe
"m" file probe state
"c" file probe index value
"l" file probe index
```

for example:

```
b 10 13
m 10 15 0
m 10 15 1
c 10 112 0 true
c 10 112 2 false
l 10 5 0
l 10 5 1
l 10 5 2
```

The first character in each record identifies the probe type, the `file` and `probe` identify uniquely the individual probes, the `file` identifies uniquely a file, and `probe`

identifies the probe inside that file. After that, the data depends on what type of probe it is, for condition probe the index of the conditions sub-expression, and the value (true or false) are written with each probe. For the case of a method probe, the state is either 0 (enter) or 1 (exit) the method. For the loop probe the index identifies what iteration of the loop the probe is executing in, values are 0, 1 and 2.

- *Coverage Reporter*: takes as input the *Coverage Data* in the format explained in the previous section, and the *Probe Data* which were output by each of the coverage visitors. The probe data file, has the following format:

```
probe = probe type : importance or counter : function calls or conditions :
path to the node
```

for example:

```
11="c":3:[a || b, a, b]:class A/{/void func1()/{/if(a || b)
12="l":10:[print(), write()]:class A/{/void func1()/{/for()
13="m":4:[write(]:class A/{/void func1()
14="b":2:[]:class A/{/void func1()/{/if(a || b)
15="b":5:[]:class A/{/void func1()/{/if(a || b)/null else
```

From the above example, we see that for each probe, the information is almost the same except the second integer, is the importance factor for the loop, branch and method probes, and is the number of sub-conditions for the condition probe. The array in between `[]` is the list of function calls in that branch, method, or loop, while it is a list of all the sub conditions for the condition probe.

The coverage reported takes these files and process them and outputs the following report:

```
Probe(b, 10, 15) was not covered (null else).
Probe(m, 10, 13, enter) was not covered.
Probe(m, 10, 13, exit) was not covered.
```

```
Probe(c, 10, 11, 2) was covered only to true.
Probe(1, 10, 12, 2) was not covered for 0, 1.
....
```

5.3. Invoking JCov

JCov is written in Java as we mentioned earlier, so to execute it, JDK 1.1 or later must be installed. It will run on any platform supporting Java, although we used it on the Windows platform only. The following show how to invoke the tool, and its command like options:

```
usage:
java kk.jcov.Tool -a -c -m -l -b --listfile list_file -i "java" dir1 dir2
....
```

Where:

-a

add the missing parts, i.e. adding the *null* else, default and finally parts to the if, switch and try statements respectively (see later for details).

-c

perform decision and condition coverage.

-m

perform method coverage.

-b, -l

perform branch/loop coverage.

-i

include ".java" files in the directories that follow.

file1 file2 ...

the files to process, if it is a directory, then process all the files under that directory.

5.4. Testing the instrumentation tool

Given that the tool was intended only to be used in this experiment, the effort that went into testing it is limited. Even so, the design is sound and generic enough that it can be used for measuring code coverage for any other Java application/compiler. We use here the strategy of *Testing in the Small* [Menzies00], i.e. the testing effort is tailored to the program and its application domain. Our approach was:

1. First test the JCov tool with the language specification (conformance) test section *lang* of the test suite. This was a by-product of our attempt to measure the code coverage of the test cases in the *lang* section of the test suite used in our experiment.
2. However, this section did not provide good coverage for either the tool or the Java grammar. Moreover it did not allow us to verify that the probes inserted are correct and are inserted in the correct places in the program. Thus, we used a

shortcut approach: since the parser (JCov) parses the Java code into trees, we create a tree of the file before instrumentation, and a tree after instrumentation. Then we compare the two trees. Since all the leaf nodes in these trees are tokens, the only differences between the two trees should be the added tokens and probes. In addition, we can get the path from the root node to the leaf for each node in the tree, and from that, we can determine and verify the correct or incorrect position of the probe and the tokens added by the `addMissingParts` component of the tool. Here we give an example to demonstrate this idea.

`[kk]`

We used this method to test the tool *on the spot* with each Java file. It is an iterative method: parse and insert probes, compare the two trees, if a problem is found, then fix it in the tool, and rerun it again on that file. We found that this method was convenient since it did not require us to create a large test suite in order to adequately test the tool.

5.5. Summary

Although we could have used an existing code coverage tool to obtain similar measurements, we chose to design and develop one since the tools available at the time were not adequate for adding the different missing parts of the various constructs. Also existing tools did not allow or support the idea of differentiating between the different types of blocks by giving them an *importance factor* or *priority*. In addition, our approach illustrates precisely what is involved in implementing such tools and in

applying them to gain important knowledge of the implementation under test. As well, the design is generic, and may be applied to build a toolset to assess the quality (completeness and adequacy) of any language test suite.

Chapter 6. Conclusions and Suggestions for Futher Work

We have contributed a new, fault-oriented approach to evaluating a test suite designed to test compilers and translators. Unfortunately we did not have enough time on site in the internship period, to use the data from this experiment to help improve the adequacy of this particular compiler test suite. As often occurs in empirical Software Engineering research, continually changing industrial constraints did not allow us to interact directly with designers and convey some of our ideas to help their testing process. We know that this work is just a start and expect it will be extended in the future. Here we will list some ideas of what can be done to extend this work and some conclusions drawn from the results we obtained from our experiments.

6.1. Conclusion

The thesis addressed the problem of evaluating the quality of an industrial test suite for compilers, formulated a high-yield code-coverage hypothesis, presented a high-yield white-box compiler test methodology, and described the design and the implementation of a support tool, JCov. Then as a case study, a commercial test suite was applied to an instrumented industrial compiler under development to investigate the high-yield hypothesis and to illustrate the feasibility and usefulness of this approach to compiler test suite evaluation.

The specific contribution of the thesis are the following:

1. A new high-yield directed approach is presented for evaluating the fault-coverage of a commercial compiler test suite.
2. New code-coverage criteria are presented for evaluating the code coverage of a Java test suite, namely *null else*, *null defaults*, *null finally*, and *method pairs* (considering polymorphism and inheritance).
3. A generic parser and probe insertion tool, JCov, is given for automating the instrumentation of the compiler code. Design rational and details are also presented.
4. A preliminary case study of the methodology is presented. In this case study we successfully applied the method to a commercial compiler test suite running against an industrial compiler under development. Our approach demonstrated conclusively that test suite used in the case study, which is accepted word-wide as a conformance test suite is virtually *usesless* for effective testing (fault-oriented testing).
5. The methodology appears to be directly applicable to *Code Inspection* and has potential usefulness for regression test case selection for compilers, but this will require further investigation.
6. Coverage can be helpful in selecting regression test cases, give indications about missing test cases, duplicate test cases, and can even be helpful in showing missing requirments or not well state requirments specially in the case of the missing parts or the fall-through constructs of the language.

6.2. Suggestions for Further Work

Below are some suggestions on what should be done to continue the work started in this thesis:

1. Improve the tool with respect to memory usage and performance and include other measurements like data flow coverage.
2. Improve the tool to include functionality to collect information about inheritance and method overloading. This will be useful for the method call pair coverage. [Binder00]
3. Implement a similar tool to automatically generate the negative (high-yield) test cases that were lacking in the test suite from the Java grammar through the usage of Grammar mutation or the method described in [Sirer99].
4. Use the results of this experiment, specifically the identification of areas of the code that were not covered, to add test cases to the test suite to increase the overall coverage. Also use the same data to remove the test cases that did not add coverage. This should be done with care, consulting the test cases and the specification is recommended here before eliminating any test case. The reduced test suite can be used in regression. [Rosenblum97]
5. Provide for repeated use of this tool in successive releases of a software work product and track the defect detection history and the test suite modification history. (Also conduct experiments to validate the usefulness of detecting high-yield regions in the source code during Code Inspection).
6. Conduct a study of a *freeware* software program, where the code is available and there are no restrictions to accessing and using it.

7. Conduct more experiments and investigate how to use this tool to select effective regression test cases. We believe that a good regression test suite should provide a certain reasonable level of test coverage.

8. Extend the tool to enable and support the assignment and use of user defined *importance/weight* for each block/function.

References

[Binder00] Robert V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*, 2000, 0-201-80938-9, Addison-Wesley.

[Frate95] Fabio Del Frate and Praerit Garg, *On the Correlation between Code Coverage and Software Reliability*, 1995, IEEE Trans. on Software Engineering, No. 95, pp. 124-132.

[Humphrey99] Watts S. Humphrey, *Introduction to the Personal Software Process*, 1999, Addison-Wesley, 0-201-54809-7.

[Humphrey95] Watts S. Humphrey, *A Discipline for Software Engineering*, 1995, Addison-Wesley, 0-201-54610-8.

[Gamma95] Erich Gamma, Richard Helm, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1995, Addison-Wesley, 0-201-63361-2.

[Glass81] Robert L. Glass, *Persistent Software Errors*, 1981, IEEE Trans. on Software Engineering, Vol. 7, No. 2, pp. 162-168.

[Gutjahr99] Walter J. Gutjahr, *Partion Tesing vs. Random Testing: The Influence of Uncertainty*, 1999, IEEE Trans. on Software Engineering, Vol. 25, No. 5, pp. 661-674.

[Jorgensen95] Jorgensen C. Paul, *Software Testing: A Craftsman's Approach*, 1995, © by CRC Press, Inc.

[Koo96] Koo Irene, *Mutation Testing and Three Variations: http://www.ee.ubc.ca/home/comlab1/irenek/etc/www/techpaps/mutate/mutation.html*, 1996, © by CRC Press, Inc.

[Lamarche98] Rodd C. Lamarche, Robert L. Probert, Vojislav D. Radonjic, and et al, *High-Yield Strategies for Testing Object-Oriented Software*, 1998, World Multiconference on Systemics, Cybernetics and Informatics (SCI'98) and 4th Intl. Conference on Information Systems Analysis and Synthesis (ISAS'98) , pp. 470-477.

[Li98] Michael Naixin Li, Yashwant K. Malaiya, and Jason Denton, *Estimating the Number of Defects: A Simple and Intuitive Approach*, 4-7 Nov. 1998, ISSRE98: International Symposium on Software Reliability Engineering, Paderborn Germany.

[Marick95] Marick Brian, *The Craft of Software Testing: Subsystem Testing*, 1995, © by Prentice Hall.

[McCabe76] Thomas J. McCabe, *A Complexity Measure*, 1976, IEEE Trans. on Software Engineering, Vol. 2, No. 4, pp. 308-320.

[Mei01] Chen Mei-Hwa, *Effect of Code Coverage on Software Reliability Measurement*, 2001, IEEE Trans. on Reliability, Vol. 50, No. 2, pp. 165-170.

[Menzies00] Tim Menzies and Bojan Cukic, *SE in the small: When to test less*, September/October 2000, IEEE Software, pp. 107-112.

[Miller01] Keith W. Miller, *A Modest Proposal for Software Testing*, 2001, IEEE Software, March, No. 4, pp. 98-100.

[Myers79] Glenford J. Myers, *The Art of Software Testing*, 1979, © by John Wiley & Sons, Inc.

[Offutt96] Jefferson A. Offutt, *A Experimental Evaluation of Data Flow and Mutation Testing*, 1996, Software-Practice and Experience, Vol. 26(2), pp. 165-176.

[Poston96] Robert M. Poston, *Automating Specification-Based Software Testing*, 1996, 0-8186-7531-4, by the IEEE Computer Society Press.

[Probert82] Robert L. Probert, *Optimal Instrumentation of Software*, 1982, IEEE Trans. on Software Engineering, Vol. 8, No. 1, pp. 63-74.

[Richardson93] Debra J. Richardson, *An Analysis of Test Data selection Criteria Using the RELAY Model of Fault Detection*, 1993, IEEE Trans. on Software Engineering, Vol. 19, No. 6, pp. 533-553.

[Roach89] Kerwyn M. Roach, *An Adaptive LCSAJ Test Strategy*, 1989, Masters Thesis of Computer Science, University of Ottawa.

[Rosenblum97] David S. Rosenblum, *Using Coverage Information to Predict the Cost-Effectiveness of Regression Testing Strategies*, 1997, IEEE Trans. on Software Engineering, Vol. 23, No. 3, pp. 146-158.

[Saleh94] A. Boujarwah and K. Saleh, *Compiler test suite: evaluation and use in an automated test environment*, 1994, Information and Software Technology, Vol. 4, No. 10, pp. 607-614.

[Saleh99] A. Boujarwah, K. Saleh, and J. Al-Dallal, *Testing syntax and semantic coverage of Java language compilers*, 1999, Information and Software Technology, No. 41, pp. 15-28.

[Sirer99] Emin G. Sirer and Brian N. Bershad, *Using Production Grammars in Software Testing*, 1999, Proceedings of the 2nd conf. on Domain Specific Languages, Austin Texas, pp. 1-13.

[Zhu97] Hong Zhu, *Software Unit Test Coverage and Adequacy*, 1997, ACM Computing Surveys, Vol. 29, No. 4, pp. 366-427.

[Antlr] *Antlr: A parser generator for Java and C++: http://www.antlr.org.*

[Coverage] *Code Coverage Analysis: http://www.bullseye.com/webCoverage.html.*

[JavaCC] *Description of the JavaCC Grammar File: http://www.webgain.com/products/metamata/java_doc.html.*

[JDKSpecs] *The Java Language Specification (1.1.8): http://java.sun.com/.*

[JDKDocs] *The Java Development Kit (JDK) 1.1.8: http://java.sun.com/products/jdk/1.1.*

[JTest] *Jtest: Automatic Error Prevention and Error Detection for Java Developers: http://www.parasoft.com/products/jtest/papers/jtestwp_3.html.*

Appendix A. JCov Program Listing

In this appendix we list some of the code that makes part of the *JCov* tool. All the code with the documentation, explanation on use and build will be included on a diskette that will accompany this thesis. The tool make use of some *Free Software* namely the *GNU* GetOpt package to process command line arguments and the *GNU* regexp, which is the Java implementation of the regular expressions used by the tool to search and compare the results.

A.1. Tree Nodes

When the parser creates a syntax tree, every node is implemented as a different class, for example, the *if* statement, is represented by the `KKIfStatement` class. This in turns contains two nodes, one for the *if* part, and the other for the *else* part. Here is the code for each of these classes:

```
package kk.jcov;

import java.io.*;
import java.util.*;

/* this class represents an if statement. The number of children is either
   1 (no else part), or 2 (with an else part).
   its children can be instances of classes KKIfNode and KKElseNode only.
 */
public class KKIfStatement extends SimpleNodeWHF
{
  public KKIfStatement(int id)
  {
    super(id);
  }
```

```
  /* if there is no else part in this if statement, then add a
     null else
   */
  private void addMissingElse()
  {
     if (getNumChildren() != 2)
     {
       KKElseNode els = new KKElseNode(KKELSENODE, null);
       this.addChild(els);
     }
  }

  /* a wrapper for the addMissingElse(), to make it polymorphic with
     the other classes in this package.
   */
  public void addMissingPart()
  {
     this.addMissingElse();
  }
}
```

Example A-1. KKIfStatement class

```
package kk.jcov;

import java.io.*;
import java.util.*;

import kk.probe.*;

/* This class represents the if part of an if statement.
 */
public class KKIfNode extends SimpleNodeWHF
{
  public KKIfNode(int id)
  {
    super(id);
  }

  /* this is a wrapper for the addChild() for this Node. It checks to see if
the
     Node to be added is a simple statement or is a block, if it is a simple
     statement, i.e. not a block, then create a block to wrap the statement
with.
```

```
      this has the effect of putting {} around the simple statements.
    */
   public void addStatement(Node n)
   {
      if (n.getID() != KKBLOCK)
      {
        KKBlock blk = new KKBlock(KKBLOCK, n);
        super.addChild(blk);
      }
      else
      {
        super.addChild(n);
      }
   }

   /* assign the condition variable to point to the condition part of the
      if statement. The code in the condition coverage references this vari-
able.
    */
   Node condition = null;
   public void addCondition(Node n)
   {
     condition = n;
     addToHeader(n);
   }

   /* force the user to use addStatement instead of addChild.
    */
   public void addChild(Node n)
   {
     System.err.println("addChild: should not use in: " +
this.getClass().getName());
   }

   /* for this node, we can apply a condition coverage and a branch coverage,
hence the two
      variables.
    */
   CProbe conditionProbe = null;
   BProbe branchProbe = null;

   /* this is the accept method that accepts a visit from the visitor, it
calls the
      visitor's visit() method passing the correct type of "this"
    */
   public void accept(Visitor v, Probe p)
   {
     v.visit(this, p);
     super.accept(v, p);
```

```
  }
}
```

Example A-2. KKIfNode class

```
package kk.jcov;

import java.io.*;
import java.util.*;

import kk.probe.*;

/* this class represents the else part of the if statement. The else part can
be null,
   this will happen when the if statement is parsed without an else part, and
we add
   a null else explicitly
 */
public class KKElseNode extends SimpleNodeWHF
{
  /* this to identify if this else part is a null else or not */
  boolean nullElse = false;

  public KKElseNode(int id)
  {
    super(id);
  }

  /* this is a wrapper for the addChild() for this Node. It checks to see if
the
     Node to be added is a simple statement or is a block, if it is a simple
     statement, i.e. not a block, then create a block to wrap the statement
with.
     this has the effect of putting {} around the simple statements.
   */
  public void addStatement(Node n)
  {
     if (n.getID() != KKBLOCK)
     {
       KKBlock blk = new KKBlock(KKBLOCK, n);
       super.addChild(blk);
     }
     else
     {
```

```
        super.addChild(n);
      }
  }

  /* force the user to use the addStatement method() */
  public void addChild(Node n)
  {
    System.err.println("addChild: should not use in: " +
this.getClass().getName());
  }

  /* creates a new else branch with null block.
   */
  public KKElseNode(int id, Node n)
  {
    super(id);

    nullElse = true;

    this.addToHeader(Token.elseToken());

    KKBlock b = new KKBlock(KKBLOCK, null);
    this.addStatement(b);
  }

  /* only branch coverage is needed for this node. */
  BProbe branchProbe = null;

  /* this is the accept method that accepts a visit from the visitor, it
calls the
      visitor's visit() method passing the correct type of "this"
   */
  public void accept(Visitor v, Probe p)
  {
    v.visit(this, p);
    super.accept(v, p);
  }
}
```

Example A-3. KKElseNode

as shown in the examples above, the same is done for all other nodes in the syntax tree. Each node representing a different language construct. We found this to be useful, in case future additions to the tool were required. For example, more coverage, or

extracting metrics from the code, such as *the list of subclasses* of a given class in a certain package or packages. Another example will be, counting how many static variables are there and where they are in the code. Such metrics might be useful automate the detection of *code inspection* metrics.

A.2. The Visitor Classes

In the design of the tool, we decided to use the *Visitor* design pattern to implement all the coverage functionality. The rational behind this decision is:

1. Centralize the coverage code in one class, for example all the code that deals with *condition* coverage is in the `ConditionVisitor` class. Without using a visitor this code will be scattered in many possibly unrelated classes (this may be hard to maintain).
2. Same visitors can be applied to a different syntax trees (reuse). Here the same visitors can be used to add probes in another syntax tree representing another language (for example C++). Without the visitors we will have to implement the same functionality in every syntax tree of every different language we decide to parse and use the tool for.
3. If the syntax tree changes (language evolution as in Java), the visitors might not need to be changed.
4. It allows the parser, the syntax tree nodes and the visitors to be developed and tested separately.

5. And finally, the visitors can be applied to a syntax tree where we do not own the code. As long as every node implements the `accept()` method, we can perform the coverage even if we do not have the source code for the nodes in the syntax tree.

Here are some examples of the different visitors we implemented for the *JCov* tool:

```
package kk.jcov;

import kk.probe.*;

/* inserts loop probes into while, for and do-while loops.
   it converts a loop form:

   while(...)
   {
     ...
   }

   to:

   int counterXXX = 0;
   writeLoop(...);
   while(...)
   {
     if(countXXX < 3) writeLoop(...);
     ...
   }

   Note that if the loop is part a labeled statement, then add
   the declaration of the counter before the labeled statement.
 */
public class LoopVisitor extends Visitor
{
  /* insert a loop probe into a do-while statement.
   */
  public void visit(KKDoStatement n, Probe p)
  {
     /* already covered?
      */
     if(n.loopProbe == null)
     {
       n.loopProbe = new LProbe(n);

       /* check to see if parent is a labeled statement, if so
          then add the counter declaration before the labeled
```

```
           statement.
          */
        if(n.parent.getID() == JCovTreeConstants.KKLABELEDSTATEMENT)
        {
          KKLabeledStatement parent = (KKLabeledStatement) n.parent;
          parent.header.insertElementAt(n.loopProbe.getVarToken(), 0);
        }
        else
          /* just insert the declaration before the header of this node.
           */
          n.header.insertElementAt(n.loopProbe.getVarToken(), 0);
        this.addProbe(n, n.loopProbe);
      }
  }

  /* insert a loop probe into a for statement.
   */
  public void visit(KKForStatement n, Probe p)
  {
     /* already covered?
      */
     if(n.loopProbe == null)
     {
       n.loopProbe = new LProbe(n);

       /* check to see if parent is a labeled statement, if so
          then add the counter declaration before the labeled
          statement.
        */
       if(n.parent.getID() == JCovTreeConstants.KKLABELEDSTATEMENT)
       {
         KKLabeledStatement parent = (KKLabeledStatement) n.parent;
         parent.header.insertElementAt(n.loopProbe.getVarToken(), 0);
       }
       else
         /* just insert the declaration before the header of this node.
          */
         n.header.insertElementAt(n.loopProbe.getVarToken(), 0);
       this.addProbe(n, n.loopProbe);
     }
  }

  /* insert a loop probe into a while statement.
   */
  public void visit(KKWhileStatement n, Probe p)
  {
     /* already covered?
      */
     if(n.loopProbe == null)
```

```
      {
        n.loopProbe = new LProbe(n);

        /* check to see if parent is a labeled statement, if so
           then add the counter declaration before the labeled
           statement.
         */
        if(n.parent.getID() == JCovTreeConstants.KKLABELEDSTATEMENT)
        {
          KKLabeledStatement parent = (KKLabeledStatement) n.parent;
          parent.header.insertElementAt(n.loopProbe.getVarToken(), 0);
        }
        else
          /* just insert the declaration before the header of this node.
           */
          n.header.insertElementAt(n.loopProbe.getVarToken(), 0);
        this.addProbe(n, n.loopProbe);
      }
  }

  /* common function used by the visit methods in this class.
     it basically looks for the first "{" in the children, and
     it inserts the following code after it:

     if(counterXXX < 3) writeLoop(...);
   */
  private void addProbe(Node node, LProbe probe)
  {
    KKBlock b = (KKBlock)node.getChild(0);

    int i;
    for(i=0; i < b.getNumChildren(); i++)
      if(b.getChild(i).getID() == JCovTreeConstants.KKTOKENID &&
        ((Token)b.getChild(i)).kind == JCovConstants.LBRACE ) break;

    b.addChild(probe.getToken(), i+1);
  }
}
```

Example A-4. LoopVisitor class

```
package kk.jcov;

import kk.probe.*;
```

```
/* inserts probes at the start and the exit points of a method or a construc-
tor
 */
public class MethodVisitor extends Visitor
{
  /* check to see if there is a constructor call? if so, then add the probe
just
     just after the call otherwise put it after the opening "{"
     Notice the difference between the method and the constructor, Java does
not allow
     any code before the call to the super class's constructor, this is basi-
cally
     why the two differ.
   */
  public void visit(KKConstructorDeclaration n, Probe p)
  {
     /* already covered? */
     if (n.methodProbe == null)
     {
       n.methodProbe = new MProbe(n);
       int i;
       if(n.constructorCall != null)
       {
         // insert the probe after the constructor call:
         for(i=0; i < n.getNumChildren(); i++)
           if(n.getChild(i) == n.constructorCall) break;

         n.addChild(n.methodProbe.getBeginToken(), i+1);
       }
       else
       {
         // if no constructor call, then add it after the first "{":
         for(i=0; i < n.getNumChildren(); i++)
           if(n.getChild(i).getID() == JCovTreeConstants.KKTOKENID &&
              ((Token)n.getChild(i)).kind == JCovConstants.LBRACE ) break;

         n.addChild(n.methodProbe.getBeginToken(), i+1);
       }

       if(! n.isTerminating())
       {
          // if it is not terminating, then add the probe before
          // the "}":
         for(i=0; i < n.getNumChildren(); i++)
           if(n.getChild(i).getID() == JCovTreeConstants.KKTOKENID &&
              ((Token)n.getChild(i)).kind == JCovConstants.RBRACE) break;

         n.addChild(n.methodProbe.getEndToken(), i);
```

```
        }

        // we need this even if it is a duplication of the accept in the node
n
        // this is how we make the return and throw statement know about the
Probe
        // otherwise it will be null, and not the probe that we want.
        // should change this and make it a call in the return statement,
        // getParentMethod for example ....
        n.accept(this, n.methodProbe);
      }
  }

  public void visit(KKMethodDeclaration n, Probe p)
  {
     /* already covered? */
     if (n.methodProbe == null)
     {
       // this is so that we do not add probe after ";" and empty method.
       if(n.getChild(0).getID() == JCovTreeConstants.KKBLOCK)
       {
         KKBlock b = (KKBlock)n.getChild(0);
         n.methodProbe = new MProbe(n);

         int i;

         for(i=0; i < b.getNumChildren(); i++)
            if(b.getChild(i).getID() == JCovTreeConstants.KKTOKENID &&
               ((Token)b.getChild(i)).kind == JCovConstants.LBRACE ) break;

         b.addChild(n.methodProbe.getBeginToken(), i+1);

         if( ! b.isTerminating())
         {
           for(i=0; i < b.getNumChildren(); i++)
              if(b.getChild(i).getID() == JCovTreeConstants.KKTOKENID &&
                 ((Token)b.getChild(i)).kind == JCovConstants.RBRACE ) break;

           // add the probe only if the last statement in the method is
           // not a return statement:
           if(b.getChild(i-1).getID() != JCovTreeConstants.KKRETURNSTATEMENT)
              b.addChild(n.methodProbe.getEndToken(), i);
         }

       // we need this even if it is a duplication of the accept in the Node
n
       // this is how we make the return and throw statement know about the
Probe
       // otherwise it will be null, and not the probe that we want.
```

```
        // should change this and make it a call in the return statement,
        // getParentMethod for example ....
          b.accept(this, n.methodProbe);
        }
      }
  }

  /* adds the probe before the return statement */
  public void visit(KKReturnStatement n, Probe p)
  {
     /* already covered? */
    if(n.methodProbe == null)
    {
       if(p == null)
          System.out.println("Return getting a null probe ...");
       n.methodProbe = (MProbe)p;
       n.addChild(n.methodProbe.getEndToken(), 0);
    }
  }

  /* adds the probe before the throw statement */
  public void visit(KKThrowStatement n, Probe p)
  {
     /* already covered? */
    if(n.methodProbe==null)
    {
       if(p == null)
          System.out.println("Throw getting a null probe ...");
       n.methodProbe = (MProbe)p;
       n.addChild(n.methodProbe.getEndToken(), 0);
    }
  }
}
```

Example A-5. MethodVisitor class

```
package kk.jcov;

import kk.probe.*;

/* inserts a probe in every block (that is after converting every
   simple statement to a block statement in the compound statements.
   Note that when crating a branch Probe, a string identifying that
   branch is used and it will be saved in the probe file.
```

```
   for example, "null else" will be added to the probe if that
   else part did not exist in the original code. same for the
   default and finally parts.
 */
public class BranchVisitor extends Visitor
{
  /* insert a writeBranch(...) after the block start, i.e "{" */
  public void visit(KKDoStatement n, Probe p)
  {
     // if u have already covered the loop, then there is no need
     // to cover the branch:
     if(n.loopProbe != null)
        return;

     /* already covered? */
     if(n.branchProbe == null)
     {
        n.branchProbe = new BProbe(n, "");
        this.addProbe(n, n.branchProbe);
     }
  }

    /* insert a writeBranch(...) after the block start, i.e "{" */
  public void visit(KKForStatement n, Probe p)
  {
     // if u have already covered the loop, then there is no need
     // to cover the branch:
     if(n.loopProbe != null)
        return;

     /* already covered? */
     if(n.branchProbe == null)
     {
        n.branchProbe = new BProbe(n, "");
        this.addProbe(n, n.branchProbe);
     }
  }

  /* insert a writeBranch(...) after the block start, i.e "{" */
  public void visit(KKWhileStatement n, Probe p)
  {
     // if u have already covered the loop, then there is no need
     // to cover the this branch:
     if(n.loopProbe != null)
        return;

     /* already covered? */
     if(n.branchProbe == null)
     {
```

```
            n.branchProbe = new BProbe(n, "");
            this.addProbe(n, n.branchProbe);
         }
      }

      /* insert a writeBranch(...) after the block start, i.e "{" */
      public void visit(KKCatchNode n, Probe p)
      {
         /* already covered? */
         if(n.branchProbe == null)
         {
            n.branchProbe = new BProbe(n, (n.nullFinally ? "null finally" : ""));
            this.addProbe(n, n.branchProbe);
         }
      }

      /* insert a writeBranch(...) after the block start, i.e "{" */
      public void visit(KKElseNode n, Probe p)
      {
         /* already covered? */
         if (n.branchProbe == null)
         {
            n.branchProbe = new BProbe(n, (n.nullElse ? " null " : "") + " else
part");
            this.addProbe(n, n.branchProbe);
         }
      }

      /* insert a writeBranch(...) after the block start, i.e "{" */
      public void visit(KKIfNode n, Probe p)
      {
         /* already covered? */
         if (n.branchProbe == null)
         {
            n.branchProbe = new BProbe(n, "");
            this.addProbe(n, n.branchProbe);
         }
      }

      /* insert a writeBranch(...) after the ":" of the case part:
       */
      public void visit(KKSwitchLabel n, Probe p)
      {
         /* already covered? */
         if(n.branchProbe == null)
         {
            n.branchProbe =  new BProbe(n, (n.nullDefault ? " null ": "") + "de-
fault");
            n.addChild(n.branchProbe.getToken(), 0);
```

```
      }
    }

    /* insert a writeBranch(...) after the block start, i.e "{" */
    public void visit(KKTryNode n, Probe p)
    {
      /* already covered? */
      if (n.branchProbe == null)
      {
          n.branchProbe = new BProbe(n, "");
          this.addProbe(n, n.branchProbe);
      }
    }

    /* common function used to all the probe after the "{" token
     */
    private void addProbe(Node node, BProbe probe)
    {
      KKBlock b = (KKBlock)node.getChild(0);

      int i;
      for(i=0; i < b.getNumChildren(); i++)
        if(b.getChild(i).getID() == JCovTreeConstants.KKTOKENID &&
          ((Token)b.getChild(i)).kind == JCovConstants.LBRACE ) break;

      b.addChild(probe.getToken(), i+1);
    }
}
```

Example A-6. BranchVisitor class

```
package kk.jcov;

import kk.probe.*;

/* this class implements the functionality to add probes on
   conditions and subconditions. for example the following condition

   a && b || c

   will be converted to

   writeCondition(...,
       writeCondition(...,
```

```
            writeCondition(..., a)
            &&
            writeCondition(..., b)
        )
        ||
        writeCondition(..., c)
    )

    The writeCondition() method writes the details of the probe
    and returns the boolean value of the boolean expression passed to it
  */
public class ConditionVisitor extends Visitor
{
   /* surround the condition in this statement with
      the tokens needed to create the call to
      writeCondition call, then recursively
      cover the sub conditions.
    */
   public void visit(KKDoStatement n, Probe p)
   {
      /* already covered? */
      if(n.conditionProbe == null)
      {
        n.conditionProbe = new CProbe(n);
        n.conditionProbe.addCondition(n);
        int count = n.conditionProbe.numOfConditions();
        n.condition.accept(this, n.conditionProbe);
        n.conditionProbe.writeProbe();
        n.condition.addChild(n.conditionProbe.getBeginToken(count), 0);
        n.condition.addChild(n.conditionProbe.getEndToken());
      }
      else
      {
         System.err.println("Node already covered ...");
      }
   }

  /* surround the condition in this statement with
     the tokens needed to create the call to
     writeCondition call, then recursively
     cover the sub conditions.
   */
  public void visit(KKForStatement n, Probe p)
  {
     /* already covered?  if not, does this for loop
        have a condition?
      */
     if(n.conditionProbe == null && n.condition != null)
     {
```

```
      n.conditionProbe = new CProbe(n);
      n.conditionProbe.addCondition(n);
      int count = n.conditionProbe.numOfConditions();
      n.condition.accept(this, n.conditionProbe);
      n.conditionProbe.writeProbe();
      n.condition.addChild(n.conditionProbe.getBeginToken(count), 0);
      n.condition.addChild(n.conditionProbe.getEndToken());
    }
    else
    {
       System.err.println("Node already covered or condition == null ...");
    }
 }

 /* surround the condition in this statement with
    the tokens needed to create the call to
    writeCondition call, then recursively
    cover the sub conditions.
  */
 public void visit(KKWhileStatement n, Probe p)
 {
    /* already covered? */
    if(n.conditionProbe == null)
    {
      n.conditionProbe = new CProbe(n);
      n.conditionProbe.addCondition(n);
      int count = n.conditionProbe.numOfConditions();
      n.condition.accept(this, n.conditionProbe);
      n.conditionProbe.writeProbe();
      n.condition.addChild(n.conditionProbe.getBeginToken(count), 0);
      n.condition.addChild(n.conditionProbe.getEndToken());
    }
    else
    {
       System.err.println("Node already covered ...");
    }
 }

 /* surround the condition in this statement with
    the tokens needed to create the call to
    writeCondition call, then recursively
    cover the sub conditions.
  */
 public void visit(KKConditionalExpression n, Probe p)
 {
    /* already covered? */
    if(n.conditionProbe == null)
    {
      n.conditionProbe = new CProbe(n);
```

```
      n.conditionProbe.addCondition(n);
      int count = n.conditionProbe.numOfConditions();
      n.condition.accept(this, n.conditionProbe);
      n.conditionProbe.writeProbe();
      n.condition.addChild(n.conditionProbe.getBeginToken(count), 0);
      n.condition.addChild(n.conditionProbe.getEndToken());
    }
    else
    {
       System.err.println("Node already covered ...");
    }
 }

 /* surround the condition in this statement with
    the tokens needed to create the call to
    writeCondition call, then recursively
    cover the sub conditions.
  */
 public void visit(KKIfNode n, Probe p)
 {
    /* already covered? */
    if(n.conditionProbe == null)
    {
      n.conditionProbe = new CProbe(n);
      n.conditionProbe.addCondition(n);
      int count = n.conditionProbe.numOfConditions();
      n.condition.accept(this, n.conditionProbe);
      n.conditionProbe.writeProbe();
      n.condition.addChild(n.conditionProbe.getBeginToken(count), 0);
      n.condition.addChild(n.conditionProbe.getEndToken());
    }
    else
    {
       System.err.println("Node already covered ...");
    }
 }

 /* surround the condition in this statement with
    the tokens needed to create the call to
    writeCondition call, then recursively
    cover the sub conditions.
    If the Probe p passed to this method is not null
    this mean that it is a sub-condition of another
    condition. if it is null, then this means it is
    part of a normal statement, i.e, a statement
    that is not one of the above covered statements.
    for example:
    boolean b = a && c;
    can be such a statement.
```

```
  */
 public void visit(KKConditionalAndExpression n, Probe p)
 {
    if(p != null)
    {
      if(n.conditionProbe == null)
      {
          n.conditionProbe = (CProbe)p;
          for(int i=0; i < n.getNumChildren(); i+=2)
          {
             Node temp = n.getChild(i);
             n.conditionProbe.addCondition(temp);
             int count = n.conditionProbe.numOfConditions();
             temp.accept(this, n.conditionProbe);
             temp.addChild(n.conditionProbe.getBeginToken(count), 0);
             temp.addChild(n.conditionProbe.getEndToken());
          }
      }
    }
    else
    {
      /* not part of another compound statement,
         create a new probe.
       */
      if(n.conditionProbe == null)
      {
         CProbe cp = new CProbe(n);
         cp.addCondition(n);
         int count = cp.numOfConditions();
         n.accept(this, cp);
         cp.writeProbe();
         n.addChild(cp.getBeginToken(count), 0);
         n.addChild(cp.getEndToken());
      }
    }
}

/* surround the condition in this statement with
   the tokens needed to create the call to
   writeCondition call, then recursively
   cover the sub conditions.
   If the Probe p passed to this method is not null
   this mean that it is a sub-condition of another
   condition. if it is null, then this means it is
   part of a normal statement, i.e, a statement
   that is not one of the above covered statements.
   for example:
   boolean b = a && c;
   can be such a statement.
```

```
     */
    public void visit(KKConditionalOrExpression n, Probe p)
    {
       if(p != null)
       {
         if(n.conditionProbe == null)
         {
            n.conditionProbe = (CProbe)p;
            for(int i=0; i < n.getNumChildren(); i+=2)
            {
               Node temp = n.getChild(i);
               n.conditionProbe.addCondition(temp);
               int count = n.conditionProbe.numOfConditions();
               temp.accept(this, n.conditionProbe);
               temp.addChild(n.conditionProbe.getBeginToken(count), 0);
               temp.addChild(n.conditionProbe.getEndToken());
            }
         }
       }
       else
       {
         /* not part of another compound statement,
            create a new probe.
          */
         if(n.conditionProbe == null)
         {
            CProbe cp = new CProbe(n);
            cp.addCondition(n);
            int count = cp.numOfConditions();
            n.accept(this, cp);
            cp.writeProbe();
            n.addChild(cp.getBeginToken(count), 0);
            n.addChild(cp.getEndToken());
         }
       }
    }
}
```

Example A-7. ConditionVisitor class

We did not include the code for the *Tool* class, plus the code for all the probe classes, We feel the reader can reference the documentation and will be able to navigate through the source code easily.

Appendix B. The JavaCC grammar

The JavaCC grammar file has the following format:

```
javacc_input ::= javacc_options
                 "PARSER_BEGIN" "(" <IDENTIFIER> ")"
                 java_compilation_unit
                 "PARSER_END" "(" <IDENTIFIER> ")"
                 ( production )*
```

Every grammar starts with an optional list of options, followed by a Java compilation unit enclosed in between the "PARSER_BEGIN" and the "PARSER_END" keywords. The Java compilation unit, must be a class which it's name matches the "<IDENTIFIER>" supplied to the PARSER_BEGIN and PARSER_END. Here is an example:

```
PARSER_BEGIN(TestParser)
. . .
class TestParser . . . {
  . . .
}
. . .
PARSER_END(TestParser)
```

JavaCC will generate the following files for the above example:

- `TestParser.java`: The generate parser class.
- `TestParserTokenManager.java`: The generated token manager (or lexical analyzer).
- `TestParserConstants.java`: Useful constants.

- `Token.java, ParseError.java ... etc`: These files are the same for every JavaCC grammar, and will be generated only if they were not found

The generated parser file contains the compilation unit plus the generated code for the parser, this is in the form of public methods for each non-terminal or javacode_production or bnf_production.

The token manager file is generated from the regular_expr_production for terminals and tokens specified inline in the grammar. It provides a public method to access the token stream:

```
Token getNextToken() throws ParseError;
```

B.1. JavaCC Options

```
javacc_options ::= [ "options" "{" ( option_binding )* "}" ]

option_binding ::= keyword "=" ( java_integer_literal |
                                 java_boolean_literal |
                                 java_string_literal  ) ";"
```

We will not show the complete list of these keywords. Refer to the JavaCC documentation for a list of the keywords and their purpose. Here is an example of an option section:

```
options {
  STATIC = true;
```

```
  LOOKAHEAD = 3;
  OUTPUT_DIRECTORY = "work_dir";
}
```

B.2. Productions

There are four productions in JavaCC grammar:

```
production ::= javacode_production
             | bnf_production
             | regular_expr_production
             | token_manager_decls
```

The first two are used to define the grammar from which the parser is generated, and as we mentioned earlier, the regular_expr_production and the inline token specifications are used to generate the token manager. Finally the token_manager_decls is used introduce declarations that get inserted into the generated token manager.

Javacode productions (javacode_production)

```
javacode_production ::= "JAVACODE"
                        java_return_type java_identifier "("
java_parameter_list ")"
                        java_block
```

The production allows the user to write Java code for some productions instead of the usual EBNF expansion. This is useful when there is the need to recognize something that is not context-free or it is very difficult to write a grammar. For

example, in the following example we illustrate a way to skip all the tokens in the beginning of the file till we find a certain token, a "begin" in this case:

```
JAVACODE void skip_to_token(String str)
{
  Token tok = getToken(1);
  while (true)
  {
    if (tok.image.equals(str))
    {
      break;
    }
    tok = getNextToken();
  }
}
```

JavaCC treats these productions as a black box, and would not know how to choose between them if they appear in a choice point in the grammar, for example:

```
void some_production() :
{}
{
  skip_to_token("begin")
  |
  some_other_production()
}
```

But this can be corrected if we give enough information for each branch, for example:

```
void some_production() :
{}
{
  "{" skip_to_token("begin")
  |
  some_other_production()
}
```

BNF Productions (bnf_production)

```
bnf_production ::= java_return_type
                   java_identifier "(" java_parameter_list ")" ":"
                   java_block
                   "{" expansion_choices "}"
```

This is the standard production used in JavaCC grammars. Each BNF production has a non-terminal on the left hand side and a BNF expansions on the right hand side. The non-terminal is written exactly as a method in Java. As we mentioned earlier, each BNF production will become a method in the final parser class, the name of the non-terminal, the return value and the parameter passed, will become the corresponding parts of the method generated. The java_block is a set of arbitrary Java declarations and code, and will be placed in the beginning of the method generated for this production. Here is a simple example:

```
IFNode IfStatement() :
{
   IFNode temp = new IFNobe();
}
{
  (
     "if" "(" Expression() ")"
     Statement()
     [
         LOOKAHEAD(1)
         "else"
         Statement()
     ]
  )
}
```

expansion_choice

The expansion_choice has the following syntax:

```
expansion_choices ::= expansion ( "|" expansion )*
```

The expansion_choices in the bnf_production is a sequence of expansions separated by a "|". A legal parse of any of the expansions is a legal parse of the expansion_choice itself. The expansion has the following syntax:

```
expansion ::= ( expansion_unit )*
```

An expansion is a sequence of expansion_units. A legal parse of the expansion is a concatenation of the expansion_units in the same order they appear in the expansion. The expansion_unit has the following syntax:

```
expansion_unit ::= local_lookahead
               | java_block
               | "(" expansion_choices ")" [ "+" | "*" | "?" ]
               | "[" expansion_choices "]"
               | [ java_assignment_lhs "=" ] regular_expression
               | [ java_assignment_lhs "=" ] java_identifier "("
java_expression_list ")"

local_lookahead ::= "LOOKAHEAD"
                    "("
                         [ java_integer_literal ] [ "," ]
                         [ expansion_choices ] [ "," ]
                         [ "{" java_expression "}" ]
                    ")"
```

The expantion_unit can be a local_lookahead, which is an instruction to the generated parser to make decisions at choice points (eliminate ambiguities). It can be a Java code (called parser actions); this code gets executed when the appropriate part of the non-terminal is matched to the input. An

expansion_unit can be a parenthesized set of one or more expansions_choices, this can be suffixed with a

- + meaning one or more repetitions
- * meaning zero or more repetitions
- *?* meaning zero or one, an alternate to this is to enclose the expansion_choices in "[" and "]" brackets.

If the expansion_unit is a regular_expression, this it will match a token and an object of type Token will be returned and can be assigned to a variable by prefixing it with "variable =". Finally the expansion_unit can be a non-terminal, it takes the form of a method call in Java, and the return value can be assigned to a Java variable as in the case of the regular_expression. Here is an example of an expansion_unit:

```
  // match an identifier and then assign it to the variable class-
Name
  className=<IDENTIFIER>

  // match an optional part that is composed of the token "extends"
and the the
  // non-terminal Name(), assign the value returned from Name() to
the variable extendList
  // note that the java_block will be executed whenever the "ex-
tends" token is matched.
  [ "extends"        { foundExtend = true; }
    extendList=Name()
  ]

  //set the lookahead to 2, then examin the next w=two tokens, if
the are "." and identifier
  //then match them.
  ( LOOKAHEAD(2) "." <IDENTIFIER> )*

  //look for a token "(" and a non-terminal Name() and a token "["
in the token stream:
  LOOKAHEAD("(" Name() "[")
```

regular_expr_production

```
regular_expr_production ::= [ lexical_state_list ]
                            regexpr_kind [ "[" "IGNORE_CASE" "]" ] ":"
                            "{" regexpr_spec ( "|" regexpr_spec )* "}"
```

A regular_expr_production is used to define tokens, from which the token manager is generated. A regular_expr_production starts with a specification of the lexical states for which it applies. There is a standard lexical state called "DEFAULT" which is the one used if lexical_state_list is omitted. Following this is a specification of what type of regular expression it is. There are four types of regular expressions:

- *SKIP*: match and throw away.
- *MORE*: used to match partial tokes, these make part of other tomes.
- *TOKEN*: match a token.
- *SPECIAL_TOKEN*: match a token, but do not insert it in the token stream. Attach it to the normal token that come after it. This is useful for some tokens that are needed for the output but not used in the program. Comments are a good example.

An option "IGNORE_CASE" can be specified to make the regular_expression case insensitive.

regexpr_spec

```
regexpr_spec ::= regular_expression [ java_block ] [ ":"
java_identifier ]
regular_expression ::= java_string_literal
                    | "<"
                      [ [ "#" ] java_identifier ":" ]
                      complex_regular_expression_choices
                      ">"
                    | "<" java_identifier ">"
                    | "<" "EOF" ">"

complex_regular_expression_choices ::= complex_regular_expression (
"|" complex_regular_expression )*
complex_regular_expression ::= ( complex_regular_expression_unit )*
complex_regular_expression_unit ::= java_string_literal
                                  | "<" java_identifier "i>"
                                  | character_list
                                  | "(" complex_regular_expression_choices
")" [ "+" | "*" | "?" ]
```

The best way to explain a regular_spec and regular expressions is to give an example: example Java floating point literals:

```
TOKEN :
{
  < FLOATING_POINT_LITERAL:
        (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)?
(["f","F","d","D"])?
      | "." (["0"-"9"])+ (<EXPONENT>)? (["f","F","d","D"])?
      | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])?
      | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"]
  >
|
  < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
}
```

In this example, the token FLOATING_POINT_LITERAL is defined using the definition of another token, namely, EXPONENT. The "#" before the

label EXPONENT indicates that this exists solely for the purpose of defining other tokens (FLOATING_POINT_LITERAL in this case). The definition of FLOATING_POINT_LITERAL is not affected by the presence or absence of the "#". However, the token manager's behavior is. If the "#" is omitted, the token manager will erroneously recognize a string like E123 as a legal token of kind EXPONENT (instead of IDENTIFIER in the Java grammar).

token_manager_decls

```
token_manager_decls ::= "TOKEN_MGR_DECLS" ":" java_block
```

The token_manager_decls are used to specify delarations and statements for the generated token manager, here is a simple example:

```
TOKEN_MGR_DECLS : { int commentSize; }
```