Flexible code coverage tool and its applications

Sergio Walter Ferrero

Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Recommended Citation
https://lib.dr.iastate.edu/rtd/20414

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Flexible code coverage tool and its applications

by

Sergio Walter Ferrero

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Suraj Kothari, Major Professor
Govindarasu Manimaran
Shashi K. Gadia

Iowa State University
Ames, Iowa
2004

Copyright © Sergio Walter Ferrero, 2004. All rights reserved.
Graduate College
Iowa State University

This is to certify that the master’s thesis of
Sergio Walter Ferrero
has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy
DEDICATION

I would like to dedicate this thesis to my wife Vero. I am sure she will not read any part of the thesis but this dedication. I would like to thank her understanding for all the time I could not share with her during these last months and for all her support. I would also like to dedicate this thesis to my family for their loving and unconditional support during these years at Iowa State University.
# TABLE OF CONTENTS

LIST OF TABLES ........................................... vi

LIST OF FIGURES ......................................... vii

CHAPTER 1. OVERVIEW ....................................... 1
  1.1 Introduction .......................................... 1
  1.2 Code Coverage ......................................... 2
    1.2.1 Decision/condition coverage .................... 5
    1.2.2 Multiple condition coverage ..................... 5
    1.2.3 Modified Condition/Decision Coverage .......... 6

CHAPTER 2. CODE COVERAGE PROBLEM ......................... 8
  2.1 Problem ............................................. 8
  2.2 Related work ......................................... 9
  2.3 Our approach ........................................ 11
  2.4 Assumptions and limitations ......................... 13

CHAPTER 3. DESIGN AND IMPLEMENTATION ....................... 15
  3.1 Overview ............................................ 15
  3.2 XCIL representation .................................. 17
  3.3 Runtime data collection ............................. 18
  3.4 Instrumentation ...................................... 18
    3.4.1 Flexible instrumentation ....................... 19
    3.4.2 Rules specification ................................ 23
    3.4.3 Weaving process .................................. 26
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1.1</td>
<td>Decision/condition coverage</td>
<td>5</td>
</tr>
<tr>
<td>Table 1.2</td>
<td>Case in which 100% condition coverage is achieved but not 100% on decision coverage</td>
<td>6</td>
</tr>
<tr>
<td>Table 1.3</td>
<td>Multiple condition coverage</td>
<td>6</td>
</tr>
<tr>
<td>Table 1.4</td>
<td>Modified Condition/Decision Coverage</td>
<td>6</td>
</tr>
<tr>
<td>Table 3.1</td>
<td>Primitives for communication with the monitor process</td>
<td>29</td>
</tr>
<tr>
<td>Table 3.2</td>
<td>Format of the packets sent through the pipe</td>
<td>30</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>Chosen configurations for the experiments</td>
<td>39</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>Results obtained while compressing a 400KB file (<em>test.pdf</em>)</td>
<td>40</td>
</tr>
<tr>
<td>Table 4.3</td>
<td>Results obtained while compressing a 25MB file (<em>test3.jar</em>)</td>
<td>40</td>
</tr>
<tr>
<td>Table 4.4</td>
<td>gzip parameters description</td>
<td>43</td>
</tr>
<tr>
<td>Table 4.5</td>
<td>Merging results from different runs</td>
<td>43</td>
</tr>
<tr>
<td>Table 4.6</td>
<td>Performance results</td>
<td>46</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1.1 Subsumption hierarchy of coverage criteria .................. 4
Figure 1.2 Code snippet with conditions ................................. 5
Figure 3.1 Source code instrumentation process ....................... 16
Figure 3.2 Runtime architecture .......................................... 16
Figure 3.3 Runtime architecture with the use of rules ................ 17
Figure 3.4 Advice location for a while statement .................... 20
Figure 3.5 Multiple advice functions .................................... 21
Figure 3.6 Predicate capture via ternary operation and function .... 22
Figure 3.7 Interface for specification of a loop-related rule .......... 24
Figure 3.8 Interface for specification of a non-loop-related rule .... 24
Figure 3.9 Interface for specification of the common rule ............ 25
Figure 3.10 I4CC architecture ............................................. 27
Figure 3.11 Runtime architecture with the use of buffers ............ 29
Figure 3.12 Visualization of results in I4CC ............................ 32

Figure 4.1 Experiment showing how simple analysis of pairing functions can be done. Spotting memory leaks. ......................... 35
Figure 4.2 Data collected on program crashes ........................... 37
CHAPTER 1. OVERVIEW

This work describes design and implementation of a code coverage tool. It presents experiments and results of a case study and it gives an overview of its possible applications areas.

1.1 Introduction

Software testing constitutes an important fraction of the entire software development cost. Since exhaustive testing is generally unfeasible, it is useful to find a way to determine how much testing should be done. Test coverage provides the means to answer the previous question. The IEEE standard glossary of software engineering terminology [17] defines test coverage as "the degree to which a given test or set of tests addresses all specified requirements for a given system or component". Y. K. Malaiya and J. A. Denton [9] state that "software test coverage directly measures the thoroughness of testing avoiding the problem of variations of test effectiveness". In order to come up with a degree of thoroughness coverage metrics has to be used. Coverage metrics measure what percentage of the code has been exercised.

Code coverage information is obtained by recording runtime data as the program runs. Thus, besides its normal functionality, a program should also log runtime data. This is done by instrumenting the original program, that is inserting pieces of code that log runtime data. A well-known problem is the high overhead incurred by the huge amount of data that has to be gathered as the program runs. This problem has been studied for several years and different approaches has been proposed [7] [20] [6] [2] [10] [15] [19] [5]. What we noticed from previously proposed tools is that they do not provide enough flexibility in the process of getting code coverage information. Since time is usually tight in the industry of software development,
it is important to reduce the overhead of getting code coverage data. Precision could not be the most important issue in some cases while runtime performance is of most concern.

This work reports a new approach to solve the problem of minimizing the time for collecting code coverage data. The main goal is to provide a flexible code coverage tool to the user, in which performance and precision can be tuned as the instrumented program runs. Design and implementation details are provided as well as results from a case study.

The remaining of this work is organized as follows. Next section gives an overview of code coverage metrics. Chapter 2 further explains the code coverage problem, gives an overview of previous approaches and presents the proposed approach. Design and implementation details are presented in chapter 3. Chapter 4 reports a case study, some results and brief analysis. Finally, conclusions and future work are covered in chapter 5.

### 1.2 Code Coverage

This section discusses the notion of code coverage and the coverage criteria that will be considered in our tool.

Code coverage provides the mechanism to measure the thoroughness of tests and thus is of most importance for testing and quality assurance. The basic idea behind code coverage is to provide testers with a measure of how much of the code was exercised before deployment. Coverage metrics are used to provide this measure and they are expressed in term of a ratio as explained in [1]

\[
Coverage = \frac{\text{items executed at least once}}{\text{total number of items}}
\]

The word *items* is used in order to give a general definition. Depending on the items being monitored different coverage criteria are defined. There exists a wide spectrum of coverage criteria but we will go over the most widely known ones in this section. Structural coverage, also known as white box testing, can be divided in control flow coverage and data flow coverage. Control flow coverage consists of the following metrics as defined in [1]

- *Statement coverage:* each statement is executed at least once.
• **Block coverage**: each basic block is executed at least once.

• **Decision coverage**: each statement is executed at least once; each decision takes on all possible outcomes at least once.

• **Condition coverage**: each statement is executed at least once; each condition in a decision takes on all possible outcomes at least once.

• **Decision/condition coverage**: each statement is executed at least once; each decision takes on all possible outcomes at least once; each condition in a decision takes on all possible outcomes at least once.

• **Multiple condition coverage**: each statement is executed at least once; all possible combinations of condition outcomes in each decision occur at least once.

• **Modified Condition/Decision Coverage**: every condition within a decision is executed in order to show that it can independently affect the outcome of the decision.

On the other hand, data flow coverage comprises the following metrics as defined in [12].

• **c-use**: a *c-use* is a variable $x$ and the set of all paths in the data flow graph from node $n_a$ to $n_b$ such that:

  - $x$ is in $\text{DEF}(n_a)$, and
  - $x$ is not in $\text{DEF}(n_i)$ for any other node $n_i$ on the paths, and
  - $x$ is in $\text{C-USE}(n_b)$

A *c-use* is covered by a set of tests if at least one of the paths in the *c-use* is executed when the test is run.

• **p-use**: a *p-use* is a variable $x$, a predicate node, $n_c$, which uses $x$ in some predicate expression, and the set of all paths in the data flow graph from node $n_a$ to $n_b$ such that:

  - $x$ is in $\text{DEF}(n_a)$, and
  - $x$ is not in $\text{DEF}(n_i)$ for any other node $n_i$ on the paths except possibly $n_c$, and
A $p$-use is covered by a set of tests if at least one of the paths in the $p$-use is executed when the test is run.

- **all-use**: an all-use is a $c$-use or a $p$-use. An all-use is covered by a set of tests if at least one of the paths in the $c$-use or $p$-use is executed when the test is run.

There are dominance and equivalence relationships among these coverage metrics. For instance, block coverage represents the same measure as statement coverage. Note that, by the definition of block coverage, 100% block coverage implies 100% statement coverage and 100% statement coverage implies 100% block coverage. They are also known as line coverage. Similarly, 100% decision coverage implies 100% block coverage. Horgan et al. [12] and Chilenski et al. [8] present diagrams showing the subsumption hierarchies of coverage criteria. Figure 1.1 shows a combination of those hierarchies. Further information about data flow coverage is beyond the scope of this thesis and can be found on [12].
In order to better understand the notions of decision and condition coverage we will introduce an example that was taken from [1]. The following definitions for condition and decision are considered throughout this document. A condition is any boolean expression that cannot be broken into simpler expressions and a decision is built from a set conditions using logical operators. Figure 1.2 shows a code snippet for the example.

```java
if(condition1 && (condition2 || condition3))
```

Figure 1.2 Code snippet with conditions

### 1.2.1 Decision/condition coverage

The basic idea in Decision/condition coverage is that every statement should execute at least once, every branch should take both, true and false values and also every condition within the decision should evaluate to true and false. Table 1.1 shows values for the condition in such a way that 100% decision/condition coverage is achieved.

<table>
<thead>
<tr>
<th>Case</th>
<th>Condition 1</th>
<th>Condition 2</th>
<th>Condition 3</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

Note that even if we achieve 100% coverage in this case, there are some combinations of condition values that remain untested.

Regarding condition and decision coverage, note that it could be the case that 100% condition coverage is achieved but not 100% on decision coverage. Table 1.2 shows values that represent this case.

### 1.2.2 Multiple condition coverage

This coverage measure addresses the shortcoming of the previous metric since it requires each combination of values to be tested. Table 1.3 shows combination of values to achieve
Table 1.2  Case in which 100% condition coverage is achieved but not 100% on decision coverage

<table>
<thead>
<tr>
<th>Case</th>
<th>Condition 1</th>
<th>Condition 2</th>
<th>Condition 3</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

100% multiple condition coverage.

Table 1.3 Multiple condition coverage

<table>
<thead>
<tr>
<th>Case</th>
<th>Condition 1</th>
<th>Condition 2</th>
<th>Condition 3</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>3</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>4</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>5</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>6</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>7</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>8</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

1.2.3 Modified Condition/Decision Coverage

The basic idea behind modified condition/decision coverage is to narrow the number of test cases that have to be done in order to determine that each condition independently affects the outcome of the decision. Table 1.4 presents a set of test cases that show how each condition affects the outcome of the decision.

Table 1.4 Modified Condition/Decision Coverage

<table>
<thead>
<tr>
<th>Case</th>
<th>Condition 1</th>
<th>Condition 2</th>
<th>Condition 3</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>3</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>4</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>6</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

It can be seen, for example, in test cases 2 and 6 how condition1 affects the output of the decision, that is, the independence of condition1. A similar analysis should be done with the other cases to see that the independence of each of the conditions.
The tool we developed includes the block coverage criterion since it is the most common criterion in industrial software [11]. It also provides function coverage, that is, the number of functions exercised over the total number of functions. We also found useful to show the user which combinations of condition values occurred for a given decision, which we refer to as multiple condition evaluation. Note that this information could be used to compute multiple condition coverage.
CHAPTER 2. CODE COVERAGE PROBLEM

This chapter gives a review on code coverage and states the problem we addressed in our work. It also includes a review of previous research on code coverage and program profiling, also known as program instrumentation. Finally, the proposed approach is described.

2.1 Problem

Code coverage analysis of large industrial software systems is very time consuming. As we already mentioned, a well-known problem is the high overhead incurred by the huge amount of data that has to be gathered as the program runs. A single loop could easily generate tremendous amounts of data on an instrumented program causing remarkable runtime degradation. There are different artifacts that can be monitored as explained in section 1.2. Data collection of those artifacts allow us to compute different code coverage metrics. The more coverage metrics we want to compute on a single run the more time the program will take to run. Similarly, the more precision we want to achieve on a particular coverage metric the higher the degradation in performance. For instance, suppose we are collecting data to compute block coverage, we might consider collecting data in a discontinuous way during the run. Collecting data discontinuously could cause the instrumented program to miss some blocks hits with a consequent loss of precision on coverage results. As a result of this precision loss block coverage values will not be completely accurate but they could be accurate enough at a reasonable runtime cost. Thus, there exists a tradeoff between precision and performance, and similarly a tradeoff between coverage metrics to be computed and performance.

Several approaches have been studied in order to cope with this performance problem. Research includes selective instrumentation, dynamic instrumentation, sampling, reduction of
instrumentation points, usually, by applying graph theory to the program AST. Section 2.2 presents a review of these previous approaches and section 2.3 discusses the approach we took to solve the problem.

2.2 Related work

Code coverage has shown to be an important issue in the system development process. As it is stated in [16] testers usually overestimate the coverage of test cases. An experiment showed that testers expected 90% of coverage and results showed only a coverage of 50%. They also state that 70-80% coverage is the ideal percentage taking into account the precision/cost tradeoff.

Program instrumentation has to be used to obtain code coverage information, which is used to compute code coverage metrics. Instrumentation can be performed at different levels as described in [18]:

- At the source-code level, as a source-to-source transformation.
- As part of compilation, by extending a compiler to use its intermediate representations for the purpose of determining where to introduce instrumentation instructions.
- As an object-code-level transformation, by modifying object-code files (such as UNIX a.o)
- As a post-loader transformation, by modifying executable files (such as UNIX a.out files)

Instrumentation can further be divided in dynamic and static [20]. In static instrumentation the instrumentation code is inserted once and it remains in the instrumented program. On the other hand, dynamic instrumentation allows inserting and removing instrumentation code as the program executes. The basic gain with dynamic instrumentation is that it permits the removal of instrumentation code when it is not needed anymore, giving the user a flexible way to reduce the overhead as the program runs.

Further analysis can be done if sensitivity is considered while profiling, that is, a notion of context is associated to the metrics. Flow sensitivity allows associating performance metrics
with paths; which is mostly useful when profiling for optimization purposes. If context sensitivity is added performance metrics can then be associated with interprocedural paths. This allows to associate metrics with calling context (sequence of procedures calls). Glenn Ammons et al. [4] developed a flow sensitive, and an approximation to context sensitive, profiler to get hardware metrics along paths in procedures. Larus [14] used a slightly modified version of the previous approach in order to capture interprocedural paths. Information about whole program paths, as he called the interprocedural paths, is used to detect heavily executed sequence of code for tuning and optimization purposes.

There has been extensive research in instrumentation techniques to handle the overhead problem of code coverage tools in different ways. Among them, we can mention reduction of instrumentation points and inference of information from the collected data. Thomas Ball and James Larus [7] developed algorithms to instrument programs in order to profile and trace them. They made use of the theory of spanning trees, applied to the control flow graph (CFG), to reduce the number of places in which instrumented code is added. Tikir et al. [20] used instrumentation to identify the parts of the program that did or did not execute. A simple flag is set to true if the statement did execute. In this case, dominator tree information was used in order to reduce the number of instrumentation points.

Instrumentation sampling is another possibility to reduce the overhead of instrumented programs, in which instrumentation code runs periodically. Arnold and Ryder [6] developed a tool that reduces the overhead of collecting runtime information by performing instrumentation sampling. They created two versions of the code, duplicated code which is highly instrumented (high overhead) and checking code (light instrumented) which just keeps checking for a sample condition to decide whether to change control to the duplicated code. The rate at which the sample condition is set to true determines the overhead and precision of the instrumentation.

A third approach is what is called selective instrumentation which allows the user to focus on particular parts of the program. The level at which selective instrumentation is done can vary from set of statements, procedures and up to modules. Rational Purify Plus [2] includes a set of tools for program analysis which are based on code instrumentation. They allow selective
instrumentation at module level. An API can be used from the instrumented program to control runtime data collection. However, compared to our approach, the API can be used in a limited manner. The execution of an API function can not be based on an iteration number. Furthermore, when recording is stopped no runtime reduction of the instrumented program is obtained.

Ding and Zhong [10] used selective instrumentation to monitor data access in applications in which complete monitoring is not necessary. They developed a source-to-source compiler that chooses what data to monitor and at which points in the program. London et al. [15] created a tool, called ATAC that allows performing selective instrumentation for code coverage. However selective instrumentation is only provided at file level. Sameer Shende et al. [19] also used selective instrumentation in order to meet requirements on runtime overhead. They performed instrumentation to evaluate execution time on functions.

Another possibility to cope with the performance issue is what is called dynamic instrumentation. In this case instrumentation is done as the program runs; therefore, the overhead of the instrumented code may be tuned up. Arnold et al. [5] considered this approach to implement an adaptive optimization system. Their approach gives more flexibility during monitoring since instrumentation can be changed based on runtime data collected from the running program. This paper presents a similar approach to our work, since they control the process of runtime data, but it is intended for a different purpose, which is optimization of java code.

Yong Woo Kim [13] states some of the problems of applying code coverage on large industrial software systems and he also proposes an approach to perform code coverage analysis. He also remarks the need for a flexible instrumentation tool in terms of data collection and performance.

2.3 Our approach

What we noticed from previous work is that all of the approaches on code coverage tools are too rigid. There is no way for the user to control how recording of runtime data is done. Similarly, it is not possible to dynamically vary the performance and precision as the instrumented
program runs. Dynamic instrumentation can use online feedback, that is take decisions based on runtime data collected, but response to that feedback is usually predetermined for the whole program run. The usual way dynamic instrumentation is used is to remove instrumentation code that records a hit of an item once the item has been exercised. The disadvantage of this approach is that it cannot compute the number of times a statements was hit.

Our approach is to develop a framework that allows the user to control runtime data collection as the instrumented program runs. The basic idea is to perform a flexible instrumentation in which precision and/or the selection of artifacts to monitor can be changed as the program runs.

One concrete case we propose is to collect data using certain sampling rate when the program executes loops. More specifically, the iteration number of a loop can be queried in order to decide whether to record data or not. Recording of values for the conditions in control structures can also be controlled as the program runs.

The user will be able to guide runtime data collection through the development and inclusion of rules into the program. Through the rules the user will specify what data should be collected, as well as to when and where. Rules can be generated at any place on the source code; the only restriction is that it has to be the beginning of a basic block or any control structure. For the case of loops, rules can be based on the current iteration number. That is, the iteration number can be checked in order to decide on the recording policy to be applied. The recording policy will state what data has to be monitored and how often in the case of loops.

Our work will extend current profiling techniques in two new directions. The first consists of varying the level of runtime information collection as the program runs. As far as we know, controlling runtime data collection has been used for optimization purposes together with dynamic instrumentation but has never been applied in code coverage tools. Guiding runtime data collection will be done by the insertion of rules in the instrumented code. The second achievement is that it will give the user a flexible, powerful way to focus precision on particular parts of the program under analysis. Rules establish what actions to take when
certain conditions hold. The inclusion of rules is the main contribution of this work, since
the user will be able to tune the process of runtime data collection. In order to achieve
more flexibility, selective instrumentation is also possible by allowing the user to select which
methods should be instrumented.

Some of the advantages of the proposed approach are increase in runtime performance and
monitoring only part of the program of interest to the user, avoidance of heavily executed code
of no interest anymore. This work will help programmers in the process of code testing as a
test coverage tool, as well as in the process of program understanding providing visualization
of results through a graphical user interface. Detail about this applications is explained in
section 4.1.

2.4 Assumptions and limitations

Assumptions made during the design and implementation of the tool are described below.

• If a function is invoked from an expression it is supposed to return a value and thus
  functions should not exit the program. Since it will not be a good programming practice
  we assumed that functions in expressions will always return control to the calling site. On
  the other hand, procedures can exit the program and so procedure calls are considered
  to be the end of a basic block.

• Functions in header files are not instrumented since they are intended for types and
  constants definition.

The tool has the following limitations with respect to the kind of programs that can be
analyzed.

• The tool does not take care of dynamic initialization. Anyway, this is not a problem
  since dynamic initialization statements will be part of the block identified by the first
  statement of that basic block.

• Recursion is not possible when rules related to sampling rate were created. The imple-
  mentation of sampling rate keeps a counter for each loop; therefore, if recursion is allowed
counters might be overridden. This problem can be solved by keeping a stack of counters that are pushed on a stack when a loop is hit and pop after the loop finishes execution. This could be part of future work in order to handle recursion. However, performance can be considerably degraded since the number of counters can be fairly large.

- *goto* statements are not handled. They can cause disrupted multiple condition evaluation results since control flow would be different from the one assumed in section 3.4.1.2 which explains how predicate values capture works.
CHAPTER 3. DESIGN AND IMPLEMENTATION

This chapter describes design and implementation details of the tool I4CC (Instrumentation for Code Coverage). First, a section with an overview of the general architecture is presented; subsequent sections present more detailed information about the process of runtime data collection.

3.1 Overview

The purpose of the tool is to obtain runtime information to compute code coverage metrics to be used in code testing. The strategy was to use source code instrumentation to obtain runtime information. Basically, the original C program is enlarged with instrumentation code that extracts runtime data. The resulting C program is compiled and run. As the program runs, runtime information is passed to a separate monitor process that collects it. Figure 3.1 presents the above mentioned process to the point in which the binary instrumented program is obtained. The runtime architecture is shown in Figure 3.2.

Note that one advantage of this run-time architecture is that even if the instrumented program crashes runtime data collected to the point of the crash is available for analysis.

As we already mentioned in section 2.3 we are able to handle user-specified rules. This slightly changes the previous architecture since the instrumented code should be able to check for the monitoring configuration. The monitoring configuration is maintained in a global structure and basically keeps track of what has to be monitored and how. More specifically, it maintains whether recording is enabled in which case runtime data is sent to the monitor. It also keeps track of whether predicates values in conditional expressions are monitored or not and it contains some other variables for internal control of the monitoring configuration.
Prognitn's source code will then be compiled into an executable file via the C compiler. When this executable runs, runtime data is logged.

Figure 3.1 Source code instrumentation process

Figure 3.2 Runtime architecture
The process of instrumenting source code requires handling a representation of a program. In our case we made use of XCIL (eXtensible Common Intermediate Language) [3] as the representation of a program. An overview of XCIL is presented in the next section.

### 3.2 XCIL representation

XCIL stands for eXtensible Common Intermediate Language. It is an intermediate language that is able to represent many programming languages in a common format. XCIL contains enough detail about the original source code to allow for many types of analysis. A translation unit is the XCIL representation of each file of the program. Each program artifact (variable,
expression, control structure, etc) is uniquely identified in XCIL by a string of digits called ID. Source correspondence information can be generated in order to map XCIL IDs to source code positions. This information is used in the process of instrumenting the original program's source code and in the process of building the abstract syntax tree (AST) nodes.

3.3 Runtime data collection

In the terminology of the monitor when a statement is reached it is called a hit. I4CC collects hits for three different artifacts:

**Function hits:** any time a function is executed a function hit is recorded

**Block hits:** each time a basic block is executed a hit for that basic block is recorded.

**Multiple condition evaluation values:** the recording of multiple condition evaluation values consists of the combination of condition values hit for each decision

Remember from section 1.2 that a condition is any boolean expression that cannot be broken into simpler expressions and a decision is built from a set conditions using logical operators. Thus, every time a decision is evaluated the combination of the truth values for the conditions in the decision is stored. This data is stored in a set, and thus, only one instance of each combination is kept. Note that this result could be used to compute multiple condition coverage.

3.4 Instrumentation

The instrumentation process is divided in three main tasks. First, a common flexible instrumentation code is generated. It is called flexible instrumentation since the instrumented code checks for the current monitoring configuration to determine what data should be sent to the monitor and how often. Secondly, the user is allowed to specify rules to control runtime data collection. The third task consists of weaving the flexible instrumentation together with the rules into the original source code.
3.4.1 Flexible instrumentation

In order to complete the first task a list of places in which to insert code, called instrumentation points, has to be determined. The functions to be inserted at each instrumentation point are called advice functions. The set of all the advice functions is called the flexible instrumentation code. The sets of instrumentation points together with the advice functions are called a markup. Summing up, the first stage of instrumentation consists of the generation of this markup that later will be woven into the original source code along with the user specified rules.

3.4.1.1 Instrumentation Points

There are six basic points that are instrumented: before any basic block, before and after a control structure, at the start and end of the body of a control structure, and around conditions. The words condition and predicate are used indistinctly in what follows. The points are also differentiated based on the type of construct being instrumented (if, for, while, etc.). These points are analogous to joinpoints in aspect-oriented programming. A basic block is any sequence of one or more consecutive, executable statements containing no branches.

There are two main kinds of advice; the start of a basic block that just reports a hit for a basic block and a set of more general advice functions that are used in control structures for the purposes of multiple condition evaluation and the handling of rules.

At a point where advice needs to be run, a corresponding advice function is called. The code in figure 3.4 illustrates where these points are in the case of a while statement and what the advice function calls might look like. Advice location is the same for other control structures; there are before and after advice functions that run just before and after the control structure is executed, respectively. Similarly, start and end advice functions run at the beginning and end of the control structure's body respectively. Actual function calls include identifiers from XCIL to identify the source code construct (basic block, if, while, etc.). Predicates are associated with the enclosing conditional statement, a while in this case.

The set of advice functions and their locations for each control structure are described in
Figure 3.4 Advice location for a \textit{while} statement
One of the problems found while instrumenting source code was that at any given point multiple advice functions might need to be inserted in certain order. Consider the noninstrumented code snippet shown in figure 3.5.

Four advice functions should be included at line 4, \texttt{cc\_end\_if}, \texttt{cc\_after\_if}, \texttt{cc\_end\_while} and \texttt{cc\_after\_while}. All of them should be inserted at the same instrumentation point but in the correct order. The process of weaving the instrumentation code in the source code includes ordering the lines in the markup file based on the instrumentation points, that is by line and column. Taking advantage of this ordering we decided to enforce the desired order by attaching a seven-digit number, called advice prefix, at the beginning of each advice when generating the markup file. A counter was maintained to keep incrementing the advice prefix each time an advice was generated. Those seven digit numbers allow us to achieve the desired order over advice functions with identical instrumentation points. Finally, the \texttt{weaver} module removes the advice prefixes when performing the code weaving.

Blocks from control structures that include a single statement might not be wrapped with brackets. The process of instrumentation has to include the corresponding brackets since we are including advice functions at the beginning and end of the control structure body and therefore the block will contain more than one statement. Advice prefixes are also attached to opening and closing brackets since they must follow a given order when inserted with advice functions with the same instrumentation point. While generating the markup, a prefix is reserved for each AST node since a rule might be created at each AST node. Since a rule can have the same instrumentation point as an advice function, the prefix for the rule allows us to
3.4.1.2 Multiple condition evaluation

In C, the ternary operator can be used to determine the value of an expression, as demonstrated in figure 3.6. As one would expect, the predicate capturing function returns the predicate value.

A hit for a decision consists of conditions evaluations for a single pass through a statement. A decision hit is also referred as a predicate bundle. To see why it is necessary to mark the start and end of evaluations, it is necessary to consider all the ways the control flow could be interrupted. Given the statement "if ( A || ( B && C ) )", there are two basic ways control can be interrupted. The first is short-circuiting in logical operations. This occurs when the predicate A is true and the expression ( B && C ) is never evaluated. This can also occur when predicates A and B are false, and C is never evaluated. The second way is when one or more predicates are function calls.

To uniquely identify a predicate bundle, the monitor is sent a message (start_pred) indicating that predicates are about to be evaluated for a particular conditional statement. Once the predicates are evaluated, another message (end_pred) is sent to the monitor indicating that evaluations are complete for the statement. Depending on the construct, it may be necessary to end evaluation both at the start of a body and after the body, due to the possible avenues of control flow.
3.4.2 Rules specification

Rules are used to control the recording process by updating the monitoring configuration. The user will be able to specify rules to enable or disable recording of runtime data at any point; that is, at the beginning of a control structure or first statement of a basic block. An AST representation of the program will be available in the graphical user interface for the purpose of specifying rules at desired places.

Rules can be divided in loop-related and non-loop-related. In the case of loop-related rules the user can specify a recording range, the sampling rate and whether multiple condition evaluation is enabled or not. They can also specify whether to enable or disable recording (recording state) before the loop executes. In the case of non-loop-related rules, only the state of multiple condition evaluation can be specified besides the recording state. The specified rules will execute just before the statement in which they are specified. There is one exception in the case of loop-related rules in which case the recording state part of the rule will execute before the loop, but the remaining parts of the rule will be executed at the beginning of the loop body. This is so because the range has to be checked at each iteration of the loop. Figures 3.7 and 3.8 present the graphical user interface for the loop-related and non-loop-related rules respectively.

The tool also allows the user to create a common rule for all loops in the program. Figure 3.9 present the interface for the rule. This will allow the user a fast creation of rules. The specification of this rule could be useful in the following cases.

- Perform monitoring at a given sampling rate for all loops in order to speed up performance.

- Disable multiple condition evaluation on loops if the user prefers performance and does not want to evaluate trivial conditions on loops.

- Perform monitoring on loops only in a given range, first k iterations for instance, in order to know if the loop body ever executed or only a coverage notion wants to be known.
Figure 3.7  Interface for specification of a loop-related rule

Figure 3.8  Interface for specification of a non-loop-related rule
The basic idea behind the common rule is to achieve performance while still getting some degree of coverage. It also speeds up considerably the process of rule specification for large code since the rule is specified only once.

Rules that modify the status of multiple condition evaluation have a different conception depending on whether the rule is a loop or non-loop related. In the case of loop related rules, the multiple condition evaluation status specified in the rule has effect only at the specified range. On the other hand, a non-loop related rule has a wider effect since the state of multiple condition evaluation set will remain until it is changed by another rule. This is done in this way since we think it gives more sense to the rule specification process and more flexibility is gained from the user point of view. Once the user specified the rules, C code is automatically generated for each rule, as well as the signature for the rule that will be used for invocation from the instrumented program. Implementation of rules resides in a file named _cc_rules.c which is compiled together with the application files.
3.4.3 Weaving process

The third task consists of weaving the instrumentation code and rules into the source code. The weaving code program\(^1\) was written in python and it takes as input a markup file in which each line represents a point and an advice function or a rule to be woven into the original source code.

3.4.4 General Architecture

Figure 3.10 shows the architecture that describes the whole process of instrumentation in I4CC. As it can be seen, first a pre-instrumentation is done which just consists of including a header file, named \texttt{_cc\_monitor.h}, in each of the program’s source code files. The included header file contains definition of types and advice prototypes needed by the final instrumented program. Next the C preprocessor is run; this step is needed in order to obtain consistent source correspondence (SC) information since Xcc runs after preprocessing. Once the source code is preprocessed, Xcc is run to generate the XCIL and SC files, which along with mkp-Config.xml are the input to the markup generation module. The file \texttt{mkpConfig.xml} maintains the instrumentation configuration specified by the user. Basically, it describes which functions have to be instrumented and whether instrumentation for multiple condition evaluation will be generated or not. The markup generation module generates a markup file, which as we already defined is a list of instrumentation points together with the advice functions to be inserted at those points. This is the main module in the instrumentation process and it was developed in java. A file, named \texttt{rules.txt}, is generated by the rules generator module which is called from the graphical user interface once the user finished with the specification of rules. This file has the same structure as the markup file; each line consist of an instrumentation point plus a rule call. Implementation of each rule is generated automatically and resides in the file \texttt{_cc\_rules.c}. The next step is to weave the markup and rule files’ lines into the preprocessed source code which results in the instrumented source code ready for compilation. The weaver module was implemented on Python. The resulting instrumented source code is then compiled with gcc

\(^1\)The first version of the weaver module was coded by Luke Bishop. Modifications were done in order to instrument multiple C files.
Program's source code → Pre-instrumentation

Pre-instrumented source code → C Preprocessor

Preprocessed source code → Pre-instrumentation

Xcc

mkpConfig.xml → XCIL

rulesGenerator

rules.txt → Weaver

rulesGenerator

rules.txt → mkpFile.txt

Weaver

Instrumented source code

cc_rules.c

cc_monitor.c

cc_rules_core.c → gcc

Instrumented program

Figure 3.10 I4CC architecture
along with the following three files:

.cc_rules.c: it contains the implementation of user specified rules

.cc_monitor.c: it contains functions to start the monitor process and to handle communication, functions for the implementation of the buffer in shared memory, code for the buddy process and the implementation of the advice functions.

.cc_rules_core.c: it declares a global structure to keep the monitoring configuration as well as methods to interact with it. It also defines some primitives for the use of rules.

Finally, the instrumented program is generated. Once run, the program will create the runtime architecture specified in figure 3.11 that will log runtime information to be later visualized in the graphical user interface.

### 3.5 Runtime architecture

Once the executable file generated from the instrumented code is run, two main processes exist, the instrumented program itself and the monitor process in charge of the recording of runtime data. Since the communication between these two processes is through a named pipe, buffering was implemented at both processes, before sending and receiving data, for performance purposes. The instrumented program will put data into a buffer which will be sent once it is full. The size of the buffer was set at 1024 bytes after some empirical analysis done with several test runs. The receiving process (the monitor) will also work with a buffer of the same size. This improvement in performance brought a problem in the case of a crash of the instrumented program since data in the buffer, which could be highly useful for detecting the crash problem, will not be flushed when the program dies. Therefore, a third buddy process that shares the buffer via shared memory with the instrumented program was designed. The buddy process is set as the parent process of the instrumented program and thus it is able to detect a crash and flush the buffer. The process architecture is depicted in figure 3.11.

A function called _cc_sm() that is invoked as the program starts is in charge of generating this process layout and the communication setup. The implementation of this function resides
in the file \_cc\_monitor.c which also defines all of the advice functions and the communication primitives.

### 3.5.1 Communication Protocol

Advice depends on the construct, but all use the same five primitive operations to communicate with the monitor. These are listed in Table 3.1.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>startPred(stmt_id)</td>
<td>Start predicate bundle associated with stmt_id.</td>
</tr>
<tr>
<td>endPred(stmt_id)</td>
<td>End predicate bundle associated with stmt_id.</td>
</tr>
<tr>
<td>pred(stmt_id, pred_id, value:Boolean)</td>
<td>Predicate pred_id, its value, and associated stmt_id.</td>
</tr>
<tr>
<td>fnHit(fn_id)</td>
<td>Reached function fn_id.</td>
</tr>
<tr>
<td>hit(stmt_id)</td>
<td>Reached statement stmt_id.</td>
</tr>
</tbody>
</table>

The first two primitives are used to bookend the evaluation of predicates. This is necessary because control may be interrupted if one of the predicates contains a function call or because of short circuiting. The third primitive is used to send each evaluated predicate to the monitor, noting the enclosing statement, the predicate in question, and its value. Unevaluated predicates
are not recorded, since control bypasses them. The last two primitives are related to hits. The first one is to tell the monitor that a function has been reached. Similarly, the last primitive is used to tell the monitor that a location in code has been reached - usually the start of a basic block or control structure statement.

All of the primitives are sent through the pipe as simple binary packets (big-endian). Packet types come first, followed by an expected format. Table 3.2 defines the packets format for each communication primitive. Note that uppercase strings represent packet types and are defined as integers values in the same way in the instrumented program and the monitor.

Table 3.2 Format of the packets sent through the pipe

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>startPred( stmt_id)</td>
<td>START_PRED stmt_id</td>
</tr>
<tr>
<td>endPred(stmt_id )</td>
<td>END_PRED stmt_id</td>
</tr>
<tr>
<td>pred(stmt_id, pred_id, value:Boolean)</td>
<td>START_PRED stmt_id pred_id value</td>
</tr>
<tr>
<td>fnHit(fn_id)</td>
<td>FNHIT fn_id</td>
</tr>
<tr>
<td>hit(stmt_id)</td>
<td>HIT stmt_id</td>
</tr>
</tbody>
</table>

3.6 Data collection

The program monitor\(^2\) was developed in java and can be divided into two major parts: the pipe decoder and the monitor control. The pipe decoder is responsible for decoding binary information from the instrumented program and the monitor control is responsible for storing coverage information. Communication with the instrumented program is performed through a named pipe, which is a pipe located via the file system. The filename is passed to the monitor by the instrumented program when it starts the monitor. The pipe decoder then reads binary information (encoded in big-endian) from the pipe. In the first phase of decoding, handled by the CMonitorStream class, binary information is converted into messages. All binary information is sent in packets based on primitive communication operations (see table 3.2 above). The format of each packet is known by the instrumented program and by the decoder. In the

\(^2\)An initial version was coded by Jon Mathews as a project in the CprE 556 course. Some modifications were done and new features added to fit 14CC needs.
second phase, messages created in the pipe decoder are sent to the monitor control. Information is gathered and sent to a bucket as appropriate. Block hits and function hits are sent directly to the bucket, while predicate bundles are gathered up and sent on only when they have been completed. The process of managing predicate bundles is facilitated by the PredicateStacks class. When a start predicate message is received, the corresponding predicate stack is pushed. Subsequent predicate hits are placed in the top bundle. A predicate end results in a pop. The predicate bundle is then sent to the bucket with the associated enclosing statement id. Empty predicate bundles are ignored, since they contain no useful information. Empty predicate bundles are generated by if statements without a matching else (this is necessary to keep the predicate stacks balanced).

When the instrumented program finishes or dies abnormally, the named pipe is closed and the pipe decoder executes cleanup code. Any unfinished predicate bundles in the predicate stacks are sent to the bucket, since they were evaluated on the way to the crash and might be useful for debugging. Finally, the monitor process dumps runtime data collected in the bucket to an XML file. The graphical user interface will read this output file to show the code coverage results of the run.

3.7 Visualization

Results will be shown in the graphical user interface. Figure 3.12 shows how code coverage results can be seen in the I4CC tool.

As it can be noticed there are two internal windows. The top window displays a summary of the results with code coverage metrics, more precisely, function coverage and block coverage metrics. The number of blocks/functions hit and missed and the total number of them is also shown. The window on the bottom consists of two panels; the one on the left shows the AST while the one on the right shows multiple condition evaluation values for control structures. The number of hits for each statement can be seen in the AST nodes’ caption as well as on top of the right panel. Whenever the user clicks on a node the panel on the right also shows multiple condition evaluation values if applicable and hits data. The outcome of each condition
Figure 3.12 Visualization of results in I4CC
is shown with three possible values; 'T' and 'F' for true and false respectively. An asterisk is used when the condition was not evaluated in a given decision.

In order to make results more readable a solid blue square appears in the nodes' icon indicating that the statement was hit. The user can easily expand and compress functions and modules providing fast focus on the part of the program of interest.
CHAPTER 4. RESULTS

There are two kinds of results, the first set is intended to show possible applications of I4CC; while the second kind is more related to a set of experiments to show the flexibility of the tool.

Applications of I4CC are covered in section 4.1 in which some characteristics of the tool are brought up. Section 4.2 deals with experiments and results that show statistical data, more specifically function coverage and block coverage. Some experiments done for correctness results are explained as well as some analysis results related to runtime performance.

4.1 Application of I4CC

The following sections bring out some of the features of I4CC and how they can be used for different purposes. The main area of application of the tool is in code testing, in which code coverage results can be used to analyze coverage of each of the test cases. Some of the characteristics of I4CC allow to do some tasks related to program understanding. For instance, section 4.1.1 describes how the tool can be used in the analysis of pairing functions and section 4.2.1.1 explains how we can relate input parameters to functions in the program.

4.1.1 Coverage counters are useful

Some code coverage tools only report whether a given item was exercised or not. The approach we took is to compute the number of times a given item was exercised. We think a number gives a broader idea of coverage since heavy executed code can be spotted. Usually, code testing involves trying a set of test cases in order to compute their coverage. In this case, having a notion of heavy executed code is useful since monitoring that code can be avoided,
Figure 4.1 Experiment showing how simple analysis of pairing functions can be done. Spotting memory leaks.
through the use of rules, with a consequent gain in runtime of test cases. On the other hand, numbers can also be useful in some specific cases for the analysis of pairing functions. Some examples of pairing functions are open-close and getmem-freemem. Usually, the logic of programs requires these functions to be paired. Numbers can be useful to find constructs not following the program’s logic. For example, if we have some knowledge about a piece of code in which getmem and freemem are used, numbers obtained from a given run can be used to check that getmem executed the same number of times as freemem. This information could be used to detect memory leaks. Again, this requires the user to have a minimum knowledge about the code and it is only applicable to cases in which the structure of the program is relatively simple. Figure 4.1 shows an example in which memory is allocated (malloc) and deallocated (free) on two places under the highlighted node. As it can be seen in the AST, the code that allocates memory was hit 5 times while the code on charge of freeing memory was hit only 3 times. Let’s keep in mind that only the first statement of each block is shown in the AST, therefore, memory allocation and deallocation calls could not be directly seen in the AST. The user has to look at the first statement of the block to determine how many times allocation/deallocation occurred.

As we already mentioned, a notion of heavily executed code can be acquired which can be useful for program understanding purposes. There is a high probability that this code corresponds to the core of the program or module. A similar analysis could be done for barely executed code which could correspond to parts of the program related to initialization or finalization code. Furthermore, if we intent to do optimization, numbers are of help since heavily executed code could be the target for it. Even if a time related analysis is more suitable for optimization, number of time a statement was hit can provide some notion of bottlenecks.

4.1.2 Crashes

As explained in section 3.1 the designed runtime architecture allows the collection of runtime data in the presence of program crashes. Whenever the instrumented program crashes, the buddy process detects the crash and proceeds to clean the buffer that is on shared memory.
Figure 4.2 Data collected on program crashes
Data in the buffer is sent to the monitor which process it and dump runtime data to a file. Finally, the user can get the results through the graphical user interface by clicking the button "Get results". The button "Last stmt hit" locates the last statement hit by expanding the AST and highlighting the node. Figure 4.2 shows an experiment in which a statement causing a division by cero was inserted right after the highlighted statement. As it can be seen, the tool is able to detect and show in a friendly interface the point of the crash.

Let's remember that the point of the crash will not be 100% accurate since instrumentation is not done at each statement but at the beginning of each basic block. Furthermore, if a sampling rate of 0.1 or 0.01 is being used, the point of the crash could be far away from the one marked by the tool. There are two possible scenarios; the first one is when the user has no idea as to where the crash could be, in which case a sampling rate of 1 should be used for a second run to get more accuracy about the crash point. The second scenario is when the user has a notion about the crash point, which can be obtained from looking at results in the AST. In this case, a sampling rate of 1 can be set for the outer loop of the statement in which we suspect the crash was and the rest of the loops can be monitored with any sampling rate.

4.1.3 Multiple condition evaluation

Let's bear in mind that multiple condition evaluation refers to keeping combinations of condition values that occurred on a given decision. We found this information useful for the development of test cases. That a given body of a control structure was not exercised can be determined from most code coverage tools. However, if we want to get feedback from code coverage for the creation of tests cases, it is useful to know what the values for the conditions were in order to determine how a test case should be created to achieve more code covered by the test case.

4.2 Experiments and Results

Next section shows analytical results while the remaining two sections show correctness results and some performance experiments done to speed up the runtime data collection process.
Runs were made on a dual pentium Xeon at 2.4GHz with 2GB of RAM. The tool runs under Linux since it makes use of libraries for named pipes.

### 4.2.1 Analytical results

In order to test I4CC a series of experiments with a compression tool called *gzip* were done. The program consisted of 20 files, 14 C files and 6 header files, totalling 8163 lines of code (LOC) and 92 functions. A first set of experiments was done on a *pdf* file (*test.pdf*) of 400KB. A second set of experiments was done again but this time compressing a larger file (*test3.jar*) of 25MB. Experiments were done to show the relation between time and coverage metrics with and without the use of rules. Table 4.1 presents the configurations while tables 4.2 and 4.3 show results for each configuration.

Legend for configurations.

- **mce** = *true/false*: indicates whether multiple condition evaluation was enabled or disabled.
- **srate** = *x*: indicates that a sampling rate of *x* was used.
- **iter(i,j)**: indicates that only iterations on the range (i,j) were monitored and it applies to all loops. Note: (i,-) is read as "from iteration i".

<table>
<thead>
<tr>
<th>Config. #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Non-instrumented version. Original program was run.</td>
</tr>
<tr>
<td>2</td>
<td>mce=true - srate=1 - iter(0,-) - but no data recording.</td>
</tr>
<tr>
<td>3</td>
<td>mce=true - srate=1 - iter(0,-)</td>
</tr>
<tr>
<td>4</td>
<td>mce=false - srate=1 - iter(0,-)</td>
</tr>
<tr>
<td>5</td>
<td>mce=false - srate=0.1 - iter(0,-)</td>
</tr>
<tr>
<td>6</td>
<td>mce=false - srate=0.01 - iter(0,-)</td>
</tr>
<tr>
<td>7</td>
<td>mce=false - srate=1 - iter(0,5)</td>
</tr>
<tr>
<td>8</td>
<td>mce=false - srate=1 - iter(0,2)</td>
</tr>
</tbody>
</table>

Configuration number 2 was created in order to determine the overhead of instrumentation and rules without sending information to the monitor.
The first column on tables 4.2 and 4.3 refers to the configuration as explained in 4.1. The time for each of the runs is shown in seconds. The column \textit{Ratio} indicates how many times longer each configuration took to run compared with the non-instrumented version. For instance, a ratio of 2 indicates that it took twice as much as the non-instrumented version. The last two columns show the function and block coverage metrics.

### Table 4.2 Results obtained while compressing a 400KB file (*test.pdf*)

<table>
<thead>
<tr>
<th>Config. #</th>
<th>Time (seconds)</th>
<th>Ratio</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Function</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Block</td>
</tr>
<tr>
<td>1</td>
<td>0.06</td>
<td></td>
<td>1.09% 0.00%</td>
</tr>
<tr>
<td>2</td>
<td>0.55</td>
<td>9.17</td>
<td>45.65% 27.13%</td>
</tr>
<tr>
<td>3</td>
<td>46.31</td>
<td>771.83</td>
<td>45.65% 27.13%</td>
</tr>
<tr>
<td>4</td>
<td>3.54</td>
<td>59.00</td>
<td>45.65% 27.13%</td>
</tr>
<tr>
<td>5</td>
<td>0.89</td>
<td>14.83</td>
<td>45.65% 26.59%</td>
</tr>
<tr>
<td>6</td>
<td>0.78</td>
<td>13.00</td>
<td>44.57% 23.83%</td>
</tr>
<tr>
<td>7</td>
<td>0.65</td>
<td>10.83</td>
<td>44.57% 23.53%</td>
</tr>
<tr>
<td>8</td>
<td>0.65</td>
<td>10.83</td>
<td>44.57% 23.05%</td>
</tr>
</tbody>
</table>

### Table 4.3 Results obtained while compressing a 25MB file (*test3.jar*)

<table>
<thead>
<tr>
<th>Config. #</th>
<th>Time (seconds)</th>
<th>Ratio</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Function</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Block</td>
</tr>
<tr>
<td>1</td>
<td>3.04</td>
<td></td>
<td>1.09% 0.00%</td>
</tr>
<tr>
<td>2</td>
<td>16.67</td>
<td>5.48</td>
<td>45.65% 27.54%</td>
</tr>
<tr>
<td>3</td>
<td>1888</td>
<td>621.05</td>
<td>45.65% 27.54%</td>
</tr>
<tr>
<td>4</td>
<td>117</td>
<td>38.49</td>
<td>45.65% 27.54%</td>
</tr>
<tr>
<td>5</td>
<td>25.63</td>
<td>8.43</td>
<td>45.65% 27.07%</td>
</tr>
<tr>
<td>6</td>
<td>20.68</td>
<td>6.80</td>
<td>45.65% 25.45%</td>
</tr>
<tr>
<td>7</td>
<td>19.62</td>
<td>6.45</td>
<td>44.57% 24.31%</td>
</tr>
<tr>
<td>8</td>
<td>19.78</td>
<td>6.51</td>
<td>44.57% 23.23%</td>
</tr>
</tbody>
</table>

The overhead of instrumentation, that is only taking into account the code inserted but not sending data to the monitor was of an average of 7.3, which means that the instrumented program will take 7 times longer than the original program.

As it can be observed from the tables, precision of coverage metrics is not seriously reduced when rules are applied. Regarding block coverage, there is a difference of less than 1% when
sampling rate of 0.1 was used, and a maximum of approximately 4% with sampling rate at 0.01 or rules to monitor only the first 2 or 5 iterations of a loop. With respect to function coverage, there is very small reduction in precision, approximately 1%, as rules in favor of performance were used. There was one function that was not hit under certain rules which causes the function coverage reduction. Looking at the ASTs we were able to determine that the function not hit was \textit{fill\_window()}. It was invoked from the body of a loop and under a condition related to the iteration number, which caused certain rules to missed the invocation and thus not hit was recorded for the function.

On the other hand, time was considerable reduced when rules were specified. A sampling rate of 0.1 seems to be the best option since it keeps accuracy while at the same time achieving a great reduction on runtime.

Multiple/condition evaluation could not be acceptable when compressing big files since it greatly increases runtime. However, it could be enabled for short runs or it could be tuned with rules for longer runs to get multiple condition evaluation data at particular statements.

Note that rules allow to create a vast number of configurations. For instance, a combination of sampling rate within a given range could also be specified. The user might want to turn off monitoring on some part of a program since it was exercised by other test. Two rules can be set to stop recording before reaching that part of the program and start after the block of code executed. We made a simple experiment with gzip with \texttt{test3.jar} as input. We realized that code in the method \texttt{treat\_file()} is in charge of the actual compressing and decompressing. We create a rule to skip monitoring at that point getting a run time of 16.63 seconds which is very close to configuration number 2 in which recording is not done. This shows the flexibility of rules to skip parts of a program that do not need to be monitored and to focus monitoring on points of interest. Similarly, rules can be used to focus monitoring at specific points of interest.

There is an observation regarding the runtime difference between the non-instrumented and instrumented versions of a program. Initial tests were done on a program taken from the electromagnetic industry. The runtime ratio for the instrumented version without multiple condition evaluation was of a factor of 2. That is, the instrumented version took twice as much
time. As it can be seen on the previous tables, the runtime factor was of 59 for the first case and 38.49 for the second one. Trying to find an explanation to this difference we computed the average and maximum number of statements per block on both programs. This experiment was done since we are doing block instrumentation; the more statements on a block the less overhead in runtime since each hit sent to the monitor represents a block hit. We found that the average statements per block in gzip was 1.44 while for the other program it was 4.95. Furthermore, the maximum block size in gzip consisted of 10 statements while in the other program was of 79 statements. This analysis led us to the conclusion, as it could be expected, that the larger the blocks the less the impact on runtime in the instrumented program.

4.2.1.1 Merging results from multiple test cases

The main purpose of code coverage metrics is to report how much of the code has been exercised. Usually, testers prepare a set of test cases expecting to exercise the highest number of items, or constructs, in the program. Therefore, it is important for code coverage tools to provide the capability of merging results from different runs. 14CC provides this feature and the following experiment was carried out.

We created eight test cases, each of them consisted of the same file to compress/decompress but using different parameters to gzip; see table 4.4. We ran gzip on each test case and results for each run were collected. Finally, results from the multiple runs were merged to get coverage metrics on all the test cases. Table 4.5 shows results on each test case and results on all test cases (last row).

As it can be noticed, coverage metrics from the merged results is low. We did not expect to achieve 100% coverage since we did not try with all possible gzip parameters, but we thought it was to low. We looked at the AST to see which functions were not exercised and we realized there were some modules in gzip that are intended to decompress some other file types, such as LZW and pack. Clearly, those functions and blocks were not exercised since we were using the gz file format. This experiment showed the ability of 14CC to merge results from multiple runs.
Table 4.4 gzip parameters description

-\textbf{-v}: verbose mode
-\textbf{-f}: force overwrite of output file and compress links
-\textbf{-q}: suppress all warnings
-\textbf{-c}: write on standard output, keep original files unchanged
-\textbf{-d}: decompress
-\textbf{-l}: list compressed file contents
-\textbf{-t}: test compressed file integrity
-\textbf{-h}: display help
-\textbf{-V}: display version number
-\textbf{-1}: compress faster
-\textbf{-9}: compress better

Table 4.5 Merging results from different runs

<table>
<thead>
<tr>
<th>Test case</th>
<th>Functions hit</th>
<th>Function Cvg.</th>
<th>Blocks hit</th>
<th>Block Cvg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>-vf -1</td>
<td>43</td>
<td>46.74%</td>
<td>450</td>
<td>26.95%</td>
</tr>
<tr>
<td>-vf -9</td>
<td>42</td>
<td>45.65%</td>
<td>457</td>
<td>27.37%</td>
</tr>
<tr>
<td>-qc</td>
<td>33</td>
<td>35.87%</td>
<td>400</td>
<td>23.95%</td>
</tr>
<tr>
<td>-vd</td>
<td>29</td>
<td>31.52%</td>
<td>365</td>
<td>21.86%</td>
</tr>
<tr>
<td>-vl</td>
<td>15</td>
<td>16.30%</td>
<td>151</td>
<td>9.04%</td>
</tr>
<tr>
<td>-vt</td>
<td>22</td>
<td>23.91%</td>
<td>332</td>
<td>19.88%</td>
</tr>
<tr>
<td>-h</td>
<td>6</td>
<td>6.52%</td>
<td>34</td>
<td>2.04%</td>
</tr>
<tr>
<td>-V</td>
<td>5</td>
<td>5.43%</td>
<td>35</td>
<td>2.10%</td>
</tr>
<tr>
<td>Merge</td>
<td>57</td>
<td>61.96%</td>
<td>763</td>
<td>45.69%</td>
</tr>
</tbody>
</table>

Another point we would like to bring out is the possibility of doing some program understanding activities. For instance, which methods were executed only under a given parameter. The function \textit{deflate.fast()} in module \textit{deflate.c}, for example, is only invoked when fast compression is used, that is using parameter \textit{-f}.

Since I4CC allows to have multiples AST views of the same program, analysis related to which functions are exercised by each of the parameters is relatively simple. For instance, when compressing files, most of the functions in the module \textit{infla.t.e.c} get called while functions on \textit{deflate.c} are not executed. The opposite occurs when decompressing files. In this case, the analysis was straightforward by just looking at results from two runs, one compressing and the other one decompressing, in two ASTs.
Similarly, when parameter -q (suppress all warnings) was used, the set of functions that perform this task could be identified by comparing with a normal run. We found the following function names related to the -q option: display_ratio, strlwr, create_outfile, get_suffix, name_too_long, check_ofname and copy_stat.

4.2.2 Correctness results

An experiment was carried out in order to be certain that results were correct. The experiment consisted of associating counters with some random selected statements. Whenever the statement was hit the corresponding counter was incremented. Code for the increment, as well as the global counters, were inserted in the already instrumented program. At the end of the run, the counters' values were compared against the number of hits reported by the tool for each of the statements. Results computed by the tool showed consistency with the counters' values.

4.2.2.1 Block coverage of gzip with different input files

We noticed that different block coverage results were obtained when different input files were compressed with the instrumented version of gzip. That is, some blocks were hit depending on the input to gzip. This behavior made us think there was a flaw in I4CC, and thus, the following experiment was carried out.

Two input files for gzip were considered, test.pdf (400KB) and test1.pdf (5.8MB). The instrumented version of gzip was run with each of the test files. We created a set containing some of the blocks IDs that were only hit in one of the runs.

Next, those blocks were identified in the instrumented version of gzip and were hardcoded to print a message to certify whether the block was hit or not. This part of the experiment was done in order to be sure whether the blocks were actually exercised during the run.

The following step consisted of running the instrumented version of gzip again on both test cases and check whether the hardcoded messages showed up. As expected, the messages appeared only while running one of the test cases, confirming our hypothesis that blocks were
hit depending on the input to gzip.

In order to be certain about the experiment we made the next experiment to determine whether the instrumented version was producing the correct output (the compressed file). Both of the compressed files were decompressed with the original gzip program and they were open with a pdf reader. The structure of the test files was correct since we were able to open them and we also checked that they contained the original information.

Therefore, we think that the fact that different blocks get hit for different input file sizes is a normal behavior of gzip.

4.2.3 Performance results

The first test runs were done on a program built for the electromagnetic industry. It consisted of 2686 lines of C code. Since the first experiments done with the tool were extremely slow, some performance optimizations were considered. A set of design and implementation improvements done are explained below. Bear in mind that the runtime architecture consists of the instrumented program (C code) communicating through a pipe with the monitor program that was implemented in Java.

While the non-instrumented version of the program ran in 3 minutes the instrumented version took 306 minutes. There was a huge overhead in process communication; this was noticed by computing the amount of data transferred and from the observation that processors' load was low. As we explained in section 3.5.1 the communication requires sending XCIL IDs through a named pipe. Since IDs consisted of a 9-digit string most of the overhead was sending IDs. Thus, the first optimization (opt#1) consisted of converting XCIL IDs to integer representation before sending them through the pipe. As it can be seen in table 4.6 this change brought a considerable reduction in runtime. The second improvement (opt#2) involves passing parameters to the java virtual machine (JVM) that executes the monitor. When no parameters were considered, the JVM allocated a default amount of memory (8MB) which caused the monitor to run very slowly. The parameters -Xmsn and -Xmxn were used to tell the JVM how much memory should be used initially and what the maximum memory
allocation pool should be. There still was a huge overhead in communication and we realized that the packets to send predicates' evaluation consisted of the packet type, the predicate value and two identifiers. Thus, our hypothesis was that the evaluation of predicates was causing the execution slowdown. In order to confirm our hypothesis we disabled evaluation of predicates, which resulted in a runtime reduction of 33% compared to the previous optimization. The next optimization, (opt#4) consisted of buffering packets on the sending size before sending them. Similarly, buffering was also implemented on the receiver side (opt#5). Finally, some miscellaneous optimization in the sending of predicate evaluations brought the program to run in 4 minutes.

Table 4.6 Performance results

<table>
<thead>
<tr>
<th>Optimization #</th>
<th>Short Description</th>
<th>Runtime (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-instrumented version</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Fully-instrumented version</td>
<td></td>
<td>306</td>
</tr>
<tr>
<td>1</td>
<td>int representation for XCIL IDs</td>
<td>105</td>
</tr>
<tr>
<td>2</td>
<td>-Xms256m and -Xmx512m JVM parameters</td>
<td>75</td>
</tr>
<tr>
<td>3</td>
<td>Predicate evaluation disabled</td>
<td>49</td>
</tr>
<tr>
<td>4</td>
<td>Buffer on sending side</td>
<td>29</td>
</tr>
<tr>
<td>5</td>
<td>Buffer on sending &amp; receiving sides</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>misc improvements</td>
<td>4</td>
</tr>
</tbody>
</table>

It is important to note that all these experiments were done in the initial stages of the tool. There were some changes as the research went on and I would like to mention the most important ones. Initially, only the body of each control structure was monitored and not each of the basic blocks. The change was intended to provide the block coverage metric. Note that the body of a control structure can easily contain more that one basic block; therefore, the current implementation possess more overhead since a packet has to be send to the monitor each time the beginning of a basic block is reached. Rules were not implemented at that time, so the reported results were computed with a fix instrumentation.

Another important issue I would like to bring up is that experiments with Rational Pure-Coverage [2] were done with the same test program. Purify ran on a Hyperthreaded Pentium 4 at 2.4GHz with 512MB of RAM and it took 24 minutes to run.
CHAPTER 5. SUMMARY AND DISCUSSION

Software development cycles are getting shorter while the size and complexity of software systems keeps increasing. There are deployment due dates that has to be met and usually little time for testing is left. Changes on existing code because of features addition or updates require developers to do testing again. This brings the need to find a way to speed up the process of testing. One kind of tools in the testing process are code coverage tools. In order to meet testing requirements they should provide enough flexibility to the developers. This work presented the design and implementation of a code coverage tool whose main purpose was to allow tuning the performance and precision variables in the process of obtaining code coverage metrics.

5.1 Conclusions

Some of the characteristics of the tool are its flexibility to focus code coverage analysis on particular parts of the program. This capability is achieved by the selective instrumentation feature in which the user can select which functions to monitor from the list functions in the program. Rules can also be used to set focus on particular parts of the program by disabling and enabling recording of runtime data. Another characteristic is the possibility of tuning runtime data precision or runtime performance. The user can perform this tuning by setting sampling rates for loops. The possibility of disabling/enabling multiple condition evaluation also helps to focus analysis at particular decisions or speed up execution by disabling it on parts of the program of no interest. One interesting feature is the possibility to detect the point of crashes as it is explained in section 4.1.2. Finally, the ability to combine results from different runs is of utmost importance in code testing, since different test cases exercise different parts of
the program. Therefore, coverage of a set of test cases can be computed by combining results from several runs as it was shown on section 4.2.1.1.

Results from the experiments carried out indicated that accurate enough results were obtained using rules to decrease runtime. A sampling rate of 0.1 for loops seems to be the ideal balance between performance and precision.

Results showed that, with the specified rules, an average of 1-4% of accuracy in block coverage is lost compared to the run with sampling rate at 1. Regarding function coverage, the reduction on precision is insignificant regardless of the applied rules. Rules for monitoring the first $x$ iterations on loops presented promising results and they could be more beneficial on larger runs.

Summing up, this work presented design and implementation of a flexible code coverage tool. Experiments on a real application were done and some results were shown and possible applications described.

5.2 Future Work

Possible extensions and improvements to this work are stated below.

- Selective instrumentation is at function level in the current implementation, but it could be extended to select/unselect modules.

- Currently, coverage metrics values are on a per program basis. It could be interesting to differentiate results for different functions or different modules. For instance, block coverage results could be presented separately for each of the functions. Similarly, function coverage results could be presented on a per module basis.

- Handling of recursion can be implemented, although as mentioned in section 2.4 there could be a great impact on overhead.

- Another interesting result could be providing the user with the number of times each combination of values in a decision occurred.
• Allow the user to select/unselect the monitoring of coverage metrics separately, that is, function coverage and block coverage.
APPENDIX. INSTRUMENTATION SCHEME

Location advice for each program construct is shown below. Words in brackets represent
joinpoints. Each joinpoint has a corresponding advice. Because of the issues with the recording
of multiple condition evaluation (see section 3.4.1.2), if/then is treated differently from
if/then/else. An if/then is referred to here as an unmatched if.

Constructs

"main"

main() {
    <declarations>
    [START MAIN]
    <body>
}

"if/then"

[BEFORE UNMATCHED\_IF ]
if ( [PRED]+ )
{
    <declarations>
    [START UNMATCHED\_THEN]
    <body>
    [END UNMATCHED\_THEN]
"if/then/else"

if ( [PRED]+ )
{
    <declarations>
    [START THEN]
    <body>
}
else
{
    <declarations>
    [START ELSE]
    <body>
}

"switch"

switch( ) {
    case 1:
        [START CASE]
        <body>
    case 2:
        [START CASE]
        <body>
    default:
"function"

\[ f() \]

\[
\begin{array}{ll}
\text{before function} & \\
\text{before for} & \\
\text{begin} & \\
\text{before while} & \\
\text{begin while} & \\
\end{array}
\]

\[
\begin{array}{ll}
\text{after function} & \\
\text{after for} & \\
\text{after while} & \\
\end{array}
\]
dowhile

do {
<dowhile>
<body>
}

"start basic block"

<first statement of a basic block>
Legend

Joinpoints and corresponding advice are shown in the following table.

<table>
<thead>
<tr>
<th>&quot;joinpoint&quot;</th>
<th>&quot;advice&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>[PRED]+</td>
<td>_cc.pred(&quot;enclosing stmt ID&quot;, &quot;pred expr ID&quot;, value);</td>
</tr>
<tr>
<td>[START MAIN]</td>
<td>_cc.sm();</td>
</tr>
<tr>
<td>[BEFORE UNMATCHED IF]</td>
<td>_cc.before.uif(&quot;if stmt ID&quot;);</td>
</tr>
<tr>
<td>[AFTER UNMATCHED IF]</td>
<td>_cc.after.uif(&quot;if stmt ID&quot;);</td>
</tr>
<tr>
<td>[START UNMATCHED_THEN]</td>
<td>_cc.start.uthen(&quot;if stmt ID&quot;);</td>
</tr>
<tr>
<td>[END UNMATCHED_THEN]</td>
<td>_cc.end.uthen(&quot;if stmt ID&quot;);</td>
</tr>
<tr>
<td>[BEFORE IF]</td>
<td>_cc.before.if(&quot;if stmt ID&quot;);</td>
</tr>
<tr>
<td>[START THEN]</td>
<td>_cc.start.then(&quot;if stmt ID&quot;);</td>
</tr>
<tr>
<td>[START ELSE]</td>
<td>_cc.start_else(&quot;if stmt ID&quot;);</td>
</tr>
<tr>
<td>[START CASE]</td>
<td>_cc.start_case(&quot;case/block ID&quot;);</td>
</tr>
<tr>
<td>[START FUNCTION]</td>
<td>_cc.start.function(&quot;function ID&quot;);</td>
</tr>
<tr>
<td>[BASIC BLOCK]</td>
<td>_cc.start.basicBlock(char * blkID);</td>
</tr>
<tr>
<td>[BEFORE FOR]</td>
<td>_cc.before.for(&quot;for stmt ID&quot;, &quot;counterIndex&quot;);</td>
</tr>
<tr>
<td>[AFTER FOR]</td>
<td>_cc.after.for(&quot;for stmt ID&quot;);</td>
</tr>
<tr>
<td>[START FOR]</td>
<td>_cc.start.for(&quot;for stmt ID&quot;, &quot;counterIndex&quot;);</td>
</tr>
<tr>
<td>[END FOR]</td>
<td>_cc.end.for(&quot;for stmt ID&quot;);</td>
</tr>
<tr>
<td>[BEFORE WHILE]</td>
<td>_cc.before.while(&quot;while stmt ID&quot;, &quot;counterIndex&quot;);</td>
</tr>
<tr>
<td>[AFTER WHILE]</td>
<td>_cc.after.while(&quot;while stmt ID&quot;);</td>
</tr>
<tr>
<td>[START WHILE]</td>
<td>_cc.start.while(&quot;while stmt ID&quot;, &quot;counterIndex&quot;);</td>
</tr>
<tr>
<td>[END WHILE]</td>
<td>_cc.end.while(&quot;while stmt ID&quot;);</td>
</tr>
<tr>
<td>[BEFORE DOWHILE]</td>
<td>_cc.before.dowhile(&quot;dowhile stmt ID&quot;);</td>
</tr>
<tr>
<td>[AFTER DOWHILE]</td>
<td>_cc.after.dowhile(&quot;dowhile stmt ID&quot;);</td>
</tr>
<tr>
<td>[START DOWHILE]</td>
<td>_cc.start.dowhile(&quot;dowhile stmt ID&quot;, &quot;counterIndex&quot;);</td>
</tr>
<tr>
<td>[END DOWHILE]</td>
<td>_cc.end.dowhile(&quot;dowhile stmt ID&quot;);</td>
</tr>
</tbody>
</table>
Bibliography


   http://dirac.ece.iastate.edu/sec/.


ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Suraj Kothari for his guidance, patience and support throughout this research and the writing of this thesis. His insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education. I would also like to thank Jon Mathews for the fruitful discussions and his insights. My committee members for their efforts and contributions to this work: Dr. Manimaran Govindarasu and Dr. Shashi Gadia.