Linköping University | Department of Computer and Information Science
Bachelor's Thesis | Computer Engineering – Software and Systems
Early-Mid Autumn of 2016 | LIU-IDA/LITH-EX-G—16/067—SE

Analysis of Test Coverage Data on a Large-Scale Industrial System

by Erik Sven Vasconcelos Jansson

Examinator at Linköping University: Dr. Lena Buffoni
Advisor at Linköping University: Dr. Bernhard Thiele
Advisor at Ericsson Linköping: Dr. Sergiu Rafiliu
Thesis Opponent: Rickard Sven Jonsson

Linköping University (LiTH)
SE-581 83 Linköping, Sweden
+46 013-28 10 00, www.liu.se

Linköping University Electronic Press

Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/her own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Erik Sven Vasconcelos Jansson

Analysis of Test Coverage Data on a Large-Scale Industrial System

Bachelor's Thesis by
Erik S. Vasconcelos Jansson
erija578@student.liu.se

Division for Software and Systems at the
Department of Computer and Information Science
at Linköping University (LiTH), Sweden

October 3, 2016

Abstract

Software testing verifies the program's functional behavior, one important process when engineering critical software. Measuring the degree of testing is done with code coverage, describing the amount of production code affected by tests. Both concepts are extensively used for industrial systems. Previous research has shown that gathering and analyzing test coverages becomes problematic on large-scale systems. Here, development experience, implementation feasibility, coverage measurements and analysis method are explored; providing potential solutions and insights into these issues.

Outlined are methods for constructing and integrating such gathering and analysis system in a large-scale project, along with the problems encountered and given remedies. Instrumentations for gathering coverage information affect performance negatively, these measurements are provided. Since large-scale test suite measurements are quite lacking, the line, branch, and function criteria are presented here. Finally, an analysis method is proposed, by using coverage set operations and Jaccard indices, to find test similarities.

Results gathered imply execution time was significantly affected when gathering coverage, $[2.656, 2.911]$ hours for instrumented software, originally between $[2.075, 2.260]$ on the system under test, given under the $\alpha = 5\%$ and $n = 4$, while both processor & memory usages were inconclusive. Measured criteria were $(59.3, 70.7, 24.6)\%$ for these suites. Analysis method shows potential areas of test redundancy.

Acknowledgments

You hold in your hands something which would never have been realised without the help from several individuals, which spent time and effort to mentor and give feedback; allowing this work to achieve higher quality & correctness. These people enabled this thesis to be interesting and fun!

Regarding the written paper, both Dr. Lena Buffoni and Dr. Bernhard Thiele, my helpful examinator and advisor, have been of invaluable help during the entire process, by giving me good feedback and advice every step of the way. Special thanks goes towards Lena's husband, who helped in explaining several important concepts within statistics. Lastly, my friend Rickard Jonsson, who drafted the theory, part of, when we worked together in a preparatory course taught by Dr. Ola Leifler, preparing us for writing research.

Huge thanks for everyone at Ericsson Linköping for their gracious hospitality and help with the system under test. Especially, I would like to thank my advisor, Sergiu Rafiliu, for his help and support during my entire stay at Ericsson, and Johan Moe, for presenting me with this opportunity. Additionally, Tony Olsson for sharing his knowledge about the build system (and being a cool C wizard in general) and Maria Lingemark for her feedback on our commits, which indeed dearly needed a review before being merged.

Erik Sven Vasconcelos Jansson
Linköping University, Sweden

Contents

1 Introduction **1**
1.1 Motivation for Study . . . 2
1.2 Problem Formulation . . . 3
1.3 Purpose of Research . . . 3
1.4 Research Questions . . . 4
1.5 Thesis Delimitations . . . 4

2 Background **5**
2.1 Ericsson Linköping . . . 5
2.2 Development Overview . . . 6
2.3 Project Description . . . 6
2.4 Main Challenges . . . 7
2.5 Specifications . . . 7

3 Theory **8**
3.1 Software Testing . . . 8
3.1.1 White-Box and Black-Box Testing . . . 8
3.1.2 Unit, Integration and System Testing . . . 9
3.1.3 Software Verification and Validation . . . 9
3.1.4 Test Case Similarity & Uniqueness . . . 9
3.2 Continuous Integration & Test Automation . . . 10
3.2.1 Measuring Relative Test Suite Quality . . . 10
3.3 Code Coverage & Criterion . . . 11
3.3.1 Different Uses of Code Coverage Data . . . 11
3.3.2 Statement, Function and Branch Criteria . . . 12
3.3.3 Code Coverage Testing Tools Overview . . . 12
3.3.4 Fallacies of Code Coverage as a Metric . . . 12
3.4 Coverage Analysis Methods . . . 13
3.4.1 Set Operations on Coverage Data . . . 13
3.4.2 Substring Hole Analysis Method . . . 13
3.5 Measuring Performance . . . 14
3.5.1 Program Running Time . . . 14
3.5.2 Memory Usage of Program . . . 14
3.5.3 Processing Time Consumed . . . 14
3.6 Statistical Confidence . . . 15
3.6.1 Student's T-Distribution . . . 15
3.6.2 Sampling of Population . . . 15
3.6.3 Confidence Interval . . . 15

4 Method 16
4.1 System Architecture . . . 17
4.2 Program Instrumentation . . . 19
4.2.1 Compiler Coverage Settings . . . 19
4.2.2 Coverage Environment Setup . . . 19
4.3 Sampling Performance Data . . . 20
4.3.1 Gathering Raw Performance Data . . . 20
4.3.2 Statistical Performance Analysis . . . 20
4.4 Flushing Daemon Coverage . . . 21
4.4.1 Forcing Coverage Data Dumps . . . 21
4.4.2 Storing Data on Remote Target . . . 21
4.4.3 Custom Daemon User Signal . . . 21
4.5 Automatic Data Gathering . . . 22
4.5.1 Changes to the Testing Flow . . . 22
4.5.2 Separating Test Case Coverage . . . 23
4.5.3 Fetching Remote Coverage Data . . . 23
4.6 Building Code Coverage Reports . . . 24
4.6.1 Coverage File Information . . . 24
4.7 Similarity-Based Coverage Analysis . . . 25
4.7.1 Coverage Set Operations & Similarity . . . 25
4.7.2 Raw Profile Coverage Data Manipulation . . . 26
4.7.3 Transforming Intermediate Coverage Data . . . 27
4.7.4 Measuring Manual Inspection Reduction . . . 27

5 Results 28
5.1 Full Test Suite Coverage . . . 28
5.1.1 Coverage by Criteria . . . 28
5.2 Instrumented Performance . . . 29
5.2.1 Suite Execution Time . . . 29
5.2.2 Average Memory Usage . . . 30
5.2.3 Average Processor Usage . . . 31
5.3 Coverage Inspection Reduction . . . 32

6 Discussion 33
6.1 Methods Utilized . . . 33
6.2 Results Encountered . . . 34
6.3 Threats to Validity . . . 34
6.4 Reference Criticism . . . 35
6.5 Safety Considered . . . 35
6.6 Future Research . . . 35

7 Conclusions 36
7.1 Development Experience . . . 36
7.2 Implementation Feasibility . . . 37
7.3 Coverage Measurements . . . 37
7.4 Analysis Interpretation . . . 38

A Suite Coverage Report

B Coverage Tool Patches

List of Acronyms

BB BitBake Build System

CLI Command Line Interface

DUT Device Under Test

GCC GNU Compiler Collection

GNU GNU's Not Unix

IBM International Business Machines

ICT Information and Communications Technology

LTP Linux Test Project

MIPS Million Instructions Per Second

NIST National Institute of Standards and Technology

RISC Reduced Instruction Set Computing

SCovAT Set Coverage Analysis Tool

SFTP SSH File Transfer Protocol

SUT System Under Test

TS Testing Server

List of Algorithms

4.1 Intersecting Arc Transitions 26

List of Figures

2.1 Development Process Overview 6

3.1 Intuitive Representation of the Testing Strategies 8
3.2 Graphical Visualisation of Coverage Set Operations 13

4.1 Software Instrumentation Overview 17
4.2 Daemon Flushing Overview 17
4.3 Performance Gathering Overview 18
4.4 Coverage Fetching Overview 18
4.5 Test Case Analysis Overview 18
4.6 Overview over the Testing Flow 22
4.7 The Coverage Fetching Operations 23
4.8 Theoretical Scenario Similarity Measure 25
4.9 Conceptual Visualization of the Arc Transitions 26

5.1 Excerpt from Coverage Report on Full Testing Suite 28
5.2 Instrumented Average Memory Usage over Time 30
5.3 Non-Instrumented Average Memory Usage over Time 30
5.4 Instrumented Average Processor Usage over Time 31
5.5 Non-Instrumented Average Processor Usage over Time 31

A.1 Continued Coverage Report on the Full Testing Suite . . . .

List of Listings

4.1 The Factorial Function 19
4.2 Instrumented Factorial 19
4.3 Sample Gathering Script 20
4.4 Performance Data Script 20
4.5 Coverage Setup Procedure for Daemon 21
4.6 Automatic Coverage Report Creation Script 24
4.7 Intermediate GCov Coverage File Format Definition 27
4.8 Session Workflow for the Set Coverage Analysis Script 27
B.1 GCC GCov-Tool Patch for Intersection and Difference . . . . . .

List of Tables

5.1 Instrumented Total Test Suite Execution Time Data 29
5.2 Non-Instrumented Total Test Suite Execution Time Data 29
5.3 Confidence Intervals of Test Suite Time Durations 29
5.4 Confidence Intervals for Average Memory Utilization 30
5.5 Confidence Intervals for Average Processor Utilization 31
5.6 Analysis Results Between the Similar `IPForwarding` Test Cases 32
5.7 Analysis Results Between Different `PL1` & `IPForwarding` Tests 32
5.8 Manual Inspection Reduction for Tables 5.6 (Left) & 5.7 (Right) 32

Chapter 1

Introduction

Software has become an essential component in modern developed societies. Solving a wide variety of practical and theoretical problems previously thought unfeasible, or increasingly tedious, without the automation enabled by computer programs; it enjoys widespread use in a multitude of different fields. Many of these applications are critical, where severe monetary or even life loss can occur if they don't function correctly (e.g. stock trading[1] or radiotherapy machines[2]). Some of the primary goals of *software engineering* have been to emphasize the importance of software quality and correctness, which became defined with the birth of the field in 1969 under a NATO [BR70] conference with the same name.

Enter the activity of *software testing*, which is a process that attempts to assert the behavior of executing programs, better described within the research of *Harrold* [Har00] as: *"testing is part of quality assurance, where one attempts to gather information about the nature of the software"*. This is done by specifying several *test cases* as input data to the executing program, where the output is observed and compared to the specification; these results are often reported automatically. For further enforcing the importance of software testing, existing research by *Boehm et al* [B$^+$81] has shown that over 50% of the total cost of software development can be attributed to testing, whereupon critical software could have an even higher cost for testing. Because of this, testing is important in the field of *software engineering* and also to the topic at hand, therefore, a short overview of its central concepts will be presented later in Section 3.1 under *Theory* in Chapter 3. However, how does one assert that the test cases themselves are testing the relevant portions of the software? When is it enough?

Since the dawn of testing, the concept of *code coverage* has been presented as one possible method to answer the above questions and similarly related topics. Usually attributed to *Miller et al* [MM63] for their paper in 1963, code coverage attempts to mediate the *degree* of testing by presenting the *amount of executed source code* of a particular *test suite* (a set of test cases) for a given criteria. Since this topic is of central importance to the thesis, Section 3.3 provides a brief overview of the topic, while also exploring how to gather coverage in practice. But is this method feasible in todays systems that are both growing larger every day while also needing to perform well? Also, even if the system is feasible to implement, how does one extract meaningful properties from the gathered data?

[1]Trading algorithm glitch in *Knight's* software caused losses of $440M in just 30 minutes.
[2]Between 1985–1987, the *Therac-25* software bugs caused six patients radiation overdose.

Several sources from existing research have stated the imminent difficulties that arise when doing function test coverage on a large-scale industrial system. One such paper, written by *Adler et al* [ABR+11] under *IBM Research*, pointed out two major reasons for this being problematic. First, collecting the actual data is not easy. Tools don't integrate perfectly with the existing developer environment (build system, compiler, test framework), since building a general solution is neigh impossible. Moreover, performance will be degraded with the overhead present from the data gathering instrumentations, many systems are in the spectrum of *real-time*, where the behaviour can therefore be impacted negatively, and will probably require special considerations. The second, more challenging problem is how to present the collected information in a meaningful way, since data inspectors will face the so called *"needle in a haystack"* problem. Section 3.4 presents results of previous research dealing with the second problem.

Request for the research and implementation of such a coverage gathering system and accompanying analysis tool was done by *Ericsson Linköping*, where the existing development process/environment would be extended to support coverage *instrumentation*, *collection*, *reporting* and *analysis* of the *function tests*. In summary, this thesis presents the *development experience* of such a system for future researchers, it gives *implementation feasibility* in respect to *performance* (so implementors know the impacts of such an extension), it also provides the *coverage measurements* of their *primary test suite* as seen by different *coverage criteria*, since such data is lacking for large-scale industrial systems, finally, the *properties* and relative *effectiveness* of the chosen *analysis method* are presented.

Now for a brief overview of the contents of this chapter: Section 1.1 follows below, providing the primary reasons why this study is needed, while Section 1.2 gives a formal definition of the problems that are to be investigated. Thereafter, Section 1.3 gives the main contributions this thesis brings forward, the research questions that are expected to be answered are provided in Section 1.4, finally, the delimitations that restrain this thesis's scope are given later in Section 1.5.

1.1 Motivation for Study

Since most software is expected to comply with a set of requirements, testing is usually used to assert the behaviour per some technical specification. Measuring the amount of code exercised by testing can be gathered with coverage, revealing potential locations of un-/tested code. Issues arise when gathering and analyzing coverage on large-scale industrial systems, as discussed by *Adler et al* [ABR+11]. Research within this area is also scarce, the articles by *Piwowarski et al* [POC93] and *Kim Woo Yong* [Kim03] dealing in this area also mention this. Additionally, the author of this paper found few articles dealing with the practical experience.

Specifically, *Adler et al* [ABR+11] points out two major problems with code coverage on large-scale systems, *gathering* and *analysis*. *Gathering* is an issue, since the extensions need to be specifically tailored for a particular build system and also because running performance will be affected. *Analysis* is hard since the amount of coverage data is huge, especially on these large-scale systems, making sense of it to extract useful information from it manually takes time. Therefore, this thesis attempts to provide results and insight into the problems which are classified as troublesome when attempting to either *gather* or *analyze test coverage* from *large-scale systems*, which are beneficial for future engineers.

1.2 Problem Formulation

Extending an large-scale industrial development ecosystem with code coverage gathering and analysis capabilities does pose a series of problems, as presented by articles such as e.g. *Adler et al* [ABR+11]. Performance degradation occurs when a program is instrumented with code coverage gathering abilities, this can cause the software to misbehave and test cases to start failing randomly. Building such a coverage system also requires special care, since many of the choices need to be tailored for a particular setup (e.g. build system, compiler), where the existing coverage gathering tools don't necessarily integrate perfectly.

However, even if the coverage gathering system is successfully integrated, the data being produced when trying to observe the test coverage of a certain test suite is overwhelming. Certainly, using it for the purpose of assessing the quality of the suite is possible and standard, by using the overall metric quantity. But, in order to extract any meaningful data regarding the relationship between the test cases and program code, more powerful ways to analyze the raw data are required, which is further enforced by the research of *Adler et al* [ABR+11]. Such transformation of the original data is required, since the engineer would otherwise be burdened with examining potentially large amounts of source code to determine any interesting properties present within the set of function tests.

1.3 Purpose of Research

Four major contributions are provided within this thesis. First, the development *experience* (problems and solutions) of integrating code coverage gathering on a large-scale industrial system is provided. This information could prove useful for both researchers and implementors attempting to construct a similar system.

Second, this thesis explores the *feasibility* of executing coverage instrumented code while testing is in progress; investigating extra resources used, additional time taken and system stability issues. Relevant data is provided for this, giving implementors concrete evidence if coverage gathering is possible on their system.

Third, it presents the coverage *measurements* gathered from running the full test suite of the industrial system, giving data about the amount of production code covered by this full test suite in the given set of available coverage criteria. These results could prove to be quite useful for future researchers, since it gives insight into the code coverage of a large-scale industrial system, which is lacking.

Finally, the fourth contribution deals with *analyzing* the large amounts of coverage data produced, proposing a few basic operations that could prove useful for extracting a certain set of meaningful answers from the target system tests. Since manually filtering through the coverage data is problematic (because of the sheer size of the code base), this could also prove useful for other researchers. The *properties* enabled by these developed *tools* will be explored in this thesis.

Therefore, in retrospective, this thesis deals in observing inherent problems that arise when coverage gathering functionality is integrated in such large-scale industrial systems. Thereafter, the thesis tries to solve and present certain such problems, dealing in both *implementation feasibility* and *coverage data analysis*. Finally, it also presents *coverage measurements* that can be used for future work, since it has been noted by *Piwowarski et al* [POC93] and *Kim Yong* [Kim03], that *coverage measurements* from *large-scale industrial software* are quite scarce.

1.4 Research Questions

After such a coverage gathering system has been integrated into the project by using the described method, results leading to the answers of the following research questions below are expected to be presented at the end of this thesis:

- **Feasibility:** what *performance effects* does the *code coverage gathering* method entail on a *large-scale software system* while *testing is in progress*? Does this inadvertently *affect the behavior* of the *software* being tested on?
- **Measurements:** how much is the *full test suite coverage* of the different *coverage criteria* on this *large-scale industrial system*? Are these results *compatible with existing research* dealing with *software on the same scale*?
- **Interpretation:** does the *selected coverage analysis utility* provide any *useful information* about the *software system under test*? By what *amount* is the *quantity* of *gathered* data *reduced* by when *analyzing* the *coverage*?

Since the questions presented above are not specific enough, they do leave a lot of room for interpretation. To produce more concrete results, delimitations have been imposed on the thesis itself. These are described in Section 1.5 below.

1.5 Thesis Delimitations

Considering that application performance can be measured in various different ways, only a few relevant metrics (in regards to this system) will be chosen. These include the *suite running time*, *total memory usage* and *processor time*. All of these will be *sampled several times* for a *statistically meaningful* answer. Something is classified as affecting the software's behavior when test cases fail.

Regarding the coverage measurements being done for the test suite coverage, are only to be done for the coverage criteria available in the chosen code coverage gathering utility, which has been selected to be *gcov* since the system uses *GCC*. It produces *statement coverage*, *branch coverage* and *function coverage*. Results will be presented as the *ratio of measured coverage by total parts instrumented*. Systems *similar in size* are classified as having *hundreds of thousands* code lines.

Precise definitions for what can be regarded as *useful information* is entirely based on context, and is therefore unsuitable as a metric. However, the issue being studied is *coverage analysis* on a *particular project*, which in this case does have precise definitions on what type of information is relevant/appropriate. Here, locations of *test case similarity* and *test case uniqueness* are important. Therefore, this thesis won't try prove these are useful, it will simply assume it. *Reductions* in the *amount of data to analyze* manually is measured as the *ratio* between the *total measured coverage* and *transformed coverage* by the method.

Since the results gathered come from this specific system, and these metrics are usually tightly coupled with it, researchers will need to carefully consider if the conclusions reached really apply to their own system. Therefore, the solution and results presented here will not be viewed from a very general perspective, instead, the reader will need to reason about the similarities the systems have. Additionally, this thesis does not mean to describe the theoretical concepts in painstaking detail, only a short overview is given here, so see relevant literature.

Chapter 2

Background

Since this thesis deals with a real-world large-scale industrial system, and also because the choices made are directly influenced by it, a thorough description of the background related to this thesis is required. This is done so the astute reader can easier relate to the problem presented at this particular company's project, and therefore determine if the results presented are of any relevance. Also, by doing this, the reader can understand the reasoning for several of the choices made in this thesis, which might not have been obvious given the context.

While not technically relevant to the thesis content, a short description of the company for which this thesis was requested, is presented in Section 2.1. This provides the reader with a concrete context of the scale and importance of the software under test, which gives most of the choices made a logical explanation.

Many of the choices made regarding the architecture and tools used, are based to accommodate the system already available at Ericsson Linköping. Therefore, a overview of the relevant components will be given under Section 2.2. Only general information will be used, any sensitive data is therefore excluded.

Finally, the proposed project is presented as viewed by the company. This gives insight into what the company's stakeholders regard as important features and requirements in the system that is going to be developed. In Section 2.3, a brief description of this document is given. Primary challenges posed by these requirements are given in Section 2.4, motivating many of the selected solutions.

2.1 Ericsson Linköping

An office of a multinational corporation providing communication technologies and services, located in Mjärdevi Science Park. Offers products in the means of services, software and infrastructure in Information and Communications Technology (ICT) for telecommunications operators. According to a report [Eri15], around 40% of the worldwide mobile infrastructure runs on Ericsson equipment.

Clearly, the software being developed there is critical to the functioning of a modern society, since several organizations and communities today rely on mobile communications working correctly. Therefore, increasing the importance of the software running on the networking hardware to be stable and thoroughly tested, since small bugs can slowly cause the malfunctioning of the system, leading to higher maintenance costs and even downtime, which is not acceptable.

2.2 Development Overview

Since this thesis deals largely with *modifying* the existing *developer workflow* to support *coverage gathering and analysis*, an overview is shown in the Figure 2.1. Developers desiring to *commit* changes to *production code* need to pass through *validation* of both *peer code reviews* and the *full testing suite*. This is automated with *continuous integration*, which does *automatic testing* on all *target devices*, with the help of the *function/system tests* written with a *Java testing framework*. Lastly, the black-box function denoted by f below is the location where relevant changes have been made, which also execute the *build system & test framework*.

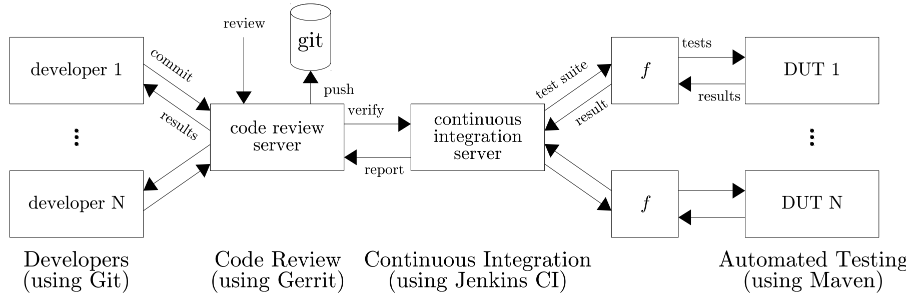


Figure 2.1: Development Process Overview

2.3 Project Description

Requested by one *project group* at the *research department* in *Ericsson Linköping*, the *research* and *construction* of a *test coverage gathering and analysis system* to be *integrated* into the existing *development workflow* shown in Section 2.2. The project uses *C/C++* for writing the *production software* and *unit tests* for the *target device*, while *function/system tests & framework* are written in *Java*. While *coverage gathering* was already in-place for the *unit tests*, the same wasn't true for *function tests*. Therefore, the *goal* is to *determine* if *coverage gathering* is *feasible* for *function tests* too, thereafter *integrating the system* into a Ericsson project, while also providing an *analysis method* for deriving useful properties. Below follow the milestones that were shown for the *project specification* [Jan].

1. Special *coverage instrumentations* for the *production code* are to be made, which will enable the *gathering* of *function test coverage* from the *target*.

2. Determine if these *instrumentations* pose any *serious performance threats* which could prove to make these changes *unfeasible* for *implementation*, especially while the *target device* executes their *full function testing suite*.

3. Setup *infrastructure* for *fetching coverage information* from *the device* to *the developer*. This should be done *automatically* to *integrate into the existing workflow*, enabling engineers to *easily fetch test case/suite coverage*.

4. *Research* and possibly *develop existing or new coverage analysis methods* for *relational properties* of *function test cases*, *similarity* and *uniqueness*, which will be used for *easing the analysis* of the *nature* of the *tests cases*.

2.4 Main Challenges

Several of the decisions made under Chapter 4 have been selected specifically because of the several constraints on both the *System Under Test (SUT)* and the *development ecosystem*. These have directly affected the *approach* taken when developing the solutions presented, and can therefore also be attributed to also affecting the *results* and *conclusions* presented later in Chapters 5 & 7.

- **Performance requirements:** while executing the *production code* with *coverage instruments*, *time* and *memory* should be minimally affected. More specifically, no test cases should start to fail because of the extra time taken. Also, the Device Under Test (DUT) has a very limited amount of memory (around 15 MiB leeway), so this resource should be minimized.
- **Daemonized processes:** most of the instrumented software running on the SUT/DUT are *daemons*, meaning they are *background processes* that don't terminate. While not problematic at first glance, existing coverage instrumentation tools don't handle this type of particular case very well.
- **Remote test target:** the production code is executing on a remote target, different from where the code was compiled, leading to the problem of retrieving the coverage data for analysis when gathering has finished.
- **Limited storage:** within the DUT, there is a small location for general purpose storage under */tmp/*, it is however shared by other functionality in the system, such as logging, meaning data shouldn't be stored for long.
- **Behaviour isolation:** since the SUT is a real product and the project is being developed by thousands of different engineers, additions to the source code would need to be very isolated, and protected by flags, so no disruption of work should occur if a feature in the coverage systems fails.
- **Complex environment:** finally, the development ecosystem at the Ericsson project is diverse and large, no universal existing solution was found. As previously shown in Section 2.2, *GCC* is used to compile the production code, *Java* to execute and define the function tests, a customized *Python* build system for automation, and a multitude of custom tooling.

While these are only some of the considerations taken into account, they are the primary reasons for the implementation having been shaped the way it has. Other reasons are to those pertaining the previous work, described in Chapter 3.

2.5 Specifications

Only essential hardware information will be divulged for confidentiality reasons, which should at least hopefully help put some of the later results in perspective. All results pertain to a *development device* running *GNU/Linux* on *kernel v3.10* compiled with *GCC 4.8.1* for an *ARMv7 12-core (logical) RISC* architecture. Insightfully, `/proc/cpuinfo` file gives a very rough processing power estimate of 512 *Bogo Million Instructions Per Second (MIPS)* for each computing core. Usable disk space in `/tmp` is around 2.7 GiB, memory leeway is 15 MiB here, which means that the thesis project should restrict itself with these resources, given that these are shared with the other development services such as logging.

Chapter 3

Theory

Several sub-sections within the thesis assume the reader has knowledge of certain non-standard *concepts*, *definitions* and *previous research findings* which are used henceforth to *explain*, *derive* and *motivate* many of the coming chapter contents. Most of these are briefly described in the coming pages, but assumptions about the readers knowledge of *fundamental computer science* & *GNU/Linux* is made. Any further details can be found in literature within the attached *Bibliography*.

3.1 Software Testing

Better described by the quote from *Myers et al* [MSB12], defining the process: *"testing is the process of executing a program with the intent of finding errors"*, which is usually done through a series of *test cases*, which are called *test suites*. Based on studies made by the National Institute of Standards and Technology (NIST) [Tas02, p. 112], the estimated cost of inadequate testing is $1.8 billion within the U.S. Aerospace and Automotive industries, where some potential $0.6 billion could have been saved with the suggested testing infrastructure. Relevant *terminology* and *concepts* follow, and are used throughout the thesis.

3.1.1 White-Box and Black-Box Testing

Strategies for testing come primarily in *two* forms, *white-box* and *black-box tests*, which are better described by *Myers et al* [MSB12, p. 8] as being two distinct ways for deriving test data. *White-box testing* examines the *program logic* with *contracts* (see *Bertrand Meyer* [Mey92]), while *black-box* completely *ignores* the *internal workings* of the program, and uses the *program behaviour specification*. Both *strategies* are used at the Ericsson project, but *black-box* will be the focus.

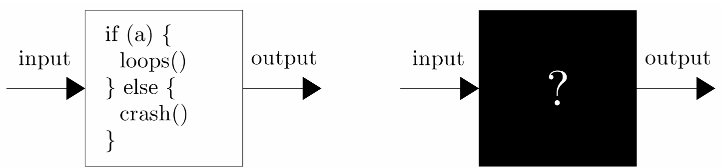


Figure 3.1: Intuitive Representation of the Testing Strategies

3.1.2 Unit, Integration and System Testing

Following descriptions derived from *Myers et al's* well known book in the area, *"The Art of Software Testing"* [MSB12, p. 85], *unit testing* is the known process of *testing small modules independently. Modules* are defined here as *independent* programming constructs, such as *functions*, *classes* and/or *namespaces.* Usually associated with *white-box testing* since *internal* knowledge of modules is needed.

Both *integration* and *system testing* are concerned with verifying that the *modules work together correctly* as per some *functional specification. Integration* attempts to combine *several independent modules* and test them for correctness, while *system testing* asserts that a *batch* of *integrated modules* function together. Both of the above are *black-box* processes, since *external specifications* are used.

Since *integration* and *system tests* are similar in nature, *function testing* commonly refers to both of these while *logic testing* is attributed to *unit tests*, described by *Myers et al* [MSB12, p. 119]. This project concerns itself with *function* and *unit tests*, written in *Java* and *C++*, where the former is the focus.

3.1.3 Software Verification and Validation

While seemingly meaning the same, *verification* and *validation* are completely different processes which should not be confused with each other. Following the description by *Hailpern et al* [HS02], *verification* is the process of checking if a program satisfies a set of specifications (program working as intended?), which implies checks for *functional correctness*, while *validation* is the process: evaluating software for *requirements* compliance (building the right product?). *Software testing* is largely concerned with only asserting that the *functional correctness* of a program is compliant, which is the focus in this thesis project.

3.1.4 Test Case Similarity & Uniqueness

Knowing the *behavioural similarity and uniqueness* between *test cases* have some applications, which will be mentioned further in Section 3.3.1 and Section 3.4. But, how does one *measure* these properties? Related articles have been found that deal in this area, *Hemmati et al* [HB10] or more recently *Noor et al* [NH15]. Commonly referred as *similarity functions*, usually come in two forms, *set-based* or *sequence-based*, but since the *set-based functions* have been shown by the *Hemmati et al* [HB10] to produce precise results, only these will be considered.

According to *Hemmati et al* [HB10], primary metrics for defining *set-based similarity* are the *Hamming distance* and *Jaccard indexing/coefficient functions. Hamming distance* is defined informally as the *"minimum amount of operations required to change one string into another"*, where strings could represent lines. More formally, it's the *minimum edge sum* between *two vertices* in a *hypercube. Jaccard indexing* is used to *compare* the *similarity* and *uniqueness* of *sets* by the application of Equation 3.1, where the *uniqueness/diversity* is $1 - J(A, B)$. Both *Hamming* and *Jaccard* will be implemented, but only the latter displayed. Following the same reasoning as *Cartaxo et al* [CNM07], the assumption that a *test case's behaviour* can be described with *coverage criteria* is also made here.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|},\ 0 \leq J(A, B) \leq 1 \tag{3.1}$$

3.2 Continuous Integration & Test Automation

Originally coined by *Grady Booch* in his book *Object Oriented Design* [Boo91], *continuous integration* is a *software practice* where work developed by a group of developers is *integrated continuously* in the mainline repository branch instead of developing modules independently, and then finally when complete, doing a massive monolithic integration, which is usually referred to as *"integration hell"*. Additionally, as described by *Fowler et al* [FF06] in their article on the subject, there are a couple of key practices for an effective continuous integration system, some points which include *automated builds* and *fast builds* for quick feedback. Several of these practices are used in the system at Ericsson, specifically on their *continuous integration servers* running *Jenkins*, which enable *automatic testing*.

Unfortunately, the *full development testing suite* takes around *two hours* to execute, which means that *fast build/test feedback* isn't available to developers, which becomes worse when triggering the *delivery testing suite*, taking *18 hours*. Clearly, reducing the time taken for testing is important, especially economically, since several developers are integrating frequently, triggering the test suites. Developers therefore have delayed feedback on the testing results, which leads to potential bugs staying longer in the system, which slows development down.

Resolving these long waiting time issues isn't trivial, since simply *removing test cases* would *reduce software reliability* and *keep bugs longer* in the system. Research sub-areas exist that try to deal with this problem, among them are the well known techniques of *test case selection* and *test case prioritization*, which will be mentioned further in Section 3.3.1 since they are coupled with this thesis. Another less formal method of achieving this is to remove *redundant test cases*. Both of these mentioned *methods* above are the *primary reasons* and *use cases* for the *gathering and analysis system* being built as part of this thesis, since, the *similarity* and *uniqueness* measures demonstrated in Section 3.1.4 can be used to infer the *potential test case redundancy*, as shown by *Cartaxo et al* [CNM07]. Similarly, *test case selection and prioritization* can be enabled by utilizing the *gathered code coverage*, which is described and motivated in the coming sections, whose tasks is to *reduce the test cases executed* and *sort for faster feedback loops*, which are as previously described, essential, when doing *continuous integration*.

3.2.1 Measuring Relative Test Suite Quality

Another *software practice* tightly linked with *continuous integration* and *testing* is *automatic verification* of *incoming commits* to detect any *testing regressions*, which can *deny* any potentially *harmful integration* from entering the mainline. However, does the described method find *all harmful additions* to the product? By what *amount of confidence* is the *software code* tested by the *testing suite*? Several *metrics* exist for determining the *quality* or *confidence* of a *testing suite*, where the widespread and actively used metric for this purpose is *code coverage*, which will be thoroughly discussed in Section 3.3, along with its additional uses. Only using this metric to determine the *quality* and *confidence* of a *test suite* has been the matter of some *controversy*, since it's constantly abused, as pointed out by *Marick et al* [$M^{+}99$], just as a general "target" that needs to be achieved. Regardless of these issues, it has been shown by *Williams et al* [WMMK01], and among others, that *code coverage* is correlated *positively* with *suite quality*, which implies that there is still value in measuring code coverage if done right.

3.3 Code Coverage & Criterion

According to *Piwowarski et al* [POC93], the topic of measuring *code coverage* using the different *code coverage criterias* has been an active area of *research* and *implementation* in the industry since the 1960s, particularly early at *IBM*. Formally, the *code coverage metric* is the *ratio* of *production code exercised* by a particular *testing suite*. It has been shown in previous research that this *metric* is an *effective estimator* for the *fault detection rate* of a *testing suite*, however, this has only been proven for exceptional test cases, where *Cai and Lyu* [CL05], have shown that *functional testing* has *higher correlation* than *random testing*. Nevertheless, it's one common consideration in safety critical systems, where 100% *statement, branch, modified condition/decision coverage* isn't an unusual requirement before delivery, such as the *aviation standard DO-178B* [Joh98]. Besides being used as one *reliability metric*, Section 3.3.1 explains further uses.

Measurements of the code coverage can be done in several different ways, each deriving from observations in the code structure, called *coverage criterias*, which produce different results since the interpretation of what is *covered* varies. Usual criteria are *functions, statements, branches & conditions covered*, shown by *Myers et al* [MSB12] to be *basic criteria*. Description of these will be given in Section 3.3.2, where only those available within *gcov* will be formally presented.

3.3.1 Different Uses of Code Coverage Data

Besides being used as one possible *quality and reliability metric* for *testing suites*, *code/test coverage data* has been used in a wealth of different research topics, while no *relevant literature review* has been found dealing with their *use cases*, the author has found the following recurring topics which utilize code coverage.

- **Finding coverage holes:** described in research by *Adler et al* [AFK+09a] at *International Business Machines (IBM) Research*, analyzing coverage becomes problematic since it's voluminous, the task of finding locations where tests don't execute production code, called *holes*, poses challenges. By using their *substring hole clustering method* to group similarly named test cases, an engineer can more easily identify locations of potential *holes*.

- **Test case clustering:** calculating *distances* between *test case coverages* allows the derivation of the *behavioral similarity*, and therefore *clustering* of *test cases*, which enabled *Pang et al* [PXN13] to reduce costs of testing by removing *non-effective test cases* while keeping the *effective test cases*. This required the calculation of the relative *Hamming distance* between the *test case coverage data*, where *unmatched criteria hits* imply distance.

- **Test case selection:** instead of executing all *test cases* when changes to the *production code* have been made, *selection* attempts to execute only *affected test cases* relating to that change, thereby *reducing testing time*. Shown by *Beszédes et al* [BGS+12] among others, that *code coverage* from *previous commits* can be processed to *derive* this *set* of *affected test cases*.

- **Test prioritization:** similarly, *Beszédes et al* [BGS+12] also used these *code coverage data* to *rank test cases* and *sort these* before execution to maximize *inclusiveness* while minimizing *selection size* for testing *WebKit*. Several other *prioritization strategies* exist, like *maximizing fail feedback*.

3.3.2 Statement, Function and Branch Criteria

Since there exists a multitude of different *coverage criteria*, only those relevant to this thesis will be explained, which are those available in *gcov* 5.3.0 [GNUb]. For a complete treatment of these, the reader is encouraged to find related work. Descriptions are derived from *Paul Johnson's Testing and Code Coverage* [Joh], additionally, where the *stronger path and condition coverage* types are explained.

- **Statement coverage:** regarded as the most basic type of code coverage, a *statement/line* is treated as *covered* if and only if it has been *executed*. Note that this type of coverage is *weak*, that is, even though high coverage can be obtained, there is no guarantee that it actually covers everything, behaviourally speaking. Achieving 100% coverage is especially hard with this criteria, since error handling is rarely taken into account when testing.
- **Branch coverage:** useful when attempting to ensure that the program has handled every possible *jump* towards every possible *destination*, which protects against errors where one faulty branch affects the final behaviour. Similarly, there exists *condition coverage* which attempts to handle if each *sub-expression* has been *evaluated*, which is somewhat trickier to achieve. 100% *branch coverage* implies 100% *statement coverage*, all *blocks* are run.
- **Function coverage:** similarly, tracks the amount of *functions* executed, which is useful since the level of *granularity* is much more *coarse*, enabling developers to easily *drill-down* to the *lower-level statement or branch* later.

3.3.3 Code Coverage Testing Tools Overview

Previously mentioned by *Adler et al* [ABR+11], there exists no *general tool* for *gathering and analyzing code coverage* since the *development ecosystems* of most *large-scale industrial systems* are complex and diverse. However, there exists a wealth of *test coverage tools* already available, some more suitable than others. *The survey* by *Yang et al* [YLW09] dealing in several *coverage-based testing tools* summarizes their properties, which can be used to decide on more suitable tools.

Since the *target programming language* is *C/C++* and the *compiler GCC*, there aren't many *free* (as in freedom) alternatives other than the *GNU's GCov*, which has also been *proven* to be *performant* when *gathering coverage* for the *Linux kernel* as described by *Larson et al* [LHRF03] to only have 3% *overhead*, regarding *execution time*, which implies that *GCov* is rightfully suitable for this.

3.3.4 Fallacies of Code Coverage as a Metric

Following conclusions derived from *code coverage measures* blindly has been the topic of some debate in the field, since it cannot really detect *faults by omission*: *"it can't identify how code that ought to have existed would have been exercised"*, as perfectly described by *Brian Marick's How to Misuse Code Coverage* [M+99]. Therefore, caution must be taken by managers, developers and testers alike so that the results by the tools presented here are not *misinterpreted* and *misused*. *"Coverage tools are only helpful if they're enhancing thought, not replacing it."* Additionally, it must be noted that *weak criteria* such as *statement* and *branch*, does *not guarantee* the same *functional behaviour* between *coverage samplings*. *Two test cases displaying* the *same coverage hit locations* might *not* be the same.

3.4 Coverage Analysis Methods

After collecting *coverage information* regarding the relationship of *test cases* and their *production code*, there is a desire to *present* these results in a *"meaningful"* way, such that relevant information about their *nature* can be *easily* retrieved. Several common methodologies exist, pointed out by *Adler et al* [AFK+09a] in their introduction, where the *drilling down technique*, searching through the data across different granularities for information, is by far the most common. Fallacies with using these primitive methods in *large-scale industrial software* is that the *volume* of the generated data could be become quite massive, essentially rendering the usage of these methods inefficient, shown by *Adler et al* [ABR+11]. *Coverage analysis methods* are techniques used to *narrow down* these properties.

3.4.1 Set Operations on Coverage Data

While not yet formally researched or analyzed, *applying set coverage operations* on *individual test case coverage data* has already been used in several existing commercial *coverage analysis tools* to provide extra information about testing. *SAP* has allowed their *tools* [SAP] to switch *merging strategy* when combining different *test case runs* if desired, two of these are *intersection* and *difference*. *Semantic Designs* have also integrated these into their commercial tool chains. Additionally, a presentation by *Nick Rutar* [Rut] on this exact subject has shown his *interpretation* on the *properties* that can be obtained from the *set operations*. Since this method is more *industry proven*, further *thesis research* will be done.

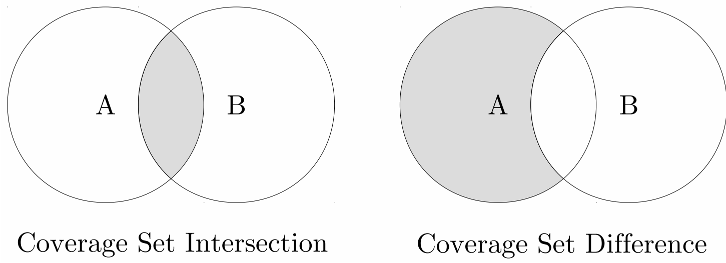


Figure 3.2: Graphical Visualisation of Coverage Set Operations

3.4.2 Substring Hole Analysis Method

Developed by *IBM Research* and then presented in the research/technical papers by *Adler et al* [AFK+09a, AFK+09b] back in 2009, the *Substring Hole Analysis* is a novel method which *aggregates test coverage data* across the *structure of the code* and then *clusters common substrings* among *elements* in the source code, such as *function* or *class names* together. Semantically, these are based under the *observation* that programmers tend to *name source code elements* related to each other *closely*. By listing the coverage results ranked by the number of *holes*, which are *related elements* which are *not covered*, analysis becomes easier. Within *Adler et al's* [AFK+09a] research, it was found that *57%–87%* of the *domain experts interviewed* thought the *30 top-ranked holes* were of *relevance*. Even though it shows promising results, it's unsuitable for the company's setup.

3.5 Measuring Performance

Determining the *relative performance cost* of a software feature is essential when reasoning about the stability and viability of the addition. Doing *performance analysis* in the field of experimental computer engineering is a combination of *measurement*, *interpretation* and *communication* of the *speed* and *resource* consumptions, as described in the book by *David Lilja* [Lil05] on the subject. There are three fundamental *techniques* for analyzing performance: *measuring*, *simulation* and *analytical modeling*. Since this paper deals with a real system, doing *measurements* will be chosen, where there are several *strategies* to do so, as described by *David Lilja* [Lil05]: *event-driven*, *tracing*, *sampling* and *indirect*. The *sampling* strategy is ideal for this system, since it produces the *statistical summary* of the *overall behaviour* of the system. Since these are *samples*, the statistical models need to be considered too, which are described in Section 3.6.

3.5.1 Program Running Time

Program execution time can be determined by using one of the most fundamental tools in performance analysis, an *interval timer*. One of the possible ways to do this is to use a *software timer*, measuring the start of the operation and the end time. By taking the difference between these two timers, one can ascertain the program running time. More specifically one calculates: $time = end - start$. Important to note, time even when the program is waiting, is taken into account.

3.5.2 Memory Usage of Program

Since memory is a resource that accumulates and changes over time, calculating the *central tendency of memory usage* is required to gain proper insight into the actual metric. Three basic indices are common in the field of performance analysis: *mean, median and mode*. Since the data is assumed to be normally distributed around its true value, the *arithmetic mean* seems appropriate to estimate the memory usage of a program under a certain interval of time, as reinforced by the literature of *David Lilja* [Lil05]. This technique is used by several tools that try to determine the *average memory consumption*, specifically, on Linux (the operating system where the *Device Under Test (DUT)* is being executed on), tools such as *top* use this method by sampling from /proc [GNUe] and drawing an average, finally presenting data as a percentage of memory used.

3.5.3 Processing Time Consumed

Similar to the situation with *memory usage* back in Section 3.5.2, the metric of *processing time* over an interval requires the data to be processed since it would otherwise only give information about a momentary sample observation. Techniques quite similar to those described above can be used for estimating *processing time*, in Linux, this sample data is usually fetched from the */proc* files. The performance measuring scripts for the DUT use these facilities provided by the files under */proc/** extensively, while calculating the arithmetic mean data.

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i, \quad s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{3.2}$$

3.6 Statistical Confidence

Measuring experimental data, such as the various performance metrics described in Section 3.5, introduce a certain *uncertainty* to their results. These are usually called *data error or data noise* as described by *David Lilja* [Lil05], and to truly present data in a format that takes into account this uncertainty, a *statistical analysis* of the data is required. While it's impossible to predict these random errors, a model can be established to describe this effect. Experimental errors can be assumed to be *normally distributed*, as pointed out by *David Lilja* [Lil05]. By using this, one can provide a *confidence interval* of the true value with a confidence of $1-\alpha$, where α is the significance level in percent. See *Norell* [JN99].

3.6.1 Student's T-Distribution

Sampling the true value of a variable x of an experiment n times, gives the mean sampled quantity $\bar{x}$ the best approximation in regards to error distance. Also, if the number of samples n is large, the central limit theorem assures that the sampled mean $\bar{x}$ is normally distributed around the true mean μ with a standard deviation of $\sigma/\sqrt{n}$ where σ can be approximated to a sampled s deviation. However, as the number of samples decrease, this isn't true anymore, as described by *Norell* [JN99]. By using the *student t-distribution* with the *probability density function* shown in Equation 3.3, one can estimate the true value of x with $n-1$ degrees of freedom as described by both *Norell and Lilja* [Lil05, JN99]. Since the samples collected in this thesis take long to process, only a few will be done. Therefore, the *student's t-distribution* is a good choice for this statistical model.

$$f(t) = \frac{\Gamma(\frac{n+1}{2})}{\sqrt{n\pi}\Gamma(\frac{n}{2})}\left(1 + \frac{t^2}{n}\right)^{-\frac{n+1}{2}}, \int_{-\infty}^{\infty} f(t)\, dt = 1 \tag{3.3}$$

3.6.2 Sampling of Population

While sampling experimental data, there are a few things that are essential to ensure the statistical model can be applied correctly. Shown by *David Lilja* [Lil05], there are two different sources of errors in measurements, *systematic errors* and *random errors*, see literature. Only *random errors* can be modeled statistically.

3.6.3 Confidence Interval

Since the error of the measurements can be statistically modeled as being part of a normal distribution, an approximation of the actual true value can be done with a statistical *confidence interval*, which defines a range of values where the true value can lie within, given a certain probability constraint. All proofs and explanations of the following equations are entirely left to literature such as the work by *Lilja and Norell* [Lil05, JN99]. In order to determine the confidence interval $[a, b]$ of values in the sampled mean $\bar{x}$ with a certain significance α, given that n samples have been gathered, the t-distribution area probability with $n-1$ degrees of freedom $t_{1-\alpha/2,n-1}$ needs to be provided, which can be derived from Equation 3.3. See Equation 3.4 for the lower/upper limit a and b.

$$a = \bar{x} - t_k\frac{s}{\sqrt{n}}, \quad b = \bar{x} + t_k\frac{s}{\sqrt{n}}, \quad t_k = t_{1-\alpha/2,n-1} \tag{3.4}$$

Chapter 4

Method

Attempting to solve the problems presented under Section 1.2, while also trying to answer the research questions in Section 1.4 and therefore fulfill the purpose of this thesis defined in Section 1.3, has required a very practical methodology. Building the *function test coverage framework* has been the thesis methodology core, since all problems and questions presented in the thesis either deal with it directly (e.g. *development experience* and *performance feasibility*), or depend on it as one prerequisite (e.g. *analysis interpretation* and *coverage measurements*). Therefore, much of the content contained herein is related to doing system implementation, which should also give researchers higher reproducibility rates.

After dealing with the implementation experience and its system details, research *data* should have been *gathered and processed* in order to answer the research questions previously posed, which should be summarized in Chapter 5 by using some techniques studied from the *previous work* presented in Chapter 3. By using these results, the *final thesis conclusions* in Chapter 7 can be derived, which can be thought of as treating the more theoretical sections of the thesis. Below follows a brief overview of the procedures that are going to be discussed.

Since the developed *function test coverage framework* is divided into several parts, that is, *modules*, an outline of these different parts and how they relate to each other is desirable. Under Section 4.1, the outline of this system is presented, with its major components. How the *production software* is *instrumented with code coverage operations* is briefly discussed in Section 4.2, and also how this is integrated into the *build system.* Given this feature, in Section 4.3, the relevant *performance effect* of running the *instrumented production code* on the target DUT is measured and statistically analyzed. As mentioned in Section 2.4, some problems arise from having both *daemons* and *remote targets* being in the set of instrumented code, these issues are explored and solved in Section 4.4, 4.5. Gathering the *coverage measurements* of the *full test suite*, require some tooling for displaying the results, which is a problem that is treated in Section 4.6. Finally, given previous research and suggestions, the *code coverage analysis tool* implementation is presented in Section 4.7, while also showing results about the amount of *reduced coverage data* to inspect with this coverage analysis method. Many of the limitations are based on the requirements of the existing system, which are summarized under the background Chapter 2, these will however, be more deeply described in this coming chapter, so more relevant parameters are given to the researcher for further analysis, in regards to these thesis's findings.

4.1 System Architecture

Understanding the relationship between these applied methods and how they work together as a whole, is essential in grasping their purpose in the system. Below follows the primary components that are to be presented in the coming sections in more detail, with a brief overview of how they function and connect. Mainly: *instrumentation*, *flushing*, *performance sampling*, *fetching* and *analysis*.

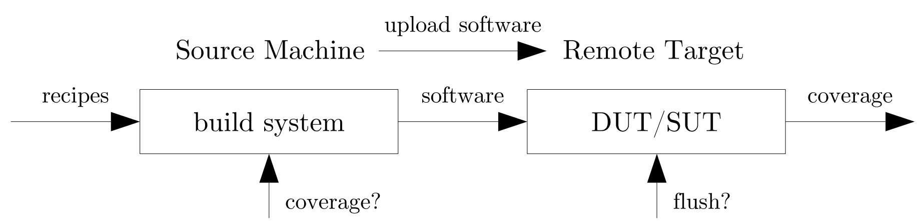


Figure 4.1: Software Instrumentation Overview

Before being able to *gather coverage* from the *target device*, *production code* must be built with special *coverage instrumentations* to enable *coverage dumps*. Section 4.2 deals with this in detail, but an overview follows in Figure 4.1 above. Modifying the *build system* to add *coverage compiler flags* when specifying a custom *build flag*, e.g. `--coverage`, builds an *instrumented software package*, which can be used to *upload and install* the software on the *remote target device*. After doing this, the *target software* can *dump coverage on termination*, which isn't a default behavior that works for this system given that all instrumented processes are *daemons*, that is, *background processes*. Manual *flushing* is needed.

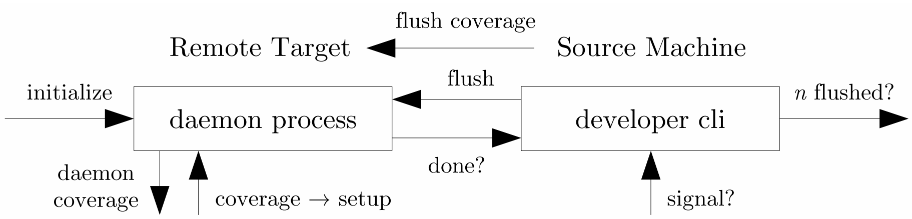


Figure 4.2: Daemon Flushing Overview

Solving these issues have required the methods presented in Section 4.4, which are outlined in Figure 4.2 above. Before being able to *manually signal* the *target daemons* to *flush coverage* down to disk, *custom user signal handlers* are required to be *setup* on *process initialization*, which is outlined in Listing 4.5. Upon doing this, the *developer's source machine* can send one *global flush* to *all* processes on the target DUT by issuing a *remote Command Line Interface (CLI) command*, thereby *flushing all coverage* from the relevant *instrumented process's memory* down to *disk* in one single monolithic step. Doing this has the advantage that at any point t, the system can be requested for its coverage which available within the range of time $[t', t]$ with one single *flush call*, where t' is the time point of the *previous* flush or the *start* of the system if no flushes have been made before, since *flushing* also *clears all available coverage counters*. Enabling the system to later *fetch coverage data* from a single *test case* or *suite*.

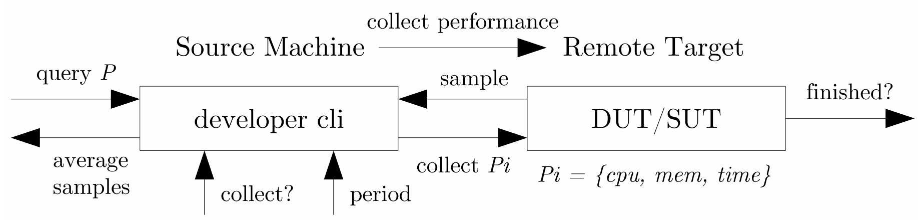


Figure 4.3: Performance Gathering Overview

Determining the *performance effects* of said *coverage instrumentations* on the *target device* running the *full testing suite* requires ways to collect the data. Described in detail under Section 4.3, but shown in Figure 4.3 for brevity, the *developer* has a *gathering script* executing locally that takes the average of every *performance sample* over time, probing the DUT at a certain fixed *frequency*. Both *local* and *remote* performance scripts are described in Listings 4.3 and 4.4.

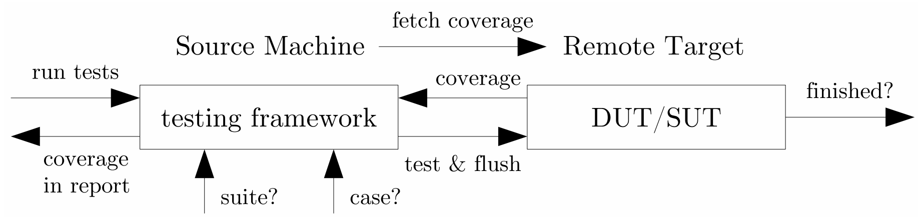


Figure 4.4: Coverage Fetching Overview

Since the chosen *analysis method* requires an *individual test case coverage*, *automatically flushing* under the period $[t'_i, t_i]$ is required for each test case i. Doing this is achieved by *modifying* the *flow* of the *testing framework* with a custom *specifiable parameter*, which enables flushing on *test case/suite end*, and then finally *downloading* the *remote coverage* to a *local report* location for later. These processes are a lot more intricate than described in the Figure 4.4 below, for a detailed description on how this actually works, see the later Section 4.5.

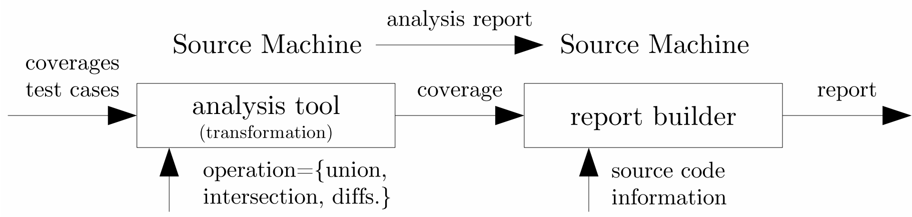


Figure 4.5: Test Case Analysis Overview

Lastly, now that the *remote test case coverage* is on the *local report directory*, these can be *processed offline* by the means of a *coverage analysis tool*, which has been designed to *merge* some *test coverage profiles* with a certain *strategy*. Engineers specify what *subset of profiles* are to be *merged* with what *strategy*, and an *output profile* is generated, which can be used to build *coverage reports*.

4.2 Program Instrumentation

Before being able to *gather* or *analyze* the *test coverage* of the *target software*, there needs to be a method to ascertain which *locations* of the *production code* have been executed. These data can then later be used to determine which parts of the *production code* have been run as a *side effect* of executing some *test cases*. Since the *target system* is compiled using the *GNU Compiler Collection (GCC)*, as mentioned in Section 1.5 and 3.3.3, the bundled *GCov* facilities are to be used. This was possible by adding additional flags, those presented in Section 4.2.1. Doing only this however, would pose problems, as mentioned in Section 2.4, the *software package* is *uploaded* to the *target DUT*, which won't work if the changes presented in Section 4.2.2 aren't applied, as reinforced in *Hofrat's* [Hof11] article.

Modifications were made to the proprietary *build system* written in *Python* to accommodate these features, while considering the desired *behaviour isolation* shown in Section 2.4. The solution was to add an extra BB flag: `--coverage`, which enables these instruments. Several solutions such as, adding a new custom target were explored, but dismissed, since they are tightly coupled in the system.

Listing 4.1: The Factorial Function

```
int fac(int n) {
    if (n == 0) {
        return 1;
    } else if (n < 0) {
        return -1;
    }

    return fac(n - 1) * n;
}
```

Listing 4.2: Instrumented Factorial

```
static int bb[4];
int fac(int n) { ++bb[0];
    if (n == 0) { ++bb[1];
        return 1;
    } else if (n < 0) { ++bb[2]
        return -1;
    } ++bb[3];
    return fac(n - 1) * n;
}
```

These examples above illustrate how code coverage tools usually instrument programs with coverage gathering capabilities. Based on the descriptions given by *Larson et al* [LHRF03] when adding coverage gathering on the *Linux kernel*, extra increment instructions are added on each *basic block*, shown in Listing 4.2.

4.2.1 Compiler Coverage Settings

Described in the *GCov* [GNUb] documentation, both `fprofile-arcs` and `ftest-coverage` are needed to correctly enable the *gcc* code coverage feature. Additionally, the program needs to be compiled *without optimizations*, or: `-O0`. Adding these incrementally on the compiler's `CFLAGS` and `LDFLAGS` instrument all compiled source code: `CFLAGS+=-fprofile-arcs -ftest-coverage`.

4.2.2 Coverage Environment Setup

Semantically, *gcov* writes coverage data to disk on the same location where it was compiled. Which poses a problem, since the software is moved to the DUT for execution, *gcov* will therefore fail since the directory doesn't actually exist. Following the *GCC* [GNUd] manuals, relocation of the standard *dump path* is done by setting the `GCOV_PREFIX` and `GCOV_PREFIX_STRIP` environment variables on the *target DUT*, which cannot be done directly, so custom *compiler variables* were exported with a valid path on target and loaded, see Section 4.4.

4.3 Sampling Performance Data

Since these recently discussed program instrumentations shown in Section 4.2 inadvertently affect the performance of the software SUT, there is a desire to quantify the *resource cost* of doing *code coverage on target.* Shown in Section 1.5, only some of the relevant properties will be measured in the interest of time: *test suite execution time*, *memory consumption*, and *processing usage* on *target.* Initially, *raw data* is gathered as shown in Section 4.3.1, which is then *processed statistically* in Section 4.3.2 to produce the results presented later in Section 5.2.

4.3.1 Gathering Raw Performance Data

Desired measurements are the *performance effects* of the *instrumented software* running on the *target device under test* while executing the *full testing suite*, where several subsequent executions are necessary for additional *data samples.* Using a script quite similar to Listing 4.3, these aforementioned preconditions could be set up and the samples gathered. Each *performance sample* is gathered with an *internal script* running on the DUT, briefly outlined under Listing 4.4. Since the target test suite in question takes about *2 hours & 10 minutes* to run, the automation enabled by these scripts was necessary, however, only *4 samples* were taken on this system in the interest of time, see Section 6.3 for the causes.

Listing 4.3: Sample Gathering Script

```
SOFTWARE=$1 ; TARGET=$2 ; SUITE=$3
SAMPLES=$4 ; SAMPLING_RATE=$5
ebuild --coverage $SOFTWARE
eupload $SOFTWARE $TARGET
for i in $(seq 1 $SAMPLES) ; do
    etest $SUITE $TARGET &
    eperf $SAMPLING_RATE $TARGET
    # continues when etest done.
done
```

Listing 4.4: Performance Data Script

```
# eperf downloads these files
cat /proc/uptime > timesample
# contains 'uptime" and 'idle'
cat /proc/meminfo > memsample
# has 'mem free', 'mem total'
cat /proc/stat > statsample
# information on 'cpu user,
# nice, system, idle' field
# also, on the 'cpus' count
```

Some commands contained within Listings 4.3 and 4.4 don't actually exist, but have a real equivalent tool in the target system. `ebuild` constructs software packages with or without instrumentations depending on the `--coverage` flag. `eupload` takes the software package and remotely uploads it to the target DUT. `etest` starts the execution of the desired testing suite. Finally, `eperf` remotely runs the performance script on target, retrieving an instant of the *memory usage* and *processor usage* on target, then calculating the average of these over time. Script measuring performance on target uses the techniques seen in Section 3.5. Lastly, raw *suite execution time*, is simply calculated by using an interval timer.

4.3.2 Statistical Performance Analysis

After *raw performance data* has been gathered, methods presented in Section 3.6 are used to determine more *statistically* precisely the *performance implications.* Both the *suite execution time* and *average memory & processor usage* variables are given a statistical confidence interval $[a, b]$ with $\alpha = 5\%$ and $n = 4$ samples. Data handling and visual graphs are produced by the *R* programming language built-in functionality, more specifically the *qt* (t-distribution quantile function).

4.4 Flushing Daemon Coverage

Assuming the *coverage instrumented processes* in Section 4.2 have a normal termination flow, the changes presented earlier regarding the *compiler flags* and *environment setup* are all that is needed to initiate *gathering coverage* on *target*. Since, upon program termination, all gathered coverage will be dumped to disk. Unfortunately, the target Device Under Test (DUT) has only instrumented code coverage data on *software daemons*, *background processes* that *never terminate*.

Solving this critical problem would require one to either *force* the *termination* of the program or to use *built-in low-level functionality* within *GCov* to *flush* it. Since the target device reboots upon termination, the second option was used. Details on how to do this is detailed in Section 4.4.1 below, with great help from *Hofrat's* [Hof11] paper, who came upon the same set of issues. Section 4.2.2 dealt with specifying a valid directory on the DUT by specifying compiler variables, these can finally be loaded in the real production environment for *gcov* to use, shown in Section 4.4.2. Synchronizing *flushing* across all *daemons*, Section 4.4.3.

4.4.1 Forcing Coverage Data Dumps

Not officially documented, there are two functions within `libgcov.h` that can be used to request an early dump of coverage data, flushing before termination. Described in *Hofrat's* [Hof11] article, one can use `__gcov_flush` to first write coverage data down to disk and then reset the in-memory counters (if desired). Doing this implies that linking towards *libgcov* is required, adding to `LDFLAGS`.

4.4.2 Storing Data on Remote Target

Since *gcov* expects both `GCOV_PREFIX` and `GCOV_PREFIX_STRIP` to be loaded into the systems *environment variables before flushing*, the *compiler variables* specified earlier with the correct dump path need to somehow write to these environment variables. By using POSIX `setenv` [GNUf] this can be achieved.

4.4.3 Custom Daemon User Signal

Simultaneously dumping the gathered coverage counters of the daemons requires a uniform way to message all of the processes. Following yet again the wisdom acquired from *Hofrat's* [Hof11] paper, a *user signal handler* can be setup for this. See Listing 4.5 for `coverage_setup` which is a function similar in nature to the one written for *each daemon* within the *production code* of the target DUT. After these have executed, all daemons should be ready for gathering coverage.

Listing 4.5: Coverage Setup Procedure for Daemon

```
extern void __gcov_flush();
void flush(int sig) { __gcov_flush(); }
void coverage_setup() {
#ifdef COVERAGE_FLAG // Exported 1 when --coverage flag in ebuild set.
    if (signal(SIGUSR1, flush) != SIG_ERR) { // or just use sigaction.
        setenv("GCOV_PREFIX", COVERAGE_PREFIX);
        setenv("GCOV_PREFIX_STRIP", COVERAGE_STRIP);
    } else abort(); // If signal setup failed, abort
#endif }
```

4.5 Automatic Data Gathering

While the purpose of implementing *automatic coverage data fetching* might not seem necessary, since the *gathering* of *coverage data* on the DUT/SUT is now possible with the introduction of *coverage instrumentations* and *flushing signals*, there are two primary reasons why this is necessary, *remote targets* and *analysis*. Discussed in both Sections 3.4.1 and 4.7, the *chosen analysis method requires* coverage data be *separated* for each *individual test case*, needing an early flush. Additionally, in order to *build coverage reports*, data needs to be present *locally*.

Since these changes require internal information about *test cases and suites*, modifications are to be made in the *testing framework* that fuels *test execution*. These modifications, or the ideas behind them, are given in Section 4.5.1 below. Before transferring coverage data to a *developer machine* from the *remote target*, data needs to be structured such that coverage can be indexed for a test case, which is dealt with in Section 4.5.2. Fetching is covered below in Section 4.5.3.

4.5.1 Changes to the Testing Flow

Making these additions to the *proprietary testing framework* written in *Java*, requires internal knowledge of how test cases and suites are executed internally. Briefly, in the interest of time, there exist two classes, each having two functions called `onStart` and `onFinish` which trigger whenever *test cases* or *test suites* have *started* or *finished* executing, as shown in Figure 4.6. Used in other features such as *fetching* test logs from the target, event triggers can be added. Complying with the *behaviour isolation* requirement, a custom test parameter: `fetch.coverage = { testcase | testsuite | false }` was added. Modifications to the existing testing flow can be divided into three components:

- **Test case finish:** first, *flush* all coverage data, writing everything to a *temporary* directory. Organize this data as per Section 4.5.2, upload it to the Testing Server (TS) since it will be fetched as shown in Section 4.5.3.
- **Test suite start:** reset all coverage counters since daemons start on boot.
- **Test suite finish:** same as the test cases, no structuring needed though, since the entire monolithic suite coverage is desired, using less disk space.

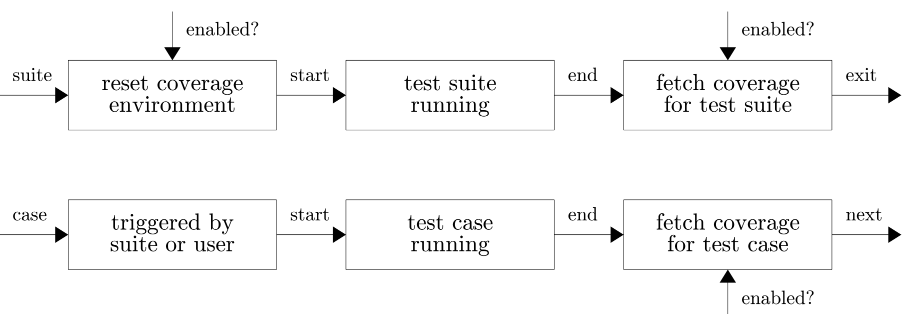


Figure 4.6: Overview over the Testing Flow

4.5.2 Separating Test Case Coverage

Another issue which might not have been obvious or relevant back in Section 4.2, is that the given coverage dumping directory is fixed (e.g. */tmp/coverage*), and subsequent executions will therefore *overwrite* the previous coverage counters. Hence why, Section 4.5.1 also deals in re-structuring and more importantly, *moving around* the coverage data after each *test case* or *test suite*. Additionally, in order to identify which test cases produced what pieces of coverage data, the *test case method name* is required. Fortunately, this data is available within the testing framework, and could be retrieved, such as: `qos.Ethernet.testIpv6`. Whenever the *flushing signal* is used, calling `kill -s SIGUSR1 $(pidof)`, coverage is written to `GCOV_PREFIX`, which is `/tmp/coverage/` on the DUT, which is then moved to `/tmp/upcoverage/` for uploading to the TS, below. Therefore: `mv /tmp/coverage/ /tmp/upcoverage/$TESTCASE`, is used for moving some particular `$TESTCASE` *temporary coverage* to the *upload path*.

4.5.3 Fetching Remote Coverage Data

After the test framework extension has successfully *flushed*, *re-structured/moved*, the *coverage data* for a particular *test case*, the last step of *downloading* it from the *remote target* to the *development machine* is required. When should this be done however? Several possibilities are available, but since *disk space* is *scarce* and also *shared* on the target as mentioned in Section 2.4, this is done upon every *test case ending*, because the data can then be *removed* from this target. Downloading the coverage data is divided in three major steps, across devices:

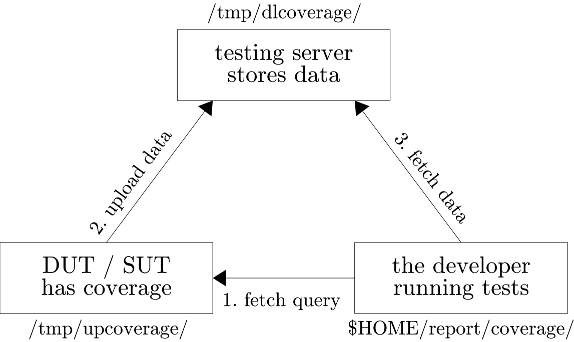


Figure 4.7: The Coverage Fetching Operations

Figure 4.7 displays some interesting and potentially unnecessary operations. Motivation why the data is *uploaded* to a proxy and only then *downloaded* is a restriction of the target, which would otherwise make an unnecessary procedure. Regardless, coverage data is uploaded from the *remote target's* directory to the *testing server's* temporary storage path, upon the *testing frameworks* request. Finally, the entire coverage data is downloaded to the *developer's test report directory* from the *testing server* by using SSH File Transfer Protocol (SFTP). All of these steps are processed automatically when requested by the commands from the *testing framework extension* upon either a *test case or test suite end.* Now that the essential steps so far of *instrumenting*, *flushing* and *fetching* the *build system*, *production code* and *testing framework* have been completed, the *test case separated coverage data* is available within the *developer device, locally.*

4.6 Building Code Coverage Reports

Upon *gathering* the *test coverage data* from the *remote target* and *retrieving* it to the *local developer machine* by using the method described in Section 4.5, there is a desire to *process* the data which is in *binary* form, to a more *graphical* form. Doing this was primarily motivated with two reasons, first, answering and displaying the *coverage measurement* results was deemed a practical solution, and second, the *test case analysis method* presented later in Section 4.7 still requires, as will be seen, an engineer to verify the *analysis results* in order to make *informed decisions* about the *conclusions*, which is better done graphically.

Following the same reasons presented in Sections 3.3.3 and 4.2, the *LGcov* graphical front-end to *GCov* was chosen as the primary tool to build these graphical *test coverage reports*. Integration would be more practical than the other available tools, as shown back in Section 3.3.3, and more *proven* since it has been used previously to gather test coverage data about the *Linux Kernel* by the *Linux Test Project (LTP)*[1], which can be considered a large-scale system.

Since the *Gcov generated files* are relevant for this section and Section 4.7, a brief overview of their format is given in Section 4.6.1 below. These *note* files are produced when the software is *compiled* with *coverage flags*, and the *data* files are produced when the *executing software* is queried to *flush* coverage data. Building a coverage report requires *both* these files to be in the *same directory*, which contributes another reason why the fetching at Section 4.5 was needed. After these have been placed in the same directory, as per the *LCov* [Obec] documentation, producing a HTML coverage report of the profile is as follows: `lcov -c -d $PROJ -o proj.info && genhtml -o $OUT proj.info` where Listing 4.6 shows what the actual report creation script looks like, which just finds all *data files* for a given *test coverage profile* and places them together with the *note files* in the project directory, where it creates the report as above.

Listing 4.6: Automatic Coverage Report Creation Script

```
PROFILE=$1 ; DIRECTORY=$2
find $PROFILE -name "*.gcda" | sed \
"s,^[^/]*/,$DIRECTORY/," > report.temp
cp -r $PROFILE/* $DIRECTORY/
lcov -c -t $PROFILE -d $DIRECTORY -o report.info
genhtml -t $PROFILE --legend -o report report.info
xargs rm < report.temp ; rm report.info ; rm report.temp
```

4.6.1 Coverage File Information

At the time of this writing, there are two file types *gcov* produces: *gcno* and *gcda*. *Note files* are produced under *compilation* and contain *structural information*. *Data files* are binary and contain the *arc transitions* generated on *program run*. Structure of these formats can be found in GCC [GNUa] source code *gcov-io.h*, which describes the binary *hierarchical structure* of both *note* and *data* files. Also found in the source is the term *profile*, which is a set of data from a *flush*. Since Section 4.7 deals extensively with the internal workings of these files and how *gcov merges & manipulates* them, this information and terms are necessary.

[1] Linux test project primary user homepage: https://github.com/linux-test-project/ltp

4.7 Similarity-Based Coverage Analysis

After retrieving the *gathered coverage data* from the *test suite executing target*, and thereafter *automatically splitting* the data into *individual test case coverage*, there exists roughly 812 *MiB* of *coverage data* to *inspect* across 100 *test cases*. Each of these test cases produced, roughly, 100 000 *coverage instrumented lines*. Following the given propositions by *Adler et al* [ABR+11], manually retrieving the *relevant properties* regarding these *test cases* isn't realistically *feasible*, since the amount of data is massive, leading to the *"needle in a haystack"* problem. Since the target *relevant properties* are *test case similarity* and *dissimilarity*, as mentioned in Section 1.5, the assertions made in Section 3.1.4 regarding *metrics* for these can be applied. Because *test coverage* displays the *test case's behaviour*, it can therefore be used to *analyze relative behavioural similarity* between *tests*.

Right below in Section 4.7.1 follows the general description of this method, explaining how these *gathered test case coverage data* can be manipulated with *set operations*, which can be used to calculate the *test case's similarity measures*. Relevant theory on *Hamming distance* & *Jaccard index* is given in Section 3.1.4. Several different attempts were made to implement these functions in practice, where Section 4.7.2 explains a failed attempt at manipulating *raw coverage data*, while, Section 4.7.3 describes methods manipulating *intermediate coverage data*. Finally, Section 4.7.4 explains how the *data inspection reduction* was measured.

4.7.1 Coverage Set Operations & Similarity

Before being able to even *implement* and thereafter *measure test case distance* and *coverage similarity/uniqueness*, proper *formal definitions* need to be given: assume *coverage criteria* have two states, either it's *hit/taken* or *not hit/taken*. The *distance* of *coverage profiles* is *defined* as the *number of unmatched criterias*, which can be represented by $d(A, B)$, *increasing linearly* for each *criteria* $a_i \neq b_i$. Similarly, *operations on sets* containing *test coverage "elements"* are *defined* here such that the $|A \cap B| = |\{a_i = b_i = hit\}|$ and $|A \cup B| = |\{a_i = hit \textbf{ or } b_i = hit\}|$. Trivially, given previous definitions, $J(A, B)$ and $1 - J(A, B)$ can be calculated.

Observe Figure 4.8 below for a more applied visualization of the definitions. Assume A and B are *two different test profiles*, which have both *executed* the shown *example scenario*, albeit in different locations since $A \to a$ while $B \to b$, leading to different *locations for statements hit* as being show in first columns. Since only three executed statements differ, the *Hamming distance* $d(A, B) = 3$, while $|A \cap B| = 2$ and $|A \cup B| = 5$, leading to *Jaccard coefficent* $J(A, B) = 0.4$.

Example Scenario	**A** hit	**B** hit	d	∩	∪
if a **is true then**	yes	yes	0	2	2
basic-block-1()	yes	no	1	0	1
elif b **is true then**	no	yes	1	0	1
basic-block-2()	no	yes	1	0	1
basic-block-3()	yes	yes	0	2	2

Figure 4.8: Theoretical Scenario Similarity Measure

4.7.2 Raw Profile Coverage Data Manipulation

After having formally defined the desired *operations* & *measures* in Section 4.7.1, there is a need to implement these features in practice, using real-world data. Initially, attempts were made by modifying *raw coverage data (binary format)*, since an existing tool was already bundled, providing a *merge/union* operation. Structure of these *raw coverage data* were discussed previously in Section 4.6.1. Since adding an *intersection operation* seemed quite trivial, a patch was written, fully presented in Appendix B, where Algorithm 4.1 was applied for all the *arcs*. By using `gcov-tool merge -i <A> <B> -o <OUT>`, the *coverage profiles* `A` and `B`, produced from *different test executions*, are *intersected*, towards `OUT`.

Algorithm 4.1 Intersecting Arc Transitions

Require: profiles A and B each having $bb[i]$
1: $bb_B[i] \leftarrow$ transition counts for B's i^{th} arc
2: $bb_A[i] \leftarrow$ transition counts for A's i^{th} arc
3: $(A_{hit}\,, B_{hit}) \leftarrow (bb_A[i] > 0\,, bb_B[i] > 0)$
4: **if** $A_{hit} \wedge B_{hit}$ **then** $bb_A[i]$ += $bb_B[i]$
5: **else** $bb_A[i] = 0$ {Result stored in A}

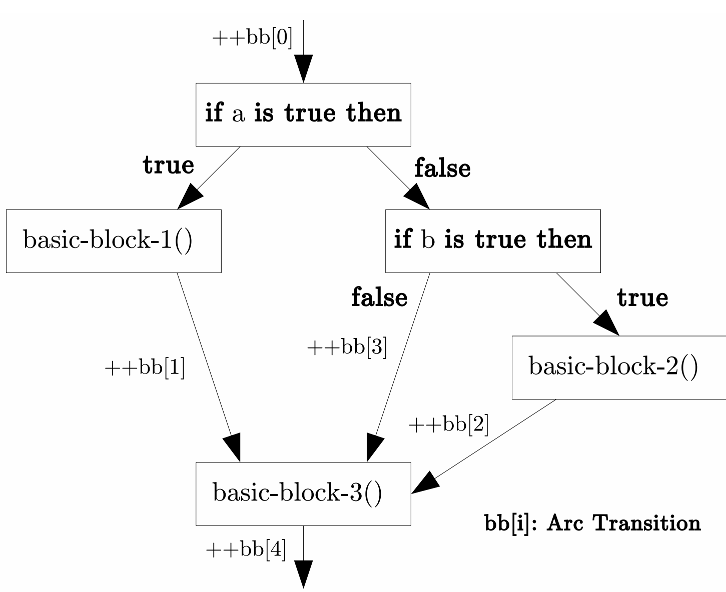


Figure 4.9: Conceptual Visualization of the Arc Transitions

While seemingly innocuous and correct, doing these operations affect data, *negatively*, since the *re-construction* of the *control flow graph* which is required by *gcov* to correctly derive the *statement/branch/function locations* from the *arc transitions*, will not succeed because *information is lost* when doing *intersection*. Observe Figure 4.9 above; notice that if Algorithm 4.1 is applied to these *arcs*, e.g. the scenario presented in Figure 4.8, there is no correct way to *rebuild flow*. Because of this, tools dependent on *gcov*, e.g. *lcov* will not report *valid* results, even though the *transformations themselves*, and *data*, are *conceptually correct*.

4.7.3 Transforming Intermediate Coverage Data

Upon realising these implementation issues presented in Section 4.7.2, another, alternative solution, manipulating *intermediate coverage* was devised instead. Since these data are produced *after re-building the control flow graph*, with the command `gcov -ibc <raw-coverage>`, it doesn't suffer from these issues. Instead of operating directly on *arc transitions*, information shown in Listing 4.7, derived from *The GCov Manual* [GNUb], could also equally represent coverage. Applying the *command* to every *profile's raw coverage*, produces an *intermediate representation* with the individual *statements, branches and functions* executed. Example: `(hsl_ifmgr.gcno, hsl_ifmgr.gcda)` $\to_{\$}$ `hsl_ifmgr.gcov`.

Listing 4.7: Intermediate GCov Coverage File Format Definition

```
file:<name> # Identifies what source file produced these sets of data.
function:<line>,<count>,<name> # Determines part functional coverages.
lcount:<line>,<count> # Contributes towards final statement coverages.
branch:<line>,<type> # Allows 'notexec', 'taken', 'nottaken' branches.
# Above fields are produced for each one of instrumented source files.
```

Building of the *Python tool/script* called `scovat.py`, realised the functions previously drafted within Section 4.7.1: *set operations, and similarity functions.* Roughly speaking, *three modes* of operation: *pre-generation*, *sets*, and *analysis* have been defined. *Pre-generation* is triggered on the `-gb` flag, which converts the *raw profile data* to *intermediate profile coverage data*, for later manipulation. *Set operations* are provided with `(-i | -u | -d)` flags, accepting several *pre-generated intermediate profiles* which are *combined* using the *specified operation.* Finally, the *analysis*, which is divided into *reporting* and *comparing*, is triggered with the `-a` flag, producing the *profile's Hamming distance* and *Jaccard index.* Further technical details regarding the implementation, besides those already in Algorithm 4.1 and Section 4.7.1, the reader is encouraged to see the `scovat.py`, located under the: `https://github.com/caffeineviking/scovat` tree.

Listing 4.8: Session Workflow for the Set Coverage Analysis Script

```
scovat.py -gb <build> -o <cache> <profile>... # Pre-generates a cache.
scovat.py -io <intersection> <cache>/<a> <cache>/<b> # A Intersection.
scovat.py -do <right-difference> <cache>/<b> <cache>/<a> # Difference.
scovat.py -do <left-difference> <cache>/<a> <cache>/<b> # Difference'.
scovat.py -ao <similarity-report> <cache>/<a> <cache>/<b> #Similarity.
scovat.py -ao <report-intersection> <intersection> #Intersection data.
scovat.py -ao <report-a> <cache>/<a> #Report of these coverage ratios.
```

4.7.4 Measuring Manual Inspection Reduction

Following the reasoning of *Adler et al's* [ABR+11] research, *analysis* is *unfeasible* since the amount of *coverage data* produced from *large-scale industrial systems* is *massive*, leading to the *"needle in a haystack"* problem. Shown in this thesis, an *analysis tool* which *measures similarity* between *test cases*, and their *location.* *Reductions* in *data inspection* for finding these *similarity properties* is *measured* by the *Jaccard indices*, interpreted here as the *manual inspection ratio*: $\frac{|A \cap B|}{|A \cup B|}$. Actual results from *three different test cases* will be presented for demonstration.

Chapter 5

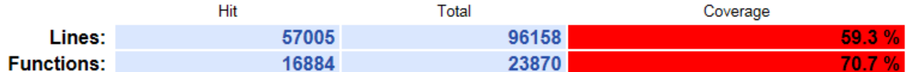
Results

Application of the methods presented in Chapter 4 have brought some results, which are presented in the coming sections, these are used to answer the research questions posed in Section 1.4, so that conclusions could be drawn in Chapter 7.

Displaying of the *test suite code coverage measurements* which were produced by *building coverage reports* back in Section 4.6, is done in the Section 5.1 below. After gathering and processing the *performance effects* of doing *coverage on target* as shown in Section 4.3, the graphs and results in Section 5.2 are shown. Finally, results regarding the *data reduction degree* of the *coverage analysis tool* previously built and described in Section 4.7 are given in the coming Section 5.3. Only plain *results* will be presented, *discussion* and *conclusions* are given later.

5.1 Full Test Suite Coverage

Below follows the *lcov report summary* produced from the *full test suite coverage*; a larger and more complete version of this excerpt can be found in Appendix A. Unfortunately, as can be seen in Figure 5.1, branch coverage couldn't be used, even after applying flags described by *Oberparleiter* [Obec]. *gcov* is used instead.



	Hit	Total	Coverage
Lines:	57005	96158	59.3 %
Functions:	16884	23870	70.7 %

Figure 5.1: Excerpt from Coverage Report on Full Testing Suite

5.1.1 Coverage by Criteria

Originally referred to as *line coverage* in the coverage report shown above, the *full test suite statement coverage* of the total 96158 *instrumented statements* has a degree of 59.3 %, which is exactly 57005 *production code* statements executed. Which *only* are from *instrumented* parts in the *program*, not directly from code.

Out of the total 23870 *instrumented production code functions*, exactly only 16884 of these were executed as a side effect of running the *full testing suite*, giving 70.7 % *function coverage* degree (also relative to methods *instrumented*). Finally, the *branch coverage degree* was reported to be 24.6% for this test suite.

5.2 Instrumented Performance

After *instrumenting* the *production code* with coverage gathering capabilities, outlined in Section 4.2, the *performance effects* relative to a normal build are derived. By using the methods in Section 4.3, the performance data is *gathered* and *analyzed statistically*. Within these next sections, the results are presented. Any available *hardware specifications* or *limitations* were shown in Section 2.4, which should ease the burden of replicating the results or reasoning about them.

5.2.1 Suite Execution Time

Raw *test suite execution time* was gathered by using the described *interval timer*, where the test suite's *start* and *finish* were collected. Both the *instrumented* and *non-instrumented* datasets are shown in the Tables 5.1, 5.2 below, where the *duration* in *hours* has been taken from each dataset *sample* for further analysis.

Table 5.1: Instrumented Total Test Suite Execution Time Data

Sample	**Start Time (24h)**	**Finish Time (24h)**	**Duration (h)**
1	17:53:00 GMT+2	20:40:00 GMT+2	2.783
2	20:53:24 GMT+2	23:42:26 GMT+2	2.817
3	23:55:14 GMT+2	02:49:02 GMT+2	2.896
4	03:02:48 GMT+2	05:41:06 GMT+2	2.638

Table 5.2: Non-Instrumented Total Test Suite Execution Time Data

Sample	**Start Time (24h)**	**Finish Time (24h)**	**Duration (h)**
1	05:52:38 GMT+2	08:09:30 GMT+2	2.281
2	08:15:28 GMT+2	10:23:43 GMT+2	2.137
3	10:29:35 GMT+2	12:38:46 GMT+2	2.153
4	12:44:14 GMT+2	14:50:12 GMT+2	2.099

After using the *statistical methods* described in Sections 3.6, 4.3.2, the results shown as the *test suite duration confidence interval* displayed in Table 5.3 for the *sampled test suite execution time* are produced for each individual dataset. Both of these results have a significance level of $\alpha = 5\%$ and 3 *degrees of freedom*. Therefore, *instrumented test suite duration* is between $[2.656, 2.911]$ *hours*, while *non-instrumented* is somewhere in-between $[2.075, 2.260]$, with 95% confidence. However, these results are only valid if the samples don't deviate from the model statistical distribution, which might be possible, as shown in Sections 3.6.2, 6.2.

Table 5.3: Confidence Intervals of Test Suite Time Durations

Sampled Dataset	**Lower Time (h)**	**Upper Time (h)**
Instrumented	2.656	2.911
Non-Instrumented	2.075	2.260

5.2.2 Average Memory Usage

Before presenting the *statistical confidence interval* of the *average memory usage* while running the *full testing suite*, *raw average memory usage data over time* is presented in the graphs shown within Figures 5.2 and 5.3. Each colored line represents an *individual sample over time*, where the horizontal line is the *mean*. Both of these were produced by applying methods in Section 4.3.1 and using *R*.

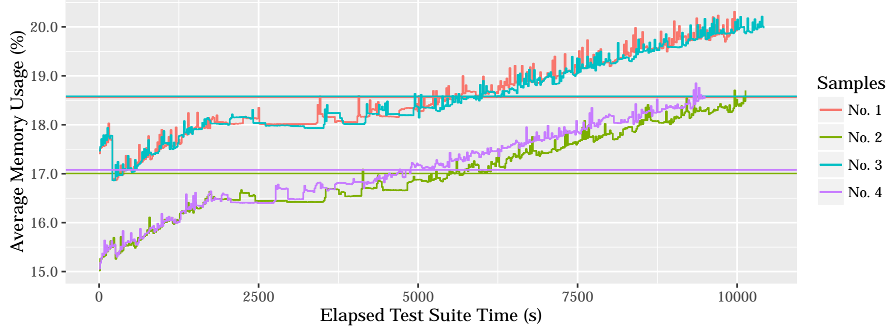


Figure 5.2: Instrumented Average Memory Usage over Time

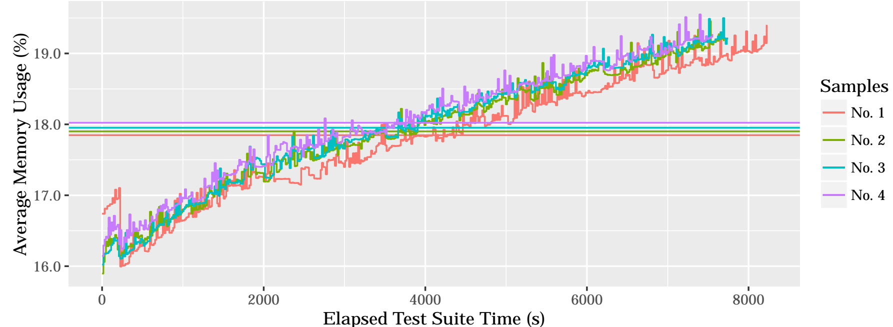


Figure 5.3: Non-Instrumented Average Memory Usage over Time

Application of the aforementioned statistical methods produced the Table 5.4 shown below. Both *instrumented* and *non-instrumented average memory usage* were modeled with a *t-distribution* by calculating the *confidence intervals* of the *sampled means* collected earlier. Average memory usage for these *instrumented builds* executing the *full test suite* seems to lie within $[16.7663, 18.8446]$ %, while an executing *non-instrumented software build* has given: $[17.8428, 18.0185]$ %, both being shown with a significance level of $\alpha = 5\%$ and 3 *degrees of freedom*.

Table 5.4: Confidence Intervals for Average Memory Utilization

Sampled Dataset	**Lower Usage (%)**	**Upper Usage (%)**
Instrumented	16.7663	18.8446
Non-Instrumented	17.8428	18.0185

5.2.3 Average Processor Usage

Similarly to the memory usage results in the previous Section 5.2.2, the data on *raw average processor usage over time* while executing the *full testing suite* has been plotted in the Figures 5.4 and 5.5. While the data is very noisy, as is usual when measuring processor usage, the *mean average processor usage* has been plotted horizontally across all samples, which should make it easier to analyze.

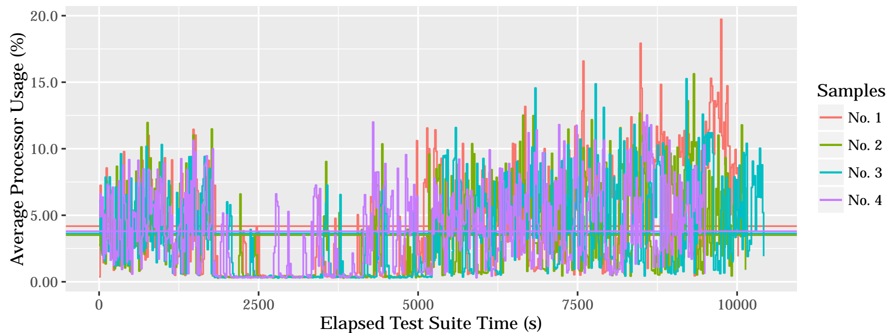


Figure 5.4: Instrumented Average Processor Usage over Time

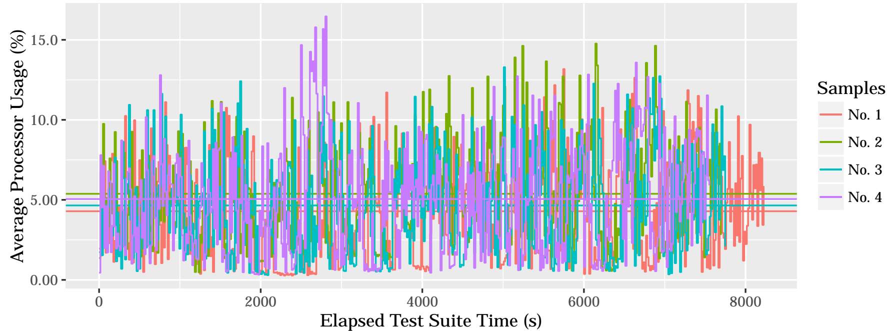


Figure 5.5: Non-Instrumented Average Processor Usage over Time

Having retrieved the *average processor usage mean* of each *sample over time*, produced the confidence intervals in Table 5.5 when being analyzed statistically. Giving the *instrumented average processor usage* from running the testing suite a range in-between $[3.43483, 4.12369]$ %, while *non-instrumented* software had an *average processor usage* interval of $[4.28527, 5.4004]$ %, after being modeled with a *t-distribution* with a significance level of $\alpha = 5\%$ and 3 *degrees of freedom*. Note that *memory* and *processor* utilization are only *assumed* to be *t-distributed*.

Table 5.5: Confidence Intervals for Average Processor Utilization

Sampled Dataset	**Lower Usage (%)**	**Upper Usage (%)**
Instrumented	3.43483	4.12369
Non-Instrumented	4.28527	5.40044

5.3 Coverage Inspection Reduction

Following the construction of the *analysis script*, presented earlier in Section 4.7, the *scovat tool* is able to determine the *similarity* and *dissimilarity* between two *test cases* by using the *Hamming distance* and *Jaccard coefficients*, described in detail under Sections 3.1.4 and 4.7.1. Now, given that the metric in Section 4.7.4 describes one method for finding the *inspection reduction amount* when looking for *potential locations* of *test case similarity*, the third initial research question presented in Section 1.4 pertaining the same subject should be explored in detail.

Demonstration of these results will be made using three different test cases, `IPForwarding#testCliRejectsInvalidAddressOnDstMo`, behaving as `IPForwarding#testCliRejectsInvalidAddressOnNexthopMo` closely, while `PL1#testPL1TestSuite` exercises completely different production code. Therefore, both `IPForwarding` tests should be *similar*, while `PL1` *dissimilar*. After executing the commands outlined in Section 4.7.3 to obtain the similarity report between these test cases, the results in Tables 5.6 and 5.7 were retrieved, after *generating* in *64.88 seconds* and *analyzing* within *55.06 + 27.19 seconds*.

Table 5.6: Analysis Results Between the Similar `IPForwarding` Test Cases

Criterion	**Test Coverage**	$d(A,B)$	$A \cap B$	$A \cup B$	$J(A,B)$
Statement	00.22% of 184 679	0	400	400	1.00000
Function	00.13% of 71 614	0	90	90	1.00000
Branch	00.03% of 385 522	0	132	132	1.00000

Table 5.7: Analysis Results Between Different `PL1` & `IPForwarding` Tests

Criterion	**Test Coverage**	$d(A,B)$	$A \cap B$	$A \cup B$	$J(A,B)$
Statement	11.96% of 184 679	21 691	398	22 089	0.01801
Function	07.62% of 71 614	5 369	90	5 459	0.01648
Branch	05.73% of 385 522	21 960	123	22 083	0.00557

Before proceeding further, an observation about the results within Table 5.6, regarding the similarity, is needed, since these might be greatly misinterpreted. Since the *Jaccard coefficient* between both `IPForwarding` tests is exactly one, they *exercise precisely* the same *instructions*, *however*, that *doesn't imply* both of these *verify the same functionality* (e.g. might provide different parameters). Given these limitations, the *reductions* in *manual inspection* are summarized below in Table 5.8, where the *intersecting areas* need *manual similarity analysis*.

Table 5.8: Manual Inspection Reduction for Tables 5.6 (Left) & 5.7 (Right)

Criterion	**Reduction (Left)**
Statement	0.0000% (100.0%)
Function	0.0000% (100.0%)
Branch	0.0000% (100.0%)

Criterion	**Reduction (Right)**
Statement	98.199% (1.801%)
Function	98.352% (1.648%)
Branch	99.443% (0.557%)

Chapter 6

Discussion

Before delving into the *conclusions* for this thesis, reached from *results* gathered, this chapter *discusses* relevant *issues* and *observations* with the *thesis processes* which were used to reach these *results*, since these might *affect* the *conclusions*. *Methods, results, validity, references, safety and future* are discussed henceforth.

6.1 Methods Utilized

Several (if not most) of the *methods* used previously are *specifically tailored* to the *target system under extension*, which means that replication isn't very easy, and that *related research* usually don't cover these problems which are unique, leading to some decisions that aren't based on research, instead on requirements.

While both *gathering and analysis* of *instrumented performance* use proven *scientific methods* from *previous research*, the *number of samples* used is *small*, which might lead to *faulty results* and *conclusions*, but was unavoidable given the time frame. Additionally, even though precautions were taken to setup the same *preconditions* before *gathering samples*, subsequent runs could be *affected*.

Application of the *instrumentations* to the entire source code was done, but, flushing *coverage data* from linked *shared libraries* is still an *open issue*, however, since the target production code only uses *static linking*, so it's acceptable here.

Automatically gathering coverage data on *test case finalization* actually adds a great deal of *extra execution time*, around 20*s* for each test case for fetching. It's important to note that the method used in Section 4.3.1 for gathering raw performance does *not* take this into account, since the given first initial research question *only* deals with the *performance costs* of *instrumentation*, nothing else.

Producing *coverage reports* of the *full testing suite* gives information about the different *coverage measurements* later in the thesis, however, the *chosen tool* only provides the *degree of testing* of *instrumented production code*, meaning, source that *isn't instrumented* won't be accounted for; an *important limitation*.

Lastly, a final *method critique* concerns the produced *analysis tool*, which was built under the assumption that *test case similarity* and *test case uniqueness* are two *interesting metrics* for analysis, which would in reality require a thorough examination if that is indeed the case. But, as described in Section 1.5, these are some of the preconditions that were setup given the companies requirements. Therefore, no method part regarding this was written, which could be criticised.

6.2 Results Encountered

While the *results* found for the *coverage measurements* are measured critically, they still only provide information relative to an amount of *instrumented parts*, which might be *misleading* since the *true* testing degree could deviate from it, however, these are properties that are inherent from the utilized coverage tool.

Additionally, *branch coverage measurements* were retrieved differently from *statement* or *function coverage*, which might affect the final results produced. However, the *report building tool* which was used for *statement and function*, utilizes *gcov* in their implementation, as shown in their *manual* [Obec], which should conclude that the final measurements of *branch coverage* are still correct.

Several *performance results* were gathered, but *suite execution time* has the most *significant* impact since the *certainty* of correctness is higher than the other two measured *performance metrics*, if *Schenker et al.* [SG01] is to be believed, however, the *overhead* of using data from the *testing framework* might deviate the *true testing time* of the suite, as will having such *small* amount of *samples*. Perhaps the most surprising result found is that *instrumented processor usage* has produced *lower* values than the *non-instrumented*, which is quite *unexpected*, and the logical explanation is that either the *sample count* is too *low* or that there are *more memory accesses* afterwards, leading towards *low processor use*. Since the sample count is low, perhaps *Wilcoxon test* would have been more suitable, and *Auto Regressive model* for *memory usage* since it removes trends. Additionally the *cpu usage* produces *noisy output*, taking the *logarithm values* would have been more suitable, *flattening* Figures 5.4 and 5.5 more cohesively.

Finally, regarding the *analysis results* found in Section 5.3, practically, these particular test candidates were chosen since they display the varied properties of *similarity* and *dissimilarity* quite nicely, which were recommended by engineers with knowledge of the testing environment, which wasn't a very scientific choice. However, the particular choices *shouldn't affect* the *research outcome directly*, since the only purpose of displaying these empirical data were to provide *real executions* and *results*, where the *reduction always* follows the *Jaccard difference*. Therefore, the *empirical results* are given for these *particular set of tests*, but, the *conclusions* on *manual inspection reduction* are derived from *general models*.

6.3 Threats to Validity

Primary concerns which could affect the final outcome from this thesis are the *low amount of performance samples collected* that are used to reach conclusions about the *effects* of *instrumentation*, especially since the assumed model falls under *four degrees of freedom*, which is still under debate, e.g. *De Winter* [DW13]. Additionally, *few conclusions* were made about *memory* and *processor usages*, since under $\alpha = 5\%$ *nothing could be asserted*, which might have been false, e.g. by *reducing the significance level* or instead manipulating values *logarithmically*.

Since the given *analysis method* is built around the *initial assumption* that *test similarity and uniqueness* are *interesting properties* to analyze, the desire to *prove* or *reference* such a claim is obvious, which couldn't be found concretely. However, the previously mentioned research paper by *Cartaxo et al.* [CNM07] proposes that *test similarity* can be utilized to determine *test case redundancy*, which on the other hand is quite useful when attempting *test case minimization*.

6.4 Reference Criticism

Several of the *formulated problems*, and therefore *research questions* are derived from *Adler et al's* [ABR+11] observations, thereby directing this thesis's results. Since the aforementioned article has quite recently been published, the citation count isn't high; however since the authors are reputable in the given area, having several published papers in well respected journals, their insight has been deemed as trustworthy. Despite this, doubt must be cast on their statement regarding the difficulty in *gathering coverage data*, which has very few references, while their other argument regarding *analysis*, has several trustworthy sources.

Certain assumptions were made earlier in Section 3.1.4 regarding metrics for measuring the *similarity and uniqueness* of *test cases*, which were derived from *Hemmati et al's* [HB10] research upon *model-based testing similarity techniques*. Note that these metrics pertain to *automatically generated tests*, quite different from regular testing techniques; assumption is made that this also applies here.

Finally, several sources are *manuals*, which do not provide scientific insight.

6.5 Safety Considered

Reaching conclusions about function tests entirely based on the *analysis tool* is not reliable since *code coverage* doesn't tell everything about the semantics between *tests cases* and *software*, input from an informed engineer is still needed. Which is further reinforced by articles like *Brian Marick* [M+99], [WMMK01], mentioning frequent misuses of coverage data by developers and managers alike. Situation was remedied by giving a presentation about this coverage gathering and analysis system, besides demonstrating practical functionality, developers and managers at Ericsson R&D were informed about these solution's limitations; hopefully leading the projects using it, to not misuse the tool and given results.

Additionally, keeping confidential information from entering this document while still providing valuable data was an important consideration throughout, permission has been granted by those responsible in releasing these thesis's data.

6.6 Future Research

Related research in *coverage analysis methods* on *large-scale systems* would be interesting to investigate, such as *IBM's Substring Hole Analysis* [AFK+09a], which is quite different from the methods considered here. More precisely, their *relevance ranking* presented in their article could be applied towards the method shown in this thesis, given that *clustering* is possible, since *distance functions* have been supplied, an important requirement for *cluster analysis*, as shown in the survey by *Pavel Berkhin* [Ber06]. Actually, *clustering* in general is desirable, since an engineer is still required to select a singular pair of tests for analysis, which can be made more intuitive by *hierarchically grouping similar test clusters*.

Since the original purpose of *analyzing test coverage* was to eventually utilize these *test case similarities* towards doing *test case selection and prioritization*, which enables *faster feedback loops* on Ericsson's *continuous integration systems*, there still remains the question whether this is can be realistically implemented. Presented here is one *gathering and test similarity analysis system*, which indeed provides the initial stepping stones towards making such an extension possible.

Chapter 7

Conclusions

Application of the *methods* described in Chapter 4 have produced a relevant set of *results* in Chapter 5 which are used to reach the *conclusions* presented in this chapter regarding the *research questions* & *thesis purpose* shown in Chapter 1. Some of these assertions are *discussed* in Chapter 6, along with possible fallacies. After developing the *coverage gathering and analysis* system, *experience* about the *problems* and *solutions* received along this thesis are shown in Section 7.1. Analysis of *criteria measures* and *performance data* is done in Sections 7.2, 7.3, while *interpretation* and *reduction* of the *coverage tool* is explored in Section 7.4.

7.1 Development Experience

Building the necessary *infrastructure* to *gather* and then later *analyze* these sets of *test coverage data* on this particular *large-scale industrial software system*, has led to some *technical insights* which could be informative for other engineers too. Assertions previously made by *Adler et al* [ABR+11] regarding the *difficulties* in *extending* a *large-scale system* with *coverage gathering* capabilities, were present in this thesis project, and as also predicted by *Adler et al*, the primary reason for this were the *inherent problems* that arose because of the *diverse toolchain* used, and the *difficulty* for existing *coverage tools* to *conform* themselves accordingly. Particularly, *primary issues* here were the *non-compatibility* with *gcov* to handle *coverage gathering* on *remote targets* and when *daemon processes* were involved, which were *resolved* with [GNUd] and from the results of *Hofrat's* [Hof11] article.

Issues that weren't explicitly presented by *Adler et al* [ABR+11] were the inherent problems that certain *coverage analysis tools* require *coverage data* to be *processed* or *organized* to a certain *format* before being *analyzed*. Doing this was also another *feature* which needed to be *tailored* for this particular system, since *individual test case coverage* was desired for doing *set operations* on these, which could be solved by modifying the *testing flow* to *flush test case coverage*. Reinforcing the statement by *Adler et al* that coverage tools can't integrate well.

However, in contrast to the early results presented by *Adler et al* [ABR+11], *performance* was *not* an issue and *didn't affect* the *development* of the system, which was reported to also be an area for which *coverage tools* need *special care*. Motivation for these particular assertions are reinforced with Sections 5.2 & 7.2. Integrating *coverage gathering & analysis* on *large-scale systems* seems possible.

7.2 Implementation Feasibility

Quantifying the *performance effects* of executing *coverage instrumented software* on the *full testing suite* of the *large-scale system* has been important for verifying the *feasibility of the implementation* and to determine links to previous research. Only three metrics were considered, *test suite execution time*, *average memory and processor usage*, which were analyzed statistically. Additionally, *stability* is determined by the number of *test cases* which have *broken* because of coverage.

Suite execution time was determined to range between $[2.656, 2.911]$ hours for *instrumented builds* while only $[2.075, 2.260]$ for *non-instrumented software*. Following the advice and results taken from *Schenker et al* [SG01], it's deemed *safe* to draw *conclusions* about the possible *implications* posed by these since the *confidence intervals* don't *overlap*, with a confidence of 95% as given before. Above intervals *imply* that *coverage instrumented software* takes *significantly longer* to execute than *non-instrumented builds* of similar software while *testing*.

Average memory usage was shown to range within $[16.766, 18.844]$ % for *instrumented* and $[17.842, 18.018]$ % out of 6 GiB for *non-instrumented builds*. Based on *Schenker et al* [SG01] yet again, *overlapping confidence intervals* are *unsuitable* to prove *relations* between datasets or for *drawing any conclusions*. Therefore, *nothing* of *significance* can be said about the *average memory effects*.

Average processor usage has been deemed to be between $[3.434, 4.123]$ % for *instrumented software* and $[4.285, 5.400]$ % for *non-instrumented builds*, $\alpha = 5\%$. Interestingly, by following *Schenker et al's* [SG01] advice this time around, leads to the illogical conclusion that *coverage instrumented builds* seems to display a *lower average processor usage* than *non-instrumented software* while *testing*. Observations have been made to try and explain the discrepancy, see Section 6.2.

Stability of the *software* has been kept since *test cases* haven't broken, which leads to the conclusion that *coverage* is *feasible*, given that the *suite testing time* isn't a problem in the *instrumented large-scale software system*, this will be true. It should be noted that while *results exists*, it depends on this *particular system*, careful consideration should be taken when attempting to apply this generally.

7.3 Coverage Measurements

Running the *full testing suite* on target while having *coverage instrumentations*, produced several *coverage measurements* for various different *coverage criteria*. Both *Piwowarski et al* [POC93] and *Kim Yong* [Kim03] have described that this type of data is *scarce* for *large-scale industrial systems*, and are provided below.

Statement coverage or *line coverage* is reported to have a *degree* of 59.3% *statements hit* out of 96 158 *instrumented* for this specific test suite and system.

Function coverage has been measured to have a *testing degree* of 70.7% *functions executed* out of 23 870 in total *instrumented* for the full testing suite.

Branch coverage is the last gathered metric, and has a *hit degree* of 24.6% *branches entered* out of 226 927 *instrumented branches* for this full testing suite. Regarding the *similarity* of this data to *large-scale* projects similar in size and existing contributions in the area, there have been open reports showing 56.6% *statement coverage* from *Marko Ivanković* [Iva14] at *Google Zürich*, which have projects similar in size and which have used the exact same set of coverage tools, which implies that Ericsson's *coverage measurements* comply with existing data.

7.4 Analysis Interpretation

Following the construction of *Set Coverage Analysis Tool (SCovAT)*, a reference implementation of the *test coverage similarity analysis method* presented earlier, the *properties/information* provided upon execution are to be formally defined; since engineers tasked with *analyzing raw coverage data* face imminent difficulty, described by *Adler et al* [ABR+11] as the *"needle in a haystack"* problem, when attempting to sift through any relevant properties in *large-scale software testing.*

Deriving from *Cartaxo et al's* [CNM07] observations, *test case similarity* and *difference* are measurements made possible with the *Jaccard coefficients* and the *Hamming distances* of *coverage criteria "hits"*, such as *statements* or *functions.* Additionally, *Cartaxo et al.* [CNM07] shows *similarity* leads to *test redundancy,* or more correctly, *test redundancy* requires *test case similarity.* Note, however, *similarity* does not *strictly* imply *redundancy,* like results in Tables 5.6 and 5.7. Therefore, the *analysis method* displays the *similarity areas between test cases,* measured using *Jaccard indices,* showing *potential locations* of *test redundancy.*

There are several applications of these *analyzed test similarity comparisons,* noted here for completeness are *test minimization, test prioritization, clustering.* Most interesting here is *clustering* since is solves the *"needle in a haystack"* issue more elegantly, not requiring *pairwise comparison* between test cases for finding the *desired properties. Pavel Berkhin's* [Ber06] survey outlines these techniques.

Tightly associated with determining *locations* of these *similarity properties* regarding tests is the task of *measuring reduction* in the amount of *coverage data* that needs to be *manually analyzed for redundancy.* Included here are *example comparisons* between *three test cases*: a *pair* of *similar tests* and *one unrelated.* All conclusions that follow are derived from Tables 5.6, 5.7 & 5.8 in Section 5.3.

1. **testCliRejectsInvalidAddressOnDstMo:** similar to **(2)**, unlike **(3)**.
2. **testCliRejectsInvalidAddressOnNexthopMo:** relates **(1)**, not **(3)**.
3. **testPL1TestSuite:** very few similarities towards the **(1)** and **(2)** test.

Results provided through the *analysis tool* indicate that **(1)** and **(2)** behave according to the *Jaccard coefficient 1.000* in respect to all the *coverage criterias,* implying both did *execute exactly* the same *functions, branches and statements.* This leads towards *0% manual inspection reduction* for finding *test redundancy,* since *all coverage areas are similar*, which might *potentially* contain *redundancy.* Analyzing **(1)** and **(3)** on the other hand reveals (*0.01648, 0.00557, 0.01801*), as the *Jaccard coefficients* of *function, branch and statement*, in that sequence. Implying that *reductions* in *98.352%* of the *functions*, *99.443%* of *branches* and finally *98.199%* of the *statements* are made upon attempting to *find redundancy.* Sifting through **(1)** and **(3)** for *redundancy* is quite more manageable, compared towards the benefits from **(1)** and **(2)**, which provided *no inspection reduction.* More generically, the *inspection reduction* follows a *Jaccard distance* $1-J(A,B)$, where $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$ is calculated with the defined *set operations* for *coverage,* provided by the reference implementation *SCovAT [Vas]*, which is free software.

> *"'The Answer to the Great Question... Of Life, the Universe and Everything... Is... Forty-two', said Deep Thought, with infinite majesty and calm."* — Douglas Adams, *The Hitchhiker's Guide to the Galaxy*

Bibliography

[ABR+11] Yoram Adler, Noam Behar, Orna Raz, Onn Shehory, Nadav Steindler, Shmuel Ur, and Aviad Zlotnick. Code coverage analysis in practice for large systems. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 736–745. ACM, 2011.

[AFK+09a] Yoram Adler, Eitan Farchi, Moshe Klausner, Dan Pelleg, Orna Raz, Moran Shochat, Shmuel Ur, and Aviad Zlotnick. Advanced code coverage analysis using substring holes. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 37–46. ACM, 2009.

[AFK+09b] Yoram Adler, Eitan Farchi, Moshe Klausner, Dan Pelleg, Orna Raz, Moran Shochat, Shmuel Ur, and Aviad Zlotnick. Automated substring hole analysis. In *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 203–206. IEEE, 2009.

[B+81] Barry W Boehm et al. *Software engineering economics*, volume 197. Prentice-hall Englewood Cliffs (NJ), 1981.

[Ber06] Pavel Berkhin. A survey of clustering data mining techniques. In *Grouping multidimensional data*, pages 25–71. Springer, 2006.

[BGS+12] Árpád Beszédes, Tamás Gergely, Lajos Schrettner, Judit Jász, Laszlo Lango, and Tibor Gyimóthy. Code coverage-based regression test selection and prioritization in webkit. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 46–55. IEEE, 2012.

[Boo91] Grady Booch. Object oriented design with applications. redwood city, 1991.

[BR70] John N Buxton and Brian Randell. *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*. NATO Science Committee; available from Scientific Affairs Division, NATO, 1970.

[CL05] Xia Cai and Michael R Lyu. The effect of code coverage on fault detection under different testing profiles. *ACM SIGSOFT software engineering notes*, 30(4):1–7, 2005.

[CNM07] Emanuela G Cartaxo, Francisco G Oliveira Neto, and Patrícia DL Machado. Automated test case selection based on a similarity function. *GI Jahrestagung (2)*, 7:399–404, 2007.

[Des] Semantic Designs. Test coverage tools. `http://www.semanticdesigns.com/Products/TestCoverage/`. Online Product Website, Accessed: 2016-05-19.

[DW13] Joost CF De Winter. Using the student's t-test with extremely small sample sizes. *Practical Assessment*, 18(10):1–12, 2013.

[Eri15] Ericsson. Ericsson annual report 2015, 2015. Fininaical Report.

[FF06] Martin Fowler and Matthew Foemmel. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, page 122, 2006.

[GNUa] GNU. Gcc 5.3 branch source code github mirror. `https://github.com/gcc-mirror/gcc/blob/gcc-5-branch`. Source Repository, Accessed: 2016-05-11.

[GNUb] GNU. The gnu/gcc manual (gcov): A test coverage program, invocation. `https://gcc.gnu.org/onlinedocs/gcc/Invoking-Gcov.html#Invoking-Gcov`. Online Manual, Accessed: 2016-05-08.

[GNUc] GNU. The gnu/gcc manual (gcov-tool): An offline coverage profile processing tool. `https://gcc.gnu.org/onlinedocs/gcc/Invoking-Gcov-tool.html#Invoking-Gcov-tool`. Online Manual, Accessed: 2016-05-08.

[GNUd] GNU. The gnu/gcc manual (ggov): A test coverage program, cross-profiling. `https://gcc.gnu.org/onlinedocs/gcc/Cross-profiling.html#Cross-profiling`. Online Manual, Accessed: 2016-05-08.

[GNUe] GNU. The linux/unix manual (proc): Process information pseudo-file system. `http://linux.die.net/man/5/proc`. Accessed: 2016-05-02.

[GNUf] GNU. The linux/unix manual (setenv): Change or add an environment variable. `http://man7.org/linux/man-pages/man3/setenv.3.html`. Online Manual, Accessed: 2016-05-08.

[Har00] Mary Jean Harrold. Testing: a roadmap. In *Proceedings of the conference on the future of software engineering*, pages 61–72. ACM, 2000.

[HB10] Hadi Hemmati and Lionel Briand. An industrial investigation of similarity measures for model-based test case selection. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 141–150. IEEE, 2010.

[HLFR04] Nigel Hinds, Paul Larson, Hubertus Franke, and Marty Ridgeway. Using code coverage tools in the linux kernel. 2004.

[Hof11] "Der Herr Hofrat". Dumping gcov data at runtime - a simple example. Technical report, OpenTech EDV Research GmbH, 2011.

[HS02] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[Iva14] Marko Ivanković. Measuring coverage at google. Technical report, Google Zürich, 2014.

[Jan] Erik Sven Vasconcelos Jansson. Analysis of test coverage data on a large-scale industrial system, description. Project Specification, Written: 2016-02-06.

[JN99] Dag Jonsson and Lennart Norell. *Ett stycke statistik*. Studentlitteratur, 1999.

[Joh] Paul Johnson. Testing and code coverage. `http://pjcj.net/testing_and_code_coverage/paper.html`. Online Article, Accessed: 2016-06-05.

[Joh98] Leslie A Johnson. Do-178b, software considerations in airborne systems and equipment certification. *Crosstalk, October*, 1998.

[Kim03] Yong Woo Kim. Efficient use of code coverage in large-scale software development. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, pages 146–155. IBM Press, 2003.

[LHRF03] Paul Larson, Nigel Hinds, Rajan Ravindran, and Hubertus Franke. Improving the linux test project with kernel code coverage analysis. In *2003 Ottawa Linux Symposium*. Citeseer, 2003.

[Lil05] David J Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge University Press, 2005.

[M+99] Brian Marick et al. How to misuse code coverage. In *Proceedings of the 16th Interational Conference on Testing Computer Software*, pages 16–18, 1999.

[Mey92] Bertrand Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

[MM63] Joan C Miller and Clifford J Maloney. Systematic mistake analysis of digital computer programs. *Communications of the ACM*, 6(2):58–63, 1963.

[MSB12] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2012.

[NH15] Tanzeem Bin Noor and Hadi Hemmati. A similarity-based approach for test case prioritization using historical failure data. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pages 58–68. IEEE, 2015.

[Obea] Peter Oberparleiter. The linux/unix manual (genhtml): Generate html views from lcov coverage data files. http://ltp.sourceforge.net/coverage/lcov/genhtml.1.php. Online Manual, Accessed: 2016-05-08.

[Obeb] Peter Oberparleiter. The linux/unix manual (geninfo): Generate tracefiles from gcda files. http://ltp.sourceforge.net/coverage/lcov/geninfo.1.php. Online Manual, Accessed: 2016-05-08.

[Obec] Peter Oberparleiter. The linux/unix manual (lcov): A graphical gcov front-end. http://ltp.sourceforge.net/coverage/lcov/lcov.1.php. Online Manual, Accessed: 2016-05-08.

[POC93] Paul Piwowarski, Mitsuru Ohba, and Joe Caruso. Coverage measurement experience during function test. In *Software Engineering, 1993. Proceedings., 15th International Conference on*, pages 287–301. IEEE, 1993.

[PXN13] Yulei Pang, Xiaozhen Xue, and Akbar Siami Namin. Identifying effective test cases through k-means clustering for enhancing regression testing. In *Machine Learning and Applications (ICMLA), 2013 12th International Conference on*, volume 2, pages 78–83. IEEE, 2013.

[Rut] Nick Rutar. Using set operations on code coverage data to discover program properties. https://www.cs.umd.edu/class/spring2005/cmsc838p/Presentations/rutarPres.ppt. Online Presentation, Accessed: 2016-05-19.

[SAP] SAP. Merging code coverage measurements. https://help.sap.com/saphelp_nw75/helpdata/en/78/047b27a4e0424fa8ca99281773567a/content.htm. Online Documentation, Accessed: 2016-05-19.

[SG01] Nathaniel Schenker and Jane F Gentleman. On judging the significance of differences by examining the overlap between confidence intervals. *The American Statistician*, 55(3):182–186, 2001.

[Tas02] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011), 2002.

[Vas] Erik Sven Jansson Vasconcelos. SCOVAT: The Set Coverage Analysis Tool. https://github.com/Caffeineviking/scovat. Source Code Repository, Accessed: 2016-07-11.

[WMMK01] TW Williams, MR Mercer, JP Mucha, and R Kapur. Code coverage, what does it mean in terms of quality? In *Reliability and Maintainability Symposium, 2001. Proceedings. Annual*, pages 420–424. IEEE, 2001.

[YLW09] Qian Yang, J Jenny Li, and David M Weiss. A survey of coverage-based testing tools. *The Computer Journal*, 52(5):589–597, 2009.

Appendix A

Suite Coverage Report

	Line Coverage		Functions	
	61.5 %	8 / 13	44.4 %	4 / 9
	100.0 %	2 / 2	100.0 %	1 / 1
	76.4 %	1658 / 2171	76.0 %	6693 / 8807
	69.0 %	29 / 42	77.1 %	1078 / 1399
	87.1 %	61 / 70	76.6 %	36 / 47
	78.4 %	2137 / 2725	90.4 %	506 / 560
	73.0 %	1153 / 1579	87.6 %	85 / 97
	33.3 %	1 / 3	33.3 %	1 / 3
	53.3 %	1158 / 2174	63.4 %	256 / 404
	63.2 %	1071 / 1694	82.1 %	92 / 112
	52.7 %	154 / 292	65.8 %	52 / 79
	68.4 %	303 / 443	83.5 %	66 / 79
	49.0 %	179 / 365	66.7 %	24 / 36
	66.7 %	2 / 3	33.3 %	1 / 3
	52.7 %	382 / 725	59.7 %	105 / 176
	38.8 %	506 / 1304	53.1 %	60 / 113
	79.3 %	452 / 570	98.1 %	51 / 52
	50.0 %	1 / 2	33.3 %	1 / 3
	56.3 %	911 / 1617	71.5 %	118 / 165
	40.0 %	2 / 5	16.7 %	1 / 6
	61.9 %	578 / 934	77.6 %	118 / 152
	44.4 %	4 / 9	25.0 %	3 / 12
	52.7 %	1356 / 2575	75.2 %	309 / 411
	55.0 %	11 / 20	28.6 %	6 / 21
	69.5 %	3562 / 5127	81.0 %	700 / 864
	50.0 %	1 / 2	33.3 %	1 / 3
	59.3 %	513 / 865	75.6 %	68 / 90
	44.0 %	11 / 25	50.0 %	1 / 2
	61.4 %	415 / 676	88.9 %	32 / 36
	68.1 %	1195 / 1754	92.3 %	84 / 91

Figure A.1: Continued Coverage Report on the Full Testing Suite

Appendix B

Coverage Tool Patches

Listing B.1: GCC GCov-Tool Patch for Intersection and Difference

```
diff --git a/gcc/gcov-tool.c b/gcc/gcov-tool.c
index fd27d7c..14ccea8 100644
--- a/gcc/gcov-tool.c
+++ b/gcc/gcov-tool.c
@@ -40,7 +40,7 @@ see the files COPYING3 and COPYING.RUNTIME
    respectively.  If not, see
 #endif
 #include <getopt.h>

-extern int gcov_profile_merge (struct gcov_info*, struct gcov_info*,
    int, int);
+extern int gcov_profile_merge (struct gcov_info*, struct gcov_info*,
    int, int, char);
 extern int gcov_profile_overlap (struct gcov_info*, struct gcov_info*)
    ;
 extern int gcov_profile_normalize (struct gcov_info*, gcov_type);
 extern int gcov_profile_scale (struct gcov_info*, float, int, int);
@@ -124,11 +124,12 @@ gcov_output_files (const char *out, struct
    gcov_info *profile)
 }

 /* Merging profile D1 and D2 with weight as W1 and W2, respectively.
-   The result profile is written to directory OUT.
+   The result profile is written to directory OUT with the STRATEGY.
    Return 0 on success.  */

 static int
-profile_merge (const char *d1, const char *d2, const char *out, int w1
    , int w2)
+profile_merge (const char *d1, const char *d2, const char *out,
+               int w1, int w2, char strategy)
 {
   struct gcov_info *d1_profile;
   struct gcov_info *d2_profile;
@@ -145,7 +146,8 @@ profile_merge (const char *d1, const char *d2,
    const char *out, int w1, int w2)
         return 1;

       /* The actual merge: we overwrite to d1_profile.  */
-      ret = gcov_profile_merge (d1_profile, d2_profile, w1, w2);
```

```
+      ret = gcov_profile_merge (d1_profile, d2_profile,
+                                w1, w2, strategy);

       if (ret)
         return ret;
@@ -167,6 +169,8 @@ print_merge_usage_message (int error_p)
   fnotice (file, "    -v, --verbose                      Verbose mode
    \n");
   fnotice (file, "    -o, --output <dir>                 Output
    directory\n");
   fnotice (file, "    -w, --weight <w1,w2>               Set weights
    (float point values)\n");
+  fnotice (file, "    -i, --intersection                 Uses
    intersection merge strategy\n");
+  fnotice (file, "    -d, --difference                   Uses
    difference merge strategy\n");
 }

 static const struct option merge_options[] =
@@ -174,6 +178,8 @@ static const struct option merge_options[] =
   { "verbose",            no_argument,       NULL, 'v' },
   { "output",             required_argument, NULL, 'o' },
   { "weight",             required_argument, NULL, 'w' },
+  { "intersection",       required_argument, NULL, 'i' },
+  { "difference",         required_argument, NULL, 'd' },
   { 0, 0, 0, 0 }
 };

@@ -196,9 +202,10 @@ do_merge (int argc, char **argv)
   int ret;
   const char *output_dir = 0;
   int w1 = 1, w2 = 1;
+  char strategy = 0;

   optind = 0;
-  while ((opt = getopt_long (argc, argv, "vo:w:", merge_options, NULL)
    ) != -1)
+  while ((opt = getopt_long (argc, argv, "vo:w:id", merge_options,
    NULL)) != -1)
     {
       switch (opt)
         {
@@ -214,6 +221,11 @@ do_merge (int argc, char **argv)
           if (w1 < 0 || w2 < 0)
             fatal_error (input_location, "weights need to be non-
    negative\n");
           break;
+        case 'i': case 'd':
+          if (strategy == 0) strategy = opt;
+          else fatal_error (input_location,
+               "multiple merge strategies\n");
+          break;
         default:
           merge_usage ();
         }
@@ -223,7 +235,8 @@ do_merge (int argc, char **argv)
     output_dir = "merged_profile";

   if (argc - optind == 2)
-    ret = profile_merge (argv[optind], argv[optind+1], output_dir, w1,
     w2);
+    ret = profile_merge (argv[optind], argv[optind+1],
```

```
+                          output_dir, w1, w2, strategy);
   else
     merge_usage ();

diff --git a/libgcc/libgcov-util.c b/libgcc/libgcov-util.c
index d76c2eb..f445260 100644
--- a/libgcc/libgcov-util.c
+++ b/libgcc/libgcov-util.c
@@ -515,6 +515,48 @@ merge_wrapper (gcov_merge_fn f, gcov_type *v1,
    gcov_unsigned_t n,
   (*f) (v1, n);
 }

+static void
+gcov_merge_add_intersection(gcov_type* counters,
+                             gcov_unsigned_t quantity)
+{
+  for (; quantity; counters++, quantity--)
+    {
+      gcov_type other_counter = gcov_get_counter ();
+      if (other_counter == 0
+          || *counters == 0) *counters = 0;
+      else  *counters += other_counter;
+    }
+}
+
+static void
+gcov_merge_add_difference(gcov_type* counters,
+                           gcov_unsigned_t quantity)
+{
+  for (; quantity; counters++, quantity--)
+    {
+      gcov_type other_counter = gcov_get_counter ();
+      if (other_counter != 0) *counters = 0;
+    }
+}
+
+static gcov_merge_fn
+gcov_merge_function (unsigned counter,
+                      char strategy)
+{
+  switch (strategy)
+    {
+    case 'i':
+      if (counter == GCOV_COUNTER_ARCS)
+        return gcov_merge_add_intersection;
+      else return ctr_merge_functions[counter]; break;
+    case 'd':
+      if (counter == GCOV_COUNTER_ARCS)
+        return gcov_merge_add_difference;
+      else return ctr_merge_functions[counter]; break;
+    default: return ctr_merge_functions[counter];
+    }
+}
+
 /* Offline tool to manipulate profile data.
    This tool targets on matched profiles. But it has some tolerance on
    unmatched profiles.
@@ -532,7 +574,8 @@ merge_wrapper (gcov_merge_fn f, gcov_type *v1,
    gcov_unsigned_t n,
 /* Add INFO2's counter to INFO1, multiplying by weight W.  */
```

```
 static int
-gcov_merge (struct gcov_info *info1, struct gcov_info *info2, int w)
+gcov_merge (struct gcov_info *info1, struct gcov_info *info2,
+            int w, char strategy)
 {
   unsigned f_ix;
   unsigned n_functions = info1->n_functions;
@@ -569,6 +612,7 @@ gcov_merge (struct gcov_info *info1, struct
    gcov_info *info2, int w)
           if (!merge1)
             continue;
           gcc_assert (ci_ptr1->num == ci_ptr2->num);
+          if (strategy != 0) merge1 = gcov_merge_function (t_ix,
    strategy);
           merge_wrapper (merge1, ci_ptr1->values, ci_ptr1->num,
    ci_ptr2->values, w);
           ci_ptr1++;
           ci_ptr2++;
@@ -621,7 +665,7 @@ find_match_gcov_info (struct gcov_info **array, int
     size,

 int
 gcov_profile_merge (struct gcov_info *tgt_profile, struct gcov_info *
    src_profile,
-                    int w1, int w2)
+                    int w1, int w2, char strategy)
 {
   struct gcov_info *gi_ptr;
   struct gcov_info **tgt_infos;
@@ -651,7 +695,8 @@ gcov_profile_merge (struct gcov_info *tgt_profile,
    struct gcov_info *src_profile
   if (w1 > 1)
     {
       for (i = 0; i < tgt_cnt; i++)
-        gcov_merge (tgt_infos[i], tgt_infos[i], w1-1);
+        gcov_merge (tgt_infos[i], tgt_infos[i],
+                    w1-1, strategy);
     }

   /* Second pass, add src_profile to the tgt_profile.  */
@@ -665,14 +710,17 @@ gcov_profile_merge (struct gcov_info *tgt_profile
    , struct gcov_info *src_profile
           in_src_not_tgt[unmatch_info_cnt++] = gi_ptr;
           continue;
         }
-      gcov_merge (gi_ptr1, gi_ptr, w2);
+      gcov_merge (gi_ptr1, gi_ptr,
+                  w2, strategy);
     }

+  if (strategy == 'i') return 0;
   /* For modules in src but not in tgt. We adjust the counter and
    append.  */
   for (i = 0; i < unmatch_info_cnt; i++)
     {
      gi_ptr = in_src_not_tgt[i];
-      gcov_merge (gi_ptr, gi_ptr, w2 - 1);
+      gcov_merge (gi_ptr, gi_ptr,
+                  w2 - 1, strategy);
      tgt_tail->next = gi_ptr;
      tgt_tail = gi_ptr;
    }
```