Bridging the Gap: Developing a Standardized Framework for any C++ Code Coverage Tool

Ahmed Abdulbaki Ibrahim
ahmed.ibrahim3@student.uva.nl

July 16, 2023, 51 pages

Academic supervisor: Dr. ir. M.C. (Martin) Bor, m.c.bor@uva.nl
Daily supervisor: Nicola Rossi, nicola@solidsands.nl
Host organisation: Solid Sands B.V., solidsands.com

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

Code coverage tools showcase the degree to which software source code has been thoroughly tested, following the execution of a specific test suite. As a measure of software quality, code coverage should ideally be accurate and consistent, allowing no room for discrepancies in its assessment of source code. Unfortunately, this is not the case in reality. For software testers to manage such inconsistencies effectively, a standardized framework is necessary for their identification and mitigation.

However, developing a systematic protocol presents a challenge due to the diverse nature of these inconsistencies; some are tool-specific shortcomings, while others arise from confusion about the coverage status of particular code statements. The highly language-dependent nature of coverage tools, along with the numerous different types of coverage criteria, further complicates the issue. As a result, we have focused solely on the C++ language specification and the statement, branch, modified condition-decision coverage criteria. Our selections are motivated by the need for coverage tools to comply with the development of safety-critical systems and their applications.

Our proposed framework comprises three key elements: a robust set of software requirements that define the necessary functionalities of any coverage tool, an innovative test suite that incorporates all aspects of the aforementioned requirement set in addition to ambiguous code statements, and a Python tool that evaluates the performance of any C++ coverage tool in line with our unique test suite.

The necessity to address these concerns is emphasized by a significant gap in current literature and research examining discrepancies in code coverage tools' behaviour. This represents a considerable risk to safety-critical systems and their compliance with pertinent guidelines and international standards, such as DO-178C and ISO26262.

Contents

Chapter 1

Introduction

Software testing serves as a critical component in ensuring the quality of all developed systems and applications [1]. We can define it as a process designed to evaluate a program or system's specific attributes to ensure it meets its required results [2]. Ideally, the intent is not just to confirm functionality, but also to uncover errors [3]. The testing process often includes writing a variety of test cases to examine the breadth of a software's functionalities, cumulatively forming a 'test suite'. The lack of a robust software testing infrastructure can have significant economic consequences, as evidenced by a case study revealing that it has cost the U.S. automotive and aerospace industries approximately 2$ billion in a single year [4].

Code coverage is a metric showcasing the percentage of source code that has been executed by the test suite implemented during the software testing phase. As a result, it is commonly regarded as a primary indicator of test suite quality [5]. In fact, it has been established that code coverage correlates to the effectiveness of test suites, where effectiveness is defined as the test suite's ability to detect and eliminate software bugs [6]. The software components responsible for this metric, code coverage tools, are therefore an essential aspect of ensuring software quality.

These tools function by 'instrumenting' the software code, a process that integrates additional data structures to track which segments are covered by testing. This instrumentation can occur either at the source-code level (pre-compilation) or at the byte-code level (post-compilation) [7].

```cpp
#include <iostream>

int square(int x) {
    int result = x * x;
    return result;
}

int main() {
    for(int i = -4; i <= 4; i++) {
    if (i % 2 == 0 && i != 0) {
    std::cout << i << "squared = "
     << square(i) << "\n";
        }
    }
    return 0;
}
```

Listing 1.1: Original Source Code

```cpp
#include <iostream>

static unsigned long long __gcov_counter[8] = {0};
// Each basic block gets a coverage counter
int square(int x) {
    __gcov_counter[0]++;
    int result = x * x;
    __gcov_counter[1]++;
    return result;
    __gcov_counter[2]++;
}

int main() {
    __gcov_counter[3]++;
    for(int i = -4; i <= 4; i++) {
        __gcov_counter[4]++;
        if (i % 2 == 0 && i != 0) {
        __gcov_counter[5]++;
        std::cout << i << "squared = "
         << square(i) << "\n";
        __gcov_counter[6]++;
        }
    }
    return 0;
    __gcov_counter[7]++;
}

void __gcov_exit() {}
```

Listing 1.2: Instrumented Source Code

The listings 1.1 and 1.2 illustrate conceptually how a coverage tool utilizing source-code instrumentation, such as Gcov [8], employs global data structures to record coverage information. The counter at position 5 (`__gcov_counter[5]`), responsible for keeping track of the 'if' statement's execution, depends on the control flow to be incremented. There are numerous code coverage criteria, one of which is **statement** coverage, showcased by the listings provided above. Additionally, there are other coverage criteria of interest to us, such as branch and modified condition/decision.

Unlike byte-code instrumenting coverage tools that may be language-agnostic (since many programming languages sharing the same environment compile identical bytecode [9][10]), source-code instrumenting coverage tools are highly language-specific [11]. Therefore, one might expect a degree of determinism in the reported coverage metrics of a particular application's source code. However, this degree isn't necessarily high for two main reasons: misalignments exist between language-specific statements and coverage tools' interpretation, and there's a lack of comprehensive specification/documentation governing the handling of each statement type. This lack of determinism significantly affects developers' confidence in code coverage tools, a situation that can and should be rectified.

1.1 Problem statement

Discrepancies in the handling of source code by coverage tools undermine confidence in their field-wide use. This is highly impractical since they play an integral part in validating software quality, particularly for safety-critical systems. At Solid Sands, our main focus is on C/C++ library and compiler validation. Still, we have encountered significant challenges due to the inconsistencies produced by code coverage tools during various stages of development. We believe these inconsistencies can be mitigated, leading to notable improvements in both economic efficiency and the pace of development.

1.1.1 Research questions

To tackle these issues, we aim to bridge the gaps between code coverage tools by reducing the frequency of such irregularities. As a result, we have deduced the following research questions in line with our objective:

Research Question 1: What essential requirements must a code coverage tool satisfy to effectively evaluate the completeness of C++ source code testing within safety-critical systems and applications?

Research Question 2: How can we design a test suite that evaluates these key requirements while handling ambiguous C++ coverage scenarios effectively?

Research Question 3: How can we implement a tool to accurately assess the effectiveness of various C++ code coverage tools in accordance with our devised test suite?

1.1.2 Research method

The research aspects of this paper are systematically approached through three primary methods:

Literature Review and Requirement Analysis: Initially, we review the existing literature on code coverage tools, with an emphasis on their utility in testing C++ source code for safety-critical systems. Also, we will analyse these systems' specifications and requirements to recognize which key features the coverage tools must possess to sufficiently validate them. The literature review's findings will promote the elicitation of a robust set of requirements that code coverage tools should satisfy.

Test Suite Design: Secondly, we design a comprehensive test suite, guided by the software requirements developed in the previous method. For this, we will formulate a collection of test units that evaluate these necessary functionalities. Among the written units, we focus on designing tests that address ambiguous coverage scenarios in C++ source code, identified through our literature review and our own observations. The end goal is to ensure that our test suite can effectively assess the reliability and quality of code coverage tools.

Tool Development and Evaluation: Thirdly, we develop a tool using Python to assess the conformance of any C++ code coverage tool capitalizing on object-oriented programming.

This developed tool will be designed to utilise our test suite and report metrics on each coverage tool's performance. An analysis stage follows, where we observe the reported metrics and evaluate each tool's conformity to requirements and coverage of test cases. Finally, we make use of our analysis to assess each coverage tool's reliability and acknowledge potential areas of improvement.

Throughout research, we aim to maintain an iterative approach by revising our three aspects of research to account for feedback and new findings. This will aid our research by keeping it up-to-date. We are motivated to document the methods we employ and the findings we reach thoroughly, to provide later researchers with transparency and the opportunity to replicate and extend our work.

1.2 Contributions

Our research makes the following significant contributions:

1. Proposing a concrete set of requirements supervising the functionality of such tools for various coverage criteria.
2. Developing a test suite that checks for all the aforementioned requirements in addition to ambiguous coverage scenarios.
3. Implementing a tool that can automate the verification of any C++ code coverage tool in accordance with our designed test-suite.

1.3 Outline

In Chapter 2 we describe the background of this thesis. Chapter 3 describes the robust set of requirements elicited for any code coverage tool operating in the safety-critical domain. It is subsequently followed by Chapter 4, where we discuss the design process of a test suite in accordance with the elicited requirements. Afterwards, Chapter 5 demonstrates how we developed a tool that can verify any C++ code coverage tool's compliance with our test suite. Results are shown in Chapter 6 and discussed in Chapter 7. We examine the work related to our thesis in Chapter 8 and, finally, we present our concluding remarks in Chapter 9 together with future work.

Chapter 2

Background

This chapter presents the necessary background information for this thesis. Initially, in section 2.1, we define some basic terminology that will be used throughout the paper. Afterwards, section 2.2 provides an overview of the various code coverage criteria. Section 2.3 showcases the requirements and coverage criteria of particular interest to safety-critical systems and their applications, whereas section 2.4 outlines the necessity of incorporating MC/DC into our focus. Section 2.5 demonstrates the key differences between source-code and byte-code instrumenting coverage tools, and section 2.6 gives insight onto the types of scenarios and statements which coverage tools may render ambiguous. Finally, the distinction between run-time coverage and compile-time coverage is made in section 2.7.

2.1 Terminology

We would like to define the following terms for clarity:
Code Coverage: A percentage metric showing the extent to which a program's source code is executed when a particular test suite runs. For this thesis, the term 'test coverage' is interchangeable with 'code coverage' as they refer to the same metric. Outside our scope, 'test coverage' can be interpreted as a metric representing an overall degree of the testing procedures.
Test unit: A single file (.cpp in our case) that tests a particular functionality or requirement.
Test Suite: A comprehensive collection of test units that span all requirements. Throughout this paper, we use 'test suite' to refer to the unit tests we've developed that we expect adequate coverage tools to pass. Not to be confused with the conventional 'test suite' that a quality engineer or software tester formulates during regular software testing procedures.
Gcov: A free, open-source code coverage analysis tool.
ISO-26262: An international functional safety standard for the development of electrical and electronic systems in automotives.
MC/DC: Modified Condition/Decision Coverage, a particular coverage criterion of great significance to safety-critical systems.

2.2 Coverage Criteria

There are various different criteria by which the degree of coverage can be assessed.

2.2.1 Statement

The most basic criterion would be **statement** coverage, which was conceptually illustrated in Listing 1.2. To achieve statement coverage, the tool traverses the source code line-by-line and reports on the execution status of every statement. 100% coverage of this criterion is achieved if every single statement is covered.

2.2.2 Function

Another common criterion would be **function** coverage, for which the tool parses every function and reports whether it has been called. 100% coverage of this criterion is achieved if every single function is called.

2.2.3 Branch

A third criterion, **branch/decision** coverage, is concerned with the tool checking every control structure (if-else, do-while, switch-case...etc.) to ensure each outcome (true/false) is taken at least once. 100% coverage of this criterion is achieved if every decision has each outcome evaluated during testing. Although this criterion is more robust than statement coverage, it doesn't measure up to the succeeding one: MC/DC [3].

```
1  #include <iostream>
2
3  int square(int x) {
4      int result = x * x;
5      return result;
6  }
7
8  int main() {
9      for(int i = -4; i <= 4; i++)
        {
10     if (i % 2 == 0 && i != 0) {
11     std::cout << i << "squared =
        " << square(i) << "\n";
12         }
13     }
14     return 0;
15 }
```

Listing 2.1: Original Source Code

```
1  #include <iostream>
2
3  static unsigned long long __gcov_counter[8] = {0};
4  // Each basic block gets a coverage counter
5  static unsigned long long __gcov_branch_t[3] = {0};
6  static unsigned long long __gcov_branch_f[3] = {0};
7  // Each boolean expression gets two branching
       counters (true/false)
8  int square(int x) {
9      __gcov_counter[0]++;
10     int result = x * x;
11     __gcov_counter[1]++;
12     return result;
13     __gcov_counter[2]++;
14 }
15
16 int main() {
17     __gcov_counter[3]++;
18     for(int i = -4; i <= 4; i++) {
19         if (i<=4) __gcov_branch_t[0]++;
20         else __gcov_branch_f[0]++;
21         __gcov_counter[4]++;
22         if (i % 2 == 0 && i != 0) {
23             if (i%2==0) __gcov_branch_t[1]++;
24             else __gcov_branch_f[1]++;
25             if (i!=0) __gcov_branch_t[2]++;
26             else __gcov_branch_f[2]++;
27             __gcov_counter[5]++;
28             std::cout << i << "squared = "
              << square(i) << "\n";
29             __gcov_counter[6]++;
30         }
31     }
32     return 0;
33     __gcov_counter[7]++;
34 }
35
36 void __gcov_exit() {}
```

Listing 2.2: Conceptual Instrumented Code (Branch)

By definition, achieving branch coverage implies the achievement of statement coverage, since evaluating every entry point and every statement is sufficient for the latter. In addition to the initial counters implemented by Gcov for statement coverage, there are further global data structures (evident in lines 5 and 6) that are added to track the potential branching taking place in the program. The 'for' loop in line 18 requires two branches to be evaluated (when the loop condition evaluates to true *and* false) whereas the 'if' statement on line 22 requires four branches for its two conditions (the first and second condition *both* being covered for true *and* false). As a result, Gcov declares six branching counters in total.

An important note is that the illustrated instrumentation is conceptual, to aid in comprehending the mechanisms of handling branching. In reality, Gcov instruments at compile-time with the use of command flags in the GCC compiler [12]. The tool and compiler work together to instrument at the source-level, but leave the original source file unaffected. The generated output looks like listing 2.3:

```
        -:    1:#include <iostream>
        -:    2:
        4:    3:int square(int x) {
        4:    4:    int result = x * x;
        4:    5:    return result;
        -:    6:}
        -:    7:
        1:    8:int main() {
       10:    9:    for(int i = -4; i <= 4; i++) {
branch  0 taken 90%
branch  1 taken 10% (fallthrough)
        9:   10:    if (i % 2 == 0 && i != 0) {
branch  0 taken 56% (fallthrough)
branch  1 taken 44%
branch  2 taken 80% (fallthrough)
branch  3 taken 20%
        4:   11:    std::cout << i << "squared = " << square(i) << "\n";
        -:   12:        }
        -:   13:    }
        1:   14:    return 0;
        -:   15:}
```

Listing 2.3: Gcov's Coverage Report

The initial branching taking place following the `for` statement demonstrates the percentage of times the loop was entered. Starting from -4 to 4 inclusive, the loop was entered 9 times. On the following increment, where `i` was equal to 5, the loop condition broke, causing an exit. Thus, the 'true' outcome branch shows a 9/10 or 90% metric. The fallthrough branch, indicative of the 'false' outcome, reports 1/10 or 10% as a result.

The branching metrics following the `if` statement are slightly more complex due to short-circuiting. Short-circuit evaluations occur when compilers ignore the remaining conditionals in a boolean expression as their evaluation is deemed unnecessary to reach the correct outcome [13]. For example, in `if (a == 0 || b == 0 || c == 0)`, if `a == 0` is found to be true, `b` and `c` are not evaluated as the outcome will be true regardless. The same concept applies to `if (a == 0 && b == 0 && c == 0)`, if the first condition is unequal to zero then the rest are ignored and the expression's outcome is deemed false.
Since C++ utilizes short-circuit evaluation for boolean expressions [14], the first boolean decision of the statement's expression (`i % 2 == 0`) will act as an outer branch to the inner branch expression of the second decision (`i != 0`) as seen in listing 2.4.

```
// Original boolean expression
if (i % 2 == 0 && i != 0) {
...
}
// Expression's short-circuit evaluation
if (i % 2 == 0){
    if(i != 0){
    ...
    }
}
```

Listing 2.4: Short-circuit evaluation example

Unlike the earlier `if` statement where the fallthrough branch was 'false', the && operator's short-circuit behaviour causes the outer branch to fallthrough to the inner expression when it is 'true' (had it been false, the inner expression would be ignored as it's unnecessary in outcome evaluation). Consequently, Gcov assigns two branches (true/false) to the outer decision (branch 0 and branch 1) and two branches to the inner decision (branch 2 and branch 3).

From the previous for loop, we know the entire boolean expression is evaluated nine times. The outer decision, `i % 2 == 0`, is true (branch 0) when `i` is equal to -4, -2, 0, 2 and 4. Achieving a total of 5/9 outcomes, or 56%. On the other hand, it is false (branch 1) when `i` is equal to -3, -1, 1 and 3 for a result of 4/9 (44%).
From the 5 fallthrough instances reaching the inner decision (`i` equating to -4, -2, 0, 2 and 4), the expression evaluates to false (branch 3) and falls through to the `cout` statement on line 11 four times for a metric of 4/5 or 80%. Only once (when `i` =0) does the innner decision evaluate to 'true' (branch 4), 1/5 times or 20%.

2.2.4 MC/DC

A particularly demanded criterion in safety-critical systems, **modified condition/decision** coverage is one of the rather robust criteria, as it demands the coverage tool to check for all the following:

1. Every point of entry and exit in the program has been invoked at least once
2. Every condition in a decision in the program has taken all possible outcomes at least once
3. Every decision in the program has taken all possible outcomes at least once
4. Each condition in a decision has been shown to independently affect that decision's outcome

In November 2022, the Gcov community released a patch that added MC/DC [15]. Despite Whalen et al.'s established methodology of extracting abstract-syntax-tree (AST) information and utilizing it to construct the instrumentation [16], Gcov uses an algorithm relying on the control-flow-graph (CFG). Our work primarily utilises Gcov's implementation of MC/DC, with awareness of a minor caveat regarding the dependence on the CFG. The limitation is that both listing 2.5 and listing 2.6 produce the **exact same** CFG:

```
1  if (a && b && c)
2          x = 1;
```

Listing 2.5: and.cpp

```
1 if (a)
2     if (b)
3         if (c)
4             x = 1;
```

Listing 2.6: ifs.cpp

As a result, branch coverage for both snippets would be identically reported, even though ideally there should be differentiation since 'and.cpp' has one branch whereas 'ifs.cpp' has three. Additionally, condition coverage would be impacted as 'and.cpp' has a single decision whereas 'ifs.cpp' has three. This disparity becomes even more apparent when accounting for short-circuit behaviour, as the single decision in 'and.cpp' will mask the remaining conditions once a single condition is found to be false. Additionally, ternary operators (i.e. int x = a ? 0 : 1) occasionally introduce conditionals that are difficult to detect in CFGs, as a result their avoidance is encouraged.

```
1 bool decision(bool a, bool b, bool c) {
2     return (a && b) || c;
3 }
```

Listing 2.7: Function to Test

```
1  void test_decision() {
2      decision(true, true, false)
3      // Expected: 1
4      decision(true, false, false)
5      // Expected: 0
6      decision(false, true, false)
7      // Expected: 0
8      decision(false, true, true)
9      // Expected: 1
10     }
11
12 int main() {
13     test_decision();
14     return 0;
15 }
```

Listing 2.8: Test Campaign

Figure 2.1: Achieving 100% MC/DC coverage for a simple function

For clarity, a boolean decision such as `if ((a || b) && c)`, has three conditions (`a` , `b` , and `c`). The four tests in Listing 2.8's test campaign achieve 100% MC/DC by ensuring every individual condition in the decision can independently affect its outcome.
The first test, `decision(true, true, false)`, returns **true**, showing that the outcome is **true** when both `a` and `b` are **true**. The second test, `decision(true, false, false)`, demonstrates that by setting `b` to **false** while keeping `a` as **true** and `c` as **false**, the function's outcome changes to **false**. This change from the previous test case demonstrates that `b` can independently affect the outcome.
The third test, `decision(false, true, false)`, changes `a` to **false** while keeping `b` as **true** and `c` as **false** (from the first test). The outcome of the function changes to **false**. This shows that the mere change of `a` causes the decision's outcome to change, demonstrating that `a` can independently affect the decision's outcome.
The final test, `decision(false, true, true)`, sets `c` to **true** while keeping `a` and `b` as **false**. The function's outcome now changes to **true**, showing that `c` can independently affect the outcome.
These tests collectively prove that every condition in the decision `(a && b) || c` can independently affect the decision's outcome, which achieves 100% MC/DC coverage.

2.2.5 Multiple Condition

An even more rigorous and exhaustive criterion exists in **multiple condition** coverage, which requires every single combination of conditions inside each decision to be covered during testing.
As outlined by NASA's Hayhurst, "In theory, multiple condition coverage is the most desirable structural coverage measure; but, it is impractical for many cases. For a decision with n inputs, multiple condition coverage requires 2^n tests." [17]

MC/DC aims to provide a practical alternative to multiple condition coverage, by keeping many of its benefits but managing to require non-exponential growth in test cases. This is due, in part, to MC/DC's capability of ensuring each condition independently affects the decision's outcome.
Along with statement and branch coverage, we maintain particular focus on MC/DC for reasons mentioned in section 2.3. There are other criteria such as parameter-value (PVC), linear code sequence and jump (LCSAJ) and data-flow coverage, but they are out of our research's scope.

2.3 Safety-Critical Systems

This section demonstrates the cruciality of MC/DC in safety-critical systems as motivation for its inclusion as a key criterion in our research. There are several key standards that deem MC/DC a necessity and require its inclusion amongst considered code coverage criteria during the development of safety-critical systems, such as ISO 26262 [18], DO-178C [19], EN 50657 [20] and IEC 61508 [21].

2.3.1 Automotive Systems

ISO 26262, titled "Road vehicles – Functional safety", is a framework that outlines a series of standards for electrical/electronic systems installed in road vehicles. The entirety of development activities taking place for such systems must abide by the functional safety objectives laid out by ISO 26262. To classify risks, the ISO provides an automotive-specific risk metric to rank the integrity levels, namely "Automotive Safety Integrity Levels" (ASIL). This metric has four stages in incremental order of risk, labelled 'A' to 'D'. For the highest integrity level designation reserved for high-risk components, MC/DC is highly recommended at the software unit level. As witnessed in the following table where "++" indicates that the method is highly recommended for the identified ASIL.

Methods		ASIL			
		A	**B**	**C**	**D**
1a	**Statement Coverage**	++	++	+	+
1b	**Branch Coverage**	+	++	++	++
1c	**MC/DC (Modified Condition/Decision Coverage)**	+	+	+	++

Table 2.1: Structural coverage criteria at the software unit level for ISO 26262 [18]

2.3.2 Aviation Systems

DO-178C, titled "Software Considerations in Airborne Systems and Equipment Certification", is a process standard that oversees the software lifecycle activities taking place in software systems responsible for airborne applications. Meeting the requirements set forth by the document is a necessity for software-based aerospace systems to be approved by certification authorities such as the FAA [22]. The standard particularly outlines safety considerations that must be respected during the integral process of software verification, for which MC/DC is strictly required. There are five Software Levels that are determined during the safety assessment process, for which MC/DC is mandatory within components in possession of the highest risk level (A - Catastrophic).

Level	Impact	Coverage Criterion
A	Catastrophic	MC/DC, Branch, Statement
B	Hazardous/Severe	Branch, Statement
C	Majorsystems	Statement
D	Minor	-
E	No Effect	-

Table 2.2: Mandatory structural coverage criteria for each Software Level in DO-178C [19]

2.3.3 Railway Systems

EN 50657, titled "Railway Applications - Software Onboard Rolling Stock", is a standard specifying the process and requirements necessary to develop software for programmable electronic systems onboard rail vehicles. It aims to oversee all software components of rolling stock applications, as well as the interactions between such software and the system on which it's built. There are four safety integrity levels (SIL) for software components, ranked in ascending degree of critical risk. As far as test coverage criteria are concerned, MC/DC is highly recommended (HR) for the most critical of integration levels (SIL 4).

Coverage Criterion	SIL 1	SIL 2	SIL 3	SIL 4
Statement	HR	HR	HR	HR
Branch	R	R	HR	HR
MC/DC	R	R	HR	HR

Table 2.3: Structural coverage criteria for each Safety Integration Level in EN 50657 [20]

2.3.4 Programmable-Electronic Systems

IEC 61508, titled "Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems", is a basic functional international standard applicable for all types of safety-critical systems. Through two principles, the standard aims to ensure every safety-related system either works as intended or fails predictably in a safe way. The first principle entails a 'safety life cycle', which is founded on best practices, to acknowledge and mitigate design errors as soon as possible. On the other hand, the second principle promotes a probabilistic failure approach to handle the safety aspect of device failures. The standard assigns a risk assessment metric, labelled 'safety integrity level' (SIL) to different software functions within programmable-electronic systems. Several industry-specific variants of IEC 61508 exist for automotive, rail, power plant, machinery and process industries. Unit testing is integral for software produced in accordance with the standard, and MC/DC is highly-recommended (HR) for the highest integrity level (SIL 4).

Method		SIL 1	SIL 2	SIL 3	SIL 4
7a	**Function Coverage**	HR	HR	HR	HR
7b	**Statement Coverage**	R	HR	HR	HR
7c	**Branch Coverage**	R	R	HR	HR
7d	**MC/DC**	R	R	R	HR

Table 2.4: Structural coverage metrics for each Safety Integration Level in IEC 61508 [21]

2.4 MC/DC Cruciality

As deducible from the requirements set forth by various international standards for safety-critical systems, MC/DC is of great importance, particularly when software components possess a high degree of risk. Satisfying the criterion on its own, by definition, implies satisfying both statement and branch coverage. This is due to the fact that MC/DC has four clauses necessary for its satisfaction as showcased in subsection 2.2.4, with clause (1) satisfying statement coverage and clause (3) satisfying branch coverage. For the scope of this research, we will extensively make use of both statement and MC/DC metrics during our work with code coverage.
Our motivation for strictly abiding by both of the aforementioned criteria stems from the usefulness of statement coverage in assessing fundamental coverage behaviour for statements of interest, and MC/DC's necessity in achieving the robust level of code coverage set forth by safety-critical standards.

2.5 Source-code vs. Byte-code Instrumentation

The instrumentation of code for the purpose of displaying coverage metrics can be performed at two levels, either the source-code or byte-code. We have witnessed the former in depth throughout the proceedings of this research, where global data structures were embedded within the original source-code to keep track of execution rates of coverage criteria throughout the testing phase. This approach is helpful in comprehending the logic behind reported metrics, but augments the source-code size and may introduce new bugs that have a detrimental effect on the original source-code. On the other hand, byte-code instrumentation does not have access to the source-code and, as a result, cannot modify it. The additional structures to track metrics now have to be instrumented to the compiled bytecode, a flexible and cost-effective method that requires thorough knowledge of the programming language to comprehend the coverage results clearly.
It has been found, however, that byte-code instrumentation is not a valid technique to measure branch coverage [23].
As a result of our exclusive focus on C++, we are primarily concerned with source-code instrumentation, since C++ usually compiles down to the machine code directly [24]. Byte-code instrumentation is rather viable in Java, as there is a standard Java API for bytecode instrumentation defined in the package 'java.lang.instrument'. The package can be used to instrument Java classes' bytecode before they are loaded by the Java virtual machine [25].

2.6 Ambiguous Scenarios

We refer to statements that do not have a single concrete interpretation regarding their coverage status, as well as statements having the same functionality yet treated differently by a coverage tool, as 'ambiguous'. Preprocessor directives, inline and template functions, macros, exception handling and object-oriented features are among the ambiguous cases thoroughly discussed in Chapter 4. Discrepancies detected in Gcov's handling of identically functioning statements are also discussed, which is alarming due to Gcov being an extremely popular open-source test coverage tool that is widely in use [26]. Our goal is to provide a robust framework that can serve as a guideline whenever such inconsistencies are found, in an attempt to standardize the output of coverage tools moving forward. The programming languages currently most popular among software engineers for writing safety critical applications are C and, more recently, C++ [27]. For safety-critical engineers to determine the effectiveness of their test suite, they must utilize code coverage tools. Ideally, there should be minimal ambiguity in order to facilitate an enhanced development process.

2.7 Compile-time vs. Run-time Coverage

Following the development of code files yet prior to running the program once, running a code coverage tool would constitute *static code analysis* and **compile-time coverage** would be achieved. This process may be helpful in detecting syntax errors, unreachable code segments, unused variables...etc. On the other hand, once a program is run once (ideally many times by a test suite), running a code coverage tool would constitute *dynamic code analysis* and **run-time coverage** would be achieved [28]. It is this type of analysis that's useful for quality engineers or software testers, as they can write more impactful tests to increase coverage.
Throughout Chapter 3, the requirements outlined for various coverage criteria make use of 'runtime' and 'compile-time' to differentiate between the different phases during which coverage requirements must apply. Our developed test-suite utilises run-time coverage, as the test units are constructed by reflecting the coverage metrics of a single parameter-less run (more in Chapter 4).

Chapter 3

Coverage Tool Requirements

A primary objective, prior to designing the test suite, is acknowledging which set of concrete requirements are necessary for any coverage tool's valid functioning. Given the vast array of programming languages and domains within which coverage tools are utilised, our requirements are primarily geared towards C++ and safety-critical applications. As such, we formulated requirements for statement, branch and MC/DC criteria.

As opposed to the conventional method of eliciting software requirements on the foundation of stakeholder demands, we are instead composing requirements based on the demands of international standards of safety-critical systems. To the best of our ability, we have abided by the quality criteria of a good requirement specification for each of the requirements in our set. These quality criteria include correctness, completeness, unambiguity, consistency and traceability [29]. The purpose of outlining such robust requirements is to provide a guideline for code coverage tools to adhere to the expectations laid out by international standards, particularly ISO 26262 and those discussed in section 2.3.

In accordance with IEEE 830-1998 [30], each requirement was deemed 'correct' by having directly reflect a concrete expectation stated by the standards. In a similar sense, each requirement was 'complete' by including all the expectation's non-trivial aspects, 'unambiguous' by being as atomic as possible, 'consistent' by possessing no conflicts with other requirements in the same set and 'traceable' in the sense that there is a direct mapping between each requirement and the designated expectation.

The formulated requirements form the backbone of our implemented test suite, as every outlined requirement has its own directory within the test suite (rightmost column in table 3.1). For each directory, numerous test units are expected to assess the conformity of coverage tools to the examined requirement.

3.1 Statement Coverage Requirements

ID	Requirement	Definition	Test Suite Path
Statement - 1	The tool shall reliably display coverage for executable source-code statements that underwent testing	For every source-code statement that is intended to undergo testing and indeed underwent testing, the tool shall communicate that this statement is tested at runtime and display the correct amount of times it was tested	/test_suite/statement/ covered/unitX.cpp
Statement - 2	The tool shall reliably display coverage for executable source-code statements that did not undergo testing	For every source code statement that is intended to undergo testing but wasn't tested, the tool shall communicate that this statement still needs testing at runtime	/test_suite/statement/ not_covered/unitX.cpp
Statement - 3	The tool shall reliably display coverage for non-executable source-code statements	For every non-executable (i.e. can never undergo testing) source code statement, the tool shall unambiguously declare that the statement does not require testing	/test_suite/statement/ not_executable/unitX.cpp
Statement - 4	The tool shall reliably display coverage of compile-time evaluated code that is present at the source-level but not in the binary	For every source-code statement intended for evaluation at compile-time and was indeed evaluated at compile time, the tool shall unambiguously declare the statement does not need compile-time testing	/test_suite/statement/ compile_time_eval/unitX.cpp
Statement - 5	The tool shall reliably display coverage of compile-time unevaluated code that is present at the source-level but not in the binary	For every source-code statement intended for evaluation at compile-time but was not evaluated at compile-time, the tool shall unambiguously declare the statement still requires compile-time testing	/test_suite/statement/ compile_time_noneval/ unitX.cpp

Table 3.1: Coverage Tool Requirements (Statement Coverage)

Statement - 1 ensures source-code statements that should be executed during the testing phase, and were **indeed executed**, have their coverage status reliably displayed by the tool. As for the exact same statements that were **not executed** during the testing phase, the tool should report that they still require testing as per **Statement - 2**. For statements that are **non-executable** by nature (comments, declarations, definitions...etc.), the tool should report that there is no testing required for them in line with **Statement - 3**. The final two requirements concern source-code statements that are not present in the binary code produced post-compilation, these are evaluated during compilation by the compiler before being run. **Statement - 4** ensures that such statements, when successfully **evaluated** at **compile-time** and prior to any runs taking place, are reported as 'not in need of compile-time testing' by the tool. On the other hand, **Statement 5** covers the case where such statements are **not evaluated** at **compile-time** (e.g. needing values unknown at compile-time, using non-literal types, stalled by compiler limitations...etc.), ensuring the tool reflects they still require compile-time testing.

3.2 Branch Coverage Requirements

ID	Requirement	Definition	Test Suite Path
Branch - 1	The tool shall evaluate whether statements containing boolean decisions are evaluated to 'true'	For every statement containing a boolean decision, the tool shall state if the decision outcome was evaluated to 'true'	/test_suite/branch/ true/unitX.cpp
Branch - 2	The tool shall evaluate whether statements containing boolean decisions are evaluated to 'false'	For every statement containing a boolean decision, the tool shall state if the decision outcome was evaluated to 'false'	/test_suite/branch/ false/unitX.cpp

Table 3.2: Coverage Tool Requirements (Branch Coverage)

In an attempt to establish branch coverage, the tool should traverse each flow control structure and determine the extent of sufficient testing that took place, demonstrated by the evaluation of each structure to both 'true' and 'false'.

```
1 void foo(int x, int y) {
2     if (x > y) {
3         std::cout << "x > y" << std::endl;
4     } else if (x < y) {
5         std::cout << "y > x" << std::endl;
6     } else {
7         std::cout << "x == y" << std::endl;
8     }
9 }
```

Listing 3.1: Simple C++ function

```
1 int main() {
2     foo(2, 1);  // Testing when x > y
3     foo(1, 2);  // Testing when x < y
4     foo(1, 1);  // Testing when x == y
5     return 0;
6 }
```

Listing 3.2: Test Campaign

Figure 3.1: Achievement of 100% branch coverage for a simple C++ function.

The branch coverage requirements do not take into account the evaluation of every sub-condition in a complex decision statement. Instead, these duties are delegated to a decision coverage metric generally, and to MC/DC in our case specifically.

3.3 MC/DC Coverage Requirements

ID	Requirement	Definition	Test Suite Path
MC/DC - 1	The tool shall correctly display how many conditions are in a boolean decision	For every statement containing a boolean decision, the tool shall correctly state how many conditions are within the decision	/test_suite/mcdc/ cond_count/unitX.cpp
MC/DC - 2	The tool shall correctly display if each condition in a decision was evaluated to boolean 'true'	For every statement containing a boolean decision, the tool shall state if every condition within the decision was evaluated to 'true'	/test_suite/mcdc/ cond_true/unitX.cpp
MC/DC - 3	The tool shall correctly display if each condition in a decision was evaluated to boolean 'false'	For every statement containing a boolean decision, the tool shall state if every condition within the decision was evaluated to 'false'	/test_suite/mcdc/ cond_false/unitX.cpp
MC/DC - 4	The tool shall correctly display if each condition in a decision was capable of independently affecting the decision outcome	For every statement containing a boolean decision, the tool shall state if every condition within the decision was independently capable of affecting the decision's outcome by altering its state whilst the rest of the conditions maintain their state	/test_suite/mcdc/ cond_indep/unitX.cpp

Table 3.3: Coverage Tool Requirements (MC/DC Coverage)

The entirety of requirements for MC/DC operate on source-code boolean decisions, for which the number of conditions should be reported, in addition to each condition's evaluation to 'true' *and* 'false', and whether it is capable of independently affecting the outcome.

Within ISO 26262-8:2018 - 11.1, the objectives of employing confidence in the use of software tools is to [18]:

- provide criteria to determine the required level of confidence in a software tool when applicable
- provide means for the qualification of the software tool when applicable, in order to create evidence that the software tool is suitable to be used to support the activities or tasks required by the ISO 26262 series of standards (i.e. the user can rely on the correct functioning of a software tool for those activities or tasks required by the ISO 26262 series of standards).

In pursuit of the initial objective, and in line with our findings in 2.3, we are confident in the provided set of requirements showcased throughout Chapter 3 to effectively evaluate the completeness of C++ source code testing within safety-critical systems and applications. After implementing the test suite modelled after the formulated requirements, we fulfil our final objective by creating a Python tool that acts as a qualification kit.

Chapter 4

Test Suite Design

Throughout this chapter, we discuss how we implement a test suite based on the requirements outlined in Chapter 3.

4.1 Coverage Files

In order for any code coverage tool to have its capabilities assessed, it must reliably produce correct output that conforms with test units. A 'test unit' is simply a .cpp file that has been modified to showcase its *expected* coverage information. Once we determine whether a coverage tool produces the exact same 'test unit', we can assess its quality. To facilitate such comparisons, we require a standard format to display the coverage metrics of interest, a shortcut method to facilitate unit production and a uniform pattern of program execution prior to deducing the coverage metrics.

4.1.1 Standard Format

Firstly, we require a standard structure for relaying coverage information of C++ files, to maintain consistency. We determine an insightful, readable and modifiable format is creating a copy of the original .cpp file, with the coverage information displayed as a comment for each line on a right-hand-aligned segment. The samples shown mainly showcase statement and MC/DC coverage for conciseness and simplicity, omitting branch coverage from display since it is a subset of MC/DC. Any of the code coverage criteria presented in section 2.2 can easily be integrated in the format's comment segment, given the tool-under test is capable of providing it. The adopted format provided emphasises criteria responsible for safety-critical systems.

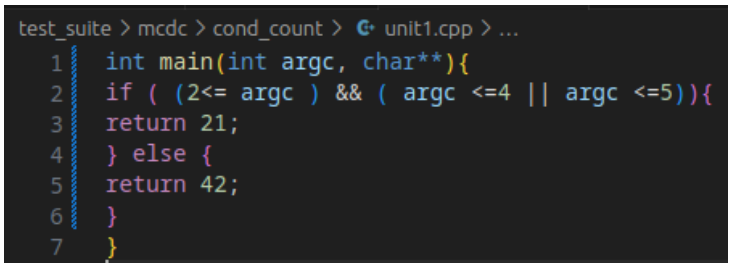


Figure 4.1: Sample .cpp file

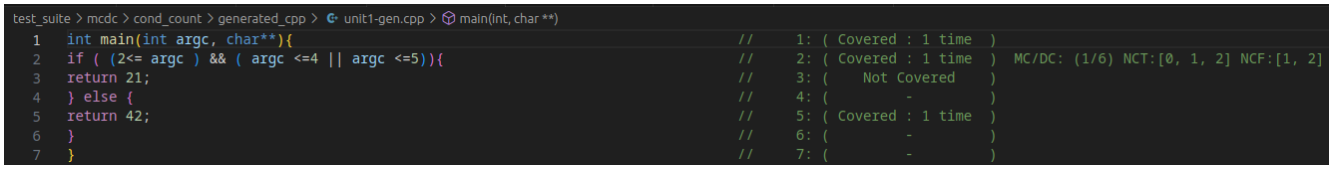


Figure 4.2: Sample file formatted post-coverage

4.1.2 Standard Execution Pattern

There are countless stages at which coverage information can be obtained, such as inbetween compiling and running (*static code analysis* in section 2.7), post-compilation following a single run, following two runs, following three runs...etc. In addition, each run post-compilation can be parameterized or non-parameterized. We illustrate this concept starting with figure 4.3:

```cpp
#include <iostream>

int main(int argc, char *argv[]) {
    if(argc == 1) {
        std::cerr << "Parameterless run" << std::endl;
        return 0;
    }

    int Iterations = std::atoi(argv[1]);

    for (int i = 0; i < Iterations; i++) {
        std::cout << i << std::endl;
    }

    return 0;
}
```

Figure 4.3: Sample .cpp file

Running this file's executable without any parameters (`./unit1`), results in the coverage information shown in figure 4.4.

```
ahmed@ahmed-Legion-5-15ACH6:~/Desktop/solidsands/test_suite/statement/TEST/generated_cpp$ cat unit1-gen.cpp
#include <iostream>                                                    //    1: (         -          )
                                                                       //    2: (         -          )
int main(int argc, char *argv[]) {                                     //    3: ( Covered : 1 time   )
    if(argc == 1) {                                                    //    4: ( Covered : 1 time   )  MC/DC: (1/2) NCT:[] NCF:[0]
        std::cerr << "Parameterless run" << std::endl;                 //    5: ( Covered : 1 time   )
        return 0;                                                      //    6: ( Covered : 1 time   )
    }                                                                  //    7: (         -          )
                                                                       //    8: (         -          )
    int Iterations = std::atoi(argv[1]);                               //    9: (    Not Covered     )
                                                                       //   10: (         -          )
    for (int i = 0; i < Iterations; i++) {                             //   11: (    Not Covered     )  MC/DC: (0/2) NCT:[0] NCF:[0]
        std::cout << i << std::endl;                                   //   12: (    Not Covered     )
    }                                                                  //   13: (         -          )
                                                                       //   14: (         -          )
    return 0;                                                          //   15: (    Not Covered     )
}                                                                      //   16: (         -          )
```

Figure 4.4: Sample after parameterless run

If we then perform a parameterized run (in this case `./unit1 2`), the coverage results are as shown in figure 4.5

```
ahmed@ahmed-Legion-5-15ACH6:~/Desktop/solidsands/test_suite/statement/TEST/generated_cpp$ cat unit1-gen.cpp
#include <iostream>                                                    //    1: (         -          )
                                                                       //    2: (         -          )
int main(int argc, char *argv[]) {                                     //    3: ( Covered : 2 times  )
    if(argc == 1) {                                                    //    4: ( Covered : 2 times  )  MC/DC: (2/2) NCT:[] NCF:[]
        std::cerr << "Parameterless run" << std::endl;                 //    5: ( Covered : 1 time   )
        return 0;                                                      //    6: ( Covered : 1 time   )
    }                                                                  //    7: (         -          )
                                                                       //    8: (         -          )
    int Iterations = std::atoi(argv[1]);                               //    9: ( Covered : 1 time   )
                                                                       //   10: (         -          )
    for (int i = 0; i < Iterations; i++) {                             //   11: ( Covered : 3 times  )  MC/DC: (2/2) NCT:[] NCF:[]
        std::cout << i << std::endl;                                   //   12: ( Covered : 2 times  )
    }                                                                  //   13: (         -          )
                                                                       //   14: (         -          )
    return 0;                                                          //   15: ( Covered : 1 time   )
}                                                                      //   16: (         -          )
```

Figure 4.5: Sample after parameterized run

If we perform yet another parameterized run (`./unit1 4`), we end up with figure 4.6.

```
ahmed@ahmed-Legion-5-15ACH6:~/Desktop/solidsands/test_suite/statement/TEST/generated_cpp$ cat unit1-gen.cpp
#include <iostream>                                                    //    1: (          -          )
                                                                       //    2: (          -          )
int main(int argc, char *argv[]) {                                     //    3: ( Covered : 3 times )
    if(argc == 1) {                                                    //    4: ( Covered : 3 times )  MC/DC: (2/2) NCT:[] NCF:[]
        std::cerr << "Parameterless run" << std::endl;                 //    5: ( Covered : 1 time  )
        return 0;                                                      //    6: ( Covered : 1 time  )
    }                                                                  //    7: (          -          )
                                                                       //    8: (          -          )
    int Iterations = std::atoi(argv[1]);                               //    9: ( Covered : 2 times )
                                                                       //   10: (          -          )
    for (int i = 0; i < Iterations; i++) {                             //   11: ( Covered : 8 times )  MC/DC: (2/2) NCT:[] NCF:[]
        std::cout << i << std::endl;                                   //   12: ( Covered : 6 times )
    }                                                                  //   13: (          -          )
                                                                       //   14: (          -          )
    return 0;                                                          //   15: ( Covered : 2 times )
}                                                                      //   16: (          -          )
```

Figure 4.6: Sample after another parameterized run

As demonstrated, the coverage information displayed for run-time coverage varies according to the file's execution pattern. Consequently, there needs to be consistency amongst the execution patterns of all files' coverage information retrieved from coverage tools, as comparing the coverage information from a parameterless run would differ from a parametrized run and two parameterless runs and so on. As a result, we have maintained a **single parameterless run** for coverage information in all the test units within our test suite.

4.1.3 Coverage File Generation

The purpose of all test units is to display the *expected* coverage information of .cpp files following a single parameterless run. For any .cpp file, if a code coverage tool produces the same output following a single non-parameterized run, it is considered compliant.

Given the desired format in subsection 4.1.1, it would be an incredibly unrealistic task to annotate each source code line manually with its coverage information. To overcome this, we utilise Gcov as a template to initially parse its output into our desired format, and then perform any necessary further changes before integrating the file into the test suite. This allows us to quickly achieve a *generated* file with Gcov's coverage that adheres to the desired format by virtue of a Python code file. Following this, amendments to the file are performed before it is deemed *correct* and joins the suite. To recap, a coverage file is a modified .cpp unit that shows the coverage criteria status commented for each line, and once it is deemed correct we label it a 'test unit'.

4.1.4 Coverage Object Generation

In addition to generating coverage files, we also capitalize on the rapidness and powerfulness of Python's object-oriented programming [31] by creating coverage objects. Each coverage object represents a 'test unit' file, and can be instantiated from JSON, XML or YML. The Coverage class has a 'lines' array, populated with CodeLine objects, which keep track of each file-line's content and coverage status. The CodeLine class defines variables to keep track of the line number along with its coverage status and MC/DC metrics. It's this array, in fact, that undergoes comparison for quicker performance. When assessing the conformity of a code coverage tool's output to our *expected* test unit, if the two files have the exact same array of CodeLines then this signifies they have the exact same coverage status for each source-code line, and as a result are deemed equal.

4.2 Test Suite Structure

The test suite is structured as shown in figure 4.7, with the bottom-level directories populated with numbered test units starting from **unit1.cpp**. The unit tests reflect the necessary requirement they fulfil, indicated by the directory's name. In addition to matching the requirements outlined in Chapter 3, the test units additionally contained cases of ambiguous scenarios (discussed in Chapter 7) A sample from each bottom-level directory is shown along with what's expected of the coverage tool, starting with 4.2.1.

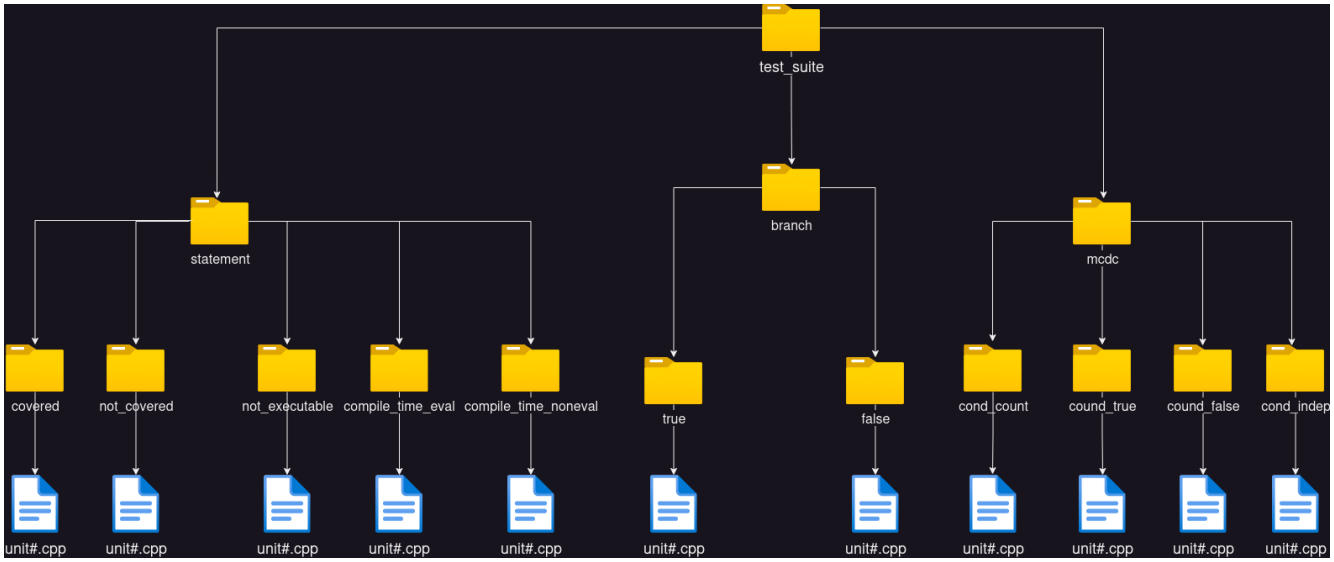


Figure 4.7: Suite Directory Structure

4.2.1 Covered Statements

A source-code statement must be declared 'covered' if it is *executable* **and** *has been tested*. An executable statement is one that performs some action when the source program is run, such as assignment operations, arithmetic operations, control-flow statements, function calls and input/output statements [32]. Once the program's statement is tested during the testing process, the tool must signal it has been 'covered' and display the correct amount of times it was 'covered'.

```
1 int main() {
2     int x = 2;
3     std::cout << "Number is: " << x << std::endl;
4     return 0;
5 }
```

Listing 4.1: Covered

Following a single parameterless run, the tool should display 'covered' once (1) for each statement in listing 4.1 except the closing bracket since it's non-executable.

4.2.2 Not Covered Statements

A source-code statement must be declared 'not covered' if it is *executable* **and** *hasn't been tested*. If an executable statement did not undergo testing following the testing process, the tool must signal that it is an 'uncovered statement' that still requires testing.

```
1 int main() {
2     int x = 2;
3     if (x > 3) {
4         std::cout << "Number is greater than 3." << std::endl;
5     }
6     return 0;
7 }
```

Listing 4.2: Not Covered

In listing 4.2, each line should demonstrate that it's covered once (1) except the line inside the 'if' condition and the closing brackets. Instead, the closing brackets are signalled as non-executable, whereas the `std :: cout << " Number is greater than 3. " << std :: endl ;` line should be marked as 'not covered' (0).

4.2.3 Non-Executable Statements

A source-code statement must be declared 'not executable' if that is indeed the case. In the context of C++, such statements are in the form of preprocessor directives (#include), comments, function prototypes, and declaration statements without initialization.

```
1  #include <iostream>
2
3  // Printing a simple message to screen
4  void printMessage();
5
6  int main() {
7      printMessage();
8      return 0;
9  }
10
11 void printMessage() {
12     std::cout << "Hello, World!" << std::endl;
13 }
```

Listing 4.3: Not Executable

In listing 4.3, the preprocessor directive to include a library header on the first line, comment on line 3, function prototype declaration on line 4, main function declaration on line 6, closing bracket on line 9, function declaration on line 11 and closing bracket on line 13 are all non-executable lines of code. Ideally, all the listing's lines are to be reported as 'non-executable' (-) except for lines 7,8 and 12. Function calls, return and output statements that undergo execution should be reported as 'covered' with the correct amount of times they were covered displayed.

4.2.4 Compile-time Evaluated Statements

A source-code statement that is intended (by the programming language's description) to be evaluated at compile-time, and does indeed undergo compile-time evaluation, should be reported as a 'compile-time evaluated statement'.

```
1  constexpr int getNumber() { return 5; }
2
3  int main() {
4      int a = getNumber();
5      return 0;
6  }
```

Listing 4.4: Compile-time Evaluated

In listing 4.4, the constant-expression function on line 1 along with its call on line 4 are compile-time statements that underwent evaluation, therefore they should be reported as 'compile-time evaluated' alongside their standard coverage status.

4.2.5 Compile-time Unevaluated Statements

A source-code statement that is intended (by the programming language's description) to be evaluated at compile-time, but does not undergo compile-time evaluation, should be additionally reported as a 'compile-time unevaluated statement'.

```
1  #include<iostream>
2
3  constexpr int doubleValue(int val) {
4      return val * 2;
5  }
6
7  int main() {
8      int x = 10;
9      constexpr int y = doubleValue(5);
10
11     int z = doubleValue(x);
12
13     std::cout << "y: " << y << ", z: " << z << std::endl;
14     return 0;
15 }
```

Listing 4.5: Compile-time Unevaluated

In listing 4.5, there are three statements intended for compile-time evaluation. The initial is the constant-expression function declaration on line 3, which does indeed undergo evaluation as the constant-expression function is called later on at line 9. As for the second and third instances, on lines 9 and 11, there's a difference. For `constexpr int y = doubleValue(5);`, the `y` variable is indeed declared with 'constexpr' and will be evaluated at compile-time with a value of 10. As for `int z = doubleValue(x);`, the `x` variable is not a 'constexpr' as witnessed on line 8, and therefore the value of the `z` variable is only evaluated at run-time. As a result, line 11 should ideally be reported as 'compile-time unevaluated'.

4.2.6 Branching

A source-code statement containing a boolean decision, at which point a change in control-flow is to ensue, should have the correct degree of branching it had undertaken displayed.

```
1  int main() {
2      int a = 2;
3      if (a > 0 && a < 10) {
4          std::cout << "Number between 0 and 10" << std::endl;
5      }
6      return 0;
7  }
```

Listing 4.6: Branching

In listing 4.6, the boolean decision `if (a > 0 && a < 10)` is only evaluated to 'true' by virtue of the value assigned to the `a` variable in the source-code. The statement should ideally be reported as having 1/2 branching outcomes covered. If, during the course of the testing phase, a value lower than 1 or higher than 9 was assigned to `a`, then the statement should showcase full (2/2) branching coverage.

4.2.7 Condition Count

A source-code statement containing a boolean decision should have its amount of conditions correctly displayed.

```
1  int main() {
2      int a(3), b(5), c(7);
3      if ((a > 0 && b < 4) or (c < 10) ){
4          std::cout << "Number meets the conditions." << std::endl;
5      }
6      return 0;
7  }
```

Listing 4.7: Condition Count

In listing 4.7, there are three conditions (namely `a`, `b` and `c`) as part of the boolean decision on line 3. Those would ideally be correctly reflected by the code coverage tool. C++ specifies `or` as an alternative spelling for the primary OR operator '||', as defined by ISO 14882:2003 Standard 2.5/2 [33].

4.2.8 Condition Evaluation

The second clause of MC/DC coverage (2.2.4) entails that 'every condition in a decision in the program has taken all possible outcomes at least once'. As a result, each of the conditions correctly identified by the coverage tool in 4.2.7 should have the extent of their outcomes taken correctly reported.

```
1 int main() {
2     int x(2), y(11), z(4);
3     if ((x > 1 && y < 9) or (z < 5)){
4         std::cout << "Number meets the conditions." << std::endl;
5     }
6     return 0;
7 }
```

Listing 4.8: Condition Evaluation

In listing 4.8, each of the three conditions in the boolean expression should be evaluated to both true and false, for a total of six evaluations. As things currently stand, following a single parameterless run where the variables assume the values assigned to them on line 2, `x > 1` has been evaluated to true but not false, `y < 9` has been evaluated to false but not true, and `z < 5` has been evaluated to true but not false. This results in a total of 3/6 condition evaluations expected to be reported by the code coverage tool.

4.2.9 Condition Independence (MC/DC)

The fourth clause of MC/DC coverage (2.2.4) entails that 'each condition in a decision has been shown to independently affect that decision's outcome'. For this, the tool should ideally be capable of detailing whether each condition has been demonstrated to independently affect a decision's outcome throughout the course of testing.

```
1 bool inRange(int a, int x, int b) {
2     return (a <= x && x <= b);
3 }
```

Listing 4.9: Condition Independence

In listing 4.9, and following a single parameterless run, the coverage tool shall ideally signal the independence of each condition (`a`, `x` and `b`) as unmet. There are at least four test cases required for full independence of each condition to be guaranteed. One possible set of such four cases would be the following:

1. `inRange(5, 7, 10)` (decision: **true**)
2. `inRange(8, 7, 10)` (decision: **false**)
3. `inRange(5, 11, 10)` (decision: **false**)
4. `inRange(5, 7, 6)` (decision: **false**)

Changing the value of condition `a` while keeping other conditions the same between tests 1 and 2 has changed the outcome from **true** to **false**, demonstrating the independence of `a`. Similarly, the change of only `x` between tests 1 and 3 being capable of altering the outcome guarantees its independence. The same applies to tests 1 and 4 for `b`, where the decision's outcome depends entirely on it. Had these four tests been a part of the test campaign, and the coverage tool was subsequently run, it would ideally report the MC/DC independence of 3/3 conditions.

It should be noted that a code coverage tool is primarily diagnostic in nature. Its main function is not to independently strive for 100% MC/DC (or any other criterion) coverage, but to accurately report the current state of coverage achieved by existing tests. The task of increasing coverage rests with the quality engineer, who ideally uses the coverage tool as a reliable resource to assess and enhance the state of test coverage.

Chapter 5

Verification Tool Development

5.1 Object-Oriented Approach

Following the declaration of requirements and the development of a test suite to cover them, it's desired to develop a tool that is capable of verifying the conformity of various code coverage tools to the established findings. In accordance with the standard format showcased in 4.1.1, we were able to generate coverage *files* (4.1.3) and *objects*(4.1.4) from .cpp files. A Coverage object is an instance of the Coverage class, which only maintains an array of CodeLines. Each CodeLine object contains the necessary information pertaining to each source-code line, such as its number, coverage and (if need be) MC/DC status along with lists of conditions not covered for 'true' [nct] and 'false' [ncf]. The CodeLine class has the structure shown in figure 5.1.

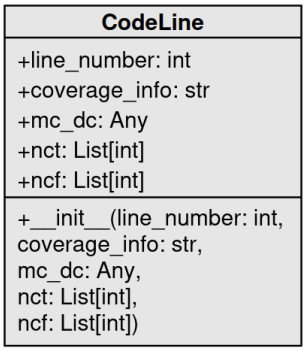


Figure 5.1: CodeLine Class Structure

A single Coverage object, like a Coverage file, is intended to represent the coverage status of an entire .cpp file on a line-by-line basis. The main benefit here from incorporating object-oriented programming is the much faster speed at which comparisons can take place between files, especially in large test suites. Both formats exist to fulfil separate needs, *files* to allow a human-readable method of assessing the coverage information and **objects** to rapidly compare between files.
By using Gcov's output as a template, we generate Coverage **files** with each source-code line annotated with its appropriate coverage metrics. A manual review ensues, where necessary changes (whether due to tool shortcomings or cases of ambiguity) are then made. Once each line is assessed to possess its *expected* coverage status alongside it, the Coverage file is labelled a 'test unit' and is added to the test suite. As a result, each bottom-level directory within the structure shown in 4.7 essentially possesses the original .cpp file along with our generated test unit.
Once the test suite is populated with units to a satisfiable extent, all the test units are converted to Coverage objects by virtue of a Python3 method, `Coverage.create_Coverage_object_from_Coverage_file(test_unit)`. This enables us to obtain a collection of .cpp files and their expected coverage information in the form of objects, in order to be compared with generated objects from the tool-under-test's output. To achieve this, we provide the same .cpp files to the tool-under-test, and convert its output into Coverage objects. To that extent, we have developed `create_Coverage_object_from_tut_output(format, cpp_file)`, which takes in the tool-under-test's coverage output and returns a Coverage object.

The formats supported include JSON, XML and YML; They do, however, require slight modifications to fit each tool's unique description of parameters (i.e. 'line_no' vs. 'line_number', 'condition_cov' vs. 'mc/dc'...etc.)
Ideally, this would be automated (discussed later in Chapter 9).

5.2 Test Driver Development

We finally develop a test driver module, which is run once we have two sets of Coverage objects taken from the two modules shown above. The test driver compares Coverage objects of our test units to the respective Coverage objects of the tool-under-test's output for each .cpp file, and generates a report displaying which objects matched and which mismatched. For mismatching objects, the exact mismatching line number(s) is showcased along with its misaligned coverage information between both objects.

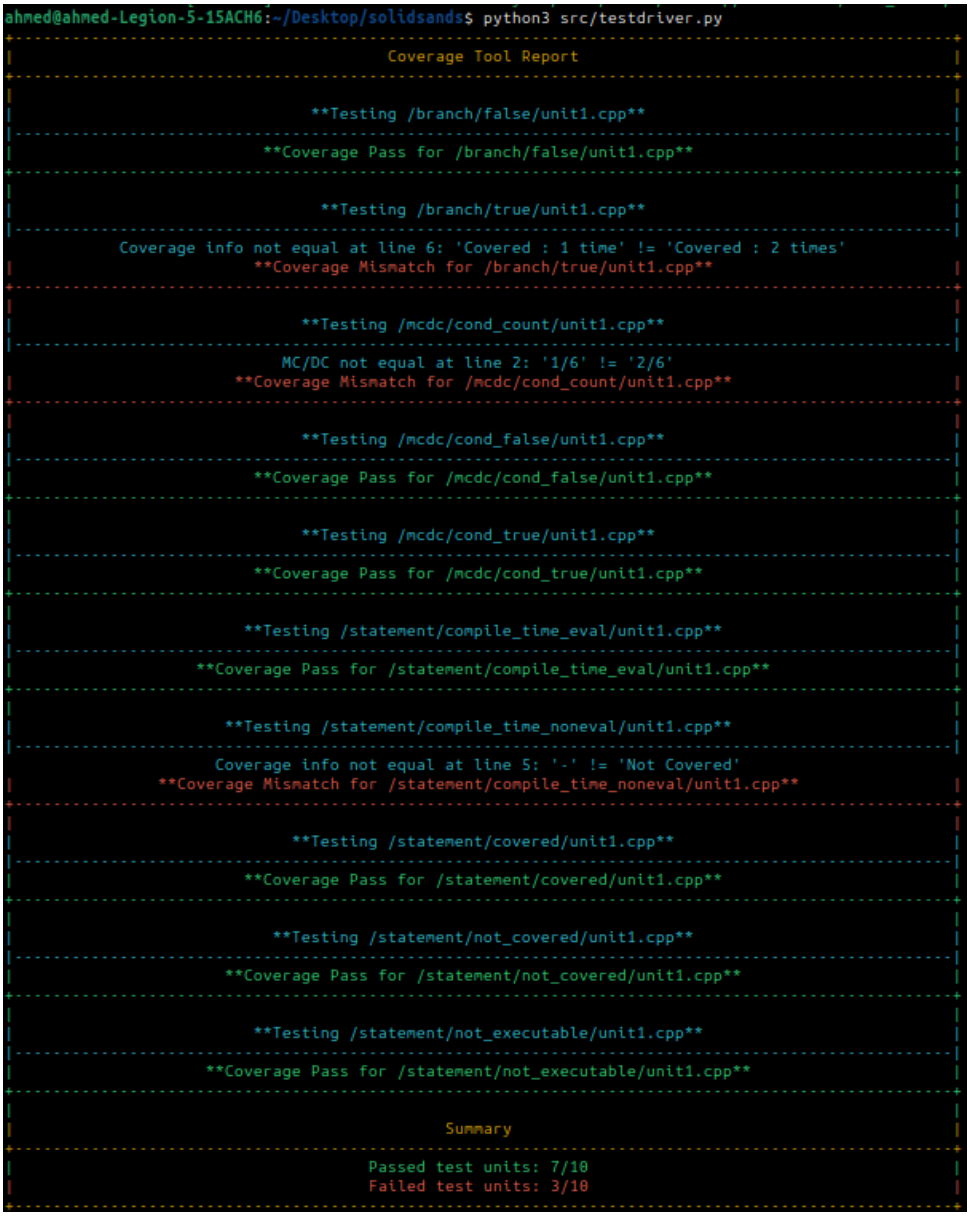


Figure 5.2: Test Driver Report Sample

5.3 Regular Expression Utility

Figure 5.2 showcases a report sample by the test driver module, where there are three Coverage objects found to possess coverage information mismatches in varying categories. We have made extensive use of **regular expressions** to aid us in the detection and assessment of coverage information when creating the Coverage objects, as they are tasked with extracting the values from the .cpp files.
To illustrate, here is a test unit file line that does not possess any boolean decisions, and thus is exempt from MC/DC metrics:

```
std::cerr << "Parameterless run" << std::endl;                    //     5: ( Covered : 1 time  )
```

To fully encapsulate all relevant information in this line, we need a regular expression that can allocate and store the values for the line number and its coverage information.
To achieve this, we construct the following regular expression:

```
line_no_mcdc = re.search(r'.*//\s*(\d+)\s*:\s*\(([^)]+)\).*', line)
```

Through Python3's **re** standard library module for regular expressions, we can capture the values of certain parts inside a regular expression by using brackets **( )**, to store as variables and use later on. The expression works to capture the information on a non-boolean-decision line as follows:

1. `.*//` : '.*' matches any character 0 or more times, immediately followed by the literal '//' that signals a comment in C++.
2. `\s*` : This part matches any whitespace character 0 or more times.
3. `(\d+)\s*:\s*` : Capture the value of one or more digits, signalling the line number. Then, allow for any whitespaces before and after the literal colon ':' that follows the line number.
4. `\(([^)]+)\)` : '\(' and '\)' match the literal parentheses characters '(' and ')'. The '([^)]+)' part allows for the capturing of all coverage information prior to the closing bracket.
5. `.*` : Match any character that may follow the closing bracket. It's used as a safeguard to avoid unnecessary expression mismatches due to accidental additions.

To illustrate, these are the targeted parts on a sample line, following the enumeration above:

For source-code lines containing a boolean decision, additional metrics to keep track of MC/DC are required. These lines are annotated in the following form:

```
if ( (2<= argc ) && ( argc <=4 || argc <=5)){        //     2: ( Covered : 1 time  )  MC/DC: (1/6) NCT:[0, 1, 2] NCF:[1, 2]
```

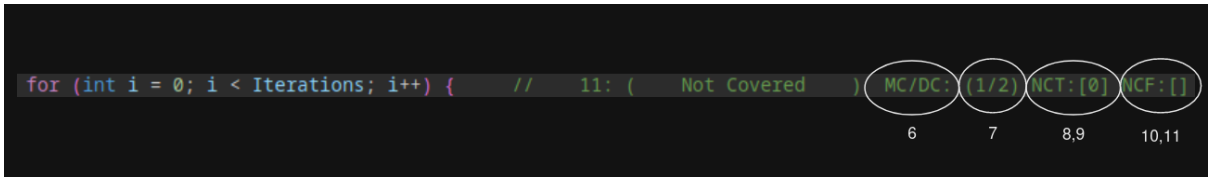
The value of MC/DC in the sample line above, **(1/6)**, represents the number of conditions tested for all possible outcomes. As the state of testing stands, the statement's boolean decision's three conditions have taken one out of six possible outcomes. The **NCT** list then demonstrates which conditions have not been covered for the 'true' outcome, starting at 0. Thus, we can deduce all three conditions have not been covered for 'true'. From the **NCF** list, we deduce that only the first condition has been covered for the 'false' outcome.
To account for the additional metrics, we construct the following regular expression:

```
line_mcdc = re.search(r'.*//\s*(\d+)\s*:\s*\(([^)]+)\)\s*MC/DC:\s*\(([^)]+)\)\s*NCT:\s*\[([^]]*)\]\s*NCF:\s*\[([^]]*)\].*',line)
```

The regular expression is the exact same as the one utilised for source-code lines without a boolean decision, with the addition of the following new segments:

6. `\s*MC/DC:\s*` : Match any whitespace prior and following the literal 'MC/DC:'.
7. `\(([^)]+)\)` : Capture all MC/DC condition information prior to the closing bracket.
8. `\s*NCT:\s*` : Match any whitespace prior and following the literal 'NCT:', which is a number list representing the conditions not covered for the 'true' outcome.
9. `\[([^]]*)\]` : '\[' and '\]' match the literal square bracket characters '[' and ']'. This captures the 'true' numbered list.
10. `\s*NCF:\s*` : Match any whitespace prior and following the literal 'NCF:', which is a number list representing the conditions not covered for the 'false' outcome.
11. `\[([^]]*)\]` : '\[' and '\]' match the literal square bracket characters '[' and ']'. This captures the 'false' numbered list.
12. `.*` : Match any character that may follow, used as a safeguard to avoid unnecessary expression mismatches due to accidental additions.

To illustrate, these are the targeted parts on a sample line, following the enumeration above:



By virtue of both these regular expressions, we ensure that the appropriate coverage information is captured and utilised during the construction of Coverage objects by the methods seen in section 5.1.

Chapter 6

Results

Throughout this chapter, we demonstrate the outcomes achieved as a result of our efforts.
The main objectives completed are in the form of eliciting concrete requirements for coverage tools (Chapter 3), designing a test suite based upon them (Chapter 4), and finally developing a tool to verify code coverage tools' conformity to the test suite (Chapter 5).
Following the achievement of the three aforementioned components, we possess a standardized framework with which ambiguous scenarios in the realm of code coverage can be tackled much more efficiently.
In the scope of C++, we deem the established framework particularly useful in the coverage scenarios of default constructors, macros, inline functions, non-deterministic code, templates, exception-handling, optimizations, multithreading, and dead code.
Throughout this chapter, we demonstrate how our efforts resulted in standardized coverage of code containing the aforementioned vague programming constructs, which we have determined much more efficiently due to the framework in place.
By virtue of suggesting a systematic coverage approach whenever such coding elements arise in accordance with our developed framework, we actively bridge the gap between various coverage tools' outputs.
The results showcased are revisited and discussed in detail throughout Chapter 7.

6.1 Default Constructors

```
1  #include <iostream>
2
3  class TestClass {
4  public:
5      TestClass();                    // Constructor declaration
6  };
7
8  TestClass::TestClass() = default;   // Constructor definition
9
10 int main() {
11     TestClass testObject;
12     return 0;
13 }
```

Listing 6.1: Default Constructor

In C++, a default constructor is one that can be called without any arguments [34]. Its main purpose is guaranteeing the automatic initialization of all appropriate parameters to their default values [35]. The declaration of a default constructor's existence to the compiler is evident on line 4 in listing 6.1, whereas its definition lies on line 8 outside the scope of the class. Despite the constructor being called by virtue of class instantiation taking place on line 11, Gcov peculiarly only reports the definition's statement as covered and not its declaration. The tester should be aware of the amount of times a class is instantiated throughout the testing phase, which would ideally be reflected both on constructors' definitions **and** declarations. Had there been three objects instantiated from the class, all coverage tools need to demonstrate a coverage count of 3 for **both** the class definition and declaration to provide clarity.

6.2 Macros

```
1 #define DIV(a, b) ((a) / (b))
2 int main() {
3     int sum = DIV(6, 3);
4 }
```

Listing 6.2: Macro

A macro in C++ is a section of code that has its body placed by the compiler wherever its name is called during compilation [36]. In listing 6.2, the definition of DIV on line 1 replaces the call on line 3 during compilation. Thus, the actual code undergoing execution is the compiler's expanded macro, not the definition on line 1. It's clear that the coverage metric of the macro call on line 3 should be reported, but the macro definition's coverage is not entirely clear as it doesn't show up in compiled code. As a result, coverage tools would ignore reporting its coverage (how many times the macro has replaced the calls for it, by the compiler). Ideally, the 4th and 5th requirements for Statement Coverage (3.1) should be fulfilled here, depending on whether the macro has been called throughout the source code. If it hasn't been, then the tool shall unambiguously declare the statement still requires compile-time testing; Whereas if it has been called at least once, then the tool shall unambiguously declare the statement does not need further compile-time testing.

6.3 Inline Functions

```
1 inline int mult(int a, int b) {
2     return a * b;
3 }
4 int main() {
5     int result = mult(2, 3);
6 }
```

Listing 6.3: Inline Function

C++ allows programmers to define functions as 'inline', allowing the code of the function to be expanded at the point where it is called, which eliminates procedure call overhead. Inline functions are capable of resulting in both a reduction of code size and an increase in execution speed [37]. In listing 6.3, the compiler will insert the function's body onto the caller on line 5, which makes it unclear where coverage should be attributed (the caller or the inline function). Coverage tools may report coverage for the caller on line 5 but not the inlined function body's statements (only line 2 in this case), because they were not technically invoked on a statement-by-statement basis. However, ideally, coverage tools should report both the caller and the inlined function body's statements as covered. Reporting one at the expense of the other may cause confusion regarding the coverage status of the function.

6.4 Non-deterministic Code

```
1  #include <random>
2  #include <iostream>
3  int main() {
4      int x = rand() % 2;
5      if (x) {
6          std::cout << "x is randomly odd" << std::endl;
7          return 0;
8      }
9      return 1;
10 }
```

Listing 6.4: Non-deterministic Code

In listing 6.4, the execution of line 6 is non-deterministic in nature since it's reliant on a randomized value. Abiding by a random probability distribution, the statement will be covered on occasion, and not covered otherwise. Coverage tools should not be burdened with this scenario, as their reporting will remain accurate regardless (the tool operates normally based on the random value produced).

It would, however, be preferred for such non-deterministic cases to be reduced for simplicity. Tests with nondeterministic executions are inherently fragile and should ideally be rewritten [38]. Optionally, coverage tools could be programmed to produce a flag/warning whenever a statement containing a random value generator is detected, to warn the software tester of non-determinism. Nonetheless, showcasing accurate coverage that matches the random value produced is sufficient in accordance with the requirements expected from coverage tools.

6.5 Templates

```
1 template<typename T>
2 T add(T a, T b) {
3     return a + b;
4 }
5 int main() {
6     int sum = add<int>(2, 3);
7     float fSum = add<float>(2.8, 3.4);
8 }
```

Listing 6.5: Template

For listing 6.5, the utility of a function template is displayed on line 1. Templates are designations for classes and functions to operate with generic types, eliminating the need to designate a certain data type as a parameter. In other words, function templates provide a functional behaviour that can be called for different types [39]. As witnessed above, the template function definition on line 2 is utilized once for the `int` assignment on line 6 and once for the `float` assignment on line 7. These two assignments call two separate, different versions of the template function. Ideally, the coverage tool should treat each distinct instantiation as a separate entity and report its coverage in accordance with whether or not it's been tested. The template blueprint statement and body should ideally reflect the amount of times the template has been instantiated correctly, in this case twice. Glancing over the template coverage status should provide the software tester with the necessary information to determine how many generic-type instances took place.

6.6 Exception Handling

```
1  #include <iostream>
2  #include <stdexcept>
3
4  int subtract(int a, int b) {
5      if (b > a) {
6          throw std::runtime_error("Negative value results unallowed.");
7      }
8      return a - b;
9  }
10
11 int main() {
12     try {
13         std::cout << subtract(3, 5) << std::endl;
14     } catch (const std::exception& err) {
15         std::cout << err.what() << std::endl;
16     }
17     return 0;
18 }
```

Listing 6.6: Exception Handling

An exception in C++ can be described as a runtime anomaly, it typically arises during program execution. The utility of exceptions is helpful when tackling abnormal behaviour such as division by zero. Exception-handling occurs by virtue of three keywords: *try*, *catch* and *throw*. The *try* keyword signals a code segment for which exceptions are expected, whereas the *catch* keyword captures a specific exception type and handles it, and finally the *throw* keyword is used to raise an error whenever an exception is known to take place [40]. In listing 6.6, an attempt to reach a negative value following subtraction is made, for which an exception is caught on line 15 due to the *throw* keyword present on line 6. It is

important to note that exception-handling causes different execution pathways to exist, depending on the *try* block's raised exceptions. Following a single parameterless run, the return statement on line 8 will remain unexecuted, which is comprehensible since an exception was thrown. Had the value of **a** been higher than **b** on line 13's function call during the testing phase, then 100% statement coverage would've been achieved. This is within the bounds of what is expected from code coverage tools, as the return statement should not showcase a need for further testing given its execution has been bypassed due to an exception. Ideally, all statements above would showcase coverage apart from the return statement, which is unreachable since an exception was thrown on line 6.

6.7 Optimizations

```
1 int main() {
2     int a = 1;
3     a = 2;
4     return 0;
5 }
```

Listing 6.7: Optimization

Compilers are capable of producing optimizations that enhance the execution of programs. GCC, as an example, categorizes three various levels of optimization that the developer can alternate between when compiling [41]. The benefit of compiler optimizations lies in the increased performance achieved by virtue of reducing code size and execution time following changes. Listing 6.7's main function declares a variable, `a`, and assigns a value to it on line 2. This value is immediately overridden by another value on the subsequent line. Depending on the degree of optimization, the initial statement may be optimized (since it's immediately assigned another value without any operations in-between) or the entire `a` variable may be optimized, since it is set but unused. Since the degree of optimization affects code coverage output, it's ideal to normalize the compilation of source code with the exact same level of optimization for the sake of assessing coverage tool performance. In our case, without any optimizations, the coverage tool should ideally report each line as covered apart from the closing bracket on line 5.

6.8 Multi-threading

```
1  #include <thread>
2  #include <iostream>
3
4  bool check;
5
6  void validateCheck() {
7      check = true;
8  }
9
10 void statusCheck() {
11     if (check) {
12         std::cout << "Check passed." << std::endl;
13     }
14 }
15
16 int main() {
17     check = false;
18
19     std::thread tA(validateCheck);
20     std::thread tB(statusCheck);
21
22     tA.join();
23     tB.join();
24
25     return 0;
26 }
```

Listing 6.8: Multi-threading

Threads and their execution operate under the guidance of operating system schedulers. Due to the

inherent non-determinism involved in the scheduling process, there is no strict guarantee which thread emerges victorious in race conditions [42]. For listing 6.8, thread scheduling determines whether or not the print statement on line 12 is executed, since it requires `tA` running prior to `tB` (which we have no guarantee of). Within the outlined requirements for code coverage tools, the software tester should be wary of non-determinism in cases of thread-reliant code files. It is ideally sufficient to showcase coverage for the statements that abide by the thread scheduler's order of execution, as the software tester can directly deduce which thread was executed first by observing the coverage status.

6.9 Dead Code

```
1 int main() {
2     return 0;
3     int a = 2;
4 }
```

Listing 6.9: Dead Code

Source code that is fully guaranteed to be unreachable can be labelled as 'dead code'. The existence of dead code is deemed a flaw that requires elimination, usually coming about by poor development practices [43]. In listing 6.9, the assignment on line 3 is considered 'dead code', for which execution is an impossibility regardless of the rigorousness of tests employed by the software tester. By virtue of never finding the dead code in the compiled binary, coverage tools are ideally expected to abide by the Statement-3 requirement outlined in section 3.1, where, for every non-executable (i.e. can never undergo testing) source code statement, the tool shall unambiguously declare that the statement does not require testing.

Chapter 7

Discussion

In this chapter, we primarily discuss the results of our experimentation showcased in Chapter 6, following the establishment of a standardized framework for any C++ code coverage tool.
In an ideal scenario, coverage tools would offer comprehensive insights into all facets of the code relevant to quality engineers or developers. Nonetheless, due to inherent technical constraints, there's disparity between the programmer's intuitive expectations from a coverage report and the actual output of code coverage tools.
The main motivation for our efforts lies in mitigating such disparity as much as possible, which we have initiated by having a framework in place that allows for rapid assessment of coverage tools' performances when subjected to our requirement-based test-suite.
The developed framework includes various test units for different coverage criteria and requirements, with some of them focusing on particularly ambiguous coverage scenarios in the scope of C++. By imposing the concrete requirements and ambiguous scenarios on any coverage tool, we can assess its degree of conformity in an attempt to bridge any existent gaps.
Since the framework's three components (requirements, test-suite and verification tool) have been discussed in detail throughout their respective chapters, we will mainly focus in this chapter on discussing the ambiguous scenarios resolved as a result of our efforts.
All figures showcased in this chapter represent, from our point of view, the ideal coverage output for the files presented following a single parameter-less and non-optimized run.

7.1 Disparity between Constructor Declaration and Definition

A constructor is a special function that is automatically called by the compiler whenever a class instance (object) is created. Its main objective is initializing the object's memory allocation and the values of its members. Despite explicit calling of the constructor being preferred, it can also be implicitly created by the compiler if the user does not define it [44].
The C++ convention does not strictly require constructors to possess both a declaration and definition. If defined within the class body, then there is no need to declare the constructor anywhere. Both are required, however, when the constructor is intended to be defined outside the class definition. In such cases, a constructor declaration is mandatory within the class' scope. When there is disparity in the reporting of coverage amongst both components, it becomes exceedingly confusing for the software tester to deduce how many object instantiations took place.
In the case of Gcov, it deems showcasing coverage metrics only for the definition, but not the declaration, as sufficient. This introduces two problems: the coverage of the declaration remains empty without providing the user any meaningful information, and the user now has to consult the entire source code to deduce how many class instantiations took place instead of a simple observation of the constructor declaration's execution count. The coverage of both declaration and definition should ideally go hand in hand, completely dependent on the number of objects created from the class.

Finding 1: Coverage tools should ideally avoid unequal coverage treatment of constructor declarations and definitions. For every class instantiation, the coverage count of both declaration and definition is ideally incremented.

```
1  #include <iostream>                                                    //     1: (          -          )
2                                                                        //     2: (          -          )
3  class TestClass {                                                     //     3: (          -          )
4  public:                                                               //     4: (          -          )
5      TestClass();                    // Constructor declaration       //     5: ( Covered : 1 time  )
6  };                                                                    //     6: (          -          )
7                                                                        //     7: (          -          )
8  TestClass::TestClass() = default;   // Constructor definition        //     8: ( Covered : 1 time  )
9                                                                        //     9: (          -          )
10 int main() {                                                          //    10: ( Covered : 1 time  )
11     TestClass testObject;                                             //    11: ( Covered : 1 time  )
12     return 0;                                                         //    12: ( Covered : 1 time  )
13 }                                                                     //    13: (          -          )
```

Figure 7.1: Ideal Coverage of Constructors

7.2 Coverage of Macro Definitions and Calls

When a preprocessor encounters a macro name in source-code, it replaces the name with the definition declared by the user using the #define directive. This process takes place prior to compilation, so when the compiler initiates its task, all necessary macro replacements should have already taken place.
Most C++ programs rely on macros to avoid function call overhead for small and frequently-called operations. Due to the semantics of macros differing vastly from the semantics of functions, the use of macros is largely prone to errors [45]. It's rare to find macros with local variables because they rapidly become illegible [46]. This dilemma has even caused Kumar et al. to create a set of *demacrofication* tools that aid in the modernization and enhancement of C++ programs [47].
Since macros are handled pre-compilation, there needs to be differentiation in coverage when handling the definition and the calls. For each macro call, it is sufficient to report its execution normally like any other statement. As for the macro definition statement, if it never gets used, then the code coverage tool should showcase that it has failed compile-time testing. When the software tester consults the #define statement, he'll ideally deduce whether the macro has indeed been called and observe how many times it's been used.

> **Finding 2:** Coverage of macro definitions should provide information regarding their utility throughout the source-code, whereas their calls can be covered regularly like any other statement.

```
1  #define DIV(a, b) ((a) / (b))                                         //     1: ( CT-Pass : 1 time  )
2  int main() {                                                          //     2: ( Covered : 1 time  )
3      int sum = DIV(6, 3);                                              //     3: ( Covered : 1 time  )
4  }                                                                     //     4: ( Covered : 1 time  )
```

Figure 7.2: Ideal Coverage of Macros

7.3 The Utility of Inline Functions

Inline functions are pre-defined functions that get expanded on the line they're called, maintaining the same legibility and safety of regular functions. This expansion takes place at compile-time by the compiler, as opposed to macros which were governed by a preprocessor, yet they maintain the same code-space and run-time efficiency as them. The main appeal for their utility lies in the elimination of function call overhead, which comes at the cost of additional registers.
A single inline function definition can be implemented multiple times throughout a program, which can easily lead to confusion regarding which statements to mark as covered. Ideally, the inline function's definition along with its statements should reflect their true execution count which signifies the number of times they have been called. As for statements containing inline function calls, they should be treated as any other statement regarding their coverage status. This is opposed to macros, which had their definitions undergo a specific designation that signals their handling prior to compile-time.

Finding 3: Coverage of inline functions' definition should ideally reflect the amount of calls made throughout the program, whereas inline function call statements are to be treated regularly.

```
1  inline int mult(int a, int b) {                    //     1: ( Covered : 1 time  )
2      return a * b;                                  //     2: ( Covered : 1 time  )
3  }                                                  //     3: (          -       )
4  int main() {                                       //     4: ( Covered : 1 time  )
5      int result = mult(2, 3);                       //     5: ( Covered : 1 time  )
6  }                                                  //     6: ( Covered : 1 time  )
```

Figure 7.3: Ideal Coverage of Inline Functions

7.4 Handling of Non-Deterministic Code

Non-deterministic code, which can be defined as code that may produce different outputs despite being given the exact same input, poses a real threat to code coverage consistency. Its effects are so undesirable that Hosek et al. have proposed for the enforcement of deterministic execution during the development process, in an attempt to quantify and optimally get rid of non-deterministic code [48]. In addition to confusion during the coverage process, non-determinism also introduces a lack of reproducibility throughout the debugging process, which complicates it further. If outright replacement with deterministic code is infeasible or detriments the testing integrity, then the non-deterministic segment could be isolated and tested separately.
Coverage tools that operate dependently on randomized or non-deterministic programs are not conflicting any of the requirements expected from them, despite the fact confusion may arise in subsequent runs. The software tester should be wary of any non-determinism, as it signifies an inherent inconsistency is likely during execution. Such inconsistencies become increasingly apparent when multiple runs of the code take place. Ideally, the coverage tool should simply approach non-deterministic segments by accurately reflecting the randomized values generated during program execution.

Finding 4: Non-determinism in code presents a real challenge for consistent code coverage. If deemed irreplaceable, coverage tools should report accurate coverage in accordance with the random values generated.

```
1  #include <random>                                  //     1: (          -          )
2  #include <iostream>                                //     2: (          -          )
3  int main() {                                       //     3: ( Covered : 1 time   )
4      int x = rand() % 2;                            //     4: ( Covered : 1 time   )
5      if (x) {                                       //     5: ( Covered : 1 time   )  MC/DC: (1/2) NCT:[] NCF:[0]
6          std::cout << "x is randomly odd" << std::endl;  //  6: ( Covered : 1 time   )
7          return 0;                                  //     7: ( Covered : 1 time   )
8      }                                              //     8: (          -          )
9      return 1;                                      //     9: (    Not Covered      )
10 }                                                  //    10: (          -          )
```

Figure 7.4: Ideal Coverage of Non-Deterministic Code

7.5 Treatment of Function Templates

Template Metaprogramming is an initiative in C++ to execute algorithms during compilation time [49]. The main appeal of resorting to templates is to gain the luxury of passing data types as parameters to a 'general' function that can perform the same operation deterministically regardless of the data types passed, given they are compatible. The operation of templates poses a real challenge to code coverage tools as, depending on the instantiation, actual code generated for execution might differ from the template definition in the source file.
Coverage tools should ideally showcase the correct execution count for the template's definition, reflecting the number of parametrized function calls made throughout the program.

As for the template function calls themselves, they ought to follow regular procedure for statement coverage. Discrepancies between the number of template function calls and the definition's execution count raises confusion regarding the extent of its utility, which complicates the software tester's task.

> **Finding 5:** Coverage of template definitions should ideally reflect the number of function calls made throughout the program, as any discrepancies will misrepresent the extent of their utility to the software tester.

```
1  template<typename T>                                          //     1: (          -          )
2  T add(T a, T b) {                                             //     2: ( Covered : 2 times )
3      return a + b;                                             //     3: ( Covered : 2 times )
4  }                                                             //     4: (          -          )
5  int main() {                                                  //     5: ( Covered : 1 time  )
6      int sum = add<int>(2, 3);                                 //     6: ( Covered : 1 time  )
7      float fSum = add<float>(2.8, 3.4);                        //     7: ( Covered : 1 time  )
8  }                                                             //     8: ( Covered : 1 time  )
```

Figure 7.5: Ideal Coverage of Templates

7.6 Behaviour of the Exception-Handling Mechanism

During the several phases of software development, source-code is susceptible to exceptional events such as insufficient resources, missing files, or invalid user input. To deal with this, C++ possesses exception-handling, a mechanism by which the handling of such events is feasible. The specially-designated functions involved in the process can throw and exception to be captured by the mechanism in the case of encountering anomalies [50]. By virtue of this, the exceptional situation is identified rapidly by the compiler, and there is no need to write further functions in order to handle the raised exception(s).
Since each and every exception-handler possesses the possibility of dealing with a thrown exception, code coverage tools have to deal with the existence of multiple paths as well as the inherent non-linear control flow. From a requirement-based perspective, it is sufficient for the tool to report on the coverage status accordingly with whether or not an exception is thrown.

> **Finding 6:** When covering the exception-handling mechanism, it is sufficient for coverage tools to report execution in accordance with the control-flow paths dictated by thrown exceptions.

```
1  #include <iostream>                                                    //     1: (          -          )
2  #include <stdexcept>                                                  //     2: (          -          )
3                                                                        //     3: (          -          )
4  int subtract(int a, int b) {                                          //     4: ( Covered : 1 time  )
5      if (b > a) {                                                      //     5: ( Covered : 1 time  )  MC/DC: (1/2) NCT:[] NCF:[0]
6          throw std::runtime_error("Negative value results unallowed."); //     6: ( Covered : 1 time  )
7      }                                                                 //     7: (          -          )
8      return a - b;                                                     //     8: (    Not Covered      )
9  }                                                                     //     9: (          -          )
10                                                                       //    10: (          -          )
11 int main() {                                                          //    11: ( Covered : 1 time  )
12     try {                                                             //    12: (          -          )
13         std::cout << subtract(3, 5) << std::endl;                     //    13: ( Covered : 1 time  )
14     } catch (const std::exception& err) {                             //    14: ( Covered : 1 time  )
15         std::cout << err.what() << std::endl;                         //    15: ( Covered : 1 time  )
16     }                                                                 //    16: ( Covered : 1 time  )
17     return 0;                                                         //    17: ( Covered : 1 time  )
18 }                                                                     //    18: (          -          )
```

Figure 7.6: Ideal Coverage for Exception-Handling

7.7 Accounting for Compiler Optimizations

Different compilers maintain different approaches to the deployment of optimizations, which is an issue that requires careful approach. When source-code undergoes optimizations, several procedures may take place such as *inlining* (where regular functions are transformed into inline ones seen in section 6.3),

strength reduction (taking expensive operations such as multiplication and transforming them to use less expensive ones such as addition) and *constant folding* (where variables with values known at compile-time are directly replaced) [51].
Given GCC allows for seven varying levels (degrees) of optimization [52], there is a real threat to coverage output consistency if a standardized setting isn't enforced. For this, we maintain optimization-less runs of our test suite. By eliminating the existence of optimizations, we can ensure the ideal coverage of source-code is produced by tools-under-test.

> **Finding 7:** Compiler optimizations pose a real threat to consistent coverage output, therefore they should ideally be disabled during the assessment of coverage tools' performance.

```
1  int main() {                                                    //    1: ( Covered : 1 time  )
2      int a = 1;                                                  //    2: ( Covered : 1 time  )
3      a = 2;      // Compiler might optimize out                  //    3: ( Covered : 1 time  )
4      return 0;                                                   //    4: ( Covered : 1 time  )
5  }                                                               //    5: (         -         )
```

Figure 7.7: Ideal Coverage for Optimizations

7.8 Inconsistencies of Multithreading

C++ threads carry an inherent notion of non-determinism with their execution, since they rely on the operating system's scheduler to allocate their necessary processor times. Running the exact same program containing threads multiple times may yield different results, in turn causing differing parts of the code to undergo execution. This lack of predictability makes coverage a daunting task for tools, as their instrumentation runs the risk of synchronization overhead within multithreaded programs.
In similar fashion to the exception-handling mechanism, the coverage tool should not concern itself with the potential inconsistencies of multithreading, since the coverage of source-code dictated by thread execution is expected behaviour. As a result, the software tester should be wary of the non-deterministic nature of threads whilst expecting the tool to reflect accurate metrics following the actual thread execution schedule.

> **Finding 8:** Coverage of threads is particularly challenging due to their unpredictable nature of execution, for which coverage tools are solely responsible to report metrics following the scheduler's decisions.

```
1  #include <thread>                                               //    1: (         -         )
2  #include <iostream>                                             //    2: (         -         )
3                                                                  //    3: (         -         )
4  bool check;                                                     //    4: (         -         )
5                                                                  //    5: (         -         )
6  void validateCheck() {                                          //    6: ( Covered : 1 time  )
7      check = true;                                               //    7: ( Covered : 1 time  )
8  }                                                               //    8: ( Covered : 1 time  )
9                                                                  //    9: (         -         )
10 void statusCheck() {                                            //   10: ( Covered : 1 time  )
11     if (check) {                                                //   11: ( Covered : 1 time  )  MC/DC: (1/2) NCT:[] NCF:[0]
12         std::cout << "Check passed." << std::endl;             //   12: ( Covered : 1 time  )
13     }                                                           //   13: (         -         )
14 }                                                               //   14: ( Covered : 1 time  )
15                                                                 //   15: (         -         )
16 int main() {                                                    //   16: ( Covered : 1 time  )
17     check = false;                                              //   17: ( Covered : 1 time  )
18                                                                 //   18: (         -         )
19     std::thread tA(validateCheck);                              //   19: ( Covered : 1 time  )
20     std::thread tB(statusCheck);                                //   20: ( Covered : 1 time  )
21                                                                 //   21: (         -         )
22     tA.join();                                                  //   22: ( Covered : 1 time  )
23     tB.join();                                                  //   23: ( Covered : 1 time  )
24                                                                 //   24: (         -         )
25     return 0;                                                   //   25: ( Covered : 1 time  )
26 }                                                               //   26: ( Covered : 1 time  )
```

Figure 7.8: Ideal Coverage of Multiple Threads

7.9 Coverage of Dead Code

Software systems are susceptible to potential problems, labelled *bad smells*, one of which is dead code [53]. It's regarded as unnecessary code since it's unreachable and/or unused, and generally regarded as harmful for development since it detracts from source-code comprehension [54]. Since dead code detection is a highly requested feature amongst software professionals [55], compilers commonly optimize out dead code to produce rather efficient executable binaries.
Due to our adoption of a standardized optimization-less approach during the assessment of tools under test, we are not particularly affected by this fact. Within the bounds of the requirements set out, it is sufficient for code coverage tools to accurately report metrics for files containing dead code, as its elimination is not the coverage tool's responsibility. The segments of dead code are unreachable to the coverage tool, and therefore there are no expectations of reported metrics.

> **Finding 9:** Dead code is a *bad smell* that harms the development process, which coverage tools could never report since it does not get executed.

```
1  int main() {                                                    //     1: ( Covered : 1 time  )
2      return 0;                                                   //     2: ( Covered : 1 time  )
3      int a = 2;  // Dead code                                    //     3: (         -         )
4  }                                                               //     4: (         -         )
```

Figure 7.9: Ideal Coverage of Dead Code

7.10 Threats to validity

There are particular decisions that may affect the validity of our efforts, both internally and externally.

7.10.1 Internal Validity

Incomplete Test Suite

There's a huge amount of constructs and functions for all types of applications in C++, which we cannot realistically populate the test suite with. As a result, we have settled for tests that cover key aspects of our chosen coverage criteria.

Bias Towards Ambiguity

Being a key objective of this thesis, the mitigation of ambiguous scenarios took a precedent when it came to designing test units, which potentially might have come at the expense of other noteworthy findings.

Version Dependency

Given support for condition coverage was only recently implemented by the GCC community for Gcov, we were dependent on the particular version for which this patch took place. Changes in versions may lead to deviations in coverage.

7.10.2 External Validity

Fixated Coverage Conditions

In the creation of test units, we utilized C++, compiled with GCC, gauged coverage using Gcov, and maintained a single parameterless unoptimized run during testing. Any changes to these fixated conditions would raise coverage mismatches, which is why their fixation is crucial.

Language Evolution

Many of the ambiguous constructs could be enhanced, replaced and/or modified in subsequent releases of C++, or even new ones may arise.

Chapter 8

Related work

Throughout this chapter, we present established work in literature that is related to our focus of standardizing the performance of C++ code coverage tools. We divide the related work into the following categories: surveying and comparing C++ code coverage tools, hunting for bugs in C/C++ code coverage tools via randomized differential testing, evaluating a C++ code coverage analyzer, and suggesting recommendations for the effective use of code coverage tools.

8.1 *eXVantage* - A Survey of Coverage Based Testing Tools

Yang *et al.* were primarily concerned with studying and comparing 17 code coverage tools, placing primary emphasis on assessing coverage measurement, and secondary emphasis on debugging assistance, automatic test case generation, and test report customization [56]. In addition to the surveying of tools on the market, the researchers have also developed their own in-house tool suite for inclusion in the survey, labelled 'eXVantage', which is capable of coverage testing, debugging and profiling. A key takeaway from the assessment of coverage measurement is the finding that source-code instrumentation, which we have relied on heavily throughout research, has less efficient compilation time but more portability and rather direct results as opposed to runtime instrumentation.
The selection of coverage criteria for inclusion in projects is entirely dependent on the domain of application, where more criteria offered for analysis provides greater merit to coverage tools as their scope of utility expands. Due to the usual infeasibility of targeting 100% code coverage for programs, the industry generally deems 60% to 70% a sufficient result. Since bypassing the 60% mark is a daunting development task, the notion of assisting in the achievement of high code coverage is regarded as one of the most important features provided by robust coverage tools. As opposed to ranking coverage criteria in terms of usefulness, which is avoided due to differing domains prioritizing differing criteria as deemed fit, the authors argue that the focus should be on attaining a code coverage tool that both allows for various coverage criteria *and* ideally employ metrics that would aid testers in recognizing the error-prone parts of the code.
Automation, regarded as a key feature for coverage tools since the task of software testing is inherently resource-consuming [57], presents itself in the form of test generation. Despite the appeal for code coverage-based test generation, it was found that none of the surveyed coverage tools are capable of generating tests for C/C++, with only three tools capable of automatically generating Java test cases. Apart from test generation, a friendly graphical interface is deemed a key feature for comparison since its existence enhances the user experience greatly. The generated coverage report, a key aspect of the graphical interface of every tool, should present coverage metrics neatly and allow for result customizations.
The conducted survey's objective is to study the various criteria considered by practitioners when employing a coverage-based testing tool, with emphasis on Java and C/C++. To conclude their efforts, the authors maintain that every single tool surveyed possesses unique features that render it useful in its respective domain of application. By focusing on the various desired aspects presented, software testers are better equipped to make an informed decision when choosing the ideal coverage tool for their project.

8.2 *C2V* - Hunting for Bugs in Code Coverage Tools via Randomized Differential Testing

Y.Yang *et al.* have noticed little attention in research is dedicated to investigating the reliability of code coverage tools. This is alarming as numerous quality assurance tasks, such as software testing, fuzzing, and debugging, depend primarily on such crucial reliability. As a result, the authors propose an unprecedented testing approach to detect software bugs in two widely used coverage tools - *gcov* and *llvm-cov* [58]. They implement a randomized testing tool, *C2V* (*Code Coverage Validation*), for application onto the two aforementioned tools to detect bugs.
The novel testing approach adopted by the authors is found to be increasingly effective, which is evident in the success witnessed during experimentation. *C2V* can be viewed as a tool-set rather than a tool, as it consists of the following components:

- random program generator
- comparer to identify inconsistencies between coverage reports
- filter to remove test programs triggering same coverage bugs
- test program reducer
- inspector to automatically determine which coverage tools have bugs for bug reporting

Gcov, the primary code coverage tool extensively used throughout our research, witnessed 46 bugs detected by *C2V*, in addition to 37 bugs in *llvm-cov*. The results achieved showcase the unideal state of reliability in two of the most widely-used C/C++ code coverage tools, which calls for further research in determining the feasible methods of enhancing their reliability.
The authors' efforts overlap with ours, but with slight differences in the desired outcome. Whereas they primarily actively seek to detect coverage defects such as incorrect execution counts and erroneous coverage information in coverage tools, we were mainly focused on developing a framework that governs the behavior of code coverage tools. In fact, many of the defects presented by Y.Yang *et al.* are useful for inclusion in our own test suite.

8.3 *nvcc* - Experimental Evaluation of a Test Coverage Analyzer for C/C++

Frakes *et al.* were surprised to witness no experimental evaluation of test-coverage analyzers in literature prior to their study, given their long history of utility in the software testing domain. As a result, they decided to investigate the effectiveness of *nvcc*, a C/C++ coverage analysis tool-set, on professional programmers at AT&T Bell Laboratories [59]. The tool-set includes automatic source-code instrumentation of programs, a printed coverage report showcasing the segments lacking testing, a generated summary of the covered basic blocks, and analysis of the differences between various test runs.
Two experiments were conducted to gauge the effectiveness fully, the first was to explore the tool's efficiency in bug detection, whereas the second was determining the tool's ability to achieve high levels of coverage. For the initial experiment, it was found that *nvcc* users were capable, on average, of allocating 65% of the total bugs in a UNIX system command seeded with 11 runtime bugs and 160 source-code lines (excluding comments). This is as opposed to non-*nvcc* users, who managed to detect 58% of the total bugs on average. Additionally, there was a positive relationship found between bug percentage detected and participant years of experience.
As for the second experiment, it was found that *nvcc* users achieved a remarkably higher level of coverage, which implies that the tool enhances the process of code coverage. The laboratory experiment concludes that the utility of *nvcc* improves testing productivity, results in a higher coverage level, improves source-code bug detection, and is perceived by participants to be a valuable testing tool.

8.4 Enhancing Software Testing by Judicious Use of Code Coverage Information

Berner *et al.*, upon closer inspection of code coverage tools' utility in various projects, asserted that it's beneficial yet not without pitfalls. Their inspection consisted of experimental studies in addition to experience from industry projects [60]. To mitigate such pitfalls, the authors employ a methodology which is thoroughly described, examine how code robustness affects coverage, and determine the potential

benefits of consolidating automated tests. Following their experimentation, a list of key recommendations for software testers' effective utility of code coverage analysis and visualization tools is presented to the reader, of which we deem the following as noteworthy:

- Make expectations clear before introducing the tool. Given a reasonably usable automated test suite, make sure that the tool will mostly influence the developers to (in decreasing order):
  1. write more robust code
  2. find bugs or anomalies in the error handling
  3. work on the consolidation of the test cases
- Do not introduce coverage analysis and visualization tools in projects without a reasonably usable automated test suite.
- Do not expect to find many new bugs in the blue-sky behavior of systems. This is a direct consequence of the preceding takeaway: not to use these tools in projects without a reasonably usable automated test suite.
- Consider the introduction around mid-construction. Ensure introduction is neither too early (tool will not be fully beneficial) nor too late (utility will be confined to bug-detection).
- Keep the feedback cycle between coding, testing and coverage visualization as short as possible.
- Emphasize that the primary goal is not to reach an ultimately high coverage rate, but to exploit coverage visualization for identifying areas of the code that are not covered by tests yet.
- If you decide to prescribe a certain coverage rate or percentage, then prescribe a reasonably high one and make sure that it cannot be reached solely by testing the blue-sky behavior.

It is only through paying close attention to the aforementioned points, that software testers would reap the most benefits from employing code coverage and visualization tools. The authors additionally determine that test automation efforts are usually closely linked to the project's testability, where systems that are not designed with testability in mind often deal with the coverage rate stalling somewhere between 70% and 80%.
This research is primarily concerned with providing a guideline for software testers to utilize code coverage tools in a judicial manner. In an attempt to enhance the code coverage process, the authors attempt to govern the utility of coverage tools as opposed to their behavior, like we did. Both our research and theirs will inevitably benefit software testers to a considerable extent in their daily work.

Chapter 9

Conclusion

We established a standardization framework in an effort to bridge the gap between C++ code coverage tools. We elicited necessary requirements for the utility of such tools in safety-critical domains, designed a test suite that guaranteed the evaluation of such requirements, and implemented a verification tool that automatically assesses whether code coverage tools are compliant to our findings. We found various ambiguous scenarios unique to the C++ language, which we were capable of assessing efficiently using the framework in place.

9.1 RQ1

What essential requirements must a code coverage tool satisfy to effectively evaluate the completeness of C++ source code testing within safety-critical systems and applications?
Statement, Branch and MC/DC coverage criteria are deemed sufficient for use within safety-critical applications. Each of these criteria possesses their own set of requirements that ensures all aspects of its utility are accounted for and tested, which we have elicited in accordance with international standards. The essential requirements include accurate coverage of executable and non-executable source-code, compile-time evaluated and unevaluated code, boolean decisions and their branches, count and evaluation of conditions in such decisions, and the independent ability of conditions to affect the decision outcome.

9.2 RQ2

How can we design a test suite that evaluates these key requirements while handling ambiguous C++ coverage scenarios effectively?
To evaluate the essential requirements as well as ambiguous scenarios, we developed a standard format to relay the coverage of .cpp files and constructed coverage files using it, employed a standard execution pattern across test units, took advantage of object-oriented programming and constructed coverage objects to represent coverage files for efficiency, and structured the test suite in line with the elicited requirements.

9.3 RQ3

How can we implement a tool to accurately assess the effectiveness of various C++ code coverage tools in accordance with our devised test suite?
To gauge the conformance of coverage tools to our findings, we developed a Python-based verification tool that converts various widely-used file formats output by coverage tools to coverage objects. These objects can be quickly compared to those of our designed test suite using a test driver, where regular expressions are employed to ensure the information in the coverage file is captured correctly during the coverage object's construction.

9.4 Future work

In the future, it would be an ideal scenario to successfully productize the tool and begin using it to assess other code coverage tools in the industry.
To increase the framework's use cases, we could expand the test suite to accommodate other coverage criteria that are not tied to safety-critical systems. Given Java is capable of being source-code instrumented, in addition to Python, JavaScript, Ruby, and Python, it's possible to augment our research and explore the ambiguities inherent with these various language specifications.
There are nuances in the ways of which coverage tools' output files name the relevant parameters, e.g. to signify the line number, the responsible variable can be labelled 'line_number', 'line_no', 'line#'...etc. We are currently fixing such nuances by manually renaming the variables of interest to us in the Python modules, which is not ideal. An incredible improvement would be to employ regular expressions that are capable of automating parameter-detection, saving us time and effort.

Acknowledgements

I would like to deeply express my sincere gratitude and appreciation to my academic supervisor, Martin Bor, for the incredible guidance and advice I received throughout the implementation of this thesis.
I am forever indebted to my daily supervisor, Nicola Rossi, for tirelessly assisting me on a daily basis and answering every single question I had. Truly, none of this would have been possible without the unwavering sense of support I felt every day in the office.
I feel immensely grateful for the opportunity to conduct this enticing research, which I attribute to UvA and Solid Sands. It's been a truly wonderful year of hard work, learning, and developing character along with the rest of the MSc Software Engineering class of 2023! Sincerely wishing the best of luck to my colleagues on their journey, as I focus on my own moving forward.
I want to particularly extend a heartfelt thank you to both of my sisters, Mariam and Nayera, for having been truly by my side through thick and thin.

Bibliography

[1] P. Tripathy and K. Naik, *Software testing and quality assurance: theory and practice*. John Wiley & Sons, 2011.

[2] B. Hetzel, *The complete guide to software testing*. QED Information Sciences, Inc., 1988.

[3] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[4] S. Planning, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology*, vol. 1, 2002.

[5] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Hyderabad, India: Association for Computing Machinery, 2014, pp. 72–82, ISBN: 9781450327565. DOI: 10.1145/2568225.2568278. [Online]. Available: https://doi.org/10.1145/2568225.2568278.

[6] P. S. Kochhar, F. Thung, and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*, IEEE, 2015, pp. 560–564.

[7] F. Horváth, T. Gergely, Á. Beszédes, D. Tengeri, G. Balogh, and T. Gyimóthy, "Code coverage differences of java bytecode and source code instrumentation tools," *Software Quality Journal*, vol. 27, pp. 79–123, 2019.

[8] GNU, "Gcov (using the GNU compiler collection (GCC))," [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[9] K. Arnold, J. Gosling, and D. Holmes, *The Java programming language*. Addison Wesley Professional, 2005.

[10] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.*, IEEE, 2004, pp. 75–86.

[11] P. K. Chittimalli and V. Shah, "Gems: A generic model based source code instrumentation framework," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, IEEE, 2012, pp. 909–914.

[12] GNU, "GCC, the GNU Compiler Collection - GNU Project," [Online]. Available: https://gcc.gnu.org/.

[13] J. A. Bergstra and A. Ponse, "Proposition algebra and short-circuit logic," in *Fundamentals of Software Engineering: 4th IPM International Conference, FSEN 2011, Tehran, Iran, April 20-22, 2011, Revised Selected Papers 4*, Springer, 2012, pp. 15–31.

[14] K. Laemmermann, *C++ the design and evolution of c++*, 2012.

[15] J. Kvalsvik, "[PATCH v2] Add condition coverage profiling," Nov. 2011. [Online]. Available: https://gcc.gnu.org/pipermail/gcc-patches/2022-November/605699.html.

[16] M. Whalen, M. Heimdahl, and I. De Silva, "Efficient test coverage measurement for mc/dc," 2013.

[17] K. J. Hayhurst, *A practical tutorial on modified condition/decision coverage*. DIANE Publishing, 2001.

[18] International Organization for Standardization, "ISO 26262:2018 Road Vehicles - Functional Safety," Dec. 2018. [Online]. Available: https://www.iso.org/standard/68383.html.

[19] Federal Aviation Administration, "AC 20-115D - Airborne Software Development Assurance Using RTCA DO-178C," Apr. 2023. [Online]. Available: https://www.faa.gov/regulations_policies/advisory_circulars/index.cfm/go/document.information/documentID/1032046.

[20] European Standards, "BS EN 50657:2017 Railways Applications. Rolling stock applications," [Online]. Available: https://www.en-standard.eu/bs-en-50657-2017-railways-applications-rolling-stock-applications-software-on-board-rolling-stock/.

[21] International Electrotechnical Commission, "IEC 61508:2010 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems (E/E/PES)," [Online]. Available: https://webstore.iec.ch/publication/5515.

[22] Boeing Commercial Airplane Group, "DO-178C Software Considerations in Airborne Systems and Equipment Certification," [Online]. Available: https://www.dcs.gla.ac.uk/~johnson/teaching/safety/reports/schad.html.

[23] N. Li, X. Meng, J. Offutt, and L. Deng, "Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools," ISSRE, 2013.

[24] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Addison-wesley Reading, 2007, vol. 2.

[25] J. Aarniala, "Instrumenting java bytecode," in *Seminar work for the Compilerscourse, Department of Computer Science, University of Helsinki, Finland*, 2005.

[26] Z. Hu Jr, "A software package for generating code coverage reports with gcov," 2021.

[27] J. S. Rogers, "Language choice for safety critical applications," *ACM SIGAda Ada Letters*, vol. 31, no. 3, pp. 81–90, 2011.

[28] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*, 2003, pp. 24–27.

[29] S. Lauesen, *Software requirements: styles and techniques*. Pearson Education, 2002.

[30] "Ieee recommended practice for software requirements specifications," *IEEE Std 830-1998*, pp. 1–40, 1998. DOI: 10.1109/IEEESTD.1998.88286.

[31] D. Phillips, *Python 3 object-oriented programming*. Packt Publishing Ltd, 2015.

[32] W. Savitch, *Problem Solving with C++, 7th Edition*, 7th. USA: Addison-Wesley Publishing Company, 2008, ISBN: 0321531345.

[33] I. ISO, "Iec 14882: 2003 (e)," *Programming Languages–C++, American National Standards Institute, New York*, 2003.

[34] S. Meyers, *Effective C++: 55 specific ways to improve your programs and designs*. Pearson Education, 2005.

[35] R. S. Wiener, "Object-oriented programming in c++—a case study," *ACM Sigplan Notices*, vol. 22, no. 6, pp. 59–68, 1987.

[36] S. Sỳkora, "Writing c/c++ macros: Rules, tricks and hints," *Stan's Library*, vol. 1, 2004.

[37] D. Jordan, "Implementation benefits of c++ language mechanisms," *Communications of the ACM*, vol. 33, no. 9, pp. 61–64, 1990.

[38] P. Marinescu, P. Hosek, and C. Cadar, "Covrig: A framework for the analysis of code, test, and coverage evolution in real software," in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 93–104.

[39] D. Vandevoorde and N. M. Josuttis, *C++ templates: The complete guide, portable documents*. Addison-Wesley Professional, 2002.

[40] C. De Dinechin, "C++ exception handling," *IEEE Concurrency*, vol. 8, no. 4, pp. 72–79, 2000.

[41] M. T. Jones, "Optimization in gcc," *Linux journal*, vol. 2005, no. 131, p. 11, 2005.

[42] H.-J. Boehm and S. V. Adve, "Foundations of the c++ concurrency memory model," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 68–78, 2008.

[43] R. Mahmood and Q. H. Mahmoud, "Evaluation of static analysis tools for finding vulnerabilities in java and c/c++ source code," *arXiv preprint arXiv:1805.09040*, 2018.

[44] Z. Yan and C. In, "On unreasonable design in c++ constructor on unreasonable design in c++ constructor,"

[45] B. Stroustrup, "An overview of c++," in *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*, 1986, pp. 7–18.

[46] D. Herity, "C++ in embedded systems: Myth and reality," *Embedded Systems Programming*, vol. 11, no. 2, pp. 48–71, 1998.

[47] A. Kumar, A. Sutton, and B. Stroustrup, "Rejuvenating c++ programs through demacrofication," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, IEEE, 2012, pp. 98–107.

[48] P. Hosek and C. Cadar, "Safe software updates via multi-version execution," in *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 612–621.

[49] Z. Porkoláb, J. Mihalicza, and Á. Sipos, "Debugging c++ template metaprograms," in *Proceedings of the 5th international conference on Generative programming and component engineering*, 2006, pp. 255–264.

[50] P. Prabhu, N. Maeda, G. Balakrishnan, F. Ivančić, and A. Gupta, "Interprocedural exception analysis for c++," in *European Conference on Object-Oriented Programming*, Springer, 2011, pp. 583–608.

[51] M. Godbolt, "Optimizations in c++ compilers," *Communications of the ACM*, vol. 63, no. 2, pp. 41–49, 2020.

[52] D. Branco and P. R. Henriques, "Impact of gcc optimization levels in energy consumption during c/c++ program execution," in *2015 IEEE 13th International Scientific Conference on Informatics*, IEEE, 2015, pp. 52–56.

[53] S. Romano, "Dead code," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2018, pp. 737–742.

[54] M. Mantyla, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, IEEE, 2003, pp. 381–384.

[55] A. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in *2013 20th working conference on reverse engineering (WCRE)*, IEEE, 2013, pp. 242–251.

[56] Q. Yang, J. J. Li, and D. Weiss, "A survey of coverage based testing tools," in *Proceedings of the 2006 international workshop on Automation of software test*, 2006, pp. 99–103.

[57] M. C. Yang and A. Chao, "Reliability-estimation and stopping-rules for software testing, based on repeated appearances of bugs," *IEEE transactions on Reliability*, vol. 44, no. 2, pp. 315–321, 1995.

[58] Y. Yang *et al.*, "Hunting for bugs in code coverage tools via randomized differential testing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, pp. 488–499.

[59] W. B. Frakes, D. J. Lubinsky, and D. Neal, "Experimental evaluation of a test coverage analyzer for c and c++," *Journal of Systems and Software*, vol. 16, no. 2, pp. 135–139, 1991.

[60] S. Berner, R. Weber, and R. K. Keller, "Enhancing software testing by judicious use of code coverage information," in *29th International Conference on Software Engineering (ICSE'07)*, IEEE, 2007, pp. 612–620.

Appendix A

Non-crucial information


```
{
    "format_version": "1",
    "gcc_version": "13.0.1 20230406",
    "current_working_directory": "/home/ahmed/Desktop/solidsands/test_suite/statement/
    covered",
    "data_file": "/home/ahmed/Desktop/solidsands/test_suite/statement/covered/unit3.cpp"
    ,
    "files": [
        {
            "file": "/home/ahmed/Desktop/solidsands/test_suite/statement/covered/unit3.
    cpp",
            "functions": [
                {
                    "name": "_ZN9TestClassC2Ev",
                    "demangled_name": "TestClass::TestClass()",
                    "start_line": 8,
                    "start_column": 1,
                    "end_line": 8,
                    "end_column": 1,
                    "blocks": 1,
                    "blocks_executed": 1,
                    "execution_count": 1
                },
                {
                    "name": "main",
                    "demangled_name": "main",
                    "start_line": 10,
                    "start_column": 5,
                    "end_line": 13,
                    "end_column": 1,
                    "blocks": 3,
                    "blocks_executed": 3,
                    "execution_count": 1
                }
            ],
            "lines": [
                {
                    "line_number": 8,
                    "function_name": "_ZN9TestClassC2Ev",
                    "count": 1,
                    "unexecuted_block": false,
                    "branches": [],
                    "conditions": []
                },
                {
                    "line_number": 10,
                    "function_name": "main",
                    "count": 1,
                    "unexecuted_block": false,
                    "branches": [],
                    "conditions": []
                },
                {
                    "line_number": 11,
```

```
                    "function_name": "main",
                    "count": 1,
                    "unexecuted_block": false,
                    "branches": [],
                    "conditions": []
                },
                {
                    "line_number": 12,
                    "function_name": "main",
                    "count": 1,
                    "unexecuted_block": false,
                    "branches": [],
                    "conditions": []
                }
            ]
        }
    ]
}
```

Listing A.1: Sample of JSON Coverage Output by Gcov

Listing A.1 showcases a sample of a JSON file output by Gcov, which is one of the accepted formats. The test unit presented is the one used to examine the ambiguous disparity between constructor declaration and definition. As discussed in section 6.1, the class declaration's coverage on line 5 is peculiarly disregarded. The JSON file, like any accepted format, is converted to a coverage file by virtue of a Python3 method and modified for inclusion in the test suite, as seen in section 5.1.