A Software Package for Generating Code Coverage Reports With Gcov

by

Zhenmai Hu

B.A.Sc., Changsha University of Science & Technology, 2013

A Report Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF ENGINEERING

in the Department of Electrical and Computer Engineering

A Software Package for Generating Code Coverage Reports With Gcov

by

Zhenmai Hu

B.A.Sc., Changsha University of Science & Technology, 2013

Supervisory Committee

Dr. Michael D. Adams, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Wu-Sheng Lu, Departmental Member
(Department of Electrical and Computer Engineering)

ABSTRACT

Code coverage is an essential tool often used in software testing. Therefore, a tool that generates well-organized and easy-to-read customized reports containing code coverage information is highly beneficial. In this report, we present the Gcov Report Generator (GRG) software, which includes a library developed for generating code coverage reports in PDF format with Gcov and a supporting application program named `coverage` that uses the library through the command line. This GRG software can work with the GCC C++ compiler version 10 onwards. The documentation of the application programming interface for the GRG library, the command-line interface for using `coverage`, and the usage example of generated PDF reports are presented. The GRG software can be used as a front-end tool to the Gcov program to generate code coverage reports in PDF format with function coverage, statement coverage, and branch coverage information. In addition, program options can be utilized to filter the file and function patterns, select coverage criteria types, specify coverage thresholds, and aggregate function information for templates, constructors, and destructors.

Contents

List of Figures

List of Listings

List of Acronyms

GCC The GNU Compiler Collection

PDF Portable Document Format

HTML HyperText Markup Language

XML Extensible Markup Language

JSON JavaScript Object Notation

GRG Gcov Report Generator

API Application Programming Interface

CFG Control Flow Graph

CLI Command Line Interface

ABI Application Binary Interface

POSIX Portable Operating System Interface

Chapter 1

Introduction

1.1 Software Testing and Code Coverage

Software plays a crucial role in society. Many critical and essential infrastructures rely on software for order and efficient operations, such as hospital medical devices, electrical grid systems, and even police tracing systems. With software constantly changing, it is difficult to guarantee, however, that the software will not contain errors. The errors that cause inappropriate and unexpected results or abnormal behaviours in computer programs or systems are called bugs, which can sometimes be very costly and may cause serious consequences. For example, on June 4, 1996, the European Space Agency's Ariane 5 Flight 501 failed forty seconds after takeoff. The prototype rocket of the flight exploded due to numerical overflow in the onboard guidance software, which caused severe losses [9] [12]. Another example of a failure caused by software bugs is the NASA Mars Climate Orbiter, which, in 1998, approached Mars at the wrong angle when entering the upper atmosphere and disintegrated. The primary cause of this failure was that one part of the software produced inaccurate results by using the wrong units of pound-force rather than Newtons when calculating thrust parameter data [4]. Therefore, finding and correcting bugs is an indispensable part of software development.

Software testing can be undertaken to reduce the occurrence of bugs and effectively minimize them. Different methods, such as software analysis and testing, can be applied to find bugs in a program. Software testing and analysis techniques can be classified into two categories: static analysis and dynamic analysis. Static analysis is a testing method that does not require execution of the software being tested, which usually profiles the software at rest, such as by utilizing code reviews or static analysis tools [31]. Many bugs can be found and fixed by applying static analysis, where static analysis might be performed by code reviews, compilers via diagnostics, or analysis tools. Still, some of the bugs may be hidden before the software runs in the actual operating environment. At this time, another complementary method is needed: dynamic analysis. Dynamic analysis is capable of exposing vulnerabilities and flaws that are too complicated for static analysis alone to reveal. In contrast to static analysis, dynamic analysis is a testing approach that works by executing the code being tested [17]. Various methods can be used in dynamic testing, such as functional, structural, and dynamic testing tools, which will be discussed later. Computing the code coverage for test suites is a standard structural testing method, which can measure the degree to which the source code of a program has been executed and the percentage of the code that has been exercised [28]. Using dynamic analysis, such as code coverage, can help confirm that the part of the code that needs attention has been executed during the test to assist in finding and reducing bugs effectively. For a given program, the more code covered during

testing, the more thorough the test. In other words, compared to lower test coverage, programs with higher test coverage execute more code during the testing process, which indicates that these programs have a lower chance of containing undetected software errors [5] [14]. The combination of static and dynamic analysis methods can help analyze and find potential bugs in the program during software testing.

As an essential technique in dynamic testing, code coverage measurement plays a vital role in improving and maintaining software code quality. Code coverage measurement helps to evaluate the effectiveness of software testing by providing different coverage criteria information, such as function, line, statement, and branch coverage [21]. Practical tools can collect code coverage data and generate reports containing different coverage information. In addition, some of these tools can analyze the data to create more readable coverage reports in various formats to clearly show the untested parts of the code detected along with their corresponding source code. Depending on the language used for coding, multiple options are available for creating code coverage reports. For example, Gcov is a very popular test coverage tool included with the GNU Compiler Collection (GCC), and Gcov helps users determine the effectiveness of their test suites [11] [10]. In addition, some graphical front-end tools can collect and then generate data from Gcov into more readable reports. For example, Lcov can generate colourful HyperText Markup Language (HTML) documents containing the source code annotated with coverage information, and Gcovr can produce different kinds of coverage reports in such formats as HTML, Extensible Markup Language (XML), or JavaScript Object Notation (JSON) [15] [32] [13].

Given the portability and convenience of Portable Document Format (PDF) documents, the author developed the Gcov Report Generator (GRG) software, which can be used as a front-end tool based on Gcov to generate code coverage reports in PDF format. Unlike in the case of some previously existing tools, the GRG software described herein allows users greater flexibility for filtering and selecting the report contents and customizing the format of the generated reports. For example, the GRG software can filter based on file and function patterns, choose to ignore branches resulting from exceptions when calculating the branch coverage, select the coverage thresholds, and aggregate the constructors, destructors, and templates that have the same names in the source code but have different mangled names [26] [29]. Thus, our tool can help users determine poorly tested parts of their code more easily to address these deficiencies to improve code quality.

1.2 Overview and Organization of Report

This report presents the GRG software that the author developed, which consists of a library based on Gcov to generate code coverage reports in PDF format and a front-end application program named `coverage` to use this library through the command line. In addition, this report introduces the essential information needed to use the GRG and includes detailed application programming interface (API) documentation, application program usage information, and code coverage report examples. The remainder of this report is organized as described below.

Chapter 2 introduces the background information necessary to understand the subsequent material presented herein. We first describe software testing techniques, such as static testing and dynamic testing. This introduction is followed by information on control flow graphs (CFGs) and their graphical representations, as well as structural coverage analysis. Next, coverage criteria used in the GRG software are presented in detail, such as function, statement, and branch coverage. After that, we introduce the commands used and files generated by the GCC C++ compiler and Gcov when collecting and generating code coverage statistics. Then, a general description of the Gcov invocation and the schema of the coverage data in JSON format generated by Gcov are given with examples. Lastly, the name mangling is discussed.

Chapter 3 presents the details of the GRG software developed by the author. This chapter begins with a general introduction to the GRG software and its functionalities, followed by instructions on building and installing this software. After that, we introduce the application programming interfaces (APIs) of the GRG library that can be used to generate the code coverage reports in PDF format with examples of how to use these APIs in the library. Then, the command-line interface (CLI) of the application program `coverage` is described with examples.

Chapter 4 concludes this report with some closing remarks. Some suggestions for future work are also made.

As supplemental information, examples of code coverage reports in PDF format generated using the GRG library and the software application `coverage` are provided in Appendices A and B. These examples can help users build a more intuitive understanding of what information is included in the report generated by the GRG software.

Chapter 2

Background

2.1 Overview

This chapter provides the necessary background information for readers to understand the work presented in this report. We start by introducing software testing, especially structural coverage testing, followed by a discussion of the coverage criteria used in structural testing. After that, GCC and Gcov and instructions on using them to collect coverage data and produce reports containing coverage information are discussed. Then, we describe options that Gcov uses for specifying the contents of the coverage report generated and the schema of coverage data in JSON format generated by Gcov. Lastly, the name mangling is introduced.

2.2 Software Testing

Software testing is an activity that tries to ensure that software meets formal requirements and works as expected. Human errors can introduce defects into software, and the consequence of such defects can potentially be catastrophic, depending on the application. Therefore, software testing aims to find software defects and bugs to ensure that the software product meets certain quality standards. Through software testing, software bugs can be caught, and the vulnerabilities of the software product can be removed at an earlier stage before bad consequences result, which is more cost-effective and helps to enhance the software security. Software testing considers many aspects of software behaviour. For example, testing considers if the software being tested [1]:

- responds correctly to all required inputs;
- has acceptable performance consumption towards time and memory;
- is sufficiently usable for applications;
- works appropriately in all intended environments;
- meets the general needs of users; and
- achieves the design and development requirements without side effects.

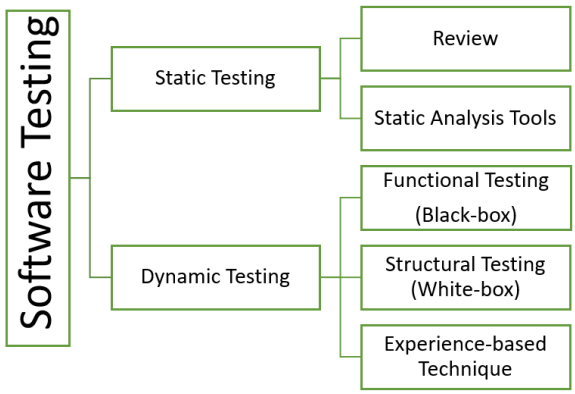


Figure 2.1: Summarization of software testing strategies.

Testing methods can be classified into two main categories: static testing and dynamic testing. In Figure 2.1, we summarize the software testing strategies covered in this report.

Static testing is a testing method that analyzes code without running it. This type of testing is performed by code reviews and static analysis tools. The examination of static testing mainly focuses on evaluating code based on its form, structure, content, or documentation without executing it [22]. For example, individuals often perform code reviews to manually examine the software source code and documents to identify bugs or other problems. Code reviews can vary from informal to formal. While two individuals can direct an informal review in various ways, a formal review often involves several knowledgeable participants along with the original author in the form of a meeting that can last several hours. The objective of a code review is to assist with discovering software bugs and to improve documentation quality. Utilizing static analysis tools is another option for static testing in finding critical defects and security weaknesses in code. Many tools are provided for undertaking static analysis, for example, the Clang Static Analyzer [6], Coverity Scan [27], and the compiler itself can report incorrect usages and invalid syntaxes. In brief, code reviews and static analysis tools can be effective at finding specific types of errors such as missing design requirements, interface specification inconsistencies, unused variables, unreachable code, standard coding violations, and syntax violations [8]. Static testing has its limitations. For example, it demands considerable time when done manually and cannot determine whether the function fundamentally matches developers' intentions. In addition, static testing tools cannot pinpoint defects that only manifest at run-time.

Dynamic testing is the process of analyzing and evaluating the code of a system or component based on its behaviour during execution [22]. The most frequently used dynamic testing strategies include black-box, white-box, and experience-based testing techniques. Black-box testing, also known as functional testing, is a testing method that checks whether the software meets its specification without using the knowledge of the code's internal structure. This type of strategy focuses on finding situations in which the software behaves differently from its specifications. Since the black-box testing process is independent of the implementation details of code design, many program paths may be left untested. White-box testing, also called structural testing, is a testing method that employs knowledge of the internal structure of the software being tested and typically exercises as much code and control flow as possible. With this type of testing, testers can access source code and documents related to the software internals and use knowledge of data structures

and algorithms to find bugs. For instance, the profiling tool Gcov that reports code coverage information is an example of a structural testing tool. A white-box technique tests more thoroughly than black-box testing but requires testers to be skilled and experienced with the software being tested. Experience-based testing techniques need the software testers to be skillful, knowledgeable and trained, since they may have insights into the areas that could contain defects or bugs during testing. Functional and structural testing can be applied to all levels of software testing. Furthermore, experience-based techniques can be used as a complementary strategy to functional and structural testing [8]. Dynamic testing helps identify weak areas in a run-time environment or defects that are problematic for static testing to find.

Static testing and dynamic testing are complementary methods for software testing because they are often effective at finding different types of defects. Based on the methods and techniques we use for software testing, different outcomes may be achieved to meet different testing standards and requirements. In brief, software testing can significantly improve software quality and durability.

2.2.1 Structural Coverage Analysis

Structural coverage analysis is frequently utilized to evaluate testing thoroughness by determining the code exercised during testing procedures. In other words, structural coverage testing measures the fraction of the code executed during testing. Structural coverage criteria can be classified into two categories: control flow and data flow criteria. Control flow criteria measure the flow of operations and paths performed during software execution, such as statement sequences and function call instructions, while data flow criteria evaluate the flow of data through variable assignments and references. To describe how control flow criteria are used in structural coverage analysis, we will first introduce the concept of a control-flow graph (CFG) and various structural components of code.

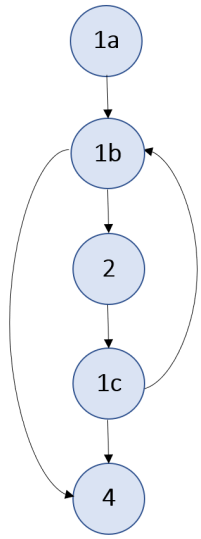
A CFG is a directed graph depicting all execution paths in code, which consists of nodes and directed edges between those nodes. Each node in a CFG represents a basic block, which is a straight-line code sequence with one entry point and one exit point [2]. In computer programming, a basic block corresponds to a statement or a set of statement fragments, which are instructions in a high-level language that perform some actions. Each edge that connects nodes in a CFG corresponds to a branch in code and shows the control flow paths. For example, suppose that we have two nodes, `a` and `b`. The edge from `a` to `b` would be present in the CFG if the statement fragment that represents node `b` can be executed immediately after statement `a`. Like a fork in the road, a node may connect to more than one path in a CFG, introducing the concept of condition and decision. A condition is a boolean expression that does not contain boolean operators like `AND`, `OR` and `NOT`, and a decision is a boolean expression composed of conditions and zero or more boolean operators [24]. Decisions are associated with branching constructs, such as `for`, `if`, and `switch` statements.

CFGs are often used in structural testing to express how structural components are chained together to impact computation sequences. Expressing code in CFGs is sometimes particularly useful in measuring coverage in software testing. To better introduce how to view and analyze program source code more clearly from the perspective of a CFG, we will give three examples of CFGs for control-flow structures that are often used in programming. Consider the codes fragments using `for`, `if`, and `switch` statements and their corresponding CFGs shown in Figures 2.2, 2.3 and 2.4. For convenience, we set the number of the node to its corresponding source code line number in these examples.

We first give an example of a `for` loop and its corresponding CFG. Figure 2.2 depicts a `for` loop that initializes an array with integers from `0` to `9`. The source code of the `for` loop is shown in Figure 2.2(a), and the corresponding CFG is given in Figure 2.2(b). The `for` loop first initializes a variable called `i` to `0` and then checks the condition that whether variable `i` is smaller than `10` is true or not. The two steps correspond

```
1       for (int i = 0; i < 10; i++) {
2           a[i] = i;
3       }
4       /*...*/
```

(a)

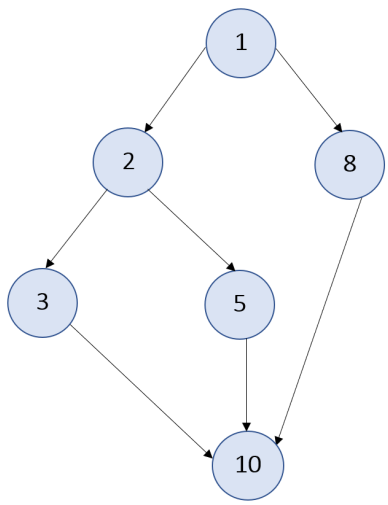


(b)

Figure 2.2: CFG example of a `for` loop. (a) Source code of the `for` loop statement. (b) CFG of the `for` loop statement.

```
1       if(x <= y) {
2           if(x == y) {
3               y = 1;
4           } else {
5               y = 2;
6           }
7       } else {
8           y = 3;
9       }
10      y = y + x;
```

(a)



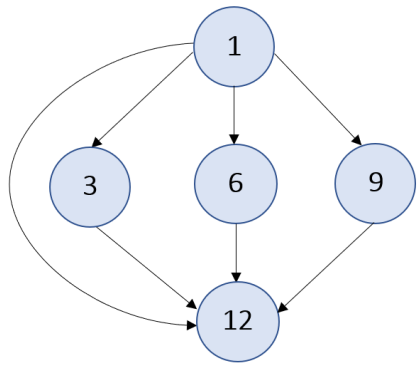
(b)

Figure 2.3: CFG example of `if` and `else`. (a) Source code of the `if` and `else` statements. (b) CFG of the `if` and `else` statements.

to nodes `1a` and `1b` in Figure 2.2(b). If the condition is true, we can move inside the `for` loop to reach the expression `a[i] = i`, which is represented by node `2`. After that, statement `i++`, which corresponds to node

```
1       switch(c) {
2       case 0:
3           x = x + 2;
4           break;
5       case 1:
6           x = x * 2;
7           break;
8       case 2:
9           x = x * x * 5;
10          break;
11      }
12      x++;
```

(a)



(b)

Figure 2.4: CFG example of a `switch`. (a) Source code of the `switch` statement. (b) CFG of the `switch` statement.

`1c`, is executed to add `1` to variable `i`. The condition of node `1b` will be rechecked after executing node `1c`. The sequence of these operations will keep running until node `1b` is not satisfied and continues to execute the statement `/*...*/` that is represented by node `4`. In another situation, node `1b` directly jumps to node `4` if the condition of `1b` is false. As shown in Figures 2.2(a) and 2.2(b), the source code of a `for` loop would typically consist of three nodes. For example, nodes `1a`, `1b` and `1c` in 2.2(b) stand for the first line of the `for` loop in 2.2(a). The operation paths of the `for` loop are particularly clearly drawn in the CFG.

Examples of the `if` and `switch` statements can more clearly show how branches appear in a CFG. Figure 2.3 shows `if` statements comparing the value of variables `x` and `y`. The source code of the `if` and `switch` statements are shown in Figure 2.3(a), and the corresponding CFG is given in Figure 2.3(b). The code first checks if the variable `x` is smaller or equal to the variable `y` at the first line in the source code, which corresponds to node `1` in the CFG. If the condition `x <= y` is true, the code will enter another `if` and `else` decision from lines 3 to 6. Otherwise, the code will directly execute the statement `y = 3` at line 8. The second `if` and `else` decision checks if the values of `x` and `y` are equal or not on the basis of the first decision being true, and then generates two branches to nodes `3` and `5`. The statement `y = y + x` represented by node `10` will be executed at the end. In summary, the `if` and `else` statements can be shown as a node that contains one or more conditions or decisions followed by two nodes that present two outcome branches in CFGs. The instructions at different branches are exercised depending upon the result of the conditions/decisions.

In CFGs, a node can connect to multiple nodes, and traversing all connected edges is usually more challenging than traversing all nodes. To illustrate, we give the Figure 2.4 that shows a `switch` statement with three different cases. The source code of the `switch` statement is shown in Figure 2.4(a), and the corresponding CFG is given in Figure 2.4(b). This `switch` example shows that a `switch` statement can have multiple results and indicates that a node can connect to many edges in a CFG. For example, Figure 2.4(b) shows that node `1`, the variable `c`, is succeeded by three nodes `3`, `6` and `9`. If we want to run all nodes, the variable `c` has to be set to `0`, `1`, and `2` respectively, and node `1` must be executed at least three times. In this situation, all the nodes are ensured to be exercised at least once. Not all branches are invoked, however. The situation that variable `c` is not equal to any value of `0`, `1`, and `2` should also be satisfied if we want to

visit all branches. By assigning the variable `c` with any value other than `0`, `1`, and `2`, we can exercise the edge between node `1` and node `12`. Consequently, this example tells us that traversing every edge is more complicated than traversing all nodes in tests.

Control flow analysis considers information about the iterations, branches, and selections in the code during testing. Moreover, control flow analysis can identify unreachable code. Data flow analysis can measure how data behaves in complex data transactions in code. In addition, it can find defects such as unused variables or undefined variables. Through testing and calculation, the control-flow and data-flow techniques can be helpful to structural coverage analysis in finding unreachable code, eliminating redundant test cases and measuring the thoroughness of testing.

2.2.2 Coverage Criteria

Having introduced the concept of a CFG and structural components of codes, now we consider the code coverage and its types. Code coverage measures the percentage of the program source code exercised based on control and data flow when running specific test suites. The coverage criteria are requirements that a test suite needs to fulfill during software testing [3]. For example, suppose the level of coverage obtained has not reached the desired threshold. In that case, additional test cases should be added for testing, and this process can be repeated until the desired threshold is achieved. In software testing, various coverage criteria exist to measure the test adequacy of programs from different testing requirements and testing levels. The primary types of coverage criteria in software include function, statement, and branch coverage, which we will introduce in detail shortly.

Function Coverage

Function coverage is a testing criterion that quantifies how many functions in the code under test are executed. We can divide the number of functions executed by the total number of functions to calculate function coverage. This coverage measure can be used to ensure that every function has been invoked during testing. If all the functions in a program have been executed, we call that full function coverage. In addition, any particular coverage type with 100% coverage can be called full coverage.

Function coverage is useful for checking if all functions are executed during a test, but it cannot ensure if all the statements inside those functions are executed. In other words, an enormous fraction of code that has full function coverage may never be invoked during the testing process. Therefore, function coverage is often considered as a weak coverage criterion if used alone. As a result, we usually use function coverage with other coverage criteria such as statement coverage and branch coverage in software testing.

Statement Coverage

Statement coverage, another testing criterion, quantifies how many statements in the code under test are executed. It can be calculated as the number of statements executed divided by the total number of statements in the code being tested. Full statement coverage ensures that every statement in the design is executed at least once. In the corresponding CFG, full statement coverage means that each node is visited at least once. Consequently, statement coverage is also called node coverage.

Unlike line coverage that measures the proportion of lines executed to the total number of lines in the code under tests, statement coverage is a more decisive coverage criterion. Each line of code may contain more than one statement, and we need to take all the potential statements into account when calculating statement coverage. To clarify, we will present a code example with the full line coverage but may not

have the full statement coverage. The code example is shown in Figure 2.2, with the source code of a `for` loop present in Figure 2.2(a) and the corresponding CFG in Figure 2.2(b). As introduced, the `for` loop at line 1 contains three nodes: `1a`, `1b` and `1c`, which correspond to statement `int i = 0`, `i < 10`, and `i++` respectively. Line coverage is 100% if any of the three nodes is executed, while statement coverage needs all nodes to be executed to obtain 100%. Compared with functions and lines in code, statements can provide more coverage details. As a result, statement coverage is more useful than function coverage and line coverage. Still, statement coverage alone is not particularly strong as it does not usually test all branch outcomes [1].

Branch Coverage

Branch coverage is meant to quantify the degree of branches taken when executing the code under test. It can be calculated by dividing the number of branch outcomes executed by the total number of branch outcomes in a program. In terms of a CFG, full branch coverage means that each edge is visited during testing. Therefore, branch coverage is stronger than statement coverage, as 100% branch coverage ensures 100% statement coverage while the reverse may not be true. The stronger the coverage criteria are, however, the more test cases are needed to achieve full coverage [8].

Next, we will present a code example with full statement coverage but not full branch coverage in Figure 2.5. The source code containing two `if` statements is shown in Figure 2.5(a) and the corresponding CFG is shown in 2.5(b). The values of the decisions in the two `if` statements can be true when `a < 0` and `b < 0`. For instance, we can execute all code fragments with a single test case `a = -1` and `b = -1` achieving full statement coverage. Not all possible branches are executed in this situation, however. For example, the branch of `a >= 0` is not exercised.

In the case of C++ code, exceptions may occur when the code is executed. So we have to consider branches taken due to exceptions. Typically, the branch coverage obtained in `lcov` or `gcovr` includes exceptional branches in the calculation. In extensive and thorough testing, users usually want to ensure that every exceptional branch is taken into consideration. In some types of testing, however, testers may not want to include exceptional branches when calculating branch coverage. Unfortunately, most tools do not support this functionality.

2.3 GCC and Gcov

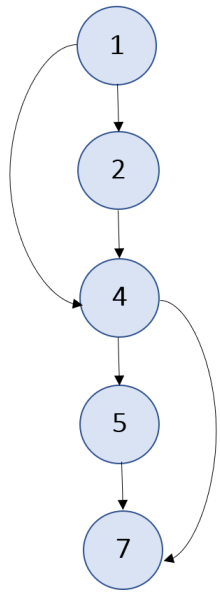
The GNU Compiler Collection, abbreviated GCC, includes the front ends and libraries to support several programming languages such as C, C++, Objective-C, Go, Fortran, Ada, and D [10]. Users invoke a language-specific driver program by commands like `gcc` for C and `g++` for C++. With these commands, GCC C/C++ compilers can parse the C/C++ statements syntactically and generate the object code. Gcov is a code coverage tool for generating information that can be used to analyze various types of code coverage. Gcov comes as a utility from GCC and only works on code compiled with GCC [11]. With the help of GCC and Gcov, users can find the places of code that are not well tested and reduce the software bugs.

2.3.1 GCC Instrumentation

GCC supports various options used to add instrumentation to the compiled code. For instance, the instrumentation for Gcov collects the statistics and calculates code coverages. Gcov uses two types of files, gcno and gcda, for coverage information. The note files with the extension `gcno`, produced during compilation, contain the data needed to reproduce the graph of basic blocks and the corresponding source line number to

```
1       if(a < 0) {
2            x = 1;
3       }
4       if(b < 0) {
5            x = 2;
6       }
7       x++;
```

(a)



(b)

Figure 2.5: An code example may have full statement coverage but not full branch coverage. (a) The source code of that example. (b) Corresponding CFG of the source code.

each block. The count files with the extension `gcda` are produced right at program termination and contain the execution count for every basic block and branch and some summary information [11]. Both types of files are generated in the same directory as the executable program. The command-line options for `gcc/g++` used to enable instrumentation and support for generating the files Gcov needs for analyzing code coverage are as follows [30]:

- `-ftest-coverage`

  This option generates note files with the extension `gcno` that the Gcov code-coverage utility can use to analyze program coverage. The note file of each source file is named with `${auxname}.gcno`, in which `${auxname}` is the basename of the source file. The C/C++ compiler will instrument the source code and add counters for every node and edge in the CFG of the code at compile time. Thus, coverage information in note files corresponds with the source files more closely if compiled without optimization.

- `-fprofile-arcs`

  This option adds code to create count files with the extension `gcda` that contain transition and value profile counts at the termination of program execution. Similarly, the count file of each source file is named with `${auxname}.gcda`. The C/C++ compiler will track basic block counts at run time with this option. The compiler loads the counts and increments them when basic blocks execute and updates the counters in the gcda file upon termination.

- `--coverage`

This option is a synonym for `-fprofile-arcs` and `-ftest-coverage`.

With the information in the gcno and gcda files, users can utilize the command `gcov` to generate code coverage statistics. By doing this, a coverage summary of files and their functions is printed to standard output. In addition, code coverage reports with the gcov extension that include more detailed coverage information are produced in the same directory as the gcno and gcda files by default. Next, we will give an example called `hello_world` to illustrate how to use Gcov and what the information generated by `gcov` looks like. The example has a single source file called `hello_world.cpp` shown in Listing 2.1. By executing the program `hello_world`, a string `Hello, World!` will be printed. Starting from the source code of `hello_world.cpp`, we will give the procedures step by step for generating the gcno, gcda, and the gcov file. The coverage summary and the generated gcov report are presented at the end.

Listing 2.1: Source code file `hello_world.cpp` for the `hello_world` program.

```
#include <iostream>

int main() {

    std::cout << "Hello, World!\n";

    int status;
    if(std::cout) {
        status = 0;
    } else {
        status = 1;
    }

    return status;
}
```

Firstly, to generate a gcno file called `hello_world.gcno` and the program called `hello_world` from the source file `hello_world.cpp`, we use the following command:

```
g++ --coverage hello_world.cpp -o hello_world
```

After that, the gcda file `hello_world.gcda` can be obtained by executing the program `hello_world`. Then, with these files and the information they carried that Gcov needs for analyzing and calculation, to generate the code coverage summary information and the coverage report for `hello_world`, we can run the command:

```
gcov hello_world
```

The summary information that should be printed to standard output would resemble that shown in Figure 2.6. According to the summary information, we know that the line coverage of file `hello_world.cpp` is 83.33% of 6, which means 5 out of 6 lines have been executed at least once. This is the basic information that `gcov` generated by default. Additional information, such as function and branch coverage, can also be obtained if enabled via command-line options of Gcov. More about the command-line options will be introduced in Chapter 2.3.2. The content of the code coverage report called `hello_world.cpp.gcov` is given in Figure 2.7. As shown in the code coverage report, the fields in the gcov file are separated by the colon character. Hence, the preamble lines consist of three fields, where the first field is always a number

```
File 'hello_world.cpp'
Lines executed:83.33% of 6
Creating 'hello_world.cpp.gcov'
File '/usr/include/c++/10/iostream'
No executable lines
Removing 'iostream.gcov'
```

Figure 2.6: Summary information of `hello_world` generated with `gcov`.

```
    -:  0:Source:hello_world.cpp
    -:  0:Graph:hello_world.gcno
    -:  0:Data:hello_world.gcda
    -:  0:Runs:1
    -:  1:#include <iostream>
    -:  2:
    1:  3:int main() {
    -:  4:
    1:  5:  std::cout << "Hello, World!\n";
    -:  6:
    -:  7:  int status;
    1:  8:  if(std::cout) {
    1:  9:      status = 0;
    -:  10: } else {
#####:  11:     status = 1;
    -:  12: }
    -:  13:
    1:  14: return status;
    -:  15:}
```

Figure 2.7: The code coverage report `hello_world.cpp.gcov`.

0, the second field is a tag name for locating a particular preamble line, and the third part is the value under that tag. In the same way, the structure of program lines is in the form of execution count, line number, and source code text separated by colons. To illustrate, the number 1 in the first column with line numbers 3, 5, 8, 9 and 14 represents the execution count of these lines, while the symbols `"#####"` with line number 11 shows that the statement `"status = 1"` is not executed. The numbers in the second column from 1 to 15 show the line number, and the contents in the third column represent each line's source code.

It is worth mentioning that the string `"0:Runs:1"` shows that the executable program has been run only once, and the number of runs accumulates across runs. That is to say, the program attempts to read the gcda files at program startup, adds the execution counts, and updates the counts to the gcda file upon program termination. If the gcda file does not exist, the execution counts are assumed to be zero. In addition, the execution counts of each line are also updated along with the program execution. Users should compile their code without optimization when using Gcov since Gcov accumulates statistics by line when analyzing coverage, and optimization may combine some lines of code together or reorder them incorrect [11].

2.3.2 Gcov Invocation

Gcov provides numerous command-line options for obtaining additional information in the coverage report. All the supported options can be printed by the `-h` or `--help` option of Gcov. To utilize the options provided by Gcov, we apply the following command-line syntax:

```
gcov [options] input_files
```

Some of the most frequently used options are as follows [11]:

- `-a` or `--all-blocks`

  When this option is specified, Gcov adds the execution count for every basic block in the coverage report. Thus, the blocks that are not executed within each line can be found with this option.

- `-b` or `--branch-probabilities`

  When this option is specified, Gcov shows the branch frequencies in percent in the coverage report and writes branch summary information to standard output. With this option, the frequency of each branch taken in the program is determined.

- `-c` or `--branch-counts`

  When this option is specified, Gcov shows the number of times each branch is taken. It should be used with `-b` or `--branch-probabilities`, otherwise, it will not affect the coverage report.

- `-f` or `--function-summaries`

  When this option is specified, Gcov prints a summary of each function in addition to the file level summary to the standard output.

- `-j` or `--json-format`

  When this option is specified, Gcov generates the coverage date in JSON format. The generated JSON file is compressed using the Gzip compression algorithm and named with suffix `".gcov.json.gz"`.

- `-t` or `--stdout`

  When this option is specified, Gcov prints the coverage information to standard output instead of generating a code coverage report in the file with the extension `gcov`.

- `-h` or `--help`

  When this option is specified, Gcov prints help information related to Gcov to standard output and exits without further processing.

To illustrate how these options work and how the corresponding coverage report looks, we will give an example called `division` that has a single source file called `division.cpp`. Two code coverage reports are generated in the example with or without Gcov options applied for comparison. The file `division.cpp` is shown in Listing 2.2 and two functions are defined in the source code. One function called `division` works to acquire the division result from the two input parameters in type `double`. The second function called `main` calls `division` with arguments `x = 50.5` and `y = 10`. The result value of executing the example is 5.05 without any exception being thrown. Based on the source code, following the steps of using Gcov to generate a code coverage report introduced in Chapter 2.3.1, we obtain a preliminary report with no options applied shown in Listing 2.3. In addition, we create a code coverage report in Listing 2.4 that contains more detailed coverage information with the `-a`, `-b`, and `-f` options.

Listing 2.2: The source code file `division.cpp` for the `division` program.

```
#include <iostream>
#include <stdexcept>

double division(double a, double b) {
    if (b == 0) {
        throw std::runtime_error("Denominator cannot be zero!");
    }
    return a / b;
}

int main () {
    double x = 50.5;
    double y = 10;
    try {
        std::cout << division(x, y) << '\n';
    } catch (std::runtime_error& e) {
        std::cerr << e.what() << '\n';
    }
    return 0;
}
```

Listing 2.3: Gcov coverage report for the `division` program with default settings.

```
        -:    0:Source:division.cpp
        -:    0:Graph:division.gcno
        -:    0:Data:division.gcda
        -:    0:Runs:1
        -:    1:#include <iostream>
        -:    2:#include <stdexcept>
        -:    3:
        1:    4:double division(double a, double b) {
        1:    5:    if (b == 0) {
    #####:    6:        throw std::runtime_error("Denominator
       cannot be zero!");
        -:    7:    }
        1:    8:    return a / b;
        -:    9:}
        -:   10:
        1:   11:int main () {
        1:   12:    double x = 50.5;
        1:   13:    double y = 10;
        -:   14:    try {
        1:   15:        std::cout << division(x, y) << '\n';
    =====:   16:    } catch (std::runtime_error& e) {
    =====:   17:        std::cerr << e.what() << '\n';
        -:   18:    }
```

```
23          1:   19:    return 0;
24          -:   20:}
```

Listing 2.4: Gcov coverage report for the `division` program with the `-a`, `-b`, and `-f` options.

```
1           -:    0:Source:division.cpp
2           -:    0:Graph:division.gcno
3           -:    0:Data:division.gcda
4           -:    0:Runs:1
5           -:    1:#include <iostream>
6           -:    2:#include <stdexcept>
7           -:    3:
8   function _Z8divisiondd called 1 returned 100% blocks executed 50%
9           1:    4:double division(double a, double b) {
10          1:    5:    if (b == 0) {
11          1:    5-block  0
12  branch  0 taken 0% (fallthrough)
13  branch  1 taken 100%
14      #####:    6:        throw std::runtime_error("Denominator
            cannot be zero!");
15      %%%%%:    6-block  0
16  call    0 never executed
17  call    1 never executed
18  branch  2 never executed
19  branch  3 never executed
20      %%%%%:    6-block  1
21  call    4 never executed
22      $$$$$:    6-block  2
23  call    5 never executed
24          -:    7:    }
25          1:    8:    return a / b;
26          1:    8-block  0
27          1:    8-block  1
28          -:    9:}
29          -:   10:
30  function main called 1 returned 100% blocks executed 43%
31          1:   11:int main () {
32          1:   12:    double x = 50.5;
33          1:   13:    double y = 10;
34          -:   14:    try {
35          1:   15:        std::cout << division(x, y) << '\n';
36          1:   15-block  0
37  call    0 returned 100%
38  branch  1 taken 100% (fallthrough)
39  branch  2 taken 0% (throw)
40          1:   15-block  1
41  call    3 returned 100%
```

```
42   branch  4 taken 100% (fallthrough)
43   branch  5 taken 0% (throw)
44            1:   15-block  2
45   call    6 returned 100%
46   branch  7 taken 100% (fallthrough)
47   branch  8 taken 0% (throw)
48       =====:   16:    } catch (std::runtime_error& e) {
49       $$$$$:   16-block  0
50   branch  0 never executed
51   branch  1 never executed
52       $$$$$:   16-block  1
53   call    2 never executed
54       $$$$$:   16-block  2
55   call    3 never executed
56       $$$$$:   16-block  3
57   call    4 never executed
58       =====:   17:        std::cerr << e.what() << '\n';
59   call    0 never executed
60   call    1 never executed
61   branch  2 never executed
62   branch  3 never executed
63       $$$$$:   17-block  0
64   call    4 never executed
65   branch  5 never executed
66   branch  6 never executed
67            -:   18:    }
68            1:   19:    return 0;
69            1:   19-block  0
70            -:   20:}
```

As can be seen by comparing the two Listings 2.3 and 2.4, more information is included in the coverage report with the `-a`, `-b`, and `-f` options than with no options. For example, lines 8 and 30 in Listing 2.4 present detailed information of `division` and `main`, including the execution counts and the percentage of executed blocks in each function. It is worth mentioning that special symbols are used to show unexecuted lines and blocks. The unexecuted lines are marked by `"#####"` or `"====="`, depending on whether they are reachable by non-exceptional paths. Specifically, the string `"#####"` is used for zero coverage lines with unexceptional paths accessible and `"====="` for exceptional paths only, such as C++ exception handlers. In addition, the execution counts for unexecuted blocks are shown as `"$$$$$"` or `"%%%%%"`. The unexecuted basic blocks that are reachable by unexceptional paths are marked by `"$$$$$"`; otherwise, the string `"%%%%%"` is used. If the executed basic blocks contain a statement with an execution count of zero, character `*` is added to the end of the execution count.

Besides the information for functions and unexecuted lines, the `-a` option also include block counts, such as on lines 11, 15 and 20 in Listing 2.4. Also, the `-b` option introduces the execution count for each branch and function call on, such as lines 12, 13, 16 and 17 in Listing 2.4. If a single source line corresponds to code with more than one branch or function call, multiple branches or calls are listed. The execution counts are shown for each branch if it has been executed at least once; otherwise, the string `"never executed"` will be printed.

Gcov also provides options to directly output the coverage information using coloured text to standard output or generate a compressed file that contains the JSON-format coverage data, which is a simpler format for programs to parse. With the help of options provided by Gcov, we can better comprehend the function, branch, and statement being achieved during testing. Moreover, we can easily find the parts of the code that are not well exercised during testing and obtain both summary and detailed code coverage information of the testing code.

2.3.3 Gcov JSON-format Coverage Data Schema

As mentioned, Gcov provides the option to produce the code coverage data in JSON format. With the `-j` or `--json-format` option, we can obtain a zipped file containing the coverage data. After decompressing the zipped file, the JSON-format data containing coverage information is acquired. Next, we describe the schema that Gcov uses to present the JSON-format coverage data.

The root object of the coverage data has the following fields:

- the current working directory where the program being tested was compiled, which is associated with the JSON key `current_working_directory`;
- the name of the data file of the testing program, which is associated with the JSON key `data_file`;
- the semantic version of the format, which is associated with the JSON key `format_version`;
- the version of the GCC compiler, which is associated with the JSON key `gcc_version`; and
- an array of the file objects, which is associated with the JSON key `files`.

The file object in the `files` array corresponds to the source file executed in running the program and contains more detailed coverage information on the function and line basis in that file. Each file object has the following fields:

- the name of the source file, which is associated with the JSON key `file`;
- an array of the function objects, which is associated with the JSON key `functions`; and
- an array of the line objects, which is associated with the JSON key `lines`.

The function object in the `functions` array contains coverage data related to the function, such as the function execution count, the blocks included in that function, the function position in the source file, and the function mangled and demangled names. Specifically, each function object has the following fields:

- the number of blocks included in the function, which is associated with the JSON key `blocks`;
- the number of executed blocks of the function, which is associated with the JSON key `blocks_executed`;
- the number of executions of the function, which is associated with the JSON key `execution_count`;
- the demangled name of the function, which is associated with the JSON key `demangled_name`;
- the mangled name of the function, which is associated with the JSON key `name`;
- the line in the source file where the function begins, which is associated with the JSON key `start_line`;

- the column in the source file where the function begins, which is associated with the JSON key `start_column`;
- the line in the source file where the function ends, which is associated with the JSON key `end_line`; and
- the column in the source file where the function ends, which is associated with the JSON key `end_column`.

Similarly, each line object in the `lines` array, which contains detailed coverage data of that line, has the following fields:

- the number of executions of the line, which is associated with the JSON key `count`;
- the line number of the line, which is associated with the JSON key `line_number`;
- the name of the function that this line belongs to, which is associated with the JSON key `function_name`;
- a boolean flag to show whether the line has unexecuted blocks, which is associated with the JSON key `unexecuted_block`; and
- an array of the branch objects, which is associated with the JSON key `branches`.

Lastly, each branch object in the `branches` array has the following fields:

- the number of executions of the branch, which is associated with the JSON key `count`;
- a boolean flag to show whether this branch is a fall through branch, which is associated with the JSON key `fallthrough`; and
- a boolean flag to show whether this branch is an exceptional branch, which is associated with the JSON key `throw`.

To show how the coverage data in the JSON format appears and clearly explain the structure, we provide partial coverage data in the JSON format shown in Listing 2.5 as an example. The coverage data is generated for the `division` program, of which the source code is shown in Listing 2.2. According to the field values in the root object, we can find general information about the program being tested, such as the data file name is `division`, the source file is `division.cpp`, and the program was only executed once.

Listing 2.5: Partial coverage data in JSON format generated for the `division` program.

```
1  {
2      "current_working_directory":
           "/home/mint/Coverage_Report/doc/software/division",
3      "data_file": "division",
4      "files": [
5          {
6              "file": "division.cpp",
7              "functions": [
8                  {
9                      "blocks": 8,
10                     "blocks_executed": 4,
11                     "demangled_name": "division(double, double)",
```

```
                        "end_column": 1,
                        "end_line": 9,
                        "execution_count": 1,
                        "name": "_Z8divisiondd",
                        "start_column": 8,
                        "start_line": 4
                    },
                    {
                        "blocks": 14,
                        "blocks_executed": 6,
                        "demangled_name": "main",
                        "end_column": 1,
                        "end_line": 20,
                        "execution_count": 1,
                        "name": "main",
                        "start_column": 5,
                        "start_line": 11
                    }
                ],
                "lines": [
                    {
                        "branches": [
                            {
                                "count": 0,
                                "fallthrough": true,
                                "throw": false
                            },
                            {
                                "count": 1,
                                "fallthrough": false,
                                "throw": false
                            }
                        ],
                        "count": 1,
                        "function_name": "_Z8divisiondd",
                        "line_number": 5,
                        "unexecuted_block": false
                    },

                    /* Many line objects are omitted here. */

                    {
                        "branches": [],
                        "count": 1,
                        "function_name": "main",
                        "line_number": 13,
```

```
58                          "unexecuted_block": false
59                      }
60                  ]
61              },
62              {
63                  "file": "/usr/include/c++/10/iostream",
64                  "functions": [],
65                  "lines": []
66              }
67          ],
68          "format_version": "1",
69          "gcc_version": "10.3.0"
70      }
```

As can be seen from lines 7 to 30 in Listing 2.5, we can obtain more detailed coverage information of the two functions, `division` and `main`, included in the source file `division.cpp`. For example, the function `division` that starts at line 4 and ends at line 9 was executed once. This function includes eight blocks, and only four of them have been executed. In addition, the function demangled name is `division(double, double)`, while the accordingly mangled name is`_Z8divisiondd`. The coverage information of the `main` function can also be obtained in the data provided.

The examples of line objects from line 31 to line 61 shown in Listing 2.5 present more information related to lines and branches. For example, the line with source code line number 5 that belongs to function `_Z8divisiondd` was executed once, and there are no unexecuted blocks included in that line. Moreover, this line has two branches, and only one branch has been executed. According to the value of the `fallthrough` flag in the branch objects of that line, we acquire that the branch executed is not the fall through branch and corresponds to the statement `"b != 0"` according to the source code of `division`.

In brief, we can obtain detailed coverage information of the entire file for the program being tested from the JSON-format data provided by Gcov. By processing the coverage data, the function, statement, and branch coverage of each file can be calculated according to the definition of those coverage criteria.

2.4 Name Mangling

One cannot avoid having to deal with function names in the context of code coverage. In C++, multiple functions can have the same name in the source code, however, due to overloading or visibility within different scopes. Unfortunately, linkers do not understand concepts like function overloading and cannot distinguish between functions that have the same names in the source code. For this reason, linkers require every function to have a unique name. Consequently, using a specific set of rules, the compiler assigns each function in the source code a unique name in the process known as name mangling. The unique name produced by name mangling is called the mangled name, and the original function name in the source code is called the demangled name. Additionally, the name mangling rules used are platform-specific. In this project, the rules are assumed to follow the Itanium application binary interface (ABI) standard for C++ programs [23]. For example, in the Itanium ABI, the functions

```
int add(int)
int add(double)
```

are mapped to the respective mangled names

```
        _Z3addi
        _Z3addd
```

The two mangled names differ in the last character representing function parameter type. For more details and examples of the name mangling, the reader is referred to [18].

Name mangling is relevant to code coverage because mangled names appear in some contexts where it is important to distinguish between multiple overloads of a function or a particular function used to implement a constructor or destructor. Specifically, the compiler can implement each constructor using multiple functions, and the same holds for destructors. For example, a virtual base destructor with the demangled name `base::~base()` can have two different mangled names: `_ZN4baseD0Ev` and `_ZN4baseD2Ev`, depending on the way the destructor is used. Therefore, testers may need to pay special attention to situations like overloaded functions, constructors, and destructors in code coverage testing.

Chapter 3

Gcov Report Generator Software

3.1 Overview

This chapter introduces the Gcov Report Generator (GRG) software developed by the author. The rest of this chapter is organized as follows. First, we give a general introduction to the working principle and overall structure of the GRG software. Then, we explain the procedures for building and installing the GRG software. Finally, we give a detailed introduction to the GRG software with usage examples.

3.2 GRG Software Introduction

The GRG software consists of a C++ library and an application program called `coverage`, which uses this library to generate nicely-formatted customized code coverage reports in PDF format. There are mainly two sections in the code coverage reports generated by the GRG software: the summary section and the detail section. The summary section includes summary tables containing execution counts calculated and coverage information in percentage format for specified files and functions, as well as file pathnames and function names related to the coverage information. The information in the summary tables are on a per-file and per-function basis for the function, statement, and branch coverage, respectively. The detail section contains the corresponding source code of the files and functions with lines, branches, and blocks details, such as execution counts of lines and branches. In addition, the file pathnames and function names in summary tables correspond to the source code in the detail section through hyperlinks so that the GRG users can easily find the specified file or function in the code coverage reports generated. More importantly, the GRG software allows users to customize their coverage reports by filtering and selecting the contents. For example, the GRG software can:

- filter coverage results using file and function patterns (i.e., regular expressions);
- calculate the branch coverage with or without exceptional branches;
- format the code coverage report by keeping only the selected section contents;
- select the coverage information included by setting the function, statement, and branch coverage threshold on a per-file and per-function basis; and

- aggregate coverage results for functions with the same names in source code but different mangled names, such as in the cases of constructors, destructors, and templates.

After describing the functionality and behaviour of this software, we will introduce the general structure of the GRG software. Under the global namespace called `grg`, the GRG software provides one primary functor class named `Report_maker` used for report generation and a few secondary classes used to specify flags or options for the primary class. The `Report_maker` class mainly consists of two parts: the constructor and the function call operator. First, the constructor parses coverage information from gcda files input as parameters, calculates the coverage statistics, and stores the statistics in an appropriate data structure. Then, the function call operator generates formatted code coverage reports in PDF format using these statistics and writes the reports to the assigned output stream. Moreover, the GRG software provides some secondary classes such as `Global_options` and `Report_options` as input parameters for the constructor and the function call operator. These classes have flags and options for users to customize the code coverage reports. The flags and options will be introduced in more detail in Section 3.4.

3.3 Software Installation

The GRG software is written in C++ and utilizes many C++17 features. Gcov, a tool provided with the C++ compiler, is used by the GRG software to acquire code coverage information. As a result, GCC version 10.3.0 or later is needed for the GRG software.

The GRG software applied the JSON for Modern C++ library to convert the coverage data in JSON format generated by Gcov into the format that fits the data structure used by the GRG library. The single header file `json.hpp` written by Niels Lohmann has already been included as part of the GRG software for convenience [16]. So, this library does not need to be installed before building the GRG software, eliminating the hassle of installing this library.

The GRG software uses the `stable_vector` container in the Boost Container library and uses the Boost Process library to manage system processes when parsing coverage information [20] [19]. In addition, the GRG software utilizes `pdflatex` to produce PDF files from LaTeX source code [25]. So, the Boost libraries and the software program `pdflatex` should be installed before building the GRG software.

In conclusion, the following versions of required software packages have been verified to work with the GRG software:

- GCC 10.3.0;
- Boost 1.71.0; and
- pdfTeX 1.40.22 (TeX Live 2021).

The GRG software uses CMake for compiling, building, testing, and installation [7]. In what follows, let `$SOURCE_DIR` denote the top-level directory of the GRG software distribution, and `$BUILD_DIR` denote a directory where build files are created. The GRG software also provides some test files in the directory named `tests` under the `$SOURCE_DIR` directory. These test files are also built and run with CMake as target tests to verify if the GRG software could run correctly after being built.

To build, test, and install the GRG software with CMake, the following steps should be:

1. Generate the native build files by running the command:

```
cmake -H$SOURCE_DIR -B$BUILD_DIR
```

2. Build the GRG software and test files by running the command:

```
cmake --build $BUILD_DIR
```

3. Next, we can test the GRG software built by running the command follows.

```
cmake --build $BUILD_DIR --target test
```

   This command first runs the programs built for the test files to generate their gcda files and then runs the GRG software with these gcda files to generate code coverage reports. The text like the following will be printed to the standard output upon successful completion of the tests. In addition, the code coverage reports in PDF format generated for the test files will be presented in the directory named `CoverageTestResults` under the `$SOURCE_DIR` directory.

```
100% tests passed, 0 tests failed out of 4
```

   The GRG software should only be used if it passes all test cases in the test suite.

4. Finally, we can install the GRG software by running the command:

```
cmake --build $BUILD_DIR --target install
```

In order to make it more convenient for users, the GRG software provides the file called `run_cmake` to execute all the commands in the above instructions. Thus, users can compile, run, test, and install the GRG software by simply running this script.

3.4 The GRG Library

As introduced in Section 3.2, the GRG library provides a primary class called `Report_maker` and mainly two supporting classes. The APIs of the GRG library are designed to be easy to understand and convenient to use with two main functions in the `Report_maker` class: the constructor and the function call operator. In addition, two supporting classes called `Global_options` and `Report_options` are used to specify customizations in reports. The `Report_maker` class is non-movable and non-copyable. Next, we will introduce how the constructor and the function call operator work in generating code coverage reports in more detail.

The constructor works for parsing coverage information and calculating coverage statistics and has two input parameters: the initialized `Global_options` class object and an array of strings of the gcda files. This constructor first invokes `gcov` on the input gcda files to generate an intermediate JSON dataset. Then, it reads coverage information from the JSON dataset and stores it on appropriate data structures. Lastly, it calculates the function, statement, and branch coverage statistics on a per-file and per-function basis using the data read. The constructor will throw an exception if an error occurs.

The class object of the `Global_options` is a parameter of the constructor. By adjusting the settings in the `Global_options` object, library users can filter using the file and function patterns, as well as aggregate information for constructors, destructors, and templates. In addition, users can choose if they want to ignore exceptional branches when calculating branch coverage. It warrants mentioning that the settings in `Global_options` are common to all code coverage reports generated by a single `Report_maker` object and cannot be modified after construction.

The function call operator works to generate code coverage reports in PDF format using the coverage statistics computed in the constructor and corresponding file source codes. This function call operator has

two input parameters: the initialized `Report_options` class object and an output stream to which to write the code coverage report generated. The function call operator first generates a LaTeX source file in a system directory for temporary files during execution. Then, it outputs summary tables that contain the coverage statistics and detail sections that include the corresponding source code and complete coverage information into the LaTeX document. In what follows, it runs the LaTeX software to produce the coverage report in PDF format. The coverage report generated is then written to the specified output stream. Finally, the temporary directory created for the LaTeX source file is deleted. Thus, each time this operator function is called, a code coverage report in PDF format is generated.

The class object of `Report_options` is a parameter of the function call operator. By adjusting the settings in the `Report_options` object, users can select the contents included in the report, filter the coverage statistics by thresholds, control the name of the temporary directory, and specify the output pathname of the coverage report generated. By adjusting the options in the `Report_options` object, users can obtain multiple code coverage reports containing different contents and various formats for a single `Report_maker` object.

In the section that follows, we will explain in more detail how the GRG library APIs and the options and flags in the `Global_options` and `Report_options` classes are used.

3.4.1 API Documentation

The API documentation of the GRG library is shown in Listing 3.1. A more detailed description of the various classes follows.

Listing 3.1: Application programming interfaces of the GRG library.

```
1  #ifndef grg_hpp
2  #define grg_hpp
3
4  #include <cstdlib>
5  #include <boost/process.hpp> // boost::process
6  #include <memory> // std::unique_ptr
7  #include <grg/code_coverage_data_structure.hpp>
8  #include <grg/nlohmann/json.hpp> // nlohmann::json
9
10
11 namespace bp = boost::process;
12 using json = nlohmann::json;
13
14 namespace grg {
15
16     class Report_maker {
17     public:
18         // Used in the process to parse and calculate coverage
               statistics
19         struct Global_options {
20             bool aggregate_ctors_dtors = false;
21             bool aggregate_templates = false;
22             bool keep_regex = true;
```

```
23                 std::vector<File_function_regex> filter_patterns;
24             };
25
26             // Used in the process to generate PDF coverage reports.
27             struct Report_options {
28                 bool ignore_exceptional_branches = false;
29                 bool function_coverage_summary = false;
30                 bool statement_coverage_summary = false;
31                 bool branch_coverage_summary = false;
32                 bool function_coverage_summary_per_file = false;
33                 bool function_coverage_summary_per_function = false;
34                 bool statement_coverage_summary_per_file = false;
35                 bool statement_coverage_summary_per_function = false;
36                 bool branch_coverage_summary_per_file = false;
37                 bool branch_coverage_summary_per_function = false;
38                 bool detail = false;
39                 bool keep_temporary_directory = false;
40                 double func_cov_per_file_threshold = 101;
41                 double func_cov_per_func_threshold = 101;
42                 double state_cov_per_file_threshold = 101;
43                 double state_cov_per_func_threshold = 101;
44                 double branch_cov_per_file_threshold = 101;
45                 double branch_cov_per_func_threshold = 101;
46                 std::string temporary_directory_name = "";
47                 std::string output_file_name = "";
48                 std::string pdflatex_program = "pdflatex";
49             };
50
51             // The contructor
52             Report_maker(Global_options options,
                   std::vector<std::string>& gcda_files);
53
54             // The function call operator
55             bool operator()(Report_options options, std::ostream& out);
56         };
57
58     } /*end of namespace grg*/
59
60     #endif /*grg_hpp*/
```

The `Global_options` Class

The `Global_options` class shown in Listing 3.1 from line 19 to line 24 allows users to specify the options settings utilized in the process of parsing data from the JSON dataset and calculating the code coverage statistics. These options mainly work to filter by file and function patterns and aggregate coverage information in constructors, destructors, and templates.

As described in Section 2.4, the constructors and destructors are implemented using multiple functions. The data member called `aggregate_ctors_dtors` is a boolean flag for users to choose if they want to aggregate the coverage information for constructors and destructors, and this data member defaults to false.

Similarly, Gcov also considers the different instantiations of a template as different functions in the report generated. Depending on the number of instantiations, a template can be present in several associated functions with different mangled names. The `aggregate_templates` data member gives the users the ability to combine the coverage information for multiple instantiations of a template into one, and this data member defaults to false.

A code coverage report produced by Gcov could easily contain thousands of functions and become quite long. Since users may find it challenging to locate the information of interest when reading a report, the GRG library provides its users with the option to filter data based on pattern matching. More specifically, users can specify regular expressions by the struct called `File_function_regex`, which consists of two strings expressed as follows:

```
struct File_function_regex {
    std::string file_regex;
    std::string function_regex;
};
```

The two members, `file_regex` and `function_regex`, are Portable Operating System Interface (POSIX) regular expressions [26]. Both regular expressions in a `File_function_regex` must match to select one file and function pattern. The GRG library users can define multiple patterns for filtering, where the relationship among patterns is `OR`. In addition, the `keep_regex` data member allows users to invert the sense of pattern matching. This data member defaults to true to keep all the patterns matched.

The `Report_options` Class

The `Report_options` class shown in Listing 3.1 from line 27 to line 49 allows users to control the option settings used when generating code coverage reports in PDF format. These options mainly work to specify and format the contents presented in the coverage reports generated.

As introduced, the GRG library provides summary tables containing function, statement, and branch coverage information on per-file and per-function bases in coverage reports. Following this, the GRG library provides six data members for users to choose whether to include these summary tables in the coverage report. Specifically, the `function_coverage_summary_per_file` data member is a boolean flag for users to choose whether they want to include summary tables of the function coverage per-file in the report, while the `function_coverage_summary_per_function` data member has the same functionality but on a per-function basis. The `function_coverage_summary` data member selects both the `function_coverage_summary_per_file` and `function_coverage_summary_per_function` at the same time. Similarly, the GRG library also offers boolean data members for the statement and branch coverage to enable their summary tables in coverage reports on per-file and per-function basis, such as the `statement_coverage_summary_per_file` and `statement_coverage_summary_per_function` data members for the statement coverage, and the two data members `branch_coverage_summary_per_file` and `branch_coverage_summary_per_function` for branch coverage. The `statement_coverage_summary` data member selects both the two data members for statement coverage mentioned above simultaneously, and the `branch_coverage_summary` data member works the same for the two data members of branch coverage. The values of these data members all default to false.

In addition, the `ignore_exceptional_branches` and `detail` data members also control the information presented in coverage reports. The `ignore_exceptional_branches` data member allows users to omit exceptional branches in the branch coverage calculation, and this data member defaults to false. The `detail` data member allows users to specify if the report should include source code and detailed coverage information, and this data member defaults to false.

To format the coverage report, the GRG library offers six data members to filter the coverage statistics by setting the coverage thresholds on a per-file and per-function basis, such as `func_cov_per_file_threshold` and `state_cov_per_file_threshold`. The value of each of these data members is of the type `double` and corresponds to a coverage threshold. A threshold value is in units of percent. The expected use case is that users simply care about coverage worse than a certain amount, so only the elements with lower percentage values than the coverage threshold will be kept in coverage reports generated by the GRG library. For example, the element with full statement coverage in the summary table of statement coverage per function will not be present in the coverage report if the value of the `state_cov_per_func_threshold` data member is set to 100. All coverage threshold values default to a value greater than 100, which means all the files and functions will be included in coverage reports.

The other data members are `temporary_directory_name`, `pdflatex_program`, and `output_file_name`, and `keep_temporary_directory`. The `temporary_directory_name` data member is used to define the name of the temporary directory containing the LaTeX source file. The value of this data member is a standard string for users to specify the name of their temporary directory, where the directory specified must already exist. This data member defaults to the system directory for temporary files. Next, the data member `keep_temporary_directory` allows users to keep the temporary directory, whereas the temporary directory is defaulted to be deleted. The `pdflatex_program` data member is applied for users to specify the pathname of the program to convert the LaTeX source file to generate the PDF coverage report. Any program that has a CLI compatible with `pdflatex` can be used to set the value of this data member. The `pdflatex`, `lualatex`, or `pdflatex_frontend` are possible values for the `pdflatex_program`, where `pdflatex_frontend` is a script provided by the GRG software based on `pdflatex` with an extended memory size for larger size files [25]. This data member defaults to run `pdflatex`. Lastly, the `output_file_name` data member allows users to specify the pathname of the code coverage report in PDF format generated by the GRG library. The GRG library prints the report to standard output if the value of this data member is an empty string.

3.4.2 Library APIs Usage Examples

In this section, we will present two files containing some usage examples on how to use the APIs of the GRG library to generate code coverage reports for a source file called `template.cpp`. The source file `template.cpp` shown in Listing 3.2 has four functions:

- a template function called `bubbleSort` to sort the input array with element type specified in ascending order;
- two overloading functions with the same name `sum` used to add two input numbers but differs in the return type and input data type; and
- a function called `main` that initializes some variables and calls the other functions.

For comparison, we will generate two code coverage reports of the `template.cpp`: one report without applying any filter option including full files and functions is generated by running the codes in `full_report.cpp`

shown in Listing 3.3, while another report utilizing specific options keeping part of files and functions is generated by running the codes in `specified_report.cpp` shown in Listing 3.4.

Listing 3.2: Source code of the `template.cpp` used in code coverage report example.

```
#include <iostream>
#include <type_traits> //std::extent

// A template function
template <class T>
void bubbleSort(T a[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = n - 1; i < j; j--) {
            if (a[j] < a[j - 1]) {
                std::swap(a[j], a[j - 1]);
            }
        }
    }
}

// Two overloading functions
int sum(int a, int b) {
    return a + b;
}

double sum(double a, double b) {
    return a + b;
}

int main() {

    // Calls template function
    int a1[5] = {3, 5, 1, 2, 4};
    double a2[5] = {2.3, 5.6, 1.1, 2.2, 10.0};
    bubbleSort<int>(a1, std::extent<int[5]>::value);
    bubbleSort<double>(a2, std::extent<int[5]>::value);

    // Calls overload functions and outputs the results
    int a = 1;
    int b =2;
    double c = 1.2;
    double d = 2.4;
    std::cout << sum(a,b) << '\n';
    std::cout << sum(c,d) << '\n';

    return 0;
}
```

Listing 3.3: Source code for the file `full_report.cpp` that uses the GRG library API with default options.

```cpp
#include <iostream>
#include <fstream>
#include "grg/grg.hpp"

int main() {

    // Initialize the global options
    grg::Report_maker::Global_options global_options;
    std::vector<std::string> gcda_files_vec;
    std::string input_file("template.gcda");
    gcda_files_vec.push_back(input_file);

    // Call the constructor of Report_maker class
    grg::Report_maker maker(global_options, gcda_files_vec);

    // Initialize the report options
    grg::Report_maker::Report_options report_options;
    report_options.function_coverage_summary = true;
    report_options.statement_coverage_summary = true;
    report_options.branch_coverage_summary = true;
    report_options.detail = true;

    // Call the operator() to genrate the coverage report stream
    // and write it to a PDF file named full_report.pdf
    std::fstream myfile("full_report.pdf", std::ios::out);
    int status = maker(report_options, myfile);
    myfile.close();

    return status;
}
```

Listing 3.4: Source code for the file `specified_report.cpp` that uses the GRG library API with particular options.

```cpp
#include <iostream>
#include <fstream>
#include "grg/grg.hpp"

int main() {

    // Initialize the global options
    grg::Report_maker::Global_options global_options;
    global_options.aggregate_templates = true;
    File_function_regex cur_pattern = {".*template.cpp", "(.*?)"};
    global_options.filter_patterns.push_back(cur_pattern);
```

```
12        std::vector<std::string> gcda_files_vec;
13        std::string input_file("template.gcda");
14        gcda_files_vec.push_back(input_file);
15
16        // Call the constructor of Report_maker class
17        grg::Report_maker maker(global_options, gcda_files_vec);
18
19        // Initialize the report options
20        grg::Report_maker::Report_options report_options;
21        report_options.function_coverage_summary = true;
22        report_options.statement_coverage_summary = true;
23        report_options.branch_coverage_summary = true;
24        report_options.detail = true;
25        double threshold = 100;
26        report_options.state_cov_per_func_threshold = threshold;
27        report_options.branch_cov_per_func_threshold = threshold;
28
29        // Call the operator() to genrate the coverage report stream
30        // and write it to a PDF file named specified_report.pdf
31        std::fstream myfile("specified_report.pdf", std::ios::out);
32        int status = maker(report_options, myfile);
33        myfile.close();
34
35        return status;
36    }
```

The general procedure for using the APIs of the GRG library to generate code coverage reports is to first initialize the settings in the `Global_options` object. When the prerequisites are ready, users can call the constructor of the `Report_maker` class with the `Global_options` object. Then, users can set up the options in the `Report_options` object and specify the output file stream. Lastly, the function call operator can be called with the chosen settings to generate the code coverage report in PDF format and write it to the desired stream. Next, we will consider the programs demonstrating the use of the GRG library in more detail by walking through the source code of the `full_report.cpp` and the `specified_report.cpp`. In the following examples, we assume the GRG software has been installed and the file `template.gcda` has been obtained.

As shown in Listing 3.3, lines 8 to 11 initialize the `Global_options` class object without setting any filter options and construct an input array that has the element `template.gcda`. After that, we called the constructor at line 14 with default options in the GRG library to calculate code coverage statistics. Then, lines 17 to 21 specify that the report should include both per-file and per-function summary tables for function, statement, and branch coverage, as well as a detail section. The coverage thresholds are not specified, which means they all default to a value larger than 100. So, the GRG library will include all the files and functions in the code coverage report. Lines 25 to 27 show how to write the generated coverage report stream in a PDF file.

Compared to `full_report.cpp`, the file `specified_report.cpp` adds a file-function pattern and specifies some options to filter the contents in the report generated. As shown in Listing 3.4, besides initializing the `Global_options` class object and the input array, lines 8 to 14 set the option that aggregates different instances of the same template to `true` and add the file-function pattern to keep only the functions in the file

`template.cpp` in coverage reports. Furthermore, lines 25 to 27 set the threshold for statement and branch coverage per function to 100%. As a result, the GRG library only includes the functions with statement and branch coverage lower than 100% in the summary tables.

The two code coverage reports generated are `full_report.pdf` and `specified_report.pdf`. The report `full_report.pdf` including all the files and functions is provided in Appendix A, and the report `specified_report.pdf` utilizing the filtering options is presented in Appendix B for comparison.

3.5 Application Program Coverage

In addition to the GRG library, we also developed an application program called `coverage` to use the library through the command-line interface (CLI). This application program allows users to generate coverage reports from the command line conveniently. In what follows, we will describe the application program in detail.

3.5.1 Command-Line Interface

The application program `coverage` is controlled by numerous command-line options. There are two types of these options: global options and per-report options. Global options apply to all reports generated by the program, while per-report options apply only to a single report. The detailed information of global options and per-report options are listed as follow:

- **Global Options**
  - `-c` or `--aggregate-constructors-destructors`
    When this option is specified, `coverage` aggregates all of the C++ compiler-generated functions from one particular C++ constructor as if they were the same function when determining code coverage. This option also has a similar effect on destructors.
  - `-t` or `--aggregate-templates`
    When this option is specified, `coverage` aggregates all instantiations of the same template as if they were the same function when determining code coverage.
  - `-h` or `--help`
    When this option is specified, `coverage` prints help information about using `coverage` to the standard output and exits without doing any further processing.

- **Global Options — Function Selection**
  - `-x` *REGEX* or `--file-pattern` *REGEX*
    When this option is specified, `coverage` sets the current value of the file pattern to the extended regular expression specified by *REGEX*. The value of *REGEX* defaults to string `"(.*?)"` (i.e., select all files).
  - `-y` *REGEX* or `--function-pattern` *REGEX*
    When this option is specified, `coverage` sets the current value of the function pattern to the extended regular expression specified by *REGEX*. The value of *REGEX* defaults to string `"(.*?)"` (i.e., select all functions).

- `-p` *EXPRESSION* or `--pattern` *EXPRESSION*

  When this option is specified, `coverage` adds a new file-function pattern to the pattern list that consists of the file-function patterns. The *EXPRESSION* is a string chosen from `keep` or `delete`, through which we can determine whether to keep or discard the current file-function pattern when generating the report. The value of *EXPRESSION* defaults to `keep`, which means the file-function pattern that matches the *EXPRESSION* will be kept.

- `-r` or `--reset`

  When this option is specified, `coverage` resets all of the per-report options to their default values.

- **Per-Report Options**

  - `-n` *DIRNAME* or `--temporary-directory-name` *DIRNAME*

    When this option is specified, `coverage` sets the *DIRNAME* as the name of the temporary directory used to store the LaTeX source file. The directory specified must already exist. For example, `"/tmp/foobar"` or `"apple"` can be used. If this option is not specified, the system directory for temporary files will be used (i.e., the directory returned by the C++ standard library function `std::filesystem::temp_directory_path`).

  - `-k` or `--keep-temporary-directory`

    When this option is specified, `coverage` keeps the temporary directory that contains the LaTeX source file. If this option is not specified, the directory will be deleted.

  - `-e` or `--ignore-exceptional-branches`

    When this option is specified, `coverage` treats exceptional branches as if they do not exist when determining branch coverage.

  - `-o` *PATHNAME* or `--output` *PATHNAME*

    When this option is specified, `coverage` sets *PATHNAME* as the output pathname to which to write the code coverage report. If not specified, the report is printed to standard output.

  - `-m` *PROGRAM* or `--pdflatex-program` *PROGRAM*

    This option sets *PROGRAM* as the pathname of the program to use to convert LaTeX source to a PDF document. Any program that has a CLI compatible with `pdflatex` can be used. The `pdflatex`, `lualatex`, or `pdflatex_frontend` are possible values for *PROGRAM*, where the `pdflatex_frontend` is the script provided by the GRG software based on `pdflatex` with an extended memory size for larger size files. The value of *PROGRAM* defaults to `pdflatex`.

- **Per-Report Options — Report Formatting**

  - `-d` or `--detail`

    When this option is specified, `coverage` includes the detail section in the report, which contains detailed coverage information for each selected function.

  - `--function-coverage-summary-per-file`

    When this option is specified, `coverage` includes the per-file function coverage summary table in the code coverage report.

- `--function-coverage-summary-per-function`
  When this option is specified, `coverage` includes the per-function function coverage summary table in the code coverage report.
- `--function-coverage-summary`
  This option works for selecting both the `--function-coverage-summary-per-file` and the `--function-coverage-summary-per-function` options.
- `--statement-coverage-summary-per-file`
  When this option is specified, `coverage` includes the per-file statement coverage summary table in the code coverage report.
- `--statement-coverage-summary-per-function`
  When this option is specified, `coverage` includes the per-function statement coverage summary table in the code coverage report.
- `--statement-coverage-summary`
  This option works for selecting both the `--statement-coverage-summary-per-file` and the `--statement-coverage-summary-per-function` options.
- `--branch-coverage-summary-per-file`
  When this option is specified, `coverage` includes the per-file branch coverage summary table in the code coverage report.
- `--branch-coverage-summary-per-function`
  When this option is specified, `coverage` includes the per-function branch coverage summary table in the code coverage report.
- `--branch-coverage-summary`
  This option works for selecting both the `--branch-coverage-summary-per-file` and the `--branch-coverage-summary-per-function` options.

- **Per-Report Options — Function Selection**
  - `-F` *PERCENT* or `--func-cov-per-file-threshold` *PERCENT*
    When this option is specified, `coverage` selects all functions in source files with function coverage less than the real number *PERCENT*, of which the value starts from 0 and is allowed to be greater than 100. The value of the *PERCENT* defaults to 101. The lines in the selected function are marked in the detail section of the report.
  - `-f` *PERCENT* or `--func-cov-per-func-threshold` *PERCENT*
    When this option is specified, `coverage` selects all functions with function coverage less than the real number *PERCENT*, of which the value starts from 0 and is allowed to be greater than 100. The value of the *PERCENT* defaults to 101. The lines in the selected function are marked in the detail section of the report.
  - `-S` *PERCENT* or `--state-cov-per-file-threshold` *PERCENT*
    When this option is specified, `coverage` selects functions in source files with statement coverage less than the real number *PERCENT*, of which the value starts from 0 and is allowed to be greater than 100. The value of the *PERCENT* defaults to 101. The lines in the selected function are marked in the detail section of the report.

- `-s` *PERCENT* or `--state-cov-per-func-threshold` *PERCENT*

  When this option is specified, `coverage` selects all functions with statement coverage less than the real number *PERCENT*, of which the value starts from 0 and is allowed to be greater than 100. The value of the *PERCENT* defaults to 101. The lines in the selected function are marked in the detail section of the report.

- `-B` *PERCENT* or `--branch-cov-per-file-threshold` *PERCENT*

  When this option is specified, `coverage` selects all functions in source files with branch coverage less than the real number *PERCENT*, of which the value starts from 0 and is allowed to be greater than 100. The value of the *PERCENT* defaults to 101. The lines in the selected function are marked in the detail section of the report.

- `-b` *PERCENT* or `--branch-cov-per-func-threshold` *PERCENT*

  When this option is specified, `coverage` selects all functions with branch coverage less than the real number *PERCENT*, of which the value starts from 0 and is allowed to be greater than 100. The value of the *PERCENT* defaults to 101. The lines in the selected function are marked in the detail section of the report.

3.5.2 Coverage Usage Examples

In Section 3.4.2, we presented two examples of how to use the GRG library APIs to generate the code coverage reports for the code in `template.cpp`. This section will show the corresponding commands on how to use the `coverage` program to match the earlier examples. The two reports generated by running `coverage` are the same as the reports generated by using the GRG library APIs and are included in Appendices A and B. To use the `coverage` program to generate the two reports, we do the following:

1. Run `coverage` to generate the file `full_report.pdf` containing all the files and functions with default option settings:

```
./coverage --function-coverage-summary \
--statement-coverage-summary --branch-coverage-summary \
--detail --output full_report.pdf template.gcda
```

2. Run `coverage` to generate the file `specified_report.pdf` with specified file-function pattern and coverage thresholds:

```
./coverage --file-pattern .*template.cpp --pattern keep \
--aggregate-templates \
--function-coverage-summary \
--statement-coverage-summary \
--branch-coverage-summary --detail \
--state-cov-per-func-threshold 100 \
--branch-cov-per-func-threshold 100 \
--output specified_report.pdf template.gcda
```

Next, we will present a comparatively complicated example of how to use the `coverage` to generate more than one report at one time. Suppose we want to produce multiple reports with different user settings from the two gcda files: `app1.gcda` and `app2.gcda`. This can be accomplished with the invocation of the

program `coverage`. We use the `--reset` option to restore the per-report settings for each report. The first report generated is called `report1.pdf` and this report:

- aggregates all of the associated constructors, destructors, and templates with different mangled names;
- calculates the branch coverage without considering exceptional branches;
- keeps the functions that match with the respective file and function patterns `src/app/.*.hpp` and `ra::(.*::)::intrusive_list<.*>::.*`; and
- includes only branch coverage summary tables.

The second report is named `report2.pdf` and this report:

- includes the function and statement coverage summary tables and the detail section;
- sets the threshold for function coverage per file to 50%;
- sets the threshold for statement coverage per function to 100%.

The commands that utilize the `coverage` to generate reports `report1.pdf` and `report2.pdf` are as follows:

```
./coverage --aggregate-templates \
--aggregate-constructors-destructors \
--ignore-exceptional-branches \
--file-pattern src/app/.*.hpp \
--function-pattern ra::(.*::)::intrusive_list<.*>::.* \
--pattern keep \
--branch-coverage-summary \
--output report1.pdf \
--reset \
--function-coverage-summary \
--statement-coverage-summary \
--func-cov-per-file-threshold 50 \
--state-cov-per-func-threshold 100 \
--detail \
--output report2.pdf \
app1.gcda app2.gcda
```

Chapter 4

Conclusions and Future Work

4.1 Conclusions

In this report, the importance of software testing and the criteria of structural coverage analysis have been studied. As a useful tool for testing code coverage for C++ code designs, the author has developed the GRG software based on Gcov. The GRG software consists of a library for generating customized nicely-formatted code coverage reports in PDF format and an application program called `coverage` to use this library. The code coverage reports generated contain summary statistics of function, statement, and branch coverage on a per-file and per-function basis and detailed information for source code, such as execution counts of lines and blocks. Moreover, the GRG software provides rich functionalities, including filtering functions using regular expressions, calculating the branch coverage with or without exceptional branches, selecting the report contents by setting coverage threshold, and aggregating associated functions with different mangled names. Finally, how to use the GRG library and the `coverage` program to generate multiple code coverage reports has been presented with code examples. The code coverage reports generated for code examples are included in appendices.

4.2 Future Work

Although the GRG software currently works quite well, there is still potential work worth exploring in the future. For example, besides generating code coverage reports in PDF format, the GRG software could allow users to output coverage statistics into a plain text file. It can be time-consuming to run LaTeX on large files, and users may not need PDF output in all cases. Despite the GRG having provided filtering and selecting options for users to specify the exact file and function patterns, all the code coverage statistics and detailed information for the report in PDF format can be quite large sometimes.

In addition, as mentioned in Section 2.4, the compiler uses mangled names when compiling and linking. The GRG software, however, currently only supports demangled function names when presenting the code coverage reports because demangled names are more human-readable and understandable. In some cases, testers may want to know mangled function names to differentiate the version of a constructor or destructor called. Also, the memory space and the execution time for running the LaTeX file to PDF file can be reduced as mangled names are usually shorter than demangled names, especially for functions with long names.

Appendix A

Code Coverage Report: `full_report.pdf`

A.1 Overview

This chapter presents the code coverage report called `full_report.pdf` generated by the GRG software for the file `template.cpp`. The `full_report.pdf` includes all the files and functions and did not use any filter options.

1 Summary

1.1 Function Coverage

1.1.1 Per-File Function Coverage

Coverage	Pathname
5/5 (100.00%)	/home/mint/Coverage/tests/template.cpp
4/4 (100.00%)	/usr/include/c++/10/bits/move.h

1.1.2 Per-Function Function Coverage

Coverage	Pathname
1/1 (100.00%)	[/home/mint/Coverage/tests/template.cpp] void bubbleSort<double>(double*, int)
1/1 (100.00%)	[/home/mint/Coverage/tests/template.cpp] void bubbleSort<int>(int*, int)
1/1 (100.00%)	[/home/mint/Coverage/tests/template.cpp] sum(int, int)
1/1 (100.00%)	[/home/mint/Coverage/tests/template.cpp] sum(double, double)
1/1 (100.00%)	[/home/mint/Coverage/tests/template.cpp] main
12/12 (100.00%)	[/usr/include/c++/10/bits/move.h] std::remove_reference<double&>::type&& std ::move<double&>(double&)
15/15 (100.00%)	[/usr/include/c++/10/bits/move.h] std::remove_reference<int&>::type&& std:: move<int&>(int&)
4/4 (100.00%)	[/usr/include/c++/10/bits/move.h] std::enable_if<std::__and_<std::__not_< std::__is_tuple_like<double> >, std::is_move_constructible<double>, std:: is_move_assignable<double> >::value, void>::type std::swap<double>(double&, double&)
5/5 (100.00%)	[/usr/include/c++/10/bits/move.h] std::enable_if<std::__and_<std::__not_ <std::__is_tuple_like<int> >, std::is_move_constructible<int>, std:: is_move_assignable<int> >::value, void>::type std::swap<int>(int&, int&)

1.2 Statement Coverage

1.2.1 Per-File Statement Coverage

Coverage	Pathname
33/33 (100.00%)	/home/mint/Coverage/tests/template.cpp
12/12 (100.00%)	/usr/include/c++/10/bits/move.h

1.2.2 Per-Function Statement Coverage

Coverage	Pathname
9/9 (100.00%)	[/home/mint/Coverage/tests/template.cpp] void bubbleSort<double>(double*, int)
9/9 (100.00%)	[/home/mint/Coverage/tests/template.cpp] void bubbleSort<int>(int*, int)
2/2 (100.00%)	[/home/mint/Coverage/tests/template.cpp] sum(int, int)
2/2 (100.00%)	[/home/mint/Coverage/tests/template.cpp] sum(double, double)
11/11 (100.00%)	[/home/mint/Coverage/tests/template.cpp] main
2/2 (100.00%)	[/usr/include/c++/10/bits/move.h] std::remove_reference<double&>::type&& std ::move<double&>(double&)
2/2 (100.00%)	[/usr/include/c++/10/bits/move.h] std::remove_reference<int&>::type&& std:: move<int&>(int&)

Continued from previous page.	
Coverage	Pathname
4/4 (100.00%)	[/usr/include/c++/10/bits/move.h] std::enable_if<std::__and_<std::__not_<std::__is_tuple_like<double> >, std::is_move_constructible<double>, std::is_move_assignable<double> >::value, void>::type std::swap<double>(double&, double&)
4/4 (100.00%)	[/usr/include/c++/10/bits/move.h] std::enable_if<std::__and_<std::__not_<std::__is_tuple_like<int> >, std::is_move_constructible<int>, std::is_move_assignable<int> >::value, void>::type std::swap<int>(int&, int&)

1.3 Branch Coverage

1.3.1 Per-File Branch Coverage

Coverage	Pathname
16/20 (80.00%)	/home/mint/Coverage/tests/template.cpp
0/0 (100.00%)	/usr/include/c++/10/bits/move.h

1.3.2 Per-Function Branch Coverage

Coverage	Pathname
4/8 (50.00%)	[/home/mint/Coverage/tests/template.cpp] main
6/6 (100.00%)	[/home/mint/Coverage/tests/template.cpp] void bubbleSort<double>(double*, int)
6/6 (100.00%)	[/home/mint/Coverage/tests/template.cpp] void bubbleSort<int>(int*, int)
0/0 (100.00%)	[/home/mint/Coverage/tests/template.cpp] sum(int, int)
0/0 (100.00%)	[/home/mint/Coverage/tests/template.cpp] sum(double, double)
0/0 (100.00%)	[/usr/include/c++/10/bits/move.h] std::remove_reference<double&>::type&& std::move<double&>(double&)
0/0 (100.00%)	[/usr/include/c++/10/bits/move.h] std::remove_reference<int&>::type&& std::move<int&>(int&)
0/0 (100.00%)	[/usr/include/c++/10/bits/move.h] std::enable_if<std::__and_<std::__not_<std::__is_tuple_like<double> >, std::is_move_constructible<double>, std::is_move_assignable<double> >::value, void>::type std::swap<double>(double&, double&)
0/0 (100.00%)	[/usr/include/c++/10/bits/move.h] std::enable_if<std::__and_<std::__not_<std::__is_tuple_like<int> >, std::is_move_constructible<int>, std::is_move_assignable<int> >::value, void>::type std::swap<int>(int&, int&)

2 Details

2.1 /home/mint/Coverage/tests/template.cpp

`void bubbleSort<double>(double*, int)`

Line	Counts	Branches	Source
6	1		void bubbleSort(T a[], int n){
7	5	4:1	for (int i = 0; i < n - 1; i++){
8	14	10:4	for (int j = n - 1; i < j; j--){
9	10	4:6	if (a[j] < a[j - 1]){
10	4		std::swap(a[j], a[j - 1]);
11			}
12			}
13			}
14			
15	1		}

void bubbleSort<int>(int*, int)

Line	Counts	Branches	Source
6	1		void bubbleSort(T a[], int n){
7	5	4:1	for (int i = 0; i < n - 1; i++){
8	14	10:4	for (int j = n - 1; i < j; j--){
9	10	5:5	if (a[j] < a[j - 1]){
10	5		std::swap(a[j], a[j - 1]);
11			}
12			}
13			}
14			
15	1		}

sum(int, int)

Line	Counts	Branches	Source
18	1		int sum(int a, int b){
19	1		return a + b;
20			}

sum(double, double)

Line	Counts	Branches	Source
22	1		double sum(double a, double b){
23	1		return a + b;
24			}

main

Line	Counts	Branches	Source
27	1		int main(){
28			
29			// Calls template function
30	1		int a1[5] = {3, 5, 1, 2, 4};
31	1		double a2[5] = {2.3, 5.6, 1.1, 2.2, 10.0};
32	1		bubbleSort<int>(a1, std::extent<int[5]>::value);
33	1		bubbleSort<double>(a2, std::extent<int[5]>::value);
34			
35			// Calls overload functions and outputs the results
36	1		int a = 1;
37	1		int b =2;
38	1		double c = 1.2;
39	1		double d = 2.4;
40	1	1:0:1:0	std::cout << sum(a,b)<< '\n';
41	1	1:0:1:0	std::cout << sum(c,d)<< '\n';
42			
43	1		return 0;
44			}

2.2 /usr/include/c++/10/bits/move.h

std::remove_reference<double&>::type&& std::move<double&>(double&)

Line	Counts	Branches	Source
101	12		move(_Tp&& __t)noexcept
102	12		{ return static_cast<typename std::remove_reference<_Tp>::type&&>(__t); }

std::remove_reference<int&>::type&& std::move<int&>(int&)

Line	Counts	Branches	Source
101	15		move(_Tp&& __t)noexcept
102	15		{ return static_cast<typename std::remove_reference<_Tp>::type&&>(__t); }

std::enable_if<std::__and_<std::__not_<std::__is_tuple_like<double> >, std::is_move_constructible<double>, std::is_move_assignable<double> >::value, void>::type std::swap<double>(double&, double&)

Line	Counts	Branches	Source
189	4		swap(_Tp& __a, _Tp& __b)
190			_GLIBCXX_NOEXCEPT_IF(__and_<is_nothrow_move_constructible<_Tp>,
191			is_nothrow_move_assignable<_Tp>>::value)
192			{
193			#if __cplusplus < 201103L
194			// concept requirements
195			__glibcxx_function_requires(_SGIAssignableConcept<_Tp>)
196			#endif
197	4		_Tp __tmp = _GLIBCXX_MOVE(__a);
198	4		__a = _GLIBCXX_MOVE(__b);
199	4		__b = _GLIBCXX_MOVE(__tmp);
200	4		}

std::enable_if<std::__and_<std::__not_<std::__is_tuple_like<int> >, std::is_move_constructible<int>, std::is_move_assignable<int> >::value, void>::type std::swap<int>(int&, int&)

Line	Counts	Branches	Source
189	5		swap(_Tp& __a, _Tp& __b)
190			_GLIBCXX_NOEXCEPT_IF(__and_<is_nothrow_move_constructible<_Tp>,
191			is_nothrow_move_assignable<_Tp>>::value)
192			{
193			#if __cplusplus < 201103L
194			// concept requirements
195			__glibcxx_function_requires(_SGIAssignableConcept<_Tp>)
196			#endif
197	5		_Tp __tmp = _GLIBCXX_MOVE(__a);
198	5		__a = _GLIBCXX_MOVE(__b);
199	5		__b = _GLIBCXX_MOVE(__tmp);
200	5		}

Appendix B

Code Coverage Report: `specified_report.pdf`

B.1 Overview

This chapter presents the code coverage report called `specified_report.pdf` generated by the GRG software for the file `template.cpp`. The `specified_report.pdf` applied some filter options and only include the functions belong to `template.cpp`.

1 Summary

1.1 Function Coverage

1.1.1 Per-File Function Coverage

Coverage	Pathname
4/4 (100.00%)	/home/mint/Coverage/tests/template.cpp

1.1.2 Per-Function Function Coverage

Coverage	Pathname
2/2 (100.00%)	[/home/mint/Coverage/tests/template.cpp] void bubbleSort<double>(double*, int)
1/1 (100.00%)	[/home/mint/Coverage/tests/template.cpp] sum(int, int)
1/1 (100.00%)	[/home/mint/Coverage/tests/template.cpp] sum(double, double)
1/1 (100.00%)	[/home/mint/Coverage/tests/template.cpp] main

1.2 Statement Coverage

1.2.1 Per-File Statement Coverage

Coverage	Pathname
24/24 (100.00%)	/home/mint/Coverage/tests/template.cpp

1.2.2 Per-Function Statement Coverage

1.3 Branch Coverage

1.3.1 Per-File Branch Coverage

Coverage	Pathname
10/14 (71.43%)	/home/mint/Coverage/tests/template.cpp

1.3.2 Per-Function Branch Coverage

Coverage	Pathname
4/8 (50.00%)	[/home/mint/Coverage/tests/template.cpp] main

2 Details

2.1 /home/mint/Coverage/tests/template.cpp

`void bubbleSort<double>(double*, int)`

Line	Counts	Branches	Source
6	2		void bubbleSort(T a[], int n){
7	10	8:2	for (int i = 0; i < n - 1; i++){
8	28	20:8	for (int j = n - 1; i < j; j--){
9	20	9:11	if (a[j] < a[j - 1]){
10	9		std::swap(a[j], a[j - 1]);
11			}
12			}
13			}
14			
15	2		}

sum(int, int)

Line	Counts	Branches	Source
18	1		int sum(int a, int b){
19	1		return a + b;
20			}

sum(double, double)

Line	Counts	Branches	Source
22	1		double sum(double a, double b){
23	1		return a + b;
24			}

main

Line	Counts	Branches	Source
27	1		int main(){
28			
29			// Calls template function
30	1		int a1[5] = {3, 5, 1, 2, 4};
31	1		double a2[5] = {2.3, 5.6, 1.1, 2.2, 10.0};
32	1		bubbleSort<int>(a1, std::extent<int[5]>::value);
33	1		bubbleSort<double>(a2, std::extent<int[5]>::value);
34			
35			// Calls overload functions and outputs the results
36	1		int a = 1;
37	1		int b =2;
38	1		double c = 1.2;
39	1		double d = 2.4;
40	1	1:0:1:0	std::cout << sum(a,b)<< '\n';
41	1	1:0:1:0	std::cout << sum(c,d)<< '\n';
42			
43	1		return 0;
44			}

Bibliography

[1] Michael D. Adams. Lecture Slides for Programming in C++, April 2021. https://ece.engr.uvic.ca/~frodo/publications.html#books.

[2] Frances E. Allen. Control Flow Analysis. ACM SIGPLAN Notices, Volume 5, Issue 7, ACM Digital Library, NY, US, 1970.

[3] FPaul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, UK, 2016.

[4] Arthur G. Stephenson and Lia S. LaPiana and Daniel R. Mulville and Peter J. Rutledge and etc. Mars Climate Orbiter Mishap Investigation Board Phase I Report, November 1999. NASA.

[5] Brader Larry and Hilliker Howie and Wills Alan. Testing for continuous delivery with visual studio 2012. https://docs.microsoft.com/en-us/previous-versions/msp-n-p/jj159345(v=pandp.10). Microsoft Corporation, Published online on March 2013; accessed November 2021.

[6] Clang Static Analyzer, a source code analysis tool for finding bugs in C, C++, and Objective-C programs. The clang team. https://clang-analyzer.llvm.org/. Published online; accessed November 2021.

[7] CMake. Kitware Inc. https://cmake.org/. Published online; accessed November 2021.

[8] I. Evans D. Graham, E. Van Veenendaal and R. Black. *Foundations of Software Testing*. International Thomson Business Press, Haryana, India, 2019.

[9] M. Dowson. The Ariane 5 Software Failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):87, March 1997.

[10] GCC, the GNU compiler collection. Free Software Foundation, Inc. https://gcc.gnu.org/. Published online; accessed November 2021.

[11] Gcov, a test coverage program. Free Software Foundation, Inc. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html. Published online; accessed November 2021.

[12] Jézéquel JM and Meyer B. Design by Contract: The Lessons of Ariane. *IEEE Computer*, 30(1):129–130, January 1997.

[13] JSON, ECMA-404 The JSON Data Interchange Standard. ECMA International. https://www.json.org/json-en.html. Published online; accessed November 2021.

[14] Williams Laurie, Smith Ben, and Heckman Sarah. Test Coverage with EclEmma. Open seminar software engineering, North Carolina State University, Raleigh, NC, USA, 2016.

[15] Lcov, a graphical front-end for GCC's coverage testing tool Gcov. Sourceforge. http://ltp.sourceforge.net/coverage/lcov.php. Published online; accessed November 2021.

[16] Niels Lohmann. JSON for Modern C++ Library. https://github.com/nlohmann/json. Published online; accessed November 2021.

[17] Egele Manuel, Scholte Theodoor, Kirda Engin, and Kruegel Christopher. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys*, 44(2):2, March 2008.

[18] Mark Mitchell. C++ ABI for IA-64: Code and Implementation Examples. https://itanium-cxx-abi.github.io/cxx-abi/abi-examples.html#mangling. Published online; accessed November 2021.

[19] Klemens David Morgenstern. Boost.Process Library. https://www.boost.org/doc/libs/1_65_1/doc/html/process.html. Published online; accessed November 2021.

[20] Joaquin M. Lopez Munoz. Class template stable vector in Boost library. https://www.boost.org/doc/libs/1_53_0/doc/html/boost/container/stable_vector.html. Published online; accessed November 2021.

[21] Glenford J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. Wiley, the 2 edition, NJ, USA, 2004.

[22] Institute of Electrical and Electronics Engineers. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE, NJ, USA, 1990.

[23] Open-source project. Itanium C++ ABI. https://itanium-cxx-abi.github.io/cxx-abi/abi.html. Published online; accessed November 2021.

[24] Bhatt C. Patel P. and Talati D. Structural coverage analysis with DO-178B standards. Conference paper, International Conference on Advanced Computing Networking and Informatics, 2019.

[25] pdfTeX. PDF producer from TeX source. https://www.tug.org/applications/pdftex/. Published online; accessed November 2021.

[26] Portable Operating System Interface (POSIX) standards, POSIX.1-2017. Regular Expressions. https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html.

[27] Scan. Coverity scan static analysis tool. https://scan.coverity.com/. Published online; accessed November 2021.

[28] Sten Pittet. An introduction to code coverage. https://www.atlassian.com/continuous-delivery/software-testing/code-coverage. Published online; accessed November 2021.

[29] The GNU C++ Library Manual Chapter 28. Demangling. Free Software Foundation, Inc. https://gcc.gnu.org/onlinedocs/libstdc++/manual/ext_demangling.html. Published online; accessed November 2021.

[30] The GNU C++ Library Manual Chapter 3.11 Program Instrumentation Options. Free Software Foundation, Inc. https://gcc.gnu.org/onlinedocs/gcc-9.3.0/gcc/Instrumentation-Options.html#Instrumentation-Options. Published online; accessed November 2021.

[31] James A. Whittaker. *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*. Addison-Wesley Professional, Boston, USA, 2009.

[32] William Hart, Lukas Atkinson, and Michael Forderer, and etc. Gcovr, a report generator for GCC code coverage. https://gcovr.com/en/stable/. Published online; accessed November 2021.