NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS & TELECOMMUNICATIONS

FINAL THESIS

Code Coverage Aid

Orestes Triantafyllos

Advisor : **Dr. Ioannis Chamodrakas,**
Member of the Instructional Lab Personnel

under the supervision of Prof. Panagiotis Stamatopoulos

ATHENS

JULY 2016

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Code Coverage Aid

Ορέστης Γ. Τριαντάφυλλος

Επιβλέπων : **Δρ. Ιωάννης Χαμόδρακας,**
Μέλος Εργαστηριακού Διδακτικού Προσωπικού

υπό την εποπτεία του καθηγητή κ. Παναγιώτη Σταματόπουλου

ΑΘΗΝΑ

ΙΟΥΝΙΟΣ 2016

FINAL THESIS

Code Coverage Aid

Orestes Triantafyllos
1115 2009 00277

Advisors : **Dr. Ioannis Chamodrakas,**
Member of the Instructional Lab Personnel

under the supervision of Prof. Panagiotis Stamatopoulos

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Code Coverage Aid

Ορέστης Γ. Τριαντάφυλλος
Α.Μ.: 1115 2009 00277

Επιβλέποντες : **Δρ. Ιωάννης Χαμόδρακας,**
Μέλος Εργαστηριακού Διδακτικού Προσωπικού

υπό την εποπτεία του καθηγητή κ. Παναγιώτη Σταματόπουλου

ABSTRACT

In this thesis we address the issue of **code coverage**, one of the most important issues of software testing, which describes the degree to which the source code of a program is tested. We compile a review on the most common techniques to perform code coverage analysis and we compare their pros and cons. Following a survey of several tools available for measuring the code coverage, and their characteristics, we present the design of the Code Coverage Aid tool (CCA) for C++ programs, developed by the author, to assist software developers and testers in the measuring of the amount of code coverage. Code Coverage Aid uses the most powerful technique, source Instrumentation, to instrument the source code upon compilation by placing a function call on every branch decision and allows to keep track of the lines of code that were executed during each run. CCA provides two sets of results; the statistics of the latest run, as well as cumulative statistics of all tests performed on the program so far. The user can review the code coverage results through a graphical interface and easily spot the areas (functions or code blocks) that were not executed. Based on this information, the user can decide to design additional tests to cover the code blocks indicated by CCA as ill-tested.

Written in portable C++, and licensed under GNU EULA, CCA may be easily become a routine task of any C++ development team, without additional cost or development effort.

SUBJECT AREA: Code Coverage

KEYWORDS: software testing, code coverage, source instrumentation

ΠΕΡΙΛΗΨΗ

Το θέμα της παρούσας πτυχιακής εργασίας είναι η ανάπτυξη ενός εργαλείου για την μέτρηση της κάλυψης κώδικα (code coverage), ένα από τα σημαντικότερα ζητήματα που αφορούν τη διαδικασία ελέγχου-δοκιμής του λογισμικού. Το εργαλείο αυτό υπολογίζει το ποσοστό του εκτελεσμένου κώδικα ενός προγράμματος έπειτα από μια σειρά δοκιμών. Περιγράφουμε και συγκρίνουμε τις πιο διαδεδομένες τεχνικές για την πραγματοποίηση της μέτρησης του code coverage. Έπειτα από μία σύντομη ανασκόπηση των διαθέσιμων εργαλείων για αυτή τη μέτρηση, παρουσιάζουμε την σχεδίαση του εργαλείου Code Coverage Aid (CCA) που δημιουργήθηκε από τον συγγραφέα και λειτουργεί σε προγράμματα γραμμένα σε C++. Το CCA χρησιμοποιεί την πιο ισχυρή τεχνική για την μέτρηση της κάλυψης κώδικα που ονομάζεται Source Instrumentation, η οποία ενεργεί πάνω στον πηγαίο κώδικα του προγράμματος πριν την μεταγλώττιση τοποθετώντας μία κλήση συνάρτησης σε κάθε διακλάδωση της ροής εκτέλεσης. Η τεχνική αυτή παρέχει έτσι τη δυνατότητα να παρακολουθεί ο προγραμματιστής ποιες γραμμές κώδικα εκτελέστηκαν σε κάθε εκτέλεση του προγράμματος. Το CCA παρέχει δυο σύνολα αποτελεσμάτων: τα στατιστικά της τελευταίας εκτέλεσης και τα συνολικά στατιστικά όλων των εκτελέσεων. Ο χρήστης μπορεί να επεξεργαστεί τα στατιστικά μέσω ενός γραφικού περιβάλοντος διεπαφής χρήστη και να αναγνωρίσει εύκολα τα κομμάτια του κώδικα που δεν έχουν εκτελεστεί. Τέλος, συγκρίνουμε το CCA με το εργαλείο GCOV που είναι ελεύθερα διαθέσιμο και παρουσιάζουμε τα αποτελέσματα της εν λόγω σύγκρισης.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Κάλυψη Κώδικα

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: έλεγχος-δοκιμή λογισμικού, κάλυψη κώδικα, μεταποίηση πηγαίου κώδικα

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor Dr. Ioannis Chamodrakas for his time, effort, valuable guidance, advice and criticism devoted to this thesis.

I must also express my gratitude towards the supervisor of my thesis Professor Panagiotis Stamatopoulos for his kind help.

Furthermore, I would like to thank all my professors at the Department of Informatics and Telecommunications for providing me with all the knowledge and tools to start my career.

Finally, my friends and family for all their support and guidance throughout my education.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Software testing is a very important and integral part of the software development process, aiming at providing the stakeholders with information about the quality of the software being developed. Its importance for software development can be seen from the enormous amount of work being done the past four decades in order to develop tools and techniques to improve its effect on the development process. Such is the interest of the scientific community, that today software testing dominates the discipline of software engineering, and tends to become a sub-discipline of its own.

This is not surprising. Virtually every person who has some contact with computer software is today a witness of a sort of computer failure, commonly referred to as bug. Numerous reports in the respective literature talk about insufficient and ad-hoc testing of released software. Others are trying to measure the remaining bugs in a software system. Numbers are astonishing. Working software today contains anything from 0.5 defects per 1000 lines of code (KLOC), to 50-60 defects per KLOC, even more, as reported by Steve McConnell in his book "*Code Complete*" [1].

(a) "Industry Average: "about 15 - 50 errors per 1000 lines of delivered code." He further says this is usually representative of code that has some level of structured programming behind it, but probably includes a mix of coding techniques."

(b) "Microsoft Applications: "about 10 - 20 defects per 1000 lines of code during in-house testing, and 0.5 defect per KLOC (KLOC IS CALLED AS 1000 lines of code) in released product (Moore 1992)." He attributes this to a combination of code-reading techniques and independent testing (discussed further in another chapter of his book)."

(c) "Harlan Mills pioneered 'cleanroom development', a technique that has been able to achieve rates as low as 3 defects per 1000 lines of code during in-house testing and 0.1 defect per 1000 lines of code in released product(Cobb and Mills 1990). A few projects - for example, the space-shuttle software - have achieved a level of 0 defects in 500,000 lines of code using a system of format development methods, peer reviews, and statistical testing."

Irrespective of the exact number, the fact remains today that released software contains many errors that can lead to computer failures. This is despite the fact that, according to Glenford J. Myers [2], software testing efforts take approximately 50 percent of the total cost and time devoted to the development of a software system.

The software testing is the software development process that inspects the produced code to determine whether the code meets the intended purpose. In other words, software testing is the vehicle to achieve software quality. Broadly speaking, it ensures the produced system adheres to the specifications (provides the functionality specified by the stakeholders) and to the desired quality (the developed functionality produces correct results). The software testing does not prove that the system is free of errors, but rather it demonstrates that if there are errors in the program these errors cannot lead to bugs when the program executes under the specific conditions (inputs) of a particular test case. Therefore, the task of measuring the quality of a program becomes a task of demonstrating that the program behaves correctly when executing under the control of a particular test suite; a set of test cases that exercise particular functionality

of the program. Obviously, the bigger and the more sophisticated the test suite is, the better the obtained results (less errors remain hidden in the code).

The software testing is classified into several [3] categories depending on

- the time (when during the development process it occurs),
- the purpose (what in particular is the point of interest), and
- the user (which test body performs the test)

A short survey of testing strategies and methodologies is presented in section 3.

The degree to which software testing is successful is measured by the software quality of the end product. The software quality of a product is defined as "*conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected to all professionally developed software.*" [4]

There are numerous other, but similar, definitions of software quality in the literature. All however "suffer" from the same characteristic. They lack the ability to be objectively measured. Philip Crosby, in his landmark book "*Quality is free*" [5], defines quality as "*the conformance to the requirements"* and argues that quality can be measured by "*the cost of non-conformance or of doing things wrong"*.

Much of the software engineering discipline is, therefore, devoted on how to achieve certain degree of software quality, via programing and testing practices. Measuring the quality is also a vital issue in this area. Not surprisingly, there are numerous quality measure metrics have been proposed in the literature.

One particular aspect of the software quality, which is the subject of this thesis, is the aspect of code coverage. It is one of the first methods invented to assist software testing, dated back in the early '60s. The code coverage (and code coverage metrics) is an attempt to investigate and measure the degree to which a program has been exercised (tested) by a particular test suite. In other words it attempts to look into the internal structures of a program and determine how much of the code has actually been used in a test. Code coverage tells the tester what percentage of the entire set of code has been executed, at least once, during any kind of testing. The above statement implies that: a) it is very likely that parts of a released program may have never been executed during testing and therefore it is likely that they contain errors to be found in the field, b) unless code coverage techniques are incorporated into the testing process, we have little evidence as to which parts of the code we are exerting during testing.

Measuring test coverage is often overlooked by testing development and testing groups, especially todays' that time-to-marker under severely constrained budgets is a norm. An additional issue that worsens the problem is that measuring the code coverage requires specialized tools that must be used thought the whole development cycle.

Though many such tools exist today, as it will be argued in this thesis, there is still need to develop more and hopefully better tools.

1.1 Framework of this thesis

This thesis is organized as follows:

- Chapter 2 presents the rationale of our thesis proposal.
- Chapter 3 presents a short introduction to the test categories and types used in the industry today for the verification of software.

- Chapter 4 talks about the code coverage techniques, which is technical information on how code coverage tools are able to find out the execution path of a program.
- Chapter 5 presents a concise survey of the existing code coverage tools
- Chapter 6 presents the design and the implementation details on our code coverage tool, named Code Coverage Aid, which the main topic of this thesis.
- Chapter 7 deals with testing aspects of CCA, and
- Chapter 8 presents some concluding remarks on this work.

2. PROPOSAL DESCRIPTION

As explained in the introduction, code coverage [cc] is a very important software quality metric that can be used to guide the developers and testers regarding the areas of code that need to shift their attention.

Nevertheless, there are several reports today [6] suggesting this metric is rarely used. When it does, it is used in organizations that produce safety critical systems, or when required by law to implement rigorous quality assurance processes that include code coverage. We can speculate on the reasons of not using this metric, some of which are the availability of flexible enough tools, the extra time needed to setup, collect and analyze the cc info, as well as budget constraints.

It is evident that if code coverage information is readily available to the software developers, it will greatly enhance the task of software testing and consequently the quality of the end product. Obviously, if a flexible enough tool exists, developers will embrace it into their routine testing activities. This is the main driving force behind our decision to design and develop yet another code coverage tool, despite the rather extensive set of such tools one may find on the internet today. It is true that many such tools are really state-of-the-art and have been around for quite some time (as discussed in chapter 5), but is also true that good systems are usually part of big development systems/platforms, provide “the whole world”, and are rarely free.

The subject of this thesis is, therefore, the design and implementation of a code coverage tool, called **Code Coverage Aid** (CCA) for C/C++ programs which is:

1) Focused on target - measure code coverage for C/C++
2) Easy to use - no complex installation, integratable with most popular automation tools (make program)
3) Flexible - no action by the user, other than initial setup of make file
4) Independent – no support system. Use it anywhere (in the lab or on the field)
5) Free – source/binary available on the internet.

Side effects of this project is the opportunity to:

1. Develop a rather complex parser for C++ programs, and thus advance my programming skills, in the area of parsers, compilers,
2. Develop a complete and useful, software system.

3. SOFTWARE TESTING STRATEGIES

The importance of software testing and its impact on the quality of the end product cannot be emphasized enough [4]. It is critical to software quality and represents the ultimate review of software specification, design and implementation. The rigorous and systematic approach to testing can lead to superior software quality and reduce the overall cost of it. Development groups must devote sufficient time and budget to derive and execute the right testing strategy for the system under development.

The software testing literature is very rich with methodologies, strategies, metrics, and best practices, as to how to go about deriving a test plan that incorporates all the essential elements of testing.

In this section we present some key elements of software testing.

3.1 Testing methodologies

Testing methodologies refers to strategies and methodologies used to test a software system to ensure it fits its purpose. Testing methodologies involve tests to ensure the system adheres to the specifications, and does not possess undesirable side effects (i.e. bugs).

Software testing methodologies include a variety of tests depicted in Figure 1, and discussed briefly in this section, as well as sections 3.2 and 3.3

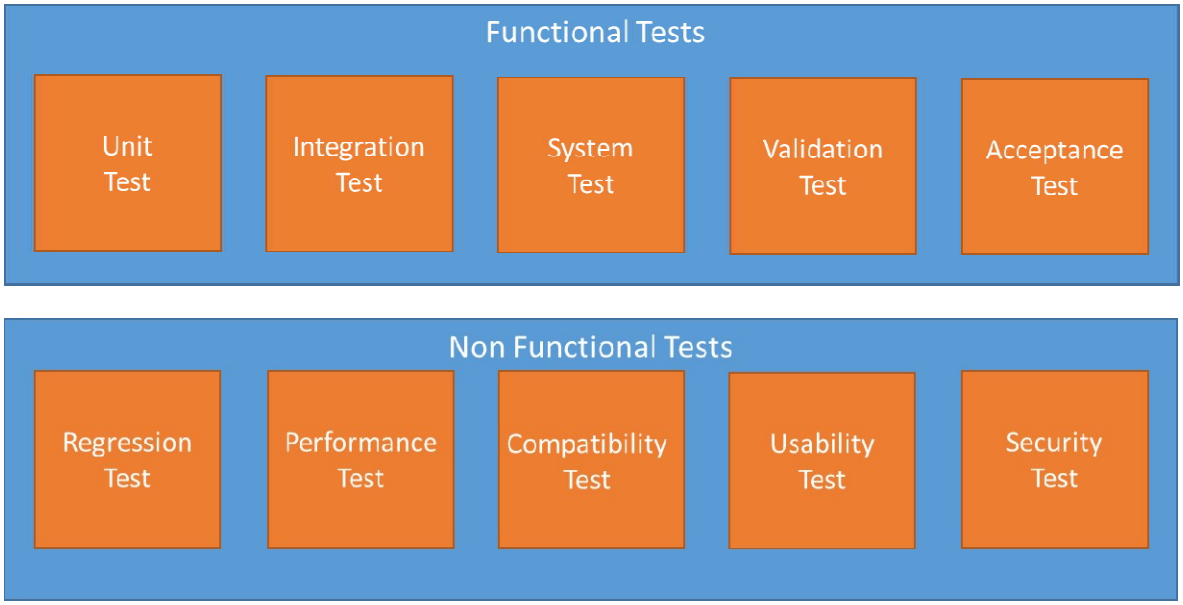

Figure 1: Software Testing Methodologies

As the goal of finding all the errors in a computer program is almost impossible, even for trivial programs (one must prove that the program behaves correctly and to specifications on every possible set of inputs, which can be billions), one must derive clever testing strategies that allow the tester to obtain a degree of confidence that a program is “bug-free”, using as little testing effort as possible. Such strategies have led to the development of techniques such as the black-box testing and the white-box testing.

3.1.1 Black-box testing

Black-box testing is a test method of testing the functionality of a system, without looking into the details of the internal structures. The tester concentrates on the external characteristics of a system, i.e. functional specifications, and derives tests to exercise these characteristics. This method of test can be applied to virtually every level of software testing: unit, integration, system and acceptance which are presented in sections 3.2 and 3.3.

In this approach, test data are derived from the specifications of the system. To find all the errors in a program using this method, one must use the criterion of *exhaustive input testing*, or to feed the program with every input combination possible. Even in this case the test will show correctness of the program with respect to what is implemented, but will provide no information with respect to what needs to be implemented.

Therefore, the choice of the “right” set of inputs to drive a block-box testing is paramount to testing itself.

3.1.2 White-box testing

White-box is a method of testing software that tests internal structures of an application, as opposed to its functionality (i.e. black-box testing). In white-box testing, testers use high and low level design information to derive appropriate test cases that test particular internal structures of a system. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs. White-box testing can be applied at the unit, integration and system levels of the software testing process.

White-box testing in essence is a test method that must test for all execution paths of a program, or “exhaustive path testing”. Testing however for all paths of a program, has the same difficulties as the testing for all possible inputs in the case of black-box testing. Even for trivial programs the number of paths to consider may be tremendous.

3.2 Testing Types

The term testing types refer to the scope of the testing to be performed at a certain point in the development cycle. That is as development process progresses, so does the testing process. Testing at various places in the development cycle, with different scope and carefully chosen strategies, can lead to discover as many errors as possible.

Notably, the most important kind of tests are:

3.2.1 Unit Testing

According to Wikipedia unit testing is *“a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use. Intuitively, one can view a unit as the smallest testable part of an application. In procedural programming, a unit could be an entire module, but it is more commonly an individual function or procedure. In object-oriented programming, a unit is often an entire interface, such as a class, but could be an individual method. Unit tests are short code fragments created by programmers or occasionally by white box testers during the development process”*

3.2.2 Integration Testing

Integration testing is according to Wikipedia definition "*the phase in software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before validation testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.*"

As the definition suggest, integration testing implies that all interfaces between the different components of a system are tested to verify the exchange of data between them, the communicating units are in accordance with the design, as well as the performance requirements.

Since a system may comprise of multiple units with several interface points, care must be taken as to how the test will be performed. Notably bottom-up, top-down, sandwich test and risky-hardest [7] are some techniques used.

3.2.3 System Testing

A software system is one element only of a larger computer-based system, where eventually it will reside and execute. As such, once software is tested stand alone, via a series of unit, integration, verification, functional tests, it must be tested integrated with the hardware it will eventually run on as well as other systems it will interact. This testing procedure is called system testing. It is a sort of black-box testing, where the focus of the test is to verify the specified system requirements (i.e. integration will external systems, performance). System testing tests the design of the software as well as aspects of the software behavior, and customer expectations.

Disaster recovery tests, security tests, stress and performance tests are some of the tests performed as part of the system test.

3.2.4 Validation Testing

Validation testing begins when the entire system has been developed and passed all tests mentioned above. As such is the last test before it is released to General Availability (GA). Validation testing can be defined in many ways, but it is commonly accepted that validation succeeds when *"software functions in a matter that can be reasonably accepted by customer*". [4] Passing validation testing is therefore, a subjective matter, and frequently a war-zone area between designer and testers.

Validation testing is always performed by Quality Assurance (QA) groups. It checks that the product design satisfies or fits the intended use, and it is done through dynamic testing and other forms of review [6] [4] [8].

Validation testing often involves the end customer into the process. This is done via limited release of the software to selective customers who agree to use the software for this purpose. These customers are feeding information back to QA group. This kind of testing is referred to as Alpha and Beta testing.

3.2.5 Acceptance Testing

Acceptance testing is a set of tests performed by the customer or end user of a system, which is installed on the customer premise, to determine if the system meets the requirements stated on a specification or contract.

Acceptance testing is a formal procedure, often detailed in a contract between the customer and the system provider, by which the customer decides if the system meets the requirements.

It involves several kinds of tests from those discussed in previous sections, such as:

- Functional tests,
- Performance tests,
- Security tests,
- Integration tests (with others systems in the customer premise).

At the end of this procedure, the customer issues a certificate of acceptance by which the system may be deemed:

- Provisionally Accepted,
- Partially Accepted,
- Finally Accepted, or
- Not Accepted

3.3 Other Types of Testing (Non-functional testing)

The types of testing presented in the previous section are performed during different times of the development process, and concentrated in the functionality of the system; making sure the system produces the desired result. Besides this type, several other testing types are also performed, depicted in Figure 1, which are not focused exclusively in the functionality of the system. The most important are briefly presented in this section.

3.3.1 Regression Testing

Regression testing is performed on a system that has undergone changes since it last verification. This can happen in two cases: when the system changes in order to correct a problem or to add new functionality.

The tests performed in this case are a special subset of the entire test suite aiming in discovering potential problems in any part of the system due to changes introduced in a specific part. Due to the frequency of regression test runs, it is a challenge to software developers and testers to select a time efficient/effective set of test cases that perform the desired task.

3.3.2 Performance Testing

Performance testing is executed on a complete system installed on its final hardware configuration (at customer site, or laboratory) and aim at ensuring that the final system performs satisfactory (i.e. stability and responsiveness) under all kinds of stress conditions.

The most important factor to successively execute a performance test is the test conditions or environment (hardware configuration of the system) on which the test will be executed. As each system (software and hardware combination) is designed to service certain kind of “quantities”, performance testing makes sense only if the system is equipped with the final configuration in which will operate.

Once the correct environment is in place, a series of tests are performed to measure certain metrics of the system, such as:

Stability – make sure the system does not collapse under normal or excessive work load,

Responsiveness – make sure the system performs the specified action in the expected amount of time.

When necessary, simulation tools may be involved. For example if the system is designed to service hundreds of users concurrently, a tool may be used to simulate the load imposed on the system by the users.

3.3.3 Compatibility Testing

Compatibility testing, is testing performed on a system to evaluate the system's compatibility with the computing environment. Computing environment contains some or all of the following elements:

Hardware: If the system is design to operate on different platforms, then the system is loaded and tested on each platform.

Operating Systems: If the system is designed to operate under different operating systems then the system in loaded and tested under each of the compatible Oss.

Database: The system may be tested when connected to different databases i.e. Oracle, SQL Server, MySQL, etc.

Browser Compatibility: Test system with different browsers I.e. Internet Explorer, Mozilla, Opera.

3.3.4 Usability Testing

Usability testing is used on software systems that are user centric. The test focuses on the interaction of a user with the system.

The system is given to a set of users, to use the system and feedback is collected based on the work performed by these users.

3.3.5 Security Testing

Security testing is a set of actions performed on a system with the intention to reveal flaws in the security mechanism that protects the data of a system from non-authorized access.

Security tests are designed to exercise/test the security requirements of the systems, such as authentication, authorization.

4. CODE COVERAGE

As stated in the introduction, a very important aspect of software testing is the aspect of code coverage. According to Wikipedia [6], *"code coverage is a measure used to describe the degree to which the source code of a program is tested by a particular test suite. A program with high code coverage has been more thoroughly tested and has a lower chance of containing software bugs than a program with low code coverage. Many different metrics can be used to calculate code coverage; some of the most basic are the percent of program subroutines and the percent of program statements called during execution of the test suite."*

By using code coverage tools the testing process can be improved and the overall cost of development can be reduced, as potentially more errors will be detected in the earlier phases of development; i.e. before release. Some of the benefits of the code coverage are [1]:

- To know whether we have enough testing in place
- To maintain the test quality over the life cycle of a project
- To know how well our tests, actually test our code
- It creates additional test cases to increase coverage
- It helps in finding areas of a program not exercised by a set of test cases
- It helps in determining a quantitative measure of code coverage, which indirectly measures the quality of the application.

4.1 Code Coverage Metrics

There exists a number of code coverage metrics in the literature [3] [9]:

- **Statement Coverage** – defined as the percentage of the executable statements that have been exercised by a test suite.
- **Path Coverage** – defined as the percentage of independent paths that have been exercised by a test suite.
- **Branch Coverage** - defined as the percentage of branches that have been exercised by a test suite.
- **Loop Coverage** – defined as the percentage of loops that have been exercised by a test suite.
- **Decision Coverage** – defined as the percentage of Boolean expressions that have been exercised by a test suite.
- **Condition Coverage** – defined as the percentage of decisions that have been exercised by a test suite.
- **Function Coverage** – defined as the percentage of functions that have been exercised by a test suite.
- **Entry/Exit Coverage** - defined as the percentage of call/return that have been exercised by a test suite.

4.2 Code Coverage Techniques

Code coverage tools use **probes** inserted into the target system before or during execution in order to capture the information that a particular code fragment has been executed. A probe is usually a call to a function of the code coverage system that conveys the information. This technique is called instrumentation.

There are two major instrumentation techniques: namely binary instrumentation, and source code instrumentation. Both of them are used in contemporary systems as each of them provides specific features to the testing tools.

4.2.1 Binary Instrumentation

Binary Instrumentation or Dynamic Binary Instrumentation (DBI) is a method of analyzing the behavior of a software system by injecting (inserting) code into the program during execution time. This code executes as part of the program transparently to the user, and records specific information that can be analyzed immediately or at a later stage. In the case of code coverage tools this technique is used to record that information that the execution path of the program reached the particular point, and therefore the code fragment in which the probe was injected has been executed.

Binary instrumentation requires that the system under test must be executed under the control of the tool capable of altering the stream of machine instructions of a compiled program. Thus, such tools exist as add-ons to IDE systems, used by developers.

The major advantage of this approach is that it is automatic in the sense that the user/tester does not need to do anything special other that invoking this function. The obvious disadvantage is that they cannot exist outside the IDE, and therefore cannot be used by non-developers.

4.2.2 Source Code Instrumentation

Source Code Instrumentation is a Source code insertion (SCI) technology that uses instrumentation techniques to automatically add specific code to the source files under analysis. This code becomes part of the system (compiled with the functional code), and during run-time it is invoked (automatically) to collect and store information that can be later used in the analysis of the software under test.

The major advantage of source code instrumentation, is that once the code is instrumented the code coverage tool becomes part of the system and therefore no external tool is needed in order to run it in comparison to tools that use BCI which depend on IDEs. This way:

- the code coverage runs seamlessly during testing,
- no special tool is needed,
- can be run in the laboratory or in the field

The disadvantage of this method is that the instrumentation process is a discrete action that must be performed every time the source code changes. Tools used to perform automatic compilations, such as the “make” program, can be used reduce the overhead of this process.

5. SURVEY OF CODE COVERAGE TOOLS

This section presents a survey of code coverage based tools. There are several such tools available in the literature and on the internet. They can be categorized around several features such as:

- commercial vs. free,
- stand-alone vs. add-ons to IDE tools,
- language support,
- operate on source code vs. binary instrumentation,
- coverage measurement,
- GUI,
- reporting

There are several papers in the literature surveying and evaluating coverage tools. Almost all of them perform a *"paper"* analysis and evaluation. Their work entails the selection of a handful of such tools and present a comparative analysis with regards to a pre-selected set of features. Most of them surveys report the availability of absence of the selected features. A typical of such papers in one by Muhammad and Suhaimi [10] who have recently evaluated 32 such tools with regards to five of the above stated categories/criteria.

Such an enumeration, however, of specific features provides only limited information about the usefulness and the ease of use of the tools. It would be very helpful if one could actually use a set of tools on one of more projects and report based on the information collected/reported by each tool, as well as the user experience, i.e. easiness of installation, usage, learning, etc.

Sneha [11] evaluates five tools and actually uses one of them. Muhammad and Suhaimi [10] cite Shanmuga on using four tools (JCover, Emma, Gretel, and CodeCover) to measure the coverage of a set of small application programs.

A more comprehensive study is presented in [12] were the authors evaluate five tools against 21 Java programs. However, their study was limited only to public domain tools that exist as plugins for eclipse, and can be used only with the Java software language. *Notably there are very few public domain tools for C++ code coverage analysis.*

5.1 Code Coverage Tools

Bellow we present some limited information about some of the C/C++ tools we surveyed. The information presented here is taken from [3], [10] and [13] which when combined present a complete list of the available tools today.

5.1.1 BullseyeCoverage

It is a C and C++ code coverage analyzer tool that tells how much of source code was tested. It is a proprietary tool. It uses source code instrumentation. The tool pinpoints areas that need attention to be reviewed. Supported coverage types are function and condition/decision. BullseyeCoverage supports the widest range of platforms of any code coverage analyzer including Windows and Linux. [3]

5.1.2 GCOV

Is a free test coverage program that can be used in concert with GCC to analyze your programs thus create more efficient, faster running code and discover untested parts of your program [14]. GCOV provides some basic performance statistics such as the following: how often each line of code is executed, which lines are actually executed and how much computing time each section of code uses. GCOV works only on code compiled with GCC and is not compatible with any other profiling or test coverage mechanism.

5.1.3 Dynamic Code Coverage (DCC)

DCC [15] uses an innovative strategy so that runtime instrumentation gathers coverage information. It measures coverage of all involving modules as well as 3rd party code. DCC provides a detailed coverage analysis including function, line, decision and branch metrics.

5.1.4 TCAT C/C++

TCAT [16] is a branch, function and call-pair coverage analyzer tool. It is an instrumentation based tool that allows to dynamically collect coverage information. Also, it provides a powerful GUI with multiple display options to easily extract the test results.

5.1.5 TestWell C/C++

TestWell [17] is a powerful instrumentation-based code coverage and dynamic analysis tool for C and C++ code. It collects extensive coverage metrics such as line, statement, function, decision, multi-condition, modified condition and condition. A rare feature in CC tools is provided by TestWell. It computes the exact number of times a certain block of code was executed instead of executed/non executed information.

5.1.6 Squish Coco Code Coverage

It is a complete code coverage tool, cross-platform, cross-compiler tool chain allowing to analyze the test coverage of C, C++, C# and Tcl code [18]. It compiles an extensive analysis as it finds untested code sections, redundant tests, unreachable code and others.

5.1.7 Open Code Coverage Framework (OCCF)

OCCF is a framework for measuring test coverage supporting multiple programming languages [19].

5.1.8 Semantics Design CC

"SD's code coverage tools operate by inserting language-specific probes for each basic block in the source files of interest before compilation/execution. At execution time, the probes record which blocks get executed ("coverage data"). On completion of execution, the coverage data is typically written to a code coverage vector file. Finally, the code coverage data is displayed on top of browsable source text for the system under test, enabling a test engineer to see what code has (not) been executed, and to see overall statistics on coverage data." [20]

5.2 Evaluation Criteria

The following criteria have been used for their evaluation of code coverage tools in [13], [10] and [3].

Language Support - Most for the tools support Java and C/C++ languages. There are some tools that support more than one languages (i.e. Java and C++) and some support JavaScript Fortran, C#, etc.

Instrumentation – What kind of instrumentation the tool uses

Code Coverage – What kind of code coverage metrics the tools uses

Reporting – What kind of reporting the tool produces.

5.3 The survey

This section summarizes the findings of the survey. Because of the large number of tools presented the information is given in tabular form in Table 1, below.

The information presented here was compiled from papers published recently with the same topic [3], [11], [13]. It should be noted that all authors, with exception of Shanmuga, performed a literature based evaluation. This is due to inherited problems associated with a hands-on experience with each tool. Each of the tools has some strong points as well as weaknesses. Common to all tools is the fact that the tool is targeted a specific domain/area. Hence they present strong points in that area whereas they have weakness in other areas.

Based on Table 1, there are 10 tools that provide code coverage functionality for C++ programs. From those, only Gcov and OCCF are stand alone and free.

Table 1: List of code coverage tools and their characteristics

Tool	Language Support			Instrumentation		Coverage Metrics				GUI	Report	License
	Java	C++	Other	Src	Bin	statement	decision	method	class			
Agitar [21]	x					x	x	x	x	x		
Dynamic [15]		x		x		x	x	x				
Gcov [14]		x			x	x						free
JTest [22]	x					x	x			x		

Koalog [23]	x									x		
PurifyPlus [24]	x	x				x		x		x		
Semantic Designs [24]	x	x	x	x		x	x	x	x	x		
TCAT [16]	x	x				x	x	x	x	x		
JavaCodeCoverage [25]	x				x	x	x	x	x	x	x	
JFeature [26]	x			x				x		x	x	
JCover [27]	x			x		x	x	x	x	x	x	
Cobertura [28]	x			x		x	x			x	x	
Emma [29]	x				x	x		x	x	x	x	
Clover [30]	x			x		x	x	x		x	x	
Quilt [31]	x				x	x	x					
Code Cover [32]	x			x		x	x			x	x	
Jester [33]	x				OT[1]					x		
GroboCodeCover [34]	x				x				x			
Hansel [35]	x				OT		x					
Gretel [36]	x			x		x						
BullseyeCoverage [37]		x		x			x	x		x	x	
NCover [38]			C#	x				x	x	x	x	
TestWell C/C++ [17]		x		x			x				x	
SquichCoco CC [18]		x		x						x	x	
OCCF [19]	x	x	x	x		x	x					
JAZZ [39]	x				OT		x					

[1] OT stands for other instrumentation

6. CODE COVERAGE AID

In this section we present the design of the Code Coverage Aid (CCA), including the program features, the reason of choosing SCI, high level design, a layered architecture of the parser, presentation of the instrumentation algorithm, how CCA is activated as well as installation and user guides.

As stated earlier CCA provides code coverage metrics for C/C++ programs. It uses the Source Code Instrumentation technique to achieve its goal because of the freedom this technique provides, namely once the source code is instrumented, no other tools are needed, and can be used everywhere.

CCA operates on a project level. That is: the set of all source files that comprise the target system under test, referred to as the *"project"*. The source files of the project must be syntactically correct. Therefore CCA must be invoked after all source files are successfully compiled.

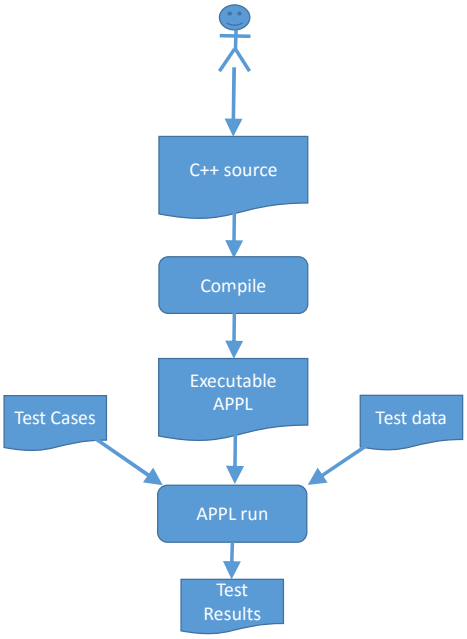

Figure 2: Stand-alone development and testing process

Once we have the compiled code we pass each source file through CCA that creates a copy of the source file and instruments the copy. This way the changes made to the source file by CCA, which may be visually annoying to the programmer, are not visible, at the expense of having two sets of files (both source and binary). Please note that even though there are two copies of the source, the programmer maintains only his own copy. There is never a need for the user to look into the CCA created source files. Figure 2 and Figure 3 depict the above process with sequence diagrams.

Figure 2 shows a typical development process, where a developer, creates a source code, then compiles the code into an executable, and tests that executable based on a chosen set of test cases and test data. The results of the test are then analyzed and the same process may be repeated again.

Figure 3 shows the same process, this time incorporating CCA. In this case, once code is developed it is passed through the CCA instrumentation module to obtain the instrumented source code. This code is compiled normally and linked with the CCA run-

time module, and then tested the same way as in the stand alone case. The tested code now produces two sets of data. The first is identical to one produced in the stand-alone case, and the second is the coverage results of CCA module. These results can be analyzed separately by the tester and additional feedback can be fed to the developer.

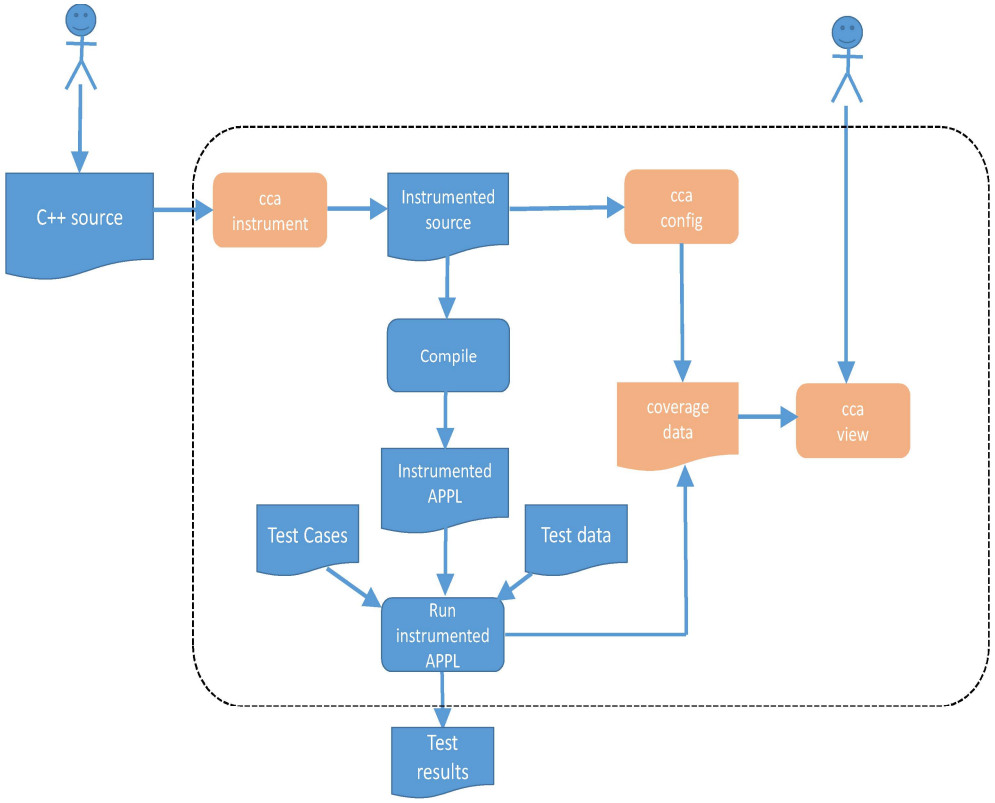

Figure 3: Development and testing under CCA control

CCA supports and operates on two instrumentation granularities, the "*user specified decisions*", and "*all decisions*". In the first case, CCA inserts probes on every code block defined by the programmer. For C programs, a code block is detected by the symbol "open brace" or '{'. In this case, CCA will track coverage to each function defined in the project, and within a function will track coverage to decisions explicitly marked by the programmer with the '{', '}' pair. In the second case, CCA will track coverage to all cases and also to all decisions not explicitly marked with the '{', '}' pair.

CCA comprises of the following executable programs:

ccaProbe – ccaProbe operates on a single C/C++ source file. As the name suggests, this program reads a source file and it inserts probes into appropriate places as instructed by the command line arguments passed to the program,

ccaConfig – ccaConfig is used at the end of the instrumentation process. The program is fed with all source files that comprise the system under test (all files that passed through ccaProbe).

ccaView – ccaView is a GUI program that may be used to visualize the test results collected by CCA. It is used after the end of a test session to display graphically the code coverage results.

cca – cca combines ccaProbe and ccaConfig in one command. It is fed with all source files that comprise the project under test and performs all needed operations in one step.

6.1 Program features

The current implementation of CCA exhibits the following features:

1) Works with C and C++ programs. It recognizes and handles all language constructs defined in ANSI C and C++ specifications.
2) Instrumentation takes place automatically (invoked by *make* program on every source file). The instrumentation granularity is user controlled and may be: a) on the entry point of every function b) on the entry point of every control structure identified by {.} pair (i.e. for/while loop and if-then-else statement), or c) on every decision point.
3) Coverage Metrics supported:
    a. Module coverage
    b. Function / Class coverage
    c. Decision Coverage
4) Statistics are collected at the end of test execution. Two sets are maintained: a) statistics for the last run, b) cumulative statistics of all runs.
5) Post-execution coverage analysis via Graphical User Interface.
6) Cross-compiler and cross-platform.

6.1.1 CCA Functional Specifications

1) Program will compute the test coverage of a complete software package written in C++.
2) The method of computing will be by means of inserting probes in various places of the source code.
3) Tester will chose which modules of the software package will participate in the test.
4) Probe insertions to the code and all setup activities needed by the tool will be invoked manually by the programmer or tester, and will be performed automatically.
5) Setup activities may be invoked at any time. That is, if programmer adds new code the tool will automatically prepare the new code for test coverage.
6) The user (tester) need not intervene with the process of preparing the code for test coverage.
7) The tool will produce test coverage statistics for
    a. Each run of the software
    b. All runs of the software (accumulative statistics)
8) The tool will assist the tester to identify which parts of the Software (i.e. function names, module names) have or have not undergo sufficient testing.
9) A Graphical User Interface will display testing results. User may click on parts of graph to view the module/functions the graph refers to.

6.1.2 CA Software Specifications

1) Written in C++.
2) Stand-alone. No dependencies on other tools to work.
3) Portable. Uses standard C functions, thus portable to any OS (by means of compile for the target OS) (GUI only windows…)
4) Other Packages used (for the tool development):
    a. Windows version: Perl Compatible Regular Expressions package PCRE 2.10
    b. Unix version: none
5) Program will identify modules, functions, and function areas (code blocks) of the target Software.

6) No limit on the number of modules the tools will handle
7) No limit on the number or size of the probes to be inserted
8) Statistics will be measured continuously, but will be saved to database every X calls (where X configurable) to a Program function.

6.2 Why Source Instrumentation

As stated previously, CCA uses source code instrumentation (SCI). We opted for this solution because it provides several advantages compared to binary code instrumentation (BCI). Specifically, it allows to develop a stand-alone tool, without any dependencies on other tools. Such tool can be easily portable to any OS. In addition, SCI allows us to control the places where probes are inserted (i.e. one could decide to instrument only switch statements), whereas in BCI one cannot determine if a branch instruction is due to a for() statement of an if() statement. With SCI there is no need to know the binary structure of a program, or to interface with other tools, and thus create dependencies on their roadmap.

This decision has its drawbacks as well. There is an overhead for instrument/compile every time a change to source is made. A second drawback is the danger that comes from the ability of the C++ compiler to redefine symbols. For example the compiler allows someone to:

```
#define begin {
#define end   }
```

And thus to write something like:

```
if ( a > b ) begin
   statements …
end ;
```

In such cases CCA will fail to recognize the decision block of the if() statement, as its parser does not have all the features of the native compiler. However, this is not a big problem, as good software development practices should prohibit such redefines of the compiler structures anyway because they create huge readability and maintainability problems.

6.3 High Level Design

The major challenge of CCA is to develop a powerful parser in order to recognize all the language features so that it can catch all the places in the source code that probes must be inserted, and at the same time make sure the semantics of the program will not change.

6.3.1 Probe Insertion points

Instrumentation in CCA is achieved by means of inserting the following function call in appropriate places in the source code.

_cca_probe(label) ;

Where **_cca_probe()** is a CCA function to be called when execution reaches this point and label is the information collected by CCA at this point as explained in 6.3.2. Probes are inserted by ccprobe.exe on every open brace token (‘{‘) of a code block, as

instructed by user arguments. Note that CCA detects implied {} pairs as in the case of a switch statement, if they are omitted. Figure 4 shows a sample C source code file containing the implementation of a function named sample, and Figure 5 show the output of ccaprobe.exe after the insertion of three probes.

```
int sample(int cnt, char *info)
{
   int sum = 0 ;
   printf("%s\n", info) ;
   for(int i=0;i<cnt;i++) {
     int v = compute(i);
     if ( v > 100 ) return -1 ;
     sum += v ;
   }
   return sum ;
}
```

Figure 4: Sample C code

6.3.2 Probe definition

The label in a _cca_probe() function call captures the information regarding which point in the source code has been executed. The label is a string having the following format:

<module name>@<function name>@<sequence number>

Where <module name> is the name of the source file where the probes are inserted,

<function name> is the name of the function (or class method), and <sequence number> is an integer starting at zero, when a new function is encountered, and incremented every time a new probe is inserted (see Figure 5).

The above format has the following properties:

1) It creates a unique label for each probe, for a given project
2) It identifies each module of the project (i.e. **sample.c**@sample@000) therefore allows to measure coverage metrics per module.
3) It identifies the entry point of each function, sequence number equal to zero (i.e. sample.c@sample@**000**), therefore it allows to collect coverage metrics per function and/or class.
4) Can be easily extended to identify other information inside a function, if need be.

CCA detects overloaded functions (functions with same name, but different set of arguments) and creates unique function names by appending to the function name the number sequence "_xx", where xx starts at 01 for the second function found with same name.

```
//--------------------------------------------------------------------------
//-- This program was automatically instrumented by ccaPROBE for code coverage measurement
//-- Compile with '/D _CCA_' switch to activate code coverage
//--------------------------------------------------------------------------
#include <cca.h>
int sample(int cnt char *info)
```

```
{    _cca_probe("sample.c@sample@000");
   int sum = 0 ;
   printf("%s\n", info) ;
   for(int i=0;i<cnt;i++) {    _cca_probe("sample.c@sample@001");
     int v = compute(i);
     if ( v > 100 ) {    _cca_probe("sample.c@sample@002"); return -1 ;}
     sum += v ;
   }
   return sum ;
}
```

Figure 5: Sample C code after instrumentation

6.3.3 Decision detection

Decisions in a C++ program occur with the following language structures: if(), for() while(), switch(), do-while(). If user has inserted braces on all decision statements, as in the example below, then CCA will instrument each one of them.

```
if ( a ) { b=5 ; } else { b=6 ; }
while (a) { a = foo(n); }
```

Since, however, braces are mandatory only to remove ambiguity of the program the above code fragment could be written as follows, in which case probes are not inserted by default (due to absence of {).

```
if ( a )  b=5 ; else b=6 ;
while (a) a = foo(n);
```

If ccaprobe.exe is invoked with the '-i' switch it will detect all cases such as the one above and insert appropriate braces before instrumentation begins, thus the end result after instrumentation will be:

```
if ( a ) { _cca_probe("f1@f1@001"); b=5 ; } else {_cca_probe("f1@f1@002");  b=  ; }
while (a) {_cca_probe("f1@f1@003");  a = foo(n); }
```

Using the '-i' switch, the current implementation of CCA is able to capture all decision points in a C/C++ program except for the following statement:

```
v = (a=b)? 1:0 ;
```

6.4 CCA parser – a layered architecture

The CCA parser uses an abstraction technique in order to minimize the complexity of a C++ program and simplify the task of the instrumentation.

To do so, for a given source file, the parser maintains two copies of it: the first copy is called the "original" and contains the source code as written by the user. The second copy is called the "working" copy. It starts out as a duplicate of the original and then it is passed through the following filters which remove information not necessary for the instrumentation. The output is constructed from the original with appropriate probe insertions, guided by the working copy.

The filters are the following:

6.4.1 Comments filter

This filter operates on a source file and creates a copy in which all user comments are replaced by spaces. The filter recognizes three types of comments, namely:

a) Comments specified with a `/* */` pair

b) Comments specified with a `//` character

c) Comments specified with the `#if 0` directive

The following shows a code fragment with comments

```
int foo( /* money */ int val )
{
  int t = val * y ; // comment
#if 0
  t = t / 2 ;
#endif
  return t ;
}
```

and the result of the comments filter

```
int foo(             int val )
{
  int t = val * y ;




  return t ;
}
```

6.4.2 Literals filter

This filter operates on a source file free of comments and creates a copy in which all literal strings are replaced by spaces. The filter recognizes special chars inside strings such as \\ or \" and handles them properly:

The following shows a code fragment with strings

```
int c = printf("This is a \\ test \' of \" the quote") ;
char *p = "this is a test" ;
char *p = "this \" a \"st" ;
char c = '}' ;
char c = '\'' ;
char c = '\"' ;
char *p = "this\\testssst" ;
char *p = "this  testss\\" ;
```

and the result of the literals filter where test string characters are replaced with the character '_' for clarity.

```
int c = printf(_____________________________________) ;
```

```
char *p = _________________ ;
char *p = _________________ ;
char c = '_' ;
char c = '_'  ;
char c = '_'  ;
char *p = _________________ ;
char *p = _________________ ;
```

6.4.3 Arrays assignment filter

This filter operates on the array assignments. It detects when an array is initialized and replaces the entire initialization data with spaces. This is necessary in order to remove the '{', '}' around the initialization data. By doing this the parser does not need to qualify the '{' it encounters. It assumes that a '{' indicates a code block.

The following shows a code fragment with array initialization data

```
int val[]={1,2,3, 5, 7 };

int foo3( /* info */ int a )
{
   int val1[2][4] =
   {
      { 1, 2, 3, 5  },
      {-1, 4, -8, 7 }
   } ;
    int v = printf("this is a test /* ok */  string \n");
}
```

and the result of the array assignment filter where assignment data is replaced with spaces.

```
int val[]=              ;

int foo3(            int a )
{
   int val1[2][4] =



     ;
   int v = printf("this is a test /* ok */  string \n");
}
```

6.4.4 Special Characters filter

The Special characters filter operates on any data (original or modified). It finds and replaces with spaces the following characters:

a) TAB
b) Line Feed
c) New Line

6.4.5 Parenthesis filter

This filter operates on code expressions enclosed in parentheses. It detects parentheses and replaces anything inside them with spaces. Nested parentheses are replaced as well.

The following shows a code fragment with parentheses

```
int foo1()
{
    for(i=01;i<20;i++){
       printf("this is index = %d\n", i ) ;
    }
    return 0 ;
}
```

and the result of the parenthesis filter.

```
int foo1()
{
    for(             ){
       printf(                          ) ;
    }
    return 0 ;
}
```

6.4.6 Brace insertion filter

A key feature of CCA is the decision coverage metric. It is able to detect when decision statements are taking the ‘true’ or the ‘false’ path and measures the coverage of these actions. When decision statements are surrounded by {} it is easy to detect, as the CCA parser is searching for open brace and inserts probes right after that.

Therefore, decisions such as the one depicted bellow are detected due to {} pairs.

```
if ( a )
{
  …
} else {
  …
}
```

However, the use of {} is mandatory in C/C++ only when a code block contains more than one statements. There is therefore many cases in which code makes a decision that are not surrounded by {}. The following code examples shows several such case.

```
if ( a )  a = 5 ;
if ( a )  a = 5 ; else a = 6 ;
if ( a )  a = 5 ; else if ( b ) a = 0 ;
if ( a )  a = 5 ; else if ( b ) a = 0 ;  else a = 4 ;
if ( a )  a = 5 ; else if ( b ) a = 0 ;  else { a = 4 ;  b = 5; }
```

The brace insertion filter scans a source code, detects each statement, decides if the statement is a decision making statement and adds {} if not surrounded by them already. It recognizes statements comprising of code blocks (i.e. a for() statement) and operates on the code block of a statement recursively. Thus, when the parser instruments the code it will encounter {} on decision statements and will measure the coverage.

The following is the output of the above code fragment passed through the brace insertion filter.

```
if ( a )  {a = 5 ;}
if ( a )  {a = 5 ;} else {a = 6 ;}
```

```
if ( a )  {a = 5 ;} else if ( b ) {a = 0 ;}
if ( a )  {a = 5 ;} else if ( b ) {a = 0 ;}  else {a = 4 ;}
if ( a )  {a = 5 ;} else if ( b ) {a = 0 ;}  else { a = 4 ;  b = 5; }
```

6.5 The instrumentation algorithm

In this section we will briefly discuss the core of the parser, namely the detection of the places where probes must be inserted. This is done in the insert_probes() method of the CCA_PARSER object. The method is called after we run all the filters discussed in the previous section 6.4.

The parser operates on three buffers

1. the original source code (original)
2. the clean source which is the output the filters (clean)
3. the output of the instrumentation (output), see

The parser algorithm involves the following steps where each step is a method that returns true or false. If a method returns false we abandon the whole process

1. load source file src.cpp
2. instrument()
    a. insert braces
    b. replace comments
    c. replace literal strings
    d. replace array assignments
    e. replace special characters
    f. replace parentheses
    g. insert probes
3. save()

The insert_probe() method which is the heart of the system works as follows:

It operates on three buffers as shown in Figure **6**. It scans the so called “clean” buffer which is identical in size and format with the “original” buffer, except for the fact that unnecessary information has been replaced by white spaces, and writes to the output buffer, by copying from the original as shown in Figure **6**.

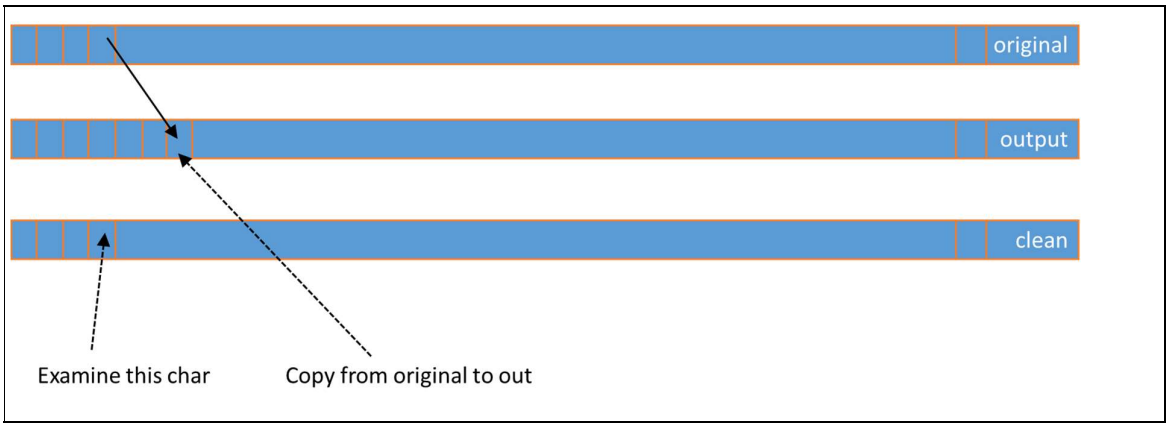

Figure 6: Parser buffers

The insert_probe() algorithm is shown below:

```
Scan the clean source buffer
while more characters in clean buffer
        let c the current character
        if c is '(' or ')' increment or decrement the parentheses counter, else
        if c is '{' then
                increment brace counter
                if brace counter == 1 then,
                check for class or struct definition, if found save class name for later use
                check for function entry and extract and save the name ( if class name
                has value prefix function name with class value)
                if inside function
                                increment function brace counter
                                insert probe into destination buffer
        if none of the above then copy corresponding character from source buffer to
        destination buffer
```

Figure 7: High-level insert probe algorithm

6.6 Software Installation

The CCA executable package comprises of the following files shown on . There are three executables needed for the operation of CCA and two files needed for the compilation of the target project.

Table 2: List of CCA executable files

File Name	Description
ccaprobe.exe	Instrumentation program
ccaconfig.exe	Project configuration program
cca.exe	Instrumentation and configuration combined in one program
ccaview.exe	Results viewer program

cca.h	Header file needed to compile project
cca.lib	Library needed to link project

The installation of the package stores a simple copy of the above files in a place that can be found by the system. I.e. it stores the executables in a folder included in the PATH variable and the *.h *.lib in a folder accessible by the C compiler.

6.7 Activation of CCA in an instrumented program

To use CCA on a project, the project must be linked with the CCA runtime library which needs initialization. In order for the library to initialize it needs the project's probes. To achieve this the CCA API provides the _cca_coverage() function which must be called at the beginning of the execution with the project's name as argument and again in the end of the execution without arguments. Having the project's name, _cca_coverage() finds the file containing the probes and initializes the object COVER.

There are two ways to call this function.

Firstly, using auto-initialization where we use the property of the C compiler to initialize global variables before handing control to main() function. The library declares a CCA_INVOKE object variable on the global data section thus the object's constructor is called before the execution of the main function. This method, which is the simplest, can be used when there is only one executable in a project. In this case the user must create a cca profile with the same name as the executable.

The second method can be used when a project has multiple executables. Now that the unique project name doesn't match with the exe files names it is mandatory to either explicitly call the _cca_coverage() function at the beginning of main() with the right cca profile name, of copy the cca profile to match the name of every executable (in which case the first method is used, but we have many profiles with the same data).

In order to explicitly initialize CCA run-time in main, the ccaprobe.exe provides a method to rename main() function to MAIN() and create its own main() in which _cca_coverage() is called with the project name to initialize coverage, then MAIN() is called to execute the project and finally _cca_coverage() is called with no arguments to do the post coverage work. To do this, when you instrument the sources with ccaProbe and specify the argument ***–app <project_name>.***

6.8 User Guide

This section presents a brief guide to users who want to use CCA to measure the test coverage of their programs.

The discussion assumes the user has created an application, called here 'a project' and is using a 'makefile' to automate the compilation process. Although CCA can be used by executing appropriate commands on a computer terminal, the easiest way to use it is to integrate it on the same makefile used to create the project. This way the user will create two versions:

a) a stand-alone version

b) a CCA instrumented version

As stated in the above section, CCA consists of four executable programs. The way they can be used is displayed on Figure **3**: Development and testing under CCA control (Figure **3**). In order to use CCA you need the source files of a system which must be syntactically correct, meaning compiler should not find any errors.

When testing a project with CCA we can distinguish three cases, discussed in the following subsections:

6.8.1 The project comprises of only one executable

The simplest case is when the project comprises of one or more source files that compile to one executable, as shown in the sample makefile below. When we have only one executable the user can use *cca.exe* which is the combination of *ccaProbe.exe* and *ccaConfig.exe* to instrument the project.

Assuming we have *prog.exe* that is created by *prog.cpp* and *src.cpp*

A simple *makefile* is shown below:

```
all: prog.exe
prog.exe: prog.obj src.obj
prog.obj: $*.cpp
src.obj: $*.cpp $*.hpp
```

In order to add the coverage the *makefile* should be converted into:

```
all: prog.exe cca_prog.cca cca_prog.exe


cca_prog.cca: prog.cpp src.cpp
   cca -app cca_prog $**


prog.exe: prog.obj src.obj
   $(LINK) $(co) $** $(libs)
cca_prog.exe: cca_prog.obj cca_src.obj
   $(LINK) $(co) $** $(libs) ccalib32.lib


prog.obj: $*.cpp
src.obj: $*.cpp $*.hpp


cca_prog.obj: $*.cpp
cca_src.obj: $*.cpp src.hpp
```

The purpose of cca.exe is to instrument ALL files that comprise the project in one step, and to create the cca *project file*.

It accepts two inputs:

- A name to be used as *cca project name*. For example, the command ‘cca –app cca_prog’ will create the cca project file cca_prog.cca. Important note: The name of the cca project must be the same as the name of the executable.
- A list of source files to be instrumented. Each source file will be instrumented and the output will be written to a file with the same name as the source prefixed with the string “cca_”. For example, the file src.cpp will be instrumented as cca_src.cpp

The instrumented code must then be compiled the same way as the original code, and linked with the **cca.lib**

6.8.2 The project comprises of more than one executables

When the test project comprises of multiple executables and we want to test them as an entity, *cca.*exe is unsuitable for the task. Instead we need to use *ccaConfig.exe* and *ccaProbe.exe*.

Assuming we have *prog1.exe* and *prog2.exe* which are created by *prog1.cpp, src1.cpp, prog2.cpp, src2.cpp* accordingly. A simple *makefile* would be:

```
all: prog1.exe prog2.exe


prog1.exe: prog1.obj src1.obj
   $(LINK) $(co) $** $(libs)
prog2.exe: prog2.obj src2.obj
   $(LINK) $(co) $** $(libs)


prog1.obj: $*.cpp
prog2.obj: $*.cpp
src1.obj: src1.cpp src1.hpp
src2.obj: src2.cpp src2.hpp
```

In order to add the coverage the *makefile* should be converted into:

```
all: prog1.exe prog2.exe cca_prog.cca cca_prog1.exe cca_prog2.exe


cca_prog.cca: cca_prog1.exe cca_prog2.exe cca_src1.cpp cca_src2.cpp
   ccaConfig -app cca_prog $**


prog1.exe: prog1.obj src1.obj
   $(LINK) $(co) $** $(libs)
prog2.exe: prog2.obj src2.obj
   $(LINK) $(co) $** $(libs)


cca_prog1.exe: cca_prog1.obj cca_src1.obj
   $(LINK) $(co) $** $(libs) ccalib32.lib
cca_prog2.exe: cca_prog2.obj cca_src2.obj
   $(LINK) $(co) $** $(libs) ccalib32.lib


cca_prog1.ccp: prog1.cpp
   ccaprobe -mainonly -app cca_prog $**
cca_prog2.ccp: prog2.cpp
   ccaprobe -mainonly -app cca_prog $**
cca_src1.cpp: src1.cpp
   ccaprobe $**
cca_src2.cpp: src2.cpp
   ccaprobe $**


prog1.obj: $*.cpp
```

```
prog2.obj: $*.cpp
src1.obj: src1.cpp src1.hpp
src2.obj: src2.cpp src2.hpp

cca_prog1.obj: $*.cpp
cca_prog2.obj: $*.cpp
cca_src1.obj: cca_src1.cpp src1.hpp
cca_src2.obj: cca_src2.cpp src2.hpp
```

Where again cca_prog is the name of the project specified with '*-app*' and ccaconfig will used this name to create the ini file cca_prog.cpp.

7. TESTING CCA

When it comes to CCA testing there is two major questions to be answered, namely:

1. Does CCA operate correctly? Meaning a) does the instrumentation capture all cases in a given program, b) are probes correctly updated during execution, c) are statistics correct? etc.
2. Does CCA preserve the semantics of the system under test? In other words, is it possible that the invasive procedure of ccaprobe altered the logic of the target program?

This section reports on the testing activities to verify the functionality of CCA, in respect to both questions above.

For this purpose we devised the following test strategy: With respect to CCA correctness itself, a set of unit tests was created to test each feature of the system. This is explained in section 7.1. Then the correctness of CCA and the correctness of the target system were verified as explained in 7.2. This test was performed with and without an instrumented version of CCA, as explained in section 7.3.

7.1 Unit Test

The purpose of this test was to verify the correctness of each function of CCA. For this purpose a small test program was written to drive each function on a set on inputs and the output of each test was visually compared against the expected result.

The tests are:

Table 3: Set of unit test cases

Test Case	Description
Test comments / arrays	Remove valid C comments and array initialization data
test parenthesis	Remove characters inside between ‘(‘ and ‘)’
test strings	Remove literal strings
test special chars	Remove special characters like LF, CR…
test braces	Add braces around single decision statement i.e. if(a) a=b;

These tests were executed against the following input amongst others, which were selected in order to capture all possible cases the author could think of:

7.1.1 Test comments and arrays

The purpose of this test is to verify CCA can remove all valid C comments and array initialization data from a source file, preserving the remaining structures intact:

Sample input to test_coments.exe:

```
int val[]={1,2,3, 5, 7 };

int foo1( /* info */ int a )
```

```
{
   int val[]={1,2,3, 5, 7 };
   return 0 ;  // ok....
}
#include "test.h"  // comment

#if 0
int foo2( /* info */ int a )
{

}
#endif

int foo3( /* info */ int a )
{
   int val[2][4] = {
      {1, 2, 3, 5  },
      {-1, 4, -8, 7 }
   } ;
   int v = printf("this is a test /* ok */  string \n");
}

int foo4( /* info */ int a )
{
   int val1[2][4] =
   {
      {1, 2, 3, 5  },
      {-1, 4, -8, 7 }
   } ;
   int val2[2][4] =  /* in your head...*/
{
      {1, 2, 3, 5  },
      {-1, 4, -8, 7 }
   } ;
   int v = printf("this is a test /* ok */  string \n");
}
```

Figure 8: Sample input for comment and array removal

Respective output of the program

```
int val[]=              ;

int foo1(            int a )
{
   int val[]=              ;
   return 0 ;
}
#include "test.h"









int foo3(            int a )
{
   int val[2][4] =


     ;
   int v = printf("this is a test /* ok */  string \n");
}

int foo4(            int a )
{
   int val1[2][4] =



     ;
   int val2[2][4] =
```

```
     ;
   int v = printf("this is a test /* ok */  string \n");
}
```

Figure 9: Output after removing comments and array initialization data

7.1.2 Test parentheses

Sample input to test_paren.exe:

```
int foo1()
{
    for(i=01;i<20;i++){
       printf("this is index = %d\n", i ) ;
    }
    return 0 ;
}

int foo2(int v)
{
    if ( v == 5 ){
       printf("v = %d. stoping\n", v ) ;
       return 0 ;
    }
    while( (v > 1) && (t < (5+v) ) ) {
       v-- ;
       v = (v > 2 ) ? 2 : 1 ;
    }
    return 0 ;
}
```

Figure 10: Sample input for parenthesis removal

Respective output of the program

```
int foo1()
{
    for(             ){
       printf(                          ) ;
    }
    return 0 ;
}

int foo2(     )
{
    if (        ){
       printf(                        ) ;
       return 0 ;
    }
    while(                          ) {
       v-- ;
       v = (      ) ? 2 : 1 ;
    }
    return 0 ;
}
```

Figure 11: Respective output of parenthesis removal

7.1.3 Test strings

Sample input to test_strings.exe:

```
int c = printf("This is a \\ test \' of \" the quote") ;

char *p = "this is a test" ;

char *p = "this \" a \"st" ;

char c = '}' ;

char c = '\'' ;

char c = '\"' ;
```

```
char *p = "this\\testssst" ;
char *p = "this  testss\\" ;
```

Figure 12: Sample input to string removal

Respective output of the program

```
int c = printf(_____________________________________) ;
char *p = _________________ ;
char *p = _________________ ;
char c = '_' ;
char c = '_'  ;
char c = '_'  ;
char *p = _________________ ;
char *p = _________________ ;
```

Figure 13: Respective output to string removal

7.1.4 Test braces

Sample input to test_braces.exe:

```
if ( a )  a = 5 ;
if ( a )  a = 5 ; else a = 6 ;
if ( a )  a = 5 ; else if ( b ) a = 0 ;
if ( a )  a = 5 ; else if ( b ) a = 0 ;  else a = 4 ;
if ( a )  a = 5 ; else if ( b ) a = 0 ;  else { a = 4 ;  b = 5; }
if(a)
  for(i=0;i<10;i++) {
    x= ';'+5 ;
  }x=5;
if(a) for(i=0;i<10;i++);
if(a) for(i=0;i<10;i++) x= ';'+5 ;

if(a) for(i=0;i<10;i++) {
  x= ';'+5 ;
}
if (a) ; else for(i=0;i<10;i++) {x= ';'+5 ;}
//------ok ----   2016-05-20  ----------
if(a) while( b != ';') ;
if(a) while( b != ';')x=a+';' ;
if(a) while( b != ';') {
  x= ';'+5 ;
}
  if (a) do
  {
    printf("xxx");
  } while (true ) ;

  if (0==strcmp(color, "black"))    return clr_black ;

  while ( a == func( b ) )   x += 5 ;
  while ( a == func( b ) )x += 5 ;
  for(i=0;i<10;i++)          x = ';'+5 ;
  for(i=0;i<10;i++)x = ';'+5 ;
```

Figure 14: Sample input to brace insertion

Respective output of the program

```
if ( a )  {a = 5 ;}
if ( a )  {a = 5 ;} else {a = 6 ;}
```

```
if ( a )  {a = 5 ;} else if ( b ) {a = 0 ;}
if ( a )  {a = 5 ;} else if ( b ) {a = 0 ;}  else {a = 4 ;}
if ( a )  {a = 5 ;} else if ( b ) {a = 0 ;}  else { a = 4 ;  b = 5; }
if(a)
  {for(i=0;i<10;i++) {
    x= ';'+5 ;
  }}x=5;
if(a) {for(i=0;i<10;i++){;}}
if(a) {for(i=0;i<10;i++) {x= ';'+5 ;}}
if(a) {for(i=0;i<10;i++) {
  x= ';'+5 ;
}}
if (a) {;} else {for(i=0;i<10;i++) {x= ';'+5 ;}}
if(a) {while( b != ';') {;}}
if(a) {while( b != ';'){x=a+';' ;}}
if(a) {while( b != ';') {
  x= ';'+5 ;
}}
  if (a) {do
  {
    printf("xxx");
  } while (true ) {;}}
  if (0==strcmp(color, "black"))    {return clr_black ;}
  while ( a == func( b ) )   {x += 5 ;}
  while ( a == func( b ) ){x += 5 ;}
  for(i=0;i<10;i++)          {x = ';'+5 ;}
  for(i=0;i<10;i++){x = ';'+5 ;}
```

Figure 15: Respective output to brace insertion

7.2 System Test

The system test of CCA serves a dual purpose. From one hand it verifies the functionality of CCA, and on the other hand it verifies the preservation of semantics of the target system.

For this purpose we selected the following “targets” and performed the following tests:

Run the tests on these targets stand alone (no CCA involvement).

Then, we instrumented the code and run the same tests again. We then compared the results for correctness (verified preservation of semantics), and analyzed the test coverage to verify CCA functionality.

Table 4: Packages used for system test of CCA

Package Name	Modules	Lines of Code	Probes	Code Coverage	Probes with ‘-i’	Code Coverage (-i)
CString class	1	594	105	62.86%	193	53.09 %
Printf class	1	708	154	52.60 %	233	57.62 %
Text Editor	11	7,542	1033	n/a	1,777	n/a

7.2.1 Cstring class

Cstring class is a c++ class that provides string manipulation functionality similar to C++ std string class. It comprises of a single module defining a set of about twenty methods.

We selected this class as a first test because it has an automated set of test cases (about 250 tests). We run the test suite stand alone and then we run exactly the same tests under CCA, using the '-i' switch which inserts probes on all decision statements. This resulted in inserting 193 probes. Running all tests available gave a code coverage of 53%. The results are shown below:

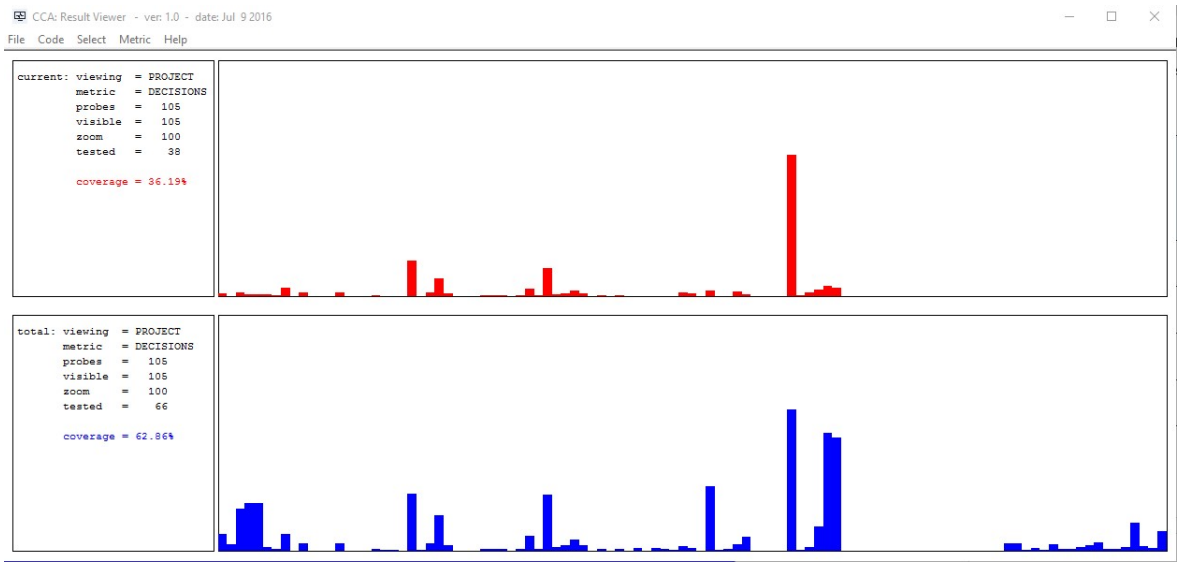

Figure 16: Test coverage of Cstring project

Running CCA without '-i' resulted in inserting 105 probes. We run the same suite of test again and computed coverage 62%, as shown below.

Comparing the test results of the CCA run with those from stand-alone run showed no differences whatsoever.

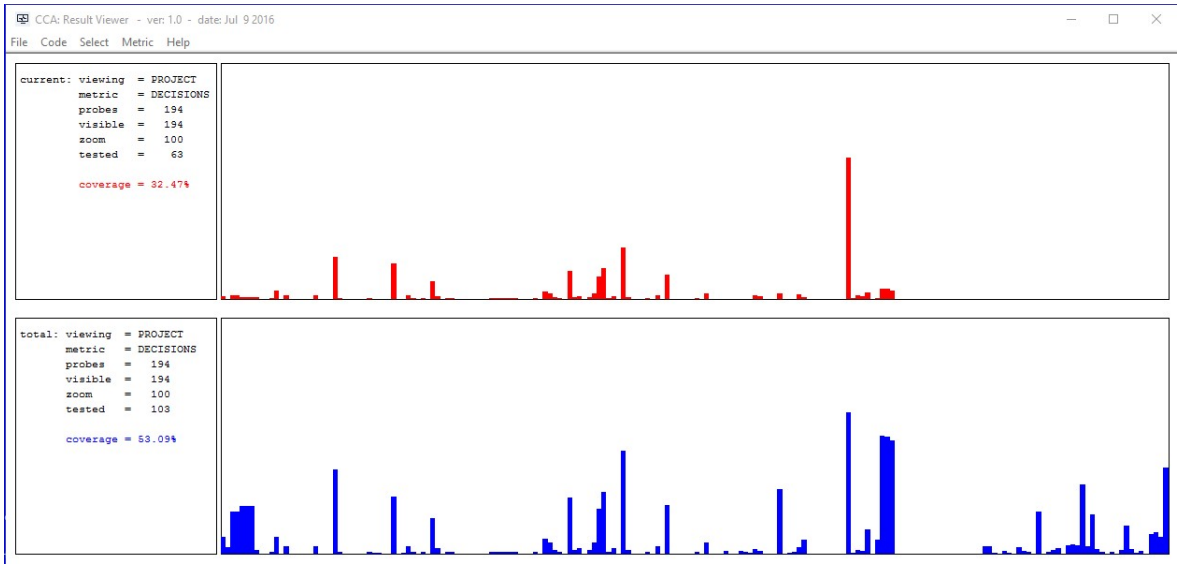

Figure 17: Test coverage of Cstring project running with no '-I' switch

7.2.2 printf class

The second test we did is similar to that of the Cstring class, which also provided automated unit tests. This time we used a class that implements the print() functionality, with all its variations (i..e sprint(), etc…).

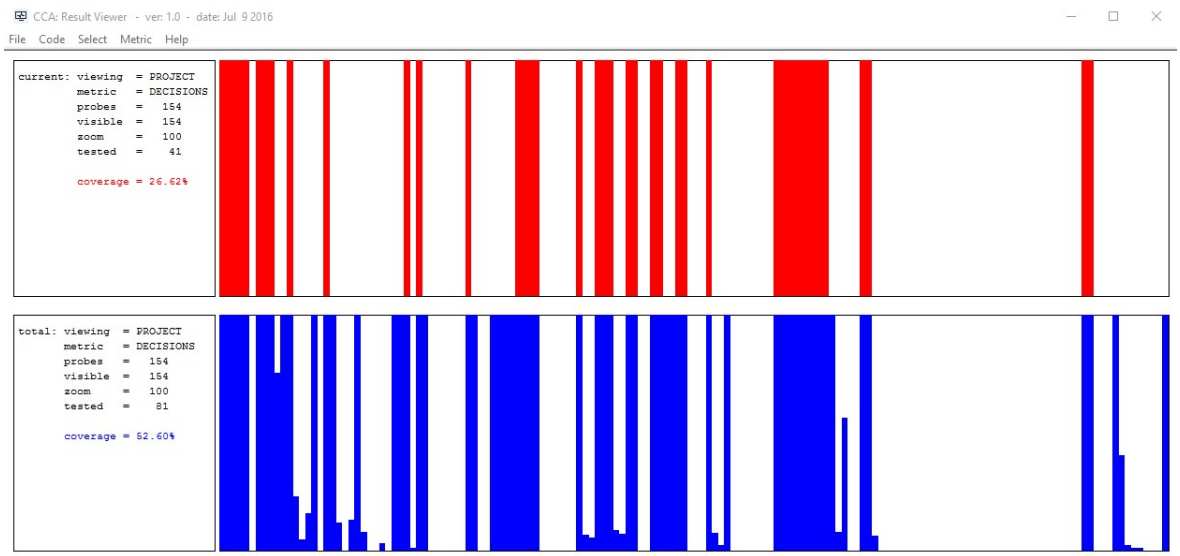

Figure 18: Test coverage of the printf class

Again, here we run the tests stand-alone, and then under CCA. The results were identical. The test coverage is shown in Figure 18.

7.2.3 Text Editor

The third test we executed with CCA is a text editor for windows, for which we had the source code. As shown in Figure 19, this program comprises of eleven modules, and about 8,000 lines of code.

The instrumentation without –i gave 1,033 probes while the instrumentation with –i resulted in 1,777 probes. There are no automated test cases available for this package, therefore, out testing effort was to use the instrumented editor in everyday tasks. We used the instrumented editor for more than two weeks in everyday activities. We did not notice any problems whatsoever. Figure 19 shows the test coverage data, after a one-day causal use of the editor.

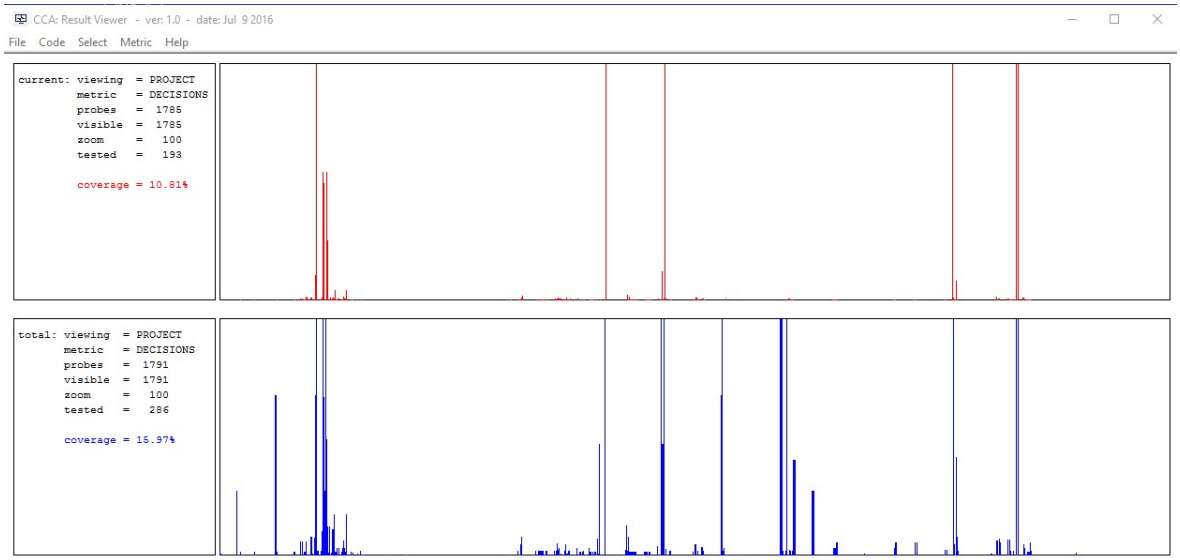

Figure 19: Test coverage of the editor after one day of normal use

7.3 Test Coverage of CCA

Finally, for the tests described in 7.1 and 7.2 we wanted to see how much of the CCA code we exercised. So we used CCA to instrument itself, and run all above tests again.

The results are shown below:

Table 5: Code coverage of the CCA testing

Test Case	Percent of Coverage	Total Coverage
Test strings	4.80 %	4.80%
Test parenthesis	9.60 %	10.65 %
Test comments	10.42 %	16.42%
Test arrays	6.07%	19.96 %
Test braces	21.95 %	33.47 %
ccaProbe running cca_parser object	45.45 %	50.62 %

The coverage we achieved after running all above test is shown bellow

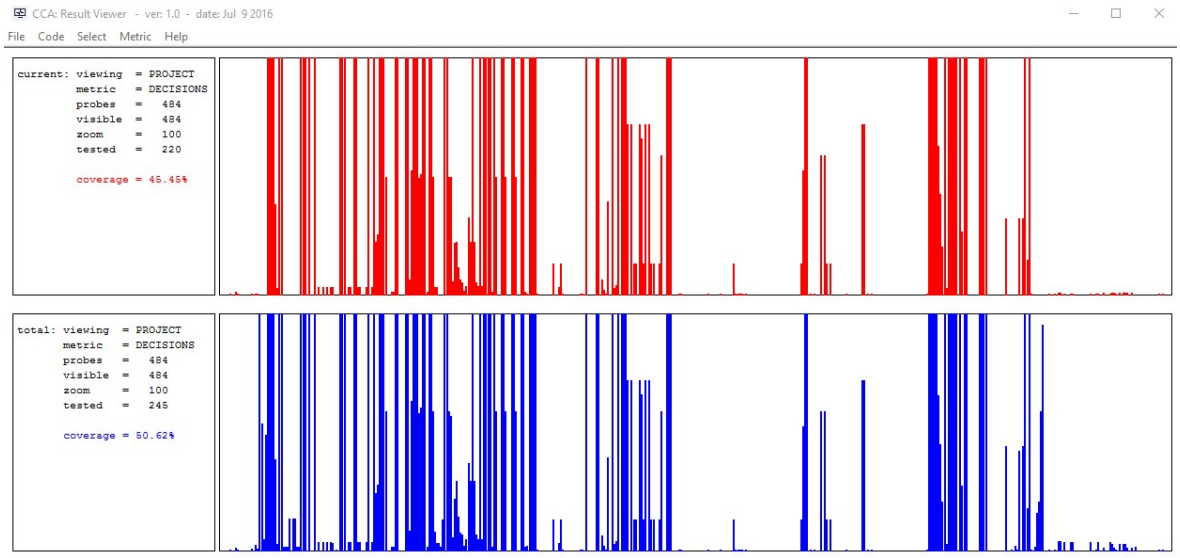

Figure 20: Test coverage of CCA test suite

7.4 Comparison of CCA with GCOV

As stated in 5.1.2 GCOV is a built-in functionality of GNU g++ compiler that provides code coverage functionality.

The last test we performed on CCA is to compare its output with that of the GCOV tool on a sample program shown on Figure 21.

Because the gcov report is very different that the CCA report, we show in Figure 21 the report taken from gcov, and we added manually, the info extracted from CCA, thus creating a table with the following columns.

- **Gcov :** shows info taken from gcov, with the following meaning:
  - **####** means that this line was not executed,
  - **-:** means no code in this line
  - **1:** means this line was executed one time.
- **CCA:** shows the same results from CCA (converted manually CCA's results to match GCOV's output for comparison reasons)
- **Code** – shows the code we used in the test

The comparison of the gcov and cca columns reveals the following:

1. For the most part both tools produce the same output.
2. There is, however some cases (marked with ** in Fig xx), where the two tools differ in the code coverage measurement.

   These cases are in the coverage of if() statements and how they interpret the result: GCOV marks an if() statement as executed if the predicate of the if is evaluated, irrespective if the branch is taken. CCA marks the same if() as executed ONLY if the branch is taken.

```
      Gcov          CCA     Code
        -:          -:    1:  #include <stdio.h>
        -:          -:    2:  #include <stdlib.h>
        -:          -:    3:  #include <string.h>
        -:          -:    4:
        -:          -:    5:  #define F_MAX 5
        -:          -:    6:
        1:          1:    7:  int func1( int val )
        -:          1:    8:  {
        1:          1:    9:     printf("%s: invoked\n", __FUNCTION__ );
        -:          -:   10:
        1:          1:   11:     if ( 1 == val )    printf("   branch on val == 1 taken\n") ;  else
    #####:      #####:   12:     if ( 2 == val )    printf("   branch on val == 2 taken\n") ;  else
    #####:      #####:   13:     if ( 3 == val )    printf("   branch on val == 3 taken\n") ;
        -:          -:   14:
        1:          1:   15:     return 0 ;
        -:          -:   16:  }
        -:          -:   17:
        1:          1:   18:  int func2( int val )
        -:          -:   19:  {
        1:          1:   20:     printf("%s: invoked\n", __FUNCTION__ );
        -:          -:   21:
        1:  **  #####:   22:     if ( 1 == val )    printf("   branch on val == 1 taken\n") ;  else
        1:          1:   23:     if ( 2 == val )    printf("   branch on val == 2 taken\n") ;  else
    #####:      #####:   24:     if ( 3 == val )    printf("   branch on val == 3 taken\n") ;
        -:          -:   25:
        1:          1:   26:     return 0 ;
        -:          -:   27:  }
        -:          -:   28:
    #####:      #####:   29:  int func3( int val )
        -:          -:   30:  {
    #####:      #####:   31:     printf("%s: invoked\n", __FUNCTION__ );
        -:          -:   32:
    #####:      #####:   33:     if ( 1 == val )    printf("   branch on val == 1 taken\n") ;  else
    #####:      #####:   34:     if ( 2 == val )    printf("   branch on val == 2 taken\n") ;  else
    #####:      #####:   35:     if ( 3 == val )    printf("   branch on val == 3 taken\n") ;
        -:          -:   36:
    #####:      #####:   37:     return 0 ;
        -:          -:   38:  }
        -:          -:   39:
    #####:      #####:   40:  int func4( int val )
        -:          -:   41:  {
    #####:      #####:   42:     printf("%s: invoked\n", __FUNCTION__ );
        -:          -:   43:
    #####:      #####:   44:     if ( 1 == val )    printf("   branch on val == 1 taken\n") ;  else
    #####:      #####:   45:     if ( 2 == val )    printf("   branch on val == 2 taken\n") ;  else
    #####:      #####:   46:     if ( 3 == val )    printf("   branch on val == 3 taken\n") ;
        -:          -:   47:
    #####:      #####:   48:     return 0 ;
        -:          -:   49:  }
        -:          -:   50:
    #####:      #####:   51:  int func5( int val )
        -:          -:   52:  {
    #####:      #####:   53:     printf("%s: invoked\n",   FUNCTION   );
```

```
        -:          -:   54:
    #####:      #####:   55:      if ( 1 == val )    printf("   branch on val == 1 taken\n") ;  else
    #####:      #####:   56:      if ( 2 == val )    printf("   branch on val == 2 taken\n") ;  else
    #####:      #####:   57:      if ( 3 == val )    printf("   branch on val == 3 taken\n") ;
        -:          -:   58:
    #####:      #####:   59:      return 0 ;
        -:          -:   60:   }
        -:          -:   61:
    #####:      #####:   62:   void Help()
        -:          -:   63:   {
    #####:      #####:   64:      printf("Sample program to demonstrate the code coverage of CCA and GCOV\n");
    #####:      #####:   65:      printf("\n");
    #####:      #####:   66:      printf("synatx:  gTest <func nom>  <func value>\n");
    #####:      #####:   67:   }
        2:          2:   68:   int main( int argc,char *argv[] )
        -:          -:   69:   {
        2:  **  #####:   70:      if ( 3 != argc ) { Help(); return 0 ; }
        -:          -:   71:
        2:          2:   72:      int fno  = atoi( argv[1] ) ;
        2:          2:   73:      int fval = atoi( argv[2] ) ;
        -:          -:   74:
        2:  **  #####:   75:      if ( fno < 1 || fno > F_MAX ){printf("fno must be 1 .. %d\n", F);return -1;}
        -:          -:   76:
        2:          2:   77:      if ( 1 == fno ) return func1( fval ) ;
        1:          1:   78:      if ( 2 == fno ) return func2( fval ) ;
    #####:      #####:   79:      if ( 3 == fno ) return func3( fval ) ;
    #####:      #####:   80:      if ( 4 == fno ) return func4( fval ) ;
    #####:      #####:   81:      if ( 5 == fno ) return func5( fval ) ;
        -:          -:   82:
    #####:  **      1:   83:      return 0 ;
        -:          -:   84:   }
        -:          -:   85:
```

Figure 21: Comparison of test results of CCA and gcov

8. CONCLUSIONS

This thesis presented the design of the CCA, a tool designed to measure the most important code coverage metrics for C/C++ programs. During this process, several such tools available today, have been reviewed and compared with our design.

From the survey it has been shown that even though many tools exist today, most of them are using binary instrumentation and are targeting Java programs.

Very few tools have been found that use source code instrumentation, target the C/C++ language and are freely provided in the public domain.

CCA, as presented in this paper, is a fully functional tool, providing a rather rich set of information to the tester, comparable with that of other mature systems. The tests performed so far do not reveal any major problems. The score, as stated in the prospectus, is fully covered. The statistics measured are sufficient to derive all metrics stated in program specifications section, even though some more work is needed in the GUI in order to present them to the user in a user-friendly way.

The strong points of CCA compared to other tools are:

1) Light-weight portable library,
2) Integrable with any other tool (by means of being an ANSI C++ library)
3) Stand-alone
4) Free

From similar tools found free on the Internet we compared the GCOV tool with CCA, and found that both tools produce similar results, but for decisions, they measure slightly different metrics: GCOV measures predicate evaluated, while CCA measures branch taken. For example, for the code fragment

```
if (p) s1; else p2;
```

if (p) evaluates to true then GCOV produces the information “(p) was executed” while CCA produces the information “(p) was executed, s1 was executed”, whereas if (p) evaluates to false then GCOV produces the information “(p) was executed” while CCA while CCA produces the information “(p) was executed, s2 was executed”,

If the code was written as follows, then both tools produce identical information.

```
if (p)
  s1;
 else
  p2;
```

Although the version of CCA described in this paper is fully functional, it is the first version to be soon released on GitHub, with a life of just a few weeks, and it is expected to have bugs. In addition, several features will be added in the next release. Amongst those are:

1. Support for Java,
2. Support for C#
3. Highlight tested code against untested and show in GUI
4. Improve performance

5. Provide statement coverage as a post-run process extracting this information from the data recorded during testing
6. Place statistics to a database

VOCABULARY

- **Software Engineering –** is an engineering discipline which encompasses all the activities of producing a software system, including the specification of the system, the development, and testing, and well as the management of its life cycle.
- **Software Testing –** is the set of activities involved to verify a software system behaves according to the specifications stated and the stake holders are happy with its behavior
- **Quality Assurance –** is a set of activities aiming in preventing mistakes or defects in a software system.
- **Error –** is a state in a program that can potentially lead to an unwanted behavior commonly referred to as fault
- **Bug** – is an error or flaw in a system that causes it to produce an unexpected behavior or incorrect result.
- **Code Coverage** – refers to the percentage of the software code being executed at least once during testing.
- **Test Coverage** – refers to measuring of how much a program has been tested. Frequently is used as synonymous to code coverage. Alternatively is used to denote the percentage of the test cases that have been executed successfully.

ACRONYMS

LOC	Lines of Code
KLOC	1000 Lines of Code
QA	Quality Assurance
GA	General Availability
CC	Code Coverage
CCA	Code Coverage Aid
SCI	Source Code Instrumentation
BCI	Binary Code Instrumentation

BIBLIOGRAPHY

[1] Steve McConnell, Code Complete, Microsoft Press, 2004.

[2] Glenford J. Myers, The art of Software Testing, 2nd, Επιμ., John Wiley & Sons, 2004.

[3] Dr. Neetu Dabas, Mrs. Kamma Solanki, "Comparison of Code Coverage Analysis Tools: A Review," *International Journal of Research in Computer Applications & Information Technology,* vol. 1, no. 1, pp. 94-99, 2013.

[4] Roger S. Pressman, Software Engineering: A Practitioner's Aproach, McGraw-Hill, 1997.

[5] P. Crosby, "Quality is Free: The Art of Making Quality Certain," McGraw-Hill Book Company, 1980.

[6] "Code Coverage," Wipikedia.org, [Online]. Available: https://en.wikipedia.org/wiki/Code_coverage.

[7] "Integration testing - Wipikedia," [Online]. Available: https://en.wikipedia.org/wiki/Integration_testing.

[8] Ian Sommerville, Software Engineering, Addison-Wesley, 1995.

[9] M. N. Norman E Fenton, Software Metrics: Roadmap.

[10] Muhammad Shahid, Suhaimi Ibrahim, "An Evaluation of Test Coverage Tools in Software Testing," *2011 Int'nal Conference on Telecomuunications Technology and Applications,* vol. 5, pp. 216-222, 2011.

[11] S. N. Sneha Shelke, «The Study of Various Code Coverage Tools,» *International Journal of Computer Trends and Technology,* τόμ. 13, αρ. 1, pp. 46-49, 2014.

[12] K. A. a. K. Magel, "Examining the Effectiveness of Testing Coverage Tools: An Empirical Study," *International Journal of Software Engineering and Its Applications,* vol. 8, no. 5, pp. 139-162, 2014.

[13] Qian Yang, J. Jenny Li, David Weiss, «A survey of Coverage Based Testing Tools,» *The Computer Journal,* pp. 99-103, 2006.

[14] "Gcov - Using the GNU Compiler Collection (GCC)," [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[15] "Solaris and Linux Code Coverage," [Online]. Available: http://www.dynamic-memory.com/ww1/products-services/quality-tools/code-coverage/.

[16] "TestWorks/Coverage TCAT," [Online]. Available: http://www.testworks.com/Products/Coverage/tcat.html.

[17] "TestWell CTC++," [Online]. Available: http://www.testwell.fi/ctcdesc.html.

[18] "Squish Coco - Test Coverage," [Online]. Available: https://www.froglogic.com/squish/coco/.

[19] «Ope Code Coverage Framework,» [Ηλεκτρονικό]. Available: https://github.com/exKAZUu/OpenCodeCoverageFramework.

[20] "Semantic Designs - Code Coverage Tools," [Online]. Available: http://www.semdesigns.com/Products/TestCoverage/CodeCoverage.html?Home=Tools.

[21] "Agitar Technologies - Puting Java to the test," [Online]. Available: http://www.agitar.com/.

[22] "Jtest - Automated Parasoft Java Testing tools," [Online]. Available: https://www.parasoft.com/product/jtest/.

[23] "Koalog Code Coverge," [Online]. Available: http://freecode.com/projects/kcc/.

[24] "IBM Rational Purify & PurifyPlus Divestiture," [Online]. Available: http://www-01.ibm.com/software/rational/products/purifyplus_divestiture/.

[25] "A Multipurpose Code Coverage Tool for Java," [Online]. Available:

https://www.researchgate.net/publication/221181402_A_Multipurpose_Code_Coverage_Tool_for_Java.

[26] "The Open Source Resuirements Coverage Tool," [Online]. Available: http://www.technobuff.net/webapp/product/showProduct.do?name=jfeature.

[27] "Java Code Coverage Analyzer - JCover," [Online]. Available: http://www.mmsindia.com/JCover.html.

[28] "Cobertura - A code coverage utility for Java," [Online]. Available: http://cobertura.github.io/cobertura/.

[29] "EMMA: a free Java code coverage tool," [Online]. Available: http://emma.sourceforge.net/.

[30] "Java and Groovy code coverage," [Online]. Available: https://www.atlassian.com/software/clover.

[31] "Code Coverage Analytics Tool," [Online]. Available: http://www.codecoverage.com.

[32] "CodeCover - An open-source glass-box testing tool," [Online]. Available: www.codecover.org.

[33] "Jester - the JUnit test tester," [Online]. Available: http://jester.sourceforge.net/.

[34] "GroboCode Coverage," [Online]. Available: http://groboutils.sourceforge.net/codecoverage/index.html.

[35] "Hansel," [Online]. Available: http://hansel.sourceforge.net/.

[36] "Gretel," [Online]. Available: http://www.cs.uoregon.edu/research/perpetual/dasada/software/Gretel/docs/index.html.

[37] "Bullsey Testing Technologies," [Online]. Available: http://www.bullseye.com/.

[38] "NCover - .NET Code Coverage," [Online]. Available: www.ncover.com.

[39] "Using Jazz Code Coverage," [Online]. Available: http://realsearchgroup.org/SEMaterials/tutorials/code_coverage/.

[40] "Code Coverage Tools," [Online]. Available: http://c2.com/cgi/wiki?CodeCoverageTools.