

TOOL SUPPORT FOR AUTOMATION OF C++ TEST CASE GENERATION

Imaran Shyabith Maher Dickwella

(168217P)

Degree of Master of Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

February 2020

TOOL SUPPORT FOR AUTOMATION OF C++ TEST CASE GENERATION

Imaran Shyabith Maher Dickwella

(168217P)

Thesis/Dissertation submitted in partial fulfillment of the requirements for the degree Master
of Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

February 2020

DECLARATION

I declare that the content of this research is my own work and the CS5999 PG Diploma Project Report does not include any content previously submitted for a Degree or Diploma in any other University or institute of higher learning without acknowledgement and to the best of my knowledge and belief, this report does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to the University of Moratuwa the non-exclusive right to regenerate and distribute my thesis/dissertation, in whole or in part in print, electronic or other media. I retain the right to use this content in whole or part in future works (such as articles or books).

.....

.....

Imaran Shyabith Maher Dickwella

Date

The above candidate has carried out this CS5999 PG Diploma Project Report under my supervision.

.....

.....

Dr. Indika Perera

Date

ABSTRACT

Testing of software plays a vital part in the software development process. It is the phase in the software development life cycle that make sure the developed software is functionally correct. Software faults can be expensive when the fault is found at the production system therefore it is essential to find software errors at the development stages of the project when the cost is minimum. Testing makes sure the software which is developed covers functional and nonfunctional requirements.

Unit tests take an essential place in software testing and it is the earliest test a developer or a tester can perform on the implementation, defects can be detected at early stages, and fixes can be done with lesser cost due to lesser dependencies. Even though it is essential to implement unit tests, it is not the case in reality. Most software companies rely heavily on end to end testing. The main reason for lack of testing at the unit test level is due to its time-consuming nature and it will not provide functional coverage whereas by performing end to end tests, end-user requirements can be captured and tested. In reality, approximately two-thirds of the development time is spent on unit testing related activities and the rest of the time is spent on designing and implementation. However, due to time constraints and budget limitations, allocating a large portion of the time for unit testing is not practical.

In this research, I came up with a solution to address the above-mentioned issues and to eliminate the requirement of writing unit tests manually. I implemented a tool which generates unit tests for applications that are implemented in C++. The process is completely automated and the unit test files generated by the tool are human-readable. Developers no longer need to implement unit tests however they may have to validate the correctness of the generated unit tests and as desired they can extend the meaningfulness of the generated unit tests.

The tool is embedded with three test data generation mechanisms; Random Value Generation, Goal Oriented Test Generation and Feedback driven Test Data Generation. The tool successfully produced test data to attain approximately 95% of code coverage with data generation mechanisms except Random value generation in multiple test experiments however when the test unit has a higher number of branches Feedback driven test data generation mechanism showed better performance as it took lesser time for data generation. Time growth is exponential when the number of branches gets increased in Goal Orientated Test Generation. This tool can be extended for complex structures and I can successfully conclude that the research is successful as the tool showed higher accuracy.

ACKNOWLEDGEMENT

My sincere thanks go to my project supervisor Dr. Indika Perera for guiding me throughout the project and giving a clear vision and a mission to accomplish during this period. It is essential to mention my friends and the staff of MillenniumIT software engineering Pvt Ltd for supporting me throughout the MSc course.

I take this moment to thank my parents for the support and the encouragement given to follow the MSc. in Computer Science amidst this busy schedule at working place. Last but not least, my sincere thanks go to the Head of Department of Computer Science and Engineering and the members of the academic staff for advising and guiding me during project evaluations and presentations.

Table of Contents

DECLARATION	i
ABSTRACT	ii
ACKNOWLEDGEMENT	iii
1 INTRODUCTION	2
1.1 Background	2
1.2 Unit Testing and Test Case Generation	5
1.2.1 Definition of unit testing	5
1.2.2 Disadvantages of manual unit tests generation	5
1.2.3 When to use unit test generation	6
1.2.4 Unit testing types	7
1.2.5 Other uses of unit testing	10
1.3 Motivation	11
1.4 Research Problem and Objectives	12
2 LITERATURE REVIEW	14
2.1 Program versus Specification-based Test Generation	14
2.2 Code Based Test Generation Methods	14
2.2.1 Static structural test data generation	15
2.2.1.1 Symbolic execution based technique	15
2.2.1.2 Domain reduction	16
2.2.1.3 Search-based techniques	16
2.2.2 Dynamic structural test data generation	16
2.2.2.1 Random selection	17
2.2.2.2 Applying local search	17
2.2.2.3 Goal oriented test generation	18
2.3 Comparison of Test Generation Approaches	19
2.4 Tools	19
2.4.1 KLOVER	19
2.4.2 KLEE	20
2.4.3 CREST	21

2.4.4	CAUT	21
2.4.5	DART	22
2.4.6	CUTE	22
2.4.7	PathCrawler	23
2.4.8	AgitarOne	24
2.4.9	CATG	24
2.4.10	EvoSuite	24
3	METHODOLOGY	27
3.1	Proposed Solution	27
3.1.1	High-level design	27
3.2	Scrutiny of the Solution	29
3.3	Progress	31
3.4	Evaluation Methodology	31
4	SYSTEM ARCHITECTURE AND THE IMPLEMENTATION	33
4.1	System Overview	33
4.1.1	Infrastructure layer	33
4.1.2	Application layer	33
4.1.3	Functional layer	33
4.2	Random Value Generator	34
4.3	Feedback Driven Value Generator	35
4.4	Goal Oriented Value Generator	36
4.5	Architecture of Code Generator	38
4.5.1	Code parser	39
4.5.2	Fitness function evaluator module	41
4.5.3	Analyzer	43
4.5.4	Code generator	44
4.5.5	Report generator	45
4.6	Parameters	45
5	EVALUATION	47
5.1	Overview	47

5.2	Evaluation of the Tool	47
5.2.1	Evaluation of feedback driven unit test generator	48
5.2.1	Evaluation of goal oriented test generator	52
5.3	Algorithm Comparison	55
5.3.1	Comparison of code coverage	61
5.3.2	Comparison of time	62
5.3.3	Proof of concept	63
5.3.4	Line coverage	66
5.3.5	Branch coverage	66
6	CONCLUSION	69
6.1	Summary	69
6.2	Contribution	71
6.3	Limitations	72
6.4	Future work	73
	Reference	74
	Appendix A – Random Test Generation Method	77

List of Figures

Figure 1 - Test pyramid	3
Figure 2 - Sample library interface.....	6
Figure 3 - Sample function implementation.....	6
Figure 4 - Sample Algorithmic function	7
Figure 5 - Sample Test Case.....	7
Figure 6 - Sample class implementation	8
Figure 7 - Expected test class	9
Figure 8 - Sample function.....	9
Figure 9 - Test generation flow	11
Figure 10 - Overall architecture of KLOVER [12]	20
Figure 11 - High level design	28
Figure 12 - Function to be tested in KLEE engine.....	29
Figure 13 - Marking symbolic variables for KLEE engine.....	30
Figure 14 - Unit test generator implementation	33
Figure 15 - Sample function II	35
Figure 16 - Execution tree	36
Figure 17 - Sample function III.....	36
Figure 18 - Unit test Module chain.....	38
Figure 19 - Field class	39
Figure 20 - Parameter class	40
Figure 21 - Method class	40
Figure 22 - Constructor class.....	41
Figure 23-Branch Distance computation	42
Figure 24-Sample function for fitness calculation.....	42
Figure 25 - Derived Fitness Function for Figure 27.....	42
Figure 26 - Generated unit test block	43
Figure 27 - Generated Test file.....	44
Figure 28 - Header Class	48
Figure 29 - Implementation class with std data structures	49
Figure 30 - LCOV report for the data structure class.....	50
Figure 31 - Time analysis for the data generation.....	51
Figure 32-Class I.....	52
Figure 33 - Header file for the above class.....	52
Figure 34 - Coverage report for class in figure 30	53
Figure 35 - Time measures for Class I with Goal Oriented Test data.....	54
Figure 36 - Unachievable branch statements.....	54
Figure 37 - Class A.....	55
Figure 38 - Random Generator Result for Class A	55

Figure 39 - Goal Oriented Result for Class A	55
Figure 40 - Class B	56
Figure 41 - Goal Oriented Result for Class B	56
Figure 42 - Random Generator Result for Class B.....	56
Figure 43 - Class C	57
Figure 44 - Goal Oriented Result for Class C	57
Figure 45 - Random Generator Result for Class C.....	57
Figure 46 - Goal Oriented Result for Class D.....	58
Figure 47 - Random Generator Result for Class D	58
Figure 48 - Class D.....	58
Figure 49 - Class E	59
Figure 50 - Random Generator Result for Class E.....	59
Figure 51 - Goal Oriented Result for Class E.....	59
Figure 52 - Random Generator Result for Class F.....	60
Figure 53 - Class F	60
Figure 54 - Goal Oriented Result for Class F.....	60
Figure 55 - Code Coverage Comparison Table.....	61
Figure 56 - Time Analysis Table.....	62
Figure 57 - Pseudo code for search logic	63
Figure 58 - Example for proof of concept case 1	64
Figure 59 - Example for proof of concept case II	65
Figure 60 - Proof of concept case III.....	65
Figure 61 - Time Comparison Vs Branches	67

List of Abbreviations

Abbreviation	Description
CUT	Class under test
E2E	End to End
API	Application Programming Interface
HTTP	Hyper Text Transfer Protocol
SO	Service Orientation
SOA	Software Oriented Architecture
TDD	Test Driven Development
GUI	Graphical User Interface
GTest	Google Test
GMock	Google Mock
CFG	Control Flow Graph
SUT	System under test

CHAPTER 1

INTRODUCTION

1 INTRODUCTION

1.1 Background

Software testing is an important phase in the software development life cycle. It improves the quality as well as the reliability of the developed software and also it can benefit in various ways for a software development project. It helps developers identify code bugs and also helps identify requirement errors. Eg: When two test cases show contradicting results it can be due to conflicting requirements. When potential conflicts are identified, these conflicts must be rectified as quickly as possible. Also, testing helps the development team to confirm that the system works as per the requirements [25]. If due to a bug system behavior differs from the expected result depending on the scenario it can cause a big issue. Testing helps identify such issues and the development team can attend to it and rectify those issues quickly. There are two ways to test a system; manual verification of the system to see it works to expectations, or automated test suite to verify the functionality. Currently, manual testing is a rare occurrence. Given the assumption that the code is changing rapidly and the system needs to be tested manually to assure the changes do not break the existing functionality, every time a feature is implemented all system scenarios need to be tested in the manual testing process, thus making the process quite costly. With rapid change requests coming from clients now the software industry has a requirement to deliver the software as quickly as possible. Hence in order to cater to this requirement, continuous integration and continuous delivery have been introduced and with that automated test suites have come into the picture. Typically, a system or a software testing is planned from the requirement elicitation phase. Designing and modeling of test cases usually start at the initial stages of the software development process. eg: In agile environments such as Test-Driven Development (TDD) tests cases are written even prior to the code being written. Assertions are agreed upon as acceptance criteria. When a new feature is introduced to the code path new test cases need to be added to the existing test suite. Typically, software or a system is thoroughly tested internally before shipped out to the customer. Hence testing is identified as one of the most crucial phases in the software development process.

There are different ways to test a software solution namely; unit testing, component testing, performance testing, GUI tests, and E2E testing and out of all mentioned methods unit testing is considered to be the most important testing procedure [11] [26].

The reason is that each function of the software is tested as units and it helps identify issues of the software at the early stages [8] [10] [25]. Also, unit testing is much closer to the implementation stage, sits at the bottom of the testing phases and hence makes easier for the developer to test their implementation. If unit tests are carried out accurately this will enable in minimizing issues encountered at the integration level therefore the cost is minimum [8].

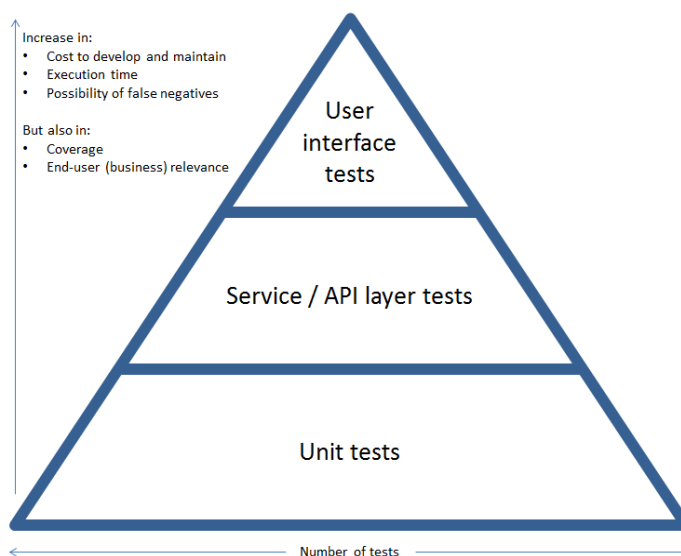


Figure 1 - Test pyramid

Implementation of the code and unit testing is tightly coupled and therefore typically are done simultaneously. Writing unit tests can be a tedious task, usually, an average developer takes more than fifty percent of the time that is allocated for the feature for the unit test implementation [10]. Once the class level design is finalized developers can easily do the implementation of the code. However, when it comes to unit testing it is a cumbersome process, the reason being there can be complex dependencies and resolving may be time-consuming. If unit testing is not considered during the designing phase it will be complicated to implement unit tests later and as there is no proper format or guidelines to follow for manual unit testing, it makes matters even worse. Some of the common problems in unit testing are mock classes and stub implementations are required in order to isolate and test a particular

function, input data must be finalized to test the function, how much variation is needed to say confidently that the function is properly tested, all the branches need to be tested. Lack of support from underlying libraries makes it more difficult to implement unit tests and therefore deep thorough analysis is required before actually implementing unit tests. However, there are a few drawbacks to testing the system with unit tests. While it is true that testing can detect presence of issues. However it cannot detect or identify missing features. If the developer failed to capture a requirement during the implementation which presents in the requirement specification there is a chance that the developer misses the requirement during the testing phase as well. Maintainability issues can arise when a function is modified and the developer may have to modify the unit test to align with the modified implementation otherwise there will be breaking test cases running on a day to day basis. As unit tests are quite tedious and time-consuming most companies are reluctant to apply unit tests. Therefore unit testing has become a frustrating and time-consuming process and becomes a burden to an organization and as a result, some organizations are reluctant to use unit tests for their application testing.

The main objective of this research is to address the problems stated above and implements a tool that will take a C++ class as the input and the tool proposed will generate unit tests for that particular input class. The tool should be able to provide a higher statement coverage, branch coverage, input coverage, and output coverage. Automating unit test generation will help developers to increase their code coverage, also help identify assertions, potential crashes and verify the functionality. A company can benefit greatly from such a tool as it will increase the confidence in its implementation. If the code is not fully tested in unit tests it may result in system misbehavior, leading to the system generating invalid outputs which could be a costly affair for a company. However, there are quite a few issues that need to be discussed before applying unit tests; how effective would be the automation process be? Can the process identify bugs in the software? How to identify whether the code catches exceptions properly? And how to identify the true meaning of the test? What would be the steps taken to solve the above issues? These areas will be discussed in detail in the chapters to follow.

This research consists of the existing literature available in the above area as well as the proposed methodology to generate unit tests for code implementations and tool

implementation details, results obtained based on the test data selected, system evaluation based on the obtained result and finally the conclusion.

1.2 Unit Testing and Test Case Generation

This section gives an introduction about the unit testing, branch coverage of function implementations, and output coverage of functions to detect functional issues and also focuses on test case generation and will discuss the impact of unit testing on software systems.

1.2.1 Definition of unit testing

“**Unit testing** is a process which takes the smallest testable parts of an application such as functions, as units, and then are independently tested for correct/expected behavior. **Unit testing** is usually an automated process however rarely it can also be done manually.” [25]

In this report, the Unit is regarded as a code segment which can be considered as the smallest building block of a program. And the smallest building block of a program is a **function**. Therefore, the smallest testable part of a program is a function implementation.

1.2.2 Disadvantages of manual unit tests generation

It takes a developer more time to write unit tests compared to actual implementation. Once the class level design is finalized converting that design into implementation is considered an easy task by many developers however due to complexities, unit test implementation can be a difficult task to achieve. The first steps would be to design test cases, model input and to finalize methods of testing. Since manual unit testing has no proper format or guidelines to follow and due to time constraints, it is impractical to test all functions and its code paths. Some of the common problems incurred in unit testing are mock classes and stub implementations that need to be available in order to test a particular function. If the stubs or mock classes are not in place when writing unit tests, the developer first needs to implement stubs/mocks in order to test implemented class functions. Developers regularly face these kinds of issues when it comes to testing and it sometimes takes more time than originally planned when writing unit tests. Another common problem is a lack of support from

underlying libraries and therefore unit tests must be designed first before the actual implementation. When unit tests are implemented it can associate with maintainability problems like when a function is modified developer may have to modify unit tests as well to assure there are no code breakages. Therefore unit testing has become a complicated and time-consuming process and has become a burden to some organizations and as a result, some organizations have taken alternatives for unit tests. Example:

Assume the following functions are provided by an underlying library: (C++ implementation)

```
bool delete();  
  
bool add();  
  
bool modified();
```

Figure 2 - Sample library interface

Function to be tested:

```
boolonAdd()  
{  
    booli = add();  
    returni;  
}
```

Figure 3 - Sample function implementation

Since add function is not virtual, testing on Add function is not possible. Even if they add function provided by the library is virtual, a mock function needs to be implemented if it is not available in the library.

1.2.3 When to use unit test generation

Even though unit testing is a time-consuming process its advantages are significant. If functions are tested as units, developers confident about the code increases and this will assist to solve most of the functions related to bugs. Unit tests may be very useful for certain applications. For example, mission-critical applications such as medical applications, banking applications, etc these types of applications need to be thoroughly tested. End to end testing

alone will not get the job done. Therefore implementation level testing needs to be emphasized.

1.2.4 Unit testing types

Unit testing can be segmented into different categories. Branch coverage testing to detect unexpected exceptions is one type of unit testing. Test code should be able to explore different paths of the code in a way that it covers the majority of the code. Another type of unit testing is output coverage testing where the output of the function is taken into consideration and assertions are generated for the developer to validate.

Following is a sample case of unit tests:

```
int array_sum(int arr[], int length)
{
    int sum = 0;
    for(int i = 0; i < length; i++)
        sum = arr[i] + sum;
    return sum;
}
```

Figure 4 - Sample Algorithmic function

If a test is written for this it would take this format;

```
TEST_F(testSumArray)
{
    int sampleArr[] = {2, 4, 6};
    EXPECT_EQ(12, array_sum);
}
```

Figure 5 - Sample Test Case

The tool generates unit tests by exploring code paths (eg: statements) and the tool makes an assumption that the code functionality is correct. This has been the main drawback of this tool that it can detect the presence of a bug but it cannot detect an absence of a feature. If the tool is required to identify a missing feature tool needs additional information to model this problem and this problem will not be captured in this research.

Following is a class to be tested:

```
class Test1
{
    Public:
        Test1 (Test2 *t) {m_t = t;}
        ~ Test1 () {}
        void Call1 ()
        {
            M_t->Call2();
        }
    Private:
        Test2* m_t;
}

class Test2
{
    Public:
        Test2 () {}
        ~ Test2 () {}
        int Call2()
        {
            Return 5;
        }
}
```

Figure 6 - Sample class implementation

In order to test this class we need to come up with a test class as follows:

```
Class TestTestA{
Public:
    Voidsetup()
    {
        a = new TestA(&b);
    }
    Voidteardown()
    {
    }
Private:
    TestA *a;
    MockB b;

    TEST_F(TestTestA, TestCall1)
    {
        a->Call1();
        EXPECT_CALL(b.Call2()).Times(1);
    }
}
```

Figure 7 - Expected test class

Branch coverage example:

```
Void addInformation(int type, string information)
{
    if(type == 1)
    {
        addName();
    }

    Else if(type == 2)
    {
        addAddress();
    }

    Else if(type == 3)
    {
        addContact();
    }

    Else if(type == 4)
    {
        addWorkInfo();
    }
}
```

Figure 8 - Sample function

To test this class we need to test all the branches therefore according to this example it is required to have 4 different tests to test this class.

1.2.5 Other uses of unit testing

Apart from bug detection, there are other advantages in writing unit tests as well. Some of them are as follows.

- Improve code quality
- Improve coding standards
- Facilitates changes
- Simplifies integration
- Provides documentation

When the function to be tested seems difficult to test we usually segment them into smaller functions. Thus it could result in improving the code quality.

A unit test plan could be used when the code is modified in order to verify that the existing functionalities have not been affected. If the unit test fails, the issue can be rectified accordingly.

1.3 Motivation

As described in the above section there are many ways to test a software system. This research focuses on creating a tool that allows developers to generate unit tests automatically instead of manually writing them. This will reduce the time spent by developers on writing the test enabling developers to focus more on designing the application and code quality when developing. This tool will allow developers to create test cases to cover branches in the code as well as test classes to a certain extent.

Following is the workflow of the tool proposed:

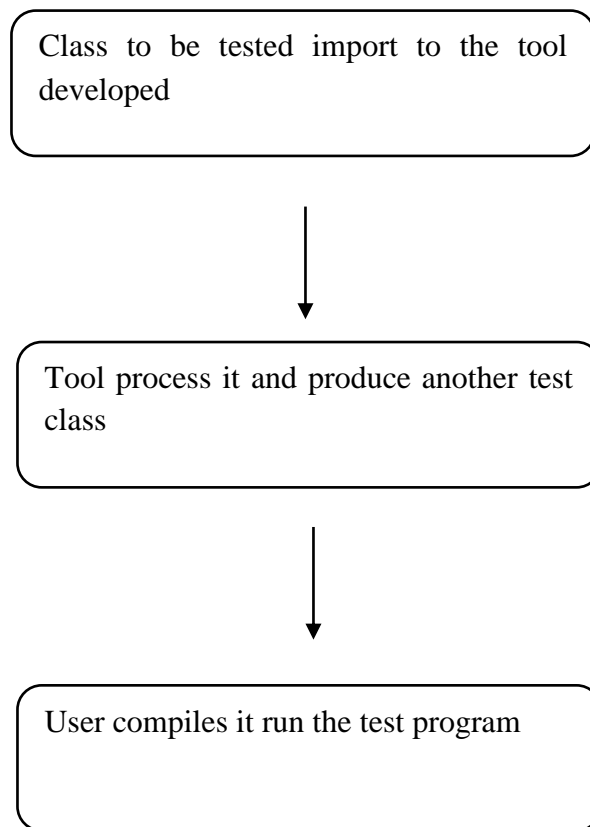


Figure 9 - Test generation flow

1.4 Research Problem and Objectives

There have been various researches done to generate unit tests for given application implementation. But most of these tools have failed due to issues in the outcome. The main issue that will be addressed in this research is how to generate unit tests for a given function first. Then the focus will extend on to the tool to generate unit tests for classes. Existing frameworks and tools such as google test framework, lcov will be used to find out various ways to generate effective unit test cases with implementation. Algorithmic function testing can be challenging and the aim here is not to demotivate developer from manually creating unit tests, but to help them save their time so that they can tweak and complete the test case if the generated one is not meaningful.

For this purpose, the objective is to develop a tool that:

- Generate branch coverage test cases
- Generate test cases to output for a given function
- Generate test cases to improve code coverage

CHAPTER 2

LITERATURE REVIEW

2 LITERATURE REVIEW

2.1 Program versus Specification-based Test Generation

There are two test cases generating methodologies; program based test generation and specification-based test generation. In specification-based test generation, formal documentation is required where the document structure is well defined to generate tests. The drawback of this methodology is specification writers need to learn an additional language to convert the specification document to a document that can be used to generate tests.

Program-based techniques only take the system as the input hence the system under is the only dependency to generate test cases. When the test suite is presented after generation developer may extend these test cases as desired. Program based test generation has few advantages over the specification-based test generation method. One advantage of program-based techniques is that these methods can achieve higher structural coverage compared to specification-based techniques since the source code implementation can provide more information for test generators than abstract specifications to generate a test suite.

2.2 Code Based Test Generation Methods

There are various approaches to generate test cases and these techniques can be categorized as random test generation, path-oriented test generation, goal-oriented test generation and intelligent approach [1]. The random technique assumes faults in the implementation are equally distributed however this is a false assumption. Path oriented approach generally use information about branches and identify the set of paths to be covered and generate tests accordingly. We can further generalize these techniques as static and dynamic test case generation methods. Static techniques often produce test cases after performing symbolic executions while dynamic approach obtains necessary data by running the program under test. The intelligent technique generally relies on complex computations to generate test cases [30].

Code based test Generation falls into following major areas [24]:

- Static Structural Test Data Generation
- Dynamic Structural Test Data Generation

2.2.1 Static structural test data generation

In static structural test data generation, the internal structure of the program is analyzed and generate test data, without actually executing the program.

Following are the main Static Structural Test generation methods:

- Symbolic Execution - Dynamic symbolic execution to generate input data
- Domain Reduction – Employed as part of Constraint based testing
- Search based technique - Generalize the test generation problem as a search-problem

2.2.1.1 Symbolic execution based technique

Symbolic Testing is a static test generation technique. In symbolic execution, program will not be executed with real values, instead it allocates symbolic expressions to variables in the program. This technique generates a set of constraints that traces the conditions that are required to achieve a certain coverage goal [6]. The solution of these constraints are provided as values for input variables. In Symbolic Testing, first it generates set of path restraints and then collects these constraints for the system under test and finally solve these constraints to obtain actual values [1] [16]. The final solution contains the concrete input data that executes these paths. Linear programming techniques can be applied in cases where the constraints are linear. If that is not the case heuristic methods can be used to find the solution [24].

2.2.1.2 Domain reduction

Domain reduction is designed as a constraint based test generation methods and it basically tries to solve domains constraints by providing values for input variables. These can be obtained from type or information from the specification, or from input by the tester. The domains are then deduced using the knowledge in the constraints, which involves the operator, a variable and a constant, or constraints involving a relation operator with two variables. In order to simplify remaining constraints are substituted with values. Likewise this process takes place until no further simplification is possible, then the input variable linked to the smallest remaining domain is selected, and a random value is assigned to it. Then this variable value is substituted in the constraint system, in order to obtain values for the rest of the remaining variables. If all variables can be assigned values with this order, then the constraint system assumed to be successfully completed; otherwise the system repeats variable assignment stage again, hoping with successfully selecting appropriate random numbers for the variables [24].

2.2.1.3 Search-based techniques

Search-based testing approach generalize the test generation problem as a search-problem and solve this problem using random or directed search techniques, such as local search, Hill Climbing, Simulated Annealing, etc [4]. The search algorithm repeatedly runs the program under test with potential input data and uses code mutation and observe the result to adjust the input accordingly. This process repeats until all desired goals are achieved or the pre-defined threshold exceeds usually a timeout.

2.2.2 Dynamic structural test data generation

It is difficult to statically analyze the relationship between input values and internal variables for constructional test data generation when there are loops and computed storage locations. As name implies dynamic methods execute the program with some input data, and then simply observe the results via some form of program instrumentation.

Following are the main Dynamic Structural Test generation methods:

- Random Testing – Treats all branches, functions equally and generate input data randomly
- Applying local search – Select a path and create a straight line through it, containing only that path
- Goal oriented test generation – Starts from a goal and starts producing data until the goal is satisfied

2.2.2.1 Random selection

In this technique, the program selects random sequences of method calls, input random values and creates a unit test with the method sequence tool selects [3]. Asserts can be generated based on the return type and this allows developers to capture the functional behavior. Random testing helps identify software vulnerabilities, like security issues, invalid logics, and behavioral issues. Random testing is considered cheap and probably the easiest to implement. One of the main advantages of random testing is, it can produce a very large number of data set and it can cover a substantial number of bugs at a lower cost. However, the major flaw with this technique is it can produce data that are irrelevant to the program since random test generation assumes the fault distribution in the program is equal in all branches however this assumption is false. Generated test cases also can be duplicated test cases and may only produce a limited number of meaningful data and also there's no guarantee that this technique will produce meaningful data in every single run.

2.2.2.2 Applying local search

A specific path selection is made in the program, and in places where branch statements exist, a path constraint of the form $c_i = 0$; $c_i > 0$; or $c_i \geq 0$ is introduced; where c_i is the estimation of how close the predicate is to being fulfilled. This approach basically constructs a set of path constraints to execute the code through a selected path. For example, suppose there is a branch where the branch condition is $a == b$ and it will be converted into the path constraint; $\text{abs}(a - b) = 0$. This is also called the fitness function. Using these fitness functions obtained, decision can be made whether the constraints are to being fulfilled based on the values returned from

the function, being negative when one or more of the constraints remains unsatisfied, and positive when all of the constraints are satisfied. To execute the code through the interested path, the program is executed with some random input data initially. If the code is executed through an unexpected path – if a deviation is observed from desired path - a local search is called on for program inputs, using the fitness function derived for the desired path, alternative branch. This fitness function explains how close the branch condition is to satisfy being true. The value obtained is referred to as the branch distance [24] [29].

2.2.2.3 Goal oriented test generation

All of the mentioned techniques focus on the execution of a path. In order to obtain a structural coverage criterion like statement coverage, this means all paths needs to discovered such that there no individual uncovered statement. Goal-oriented test generation however neglect this statement. First and foremost Goal oriented test generation constructs a control flow graph from the input program and then, it classify branches in the control flow graph of the program node as either critical, semi-critical or non-essential with respect to a target.

Critical branch is denoted as the edge which drives the execution away from the expected execution path. If the execution follows through a critical branch program will never reach the target hence it is essential in finding an input value such that program does not execute through critical branches. If proper input cannot be found tool terminates. Semi critical branches are the branches that lead the execution to the target node. If the provided input takes the program through a semi critical branch, like in critical branch program doesn't terminate because hoping that correct input will be found in next execution cycle. Nonessential branches do not determine whether the target will be reached, regardless where these nodes are located in the CFG. Hence, execution is allowed through these branches since there is no impact on the final outcome [24].

2.3 Comparison of Test Generation Approaches

In this section test generation methods such as Search-based Testing, Symbolic Testing and Concolic Testing will be compared and evaluated. Recent studies are available to compare performances of Search-based Testing and Symbolic Testing, there have been many attempts to improve the performance of Symbolic Testing and Concolic Testing approaches considerably [19]. Moreover, since constraint solvers takes a very important place in deriving final concrete values over the years the performance of the constraint solvers has improved greatly and recent studies has given significant attention to improve the performance further. However Search-based testing has not been treated with the same focus and hence there is no significant improvement compared to Symbolic and Concolic Testing. Therefore as per the recent studies, Symbolic and Concolic Testing is expected to be outperformed and expected to be improved further [19].

2.4 Tools

2.4.1 KLOVER

KLOVER is considered the first symbolic execution and automatic test generation tool for C++ programs [12]. KLOVER developed in a time where symbolic execution engines are only available for languages like C and JAVA.

Unlike in concrete execution where the program is executed with real values, in symbolic execution program is executed symbolically. KLOVER collects each program path it explores with a path condition including a series of branching decisions. When the test tool has a list of potential execution paths with mapped symbols, during concrete execution these symbols will be replaced by real values to explore collected paths. KLOVER uses a decision maker called SMT (Satisfiability Modulo Theory) solver which is the constraint solver in KLOVER to find concrete values and remove false paths. KLOVER also does a set of sanity checks and KLOVER has the capability to detect memory out-of-bound accesses, seg-faults such as divide-by-zero, and set of user-defined assertions [12].

KLOVER needs user's input about the program to determine variables that needs to be captured as symbolic inputs. A user can select the list of inputs the user wishes to mark as symbols, however, it is the users' responsibility to ensure that the relationship between the inputs is appropriate. When the C++ program is compiled into LLVM byte code, KLOVER then interprets it for symbolic execution. To handle the C++ library constructs KLOVER uses its own C++ library. After the execution, statistical information as well as results from sanity checks are fed, and source program coverage report is produced by gcov. Below diagram shows the high-level architecture of KLOVER [12]:

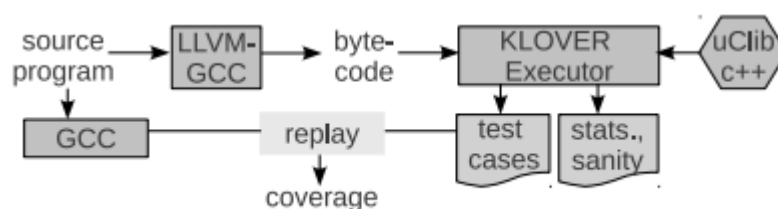


Figure 10 - Overall architecture of KLOVER [12]

2.4.2 KLEE

KLEE is also a symbolic execution tool that is capable of producing high code coverage test cases by mimicking the LLVM bit code in a custom runtime environment. KLEE's main advantage is that it operates on LLVM bit code and due to this KLEE can run on anything that the Clang compiler tool chain can compile: C, C++, Swift, Rust, etc [13] [31].

The major drawback in KLEE is that even when the bit code is available, a user has to manually inject KLEE API calls into the source code, link in the KLEE runtime, and may have to stub out or manually model external library dependencies [13] [17]. When the code is large tasks like this become daunting and complicates the build systems. KLEE also cannot determine all possible routes when the program size is large. It even can fail on a utility function such as sort. The problem becomes a halting problem rather we say an unreachable problem. KLEE is a heuristic, hence it is capable in only reaching some of the code paths in limited time. Also, it can't produce results in quick time, according to the paper, KLEE took roughly 89-hours to generate tests for COREUTILS library which has about 141000 lines of code in it (with libc code used in them) [13].

2.4.3 CREST

CREST is also a test data generation tool specifically targeting the C language. CREST inserts mutations to the code into the target program using CIL and it performs both symbolic executions as well as concrete execution concurrently. Once the program is executed symbolically, constraints that are generated are solved using Yices to generate concrete values which can then be used to create new, unexplored program paths. “Yices is a constraint resolver that decides the satisfiability of formulas containing uninterpreted function symbols with equality, real and integer arithmetic, bit vectors, scalar types, and tuples.”[17] Yices is capable of handling both linear and nonlinear arithmetic.

However the latest version of CREST can only resolve linear and integer arithmetic, symbolically. And for the CREST tool to be able to inject instrumentation code into the program user must use functions such as `CREST_int`, `CREST_char`, etc., declared in "crest.h". CREST will then identify these variables and generate symbolic inputs for the program accordingly [17].

2.4.4 CAUT

CAUT is also a symbolic execution engine based program to generate input data for C programs for unit tests automatically. It basically is a tool that provides coverage driven testing on the branch [20].

Currently, CAUT provides following search strategies:

- CREST cfg-guided search strategy
- KLEE rp-md2u search strategy
- Predictive Path Search strategy based on the Coverage Structure

2.4.5 DART

DART is tool which is capable of generating unit tests for applications that are developed in C language and it has achieved automation of software testing combining three main techniques:

- Automatically extract interfaces details by parsing the source code
- Perform tests with randomly generate test drivers for interfaces extracted
- Analyze results dynamically to see how the code conducts under random testing and a new set of test inputs are derived based on this information to explore alternative program paths.

The main strength of DART is it can perform on any program that can compile. DART has the capability to identify standard errors like segmentation faults (crashes), violations like assertion and non-terminations. DART is capable of dynamically collecting knowledge about the execution of the program is called a directed search. DART starts with a random input, and it calculates and generates an input vector for the next cycle of execution during each cycle. This vector includes values that are the solution of symbolic constraints gathered from predicates in branch statements during the previous execution. And as a result the new input vector will execute the program through a different path. By iterating this process, all related feasible execution paths can be explored [21].

2.4.6 CUTE

CUTE first instruments the code under test to explore execution paths, and then constructs a logical input map for the class under test. This generated map can be regarded as a memory graph in a symbolic way. CUTE then iteratively runs the modified code as below [22]:

- CUTE uses the generated map to constructe a concrete input memory graph for the program and two symbolic states, one for pointer values and another for primitive values.

- CUTE runs the code on the concrete input graph, collecting constraints (as symbolic values in the symbolic state) that characterize the set of inputs that would take the exact path of the execution as the current execution path.
- CUTE reverses one of the collected constraints and solves the resulting restraint system to obtain a new logical map that is similar to previously generated one but with a better chance of execution through a different path. It then assigns the new logical map and repeats the process.

CUTE executes the program in both with actual values and also symbolically. The actual CUTE implementation first modifies the source code under test, by adding functions that perform the symbolic execution. CUTE then repeatedly executes the instrumented code only concretely.

2.4.7 PathCrawler

PathCrawler is test generator which is developed for C applications and it is developed based on code instrumentation and constraint solving. Hence the tool is efficient and also can be extended. The code instrumentation is basically a process where it automatic transform the source code into another source code which helps identify the executed path. The PathCrawler does not have issues from approximations and is not complex compared to static analysis, nor from the number of executions demanded by the use of heuristic algorithms in function minimization and the possibility that they fail to find a solution. PathCrawler includes standard output into the code as code instrumentation to identify the symbolic execution path each time the program is executed. The mutated code is executed using a test scenario which consist of set of inputs from the domain related values. From the execution, symbolic path can be discovered and this path can then be transformed into a path statement which defines the “domain” of the path covered by the test-case, set of input values can cause execution of the same path. The next test scenario is discovered by solving the restraints by defining the input values which are from outside from the path which is already covered. The modified code is then executed on these test cases, until all the feasible paths have been covered [23].

2.4.8 AgitarOne

AgitarOne is a commercial tool that is capable of generating test data for programs. It executes classes with various input data and creates a set of observations that represent the function or the unit behavior, which developers can use to generate assertions [5].

2.4.9 CATG

CATG is an example for concolic unit testing engine based tool and it is developed specifically for Java programs. CATG uses assembly (ASM) for code mutation or for instrumentation. CATG mutates class files during runtime and dumps all executed instructions by the program and all local and global variable values loaded from different memory areas into a log file. Then CATG loads these log files and interprets logged instructions and values both symbolic and concretely.

2.4.10 EvoSuite

EvoSuite is a tool that is developed to generate test cases for programs that are developed in Java. EvoSuite not only provide a higher coverage it also provides assertions as well. To achieve this, EvoSuite introduced a unique approach to generate and to optimize whole test suite generation towards gratifying a coverage criterion [4]. EvoSuite implements hybrid search, dynamic symbolic execution and testability transformation.

User has the option to select a coverage criteria in EvoSuite and based on the selected criteria EvoSuite evolves an entire test suite. The result generated by the EvoSuite is not adversely influenced by the order nor by the difficulty or infeasibility of individual coverage goals.

EvoSuite uses an instrumentation-based testing mechanism to produce a set of assertions for a given class. Asserts can be used to identify the current behavior of the class so that developers can identify defects, and also the assertions protect against regression faults.

To lead the test suites using EvoSuite for a given Java class, the tool only requires the Java byte code of the class and its dependencies [4]. EvoSuite analyze the byte code first and then instrument it to produce assertions, and at the end of the search, EvoSuite generates a test suite

for the given class which complies with JUnit. EvoSuite is completely automatic, capable of generating test suites for the entire packages and no manual intervention required during the test generation process.

When EvoSuite generates tests for a package, it takes one class at a time (not the entire package), and the goal is to produce a test suite to maximize the code coverage rather we say branch coverage for a particular class. EvoSuite is not by any means limited to primitive data types, it can handle arrays and any class object. To instantiate a class EvoSuite considers all possible constructors and produces an instance of it, and creates method calls for each method by passing the required type of variables into it and recursively tries to satisfy all its dependencies.

CHAPTER 3

METHODOLOGY

3 METHODOLOGY

As described in the above chapters generating test cases automatically will save a lot of time for developers. Unit testing is an essential component in delivering a quality product and it is the testing method that gives the highest level of confidence about the program quality. Therefore automating the test creation process is beneficial in many cases.

3.1 Proposed Solution

3.1.1 High-level design

The ultimate objective of this research project is to build a unit test generation tool for programs that are implemented in C++. When a C++ class is given, the tool processes it by analyzing the header file and the implementation file and generates unit tests such a way that the tool satisfies the criteria defined in the coverage goal. In the current phase, the user cannot input the coverage criteria into the demo tool, however, in the next phases, the user should be able to pass the coverage criteria as a parameter to generate results accordingly. The demo tool supports following inbuilt coverage criteria; output coverage, branch coverage, and line coverage. The demo tool only supports the integer data type at the moments and it can be easily extended to support other complex data types such as string, structs, and objects.

Two test generation algorithms have been implemented and the default algorithm set in the tool is random test generation. And the user has the capability to pass an argument to select the searched based test generation algorithm. Please refer the Chapter 04 - System Architecture for more details.

Following are the prerequisites for the tool:

- G++ - Compiler to compile the C++ code
- Lcov/gcov – Generate coverage report
- Google Test framework – Unit test framework
- Java 8 – Tool is developed in java
- JNI – Communicate with C program

- Unix environment – Tool is built for unix environments only

The following diagram shows the high-level design of the system.

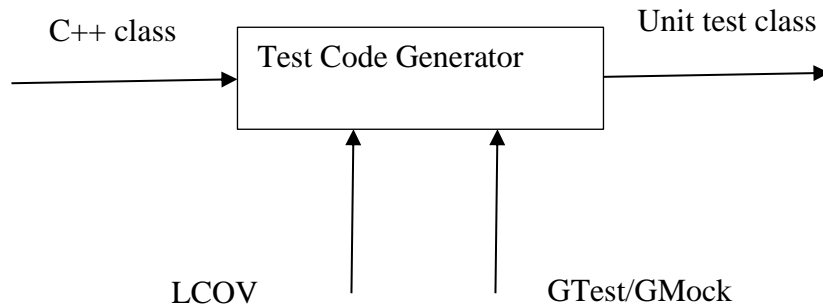


Figure 11 - High level design

As shown in the above diagram, the tool takes the C++ class file as an argument (ex: main.cpp). Gtest/GMock unit test framework has integrated into the tool as the unit test suite is generated using Gtest/GMock unit test framework. GCOV/LCOV is used to generate coverage reports hence GCOV/LCOV frameworks have been integrated into the tool. Based on the input parameters, the tool will generate a class that consists of unit tests (unit test suite) satisfying inbuilt coverage criteria and LCOV report which consist of the code coverage.

The challenging part in generating test cases is the generation of input data set. To achieve data creation, three methodologies have been used; random unit test generation, feedback driven value generation and goal-based test data generation with the assistance of dynamic code execution. In all methods, the class under test is executed and evaluated multiple times to generate the output.

Essentially the tool will cover the following unit testing activities.

- Branch coverage
- Output coverage
- Statement coverage

3.2 Scrutiny of the Solution

When existing researches for unit test generation for C++ programs are considered, almost all require some level of manual intervention to generate results. Meaning user input is required even it states automated unit test generation (for example KLEE, KLOVER). The main reason being these unit test generation tools have been implemented using symbolic execution engines. Meaning that a custom runtime environment is used by the tool to produce test cases for code coverage by emulating LLVM byte code [18]. Basically, tools that are developed using symbolic execution operate on the LLVM byte code.

If the KLEE engine is taken as an example: KLEE can only generate unit tests for programs where source code is available, however, it requires the LLVM bit code of the class as well. However getting the LLVM byte code from the source code can be tricky due to its dependencies such as build systems, configurations [13]. Even when byte code is available, users may have to manually inject KLEE API calls into the source code, link in the KLEE runtime, and possibly stub out or manually model external library dependencies [18]. Generated unit tests are in binary form hence it is not easily readable and if it is required to verify the correctness of a function by looking at the generated unit test function return values, it will be difficult to do. These tasks can become exhaustive when dealing with large codebases and complicated build systems.

Sample function to be tested in KLEE:

```
int get_sign(int x) {
    if (x == 0)
        return 0;

    if (x < 0)
        return -1;
    else
        return 1;
}
```

Figure 12 - Function to be tested in KLEE engine

In order to test the above function in the KLEE engine, to make the KLEE engine aware of symbolic inputs, the user needs to manually mark the parameter as symbols as follows.

```
int main() {  
    int a;  
    klee_make_symbolic(&a, sizeof(a), "a");  
    return get_sign(a);  
}
```

Figure 13 - Marking symbolic variables for KLEE engine

As mentioned above since the KLEE engine works on top of LLVM bit code user has to compile the source code to generate the LLVM bit code. The test cases generated by the KLEE engine are written into a file with .ktest extension and since it is also in binary format, KLEE is given a tool called ktest-tool to view the content of the file [13].

The tool proposed in this research does not need anything other than the library dependencies mentioned in the above section and the source code, the proposed tool focuses on the source code and generates unit tests exploring source code statements and branches. The tool basically contains two test data generation algorithms: feedback-driven random test generation and goal-oriented test data generation and based on the algorithm selects data generation can differ.

In the random test generator, a test input is derived randomly and pass on to the function under test and evaluate the result if the function returns a value and feedback is used determine the path taken by the given input, and a duplicate can be sliced with the feedback obtained. Pre-defined number of iterations takes place to obtain test cases to cover all paths which is the worst case and if all paths are covered test generation stops without reaching maximum iterations defined. In the case of goal-oriented tests, test generation starts from a goal and it starts producing data until the goal is satisfied or until the defined time expires. The goal is to cover all branches in the code by the exploration of path space. Statements in the code are modeled into a graph and by providing inputs traverse the tree to obtain different test functions and paths that are not traversable are notified to the user for further inspection.

3.3 Progress

The detailed literature review is completed and analyzed existing implementations related to the project topic. After the preliminary studies, detailed design has been drafted before implementing the final product, the code generator. The code generator was implemented and the code coverage is measured using lcof reports. The tool is a java based application as mentioned in the pre-requisites. The research has been partially implemented under a selected set of criteria and limitations. The study limitations have been mentioned under chapter 6 study limitations.

3.4 Evaluation Methodology

For the final evaluation of the research, the code generator will be provided however it will be a tool that is developed for the demonstration purpose. The tool should be able to generate unit tests for classes which have setter methods or constructor with primitive data types (only integer is considered in the tool). Also, in the final demonstration how the code generator works will be explained. Evaluation of the code generator will be done by comparing coverage reports. If the model is correct and accurate generated unit test should cover a significant portion of the implementation.

CHAPTER 4

SYSTEM ARCHITECTURE AND THE IMPLEMENTATION

4 SYSTEM ARCHITECTURE AND THE IMPLEMENTATION

4.1 System Overview

The Unit test generator works based on a transactional model and the unit test generator is modeled into 3 main layers.

4.1.1 Infrastructure layer

This layer manages the underlying low-level functions such as thread creation, socket connection, etc. Changes to the underlying implementation should not affect the interface given to the application layer by the infrastructure layer. Therefore the upgrades to the underlying frameworks do not cause code changes in the above layer.

4.1.2 Application layer

This layer facilitates the application with functionality to implement data generation algorithms. This layer doesn't have any impact on underlying infrastructure when defining the application structure. The framework itself creates the required infrastructure for the application to run based on the application configuration. Modules can be implemented on top of this layer.

4.1.3 Functional layer

This layer is responsible for managing modules and modules can be hosted on top of a modular framework. The association of each functional module can be defined using the functionality provided by this layer. This is implemented on top of the Application layer of the framework.

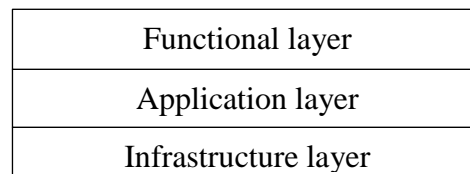


Figure 14 - Unit test generator implementation

The module is a class that has logics to perform a well-defined task. There should be a predefined set of pre-conditions and post-conditions for a Module.

The unit test generator consists of the following modules.

- Mutate Module
- Code parser Module
- Analyzer Module
- Code Generator Module
- Report Generator Module

Three test generators have been integrated into this tool which is a random value generator, feedback-driven test data generator, and goal-oriented test generator. In the following subsections approach for the three generators will be discussed.

4.2 Random Value Generator

As the name implies test data will be generated randomly using random value generators. With this approach, we can exercise the class under test with random values [3]. Input data can be either domain values or values that are invalid for that software. Even if the data provided for the test is not domain-specific still it is a valid test since the application should be able to handle those values and the scenarios. First, the class needs to be executed with the generated test data then we need to observe the outputs and based on the result we need to generate asserts. Due to the simplicity of the random data generation, this approach can apply for any given software. We can explore unexpected security problems, and we can identify potential problems that are not handled in the application with the random value generator approach.

However, there are few drawbacks to this methodology. Since inputs are generated randomly without taking the class under test into consideration, it raises a concern about the validity of the input data used to test the class. And also there is a chance for duplicate test cases as inputs are generated randomly, some execution paths may never get executed, and some functions may never get tested.

Take the following class as an example:

```
std::string intToString(int num)
{
    if (i == 1)
        return "One";
    else if (i == 2)
        return "Two";
    else
        return "Undefined";
}
```

Figure 15 - Sample function II

If the task is given to generate unit tests for the above example class with pure random nature, there is a chance that generated test cases might get executed on the same path. Since random value generators are used unless a branch condition specifies a range the chances that a randomly generated value falls into the branch condition have a very low probability. In the above scenario in most cases randomly generated value could be larger than 3 or can be a negative value and as a result, the function will return Undefined in most cases. And also we don't need multiple test cases to test else path. Hence random value generation may not be very effective if we adapt to pure random test generation.

4.3 Feedback Driven Value Generator

In this mechanism, the suggested proposal is to improve the test data generation by adapting the feedback-driven test case generation. For each path in the code, a fitness function will be defined [4]. Starting with a random value, the value will be fed to each fitness function and the return value is taken into consideration as the feedback which is used to map input data with execution path [14]. This way we can eliminate redundant test cases insist tool generates extra test cases incase tool generates a redundant test case, and this process carries out until all branches are covered or until set timeout expires. The main objective here is to explore paths in the code effectively and efficiently. Test data generator selects method calls of the class under test randomly and then generated random inputs are passed onto the selected method calls, construct a sample test case and execute the sample test program[28]. The output

of the generated test result is evaluated against execution paths (filters) to determine whether the generated results are illegal, redundant or useful data.

4.4 Goal Oriented Value Generator

A goal-oriented value generator starts from a goal and it starts producing data until the goal is satisfied or until the defined time expires. The problem statement here is to the exploration of path space hence the goal is to cover all branches in the code [3]. Goal-oriented test data generation has a number of practical examples. The main advantage of using this generator is that this method is capable of identifying data dependencies and carry these data dependencies up to the goals to influence the data generation [27].

```
(S)void example(int r1, int r2) {  
  (1)  bool invalid = false;  
  (2)  if(r1==10) {  
  (3)      if(r2==20) {  
  (4)          invalid = true;  
          }  
  }  
  (5)  if(invalid) {  
  (6)      THROW_ERROR();  
  }  
}
```

Figure 17 - Sample function III

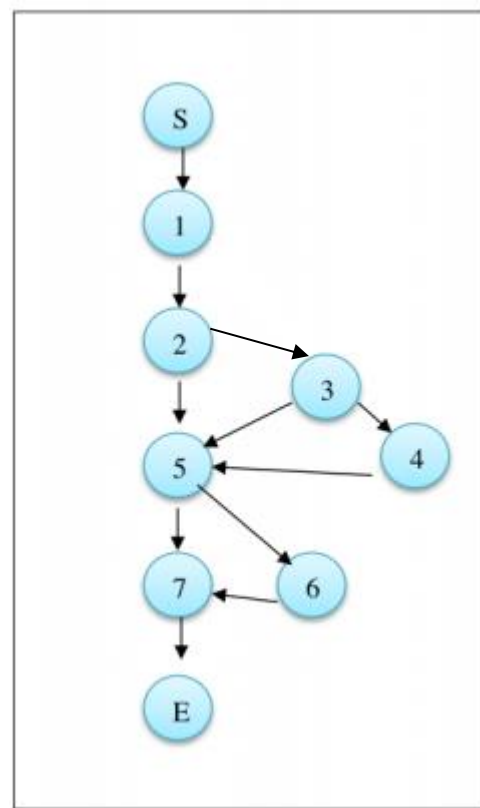


Figure 16 - Execution tree

As denoted in the above diagram the test method or the test function is modeled into a controlled flow graph where each statement in the program is symbolized as a node in the graph [11] [15]. For example assignment statement, branch statement, loop statements such as while/for are all modeled into nodes. Each edge represents the possible execution path. If there is a conditional node there will be two branch edges denoted as a true branch and false branch [11] [15]. S denotes the entry point and the E denotes the endpoint. In the goal-oriented approach, the goal should be to traverse through a set of desired nodes. Basically, traversal should satisfy the exploration of desired nodes. And the ultimate goal should be to generate test cases such a way that it satisfies all possible code execution paths. But in certain cases path exploration can be infeasible. Hence tool may fail to generate a test case for the path which is infeasible to access. These infeasible paths will require manual intervention, hence the tool will report these infeasible paths in the log file as warning and developer should attend and should take corrective actions. In goal-oriented test case generation, the tool should generate input to satisfy a particular goal. As mentioned above the motivation is to find a test (a statement or a branch) in the program, the goal is to find inputs on which the test is executed.

As an example, if the goal is to get `THROW_ERROR` method to get called, the execution path should be nodes <1, 2, 3, 4, 5, 6>. To achieve this particular goal generation tool must generate inputs; integer 10 for r1 variable and integer 20 for r2 variable respectively. The tool should generate these values within the configured time limit and if the tool fails to generate inputs it must be reported in the logs for further investigations. When the tool comes to the point where it needs to generate inputs it does a binary search to detect input values. The drawback of doing a binary search is it has a time complexity of $O(\log n)$ and once the input values are determined based on this information, tool can exercise the true and false branches. Compare to random testing this approach does not generate a whole lot of inputs but a set of well-defined inputs to satisfy all the branches unless the branch is infeasible to access. The random testing helps in certain ways when invalid or unhandled inputs that are generated randomly and pass on to the method. Random testing can be effective in some ways and it can help detect unhandled exceptions whereas in goal-oriented approach variable values are known beforehand by searching for possibilities in the CUT and it will only exercise the class for the known set of variables derived.

4.5 Architecture of Code Generator

This section covers the functionality of each module.

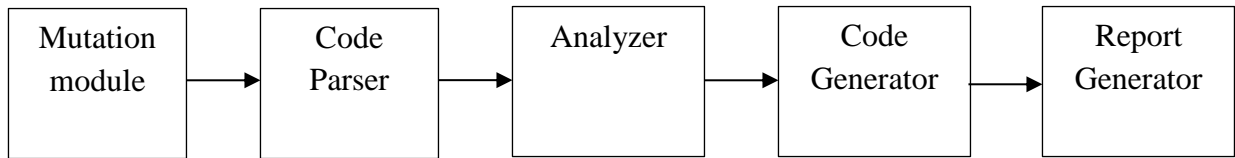


Figure 18 - Unit test Module chain

The class under test passes through the module chain as depicted in the above graph. Each module has a set of pre-conditions and post-conditions and each module operates within a well-defined boundary.

Fitness Function Evaluator is used to calculate the branch distance when the feedback driven test data generator is selected meaning this function will only be evaluated at the analyzer module when feedback driven test data generator is selected as the data generator.

Code parser module checks whether the class is compile able. If this module finds any compile errors/issues then the tool stops generating unit tests and report errors. If there are no errors in the class, the class is parsed and the collected data will be moved forward on to the analyzer module.

Analyzer module is where the test data generation algorithms work. Then the generator module takes this generated input data/selected methods and constructs unit tests according to the gtest framework.

Once the code is successfully generated tool itself compiles the generated unit test suite and runs test code and generates the lcov report. The report generator module is an optional module and it will only be enabled if the parameter required for report generation is passed as an argument during the tool run.

4.5.1 Code parser

Since the tool is developed in JAVA 8, there is a requirement for a C++ parser library written in JAVA. The parser used is “Eclipse CDT Parser” version 5.6.0. First, the Code parser module checks for syntax errors by compiling the source code and unit test generation process stops if compiler gives any error. An assumption is made that g++ is available in the environment to compile the source code. The source code is then parsed and the tool keeps constructors, methods, and fields for that particular class in data structures. Below figures show class structures:

```
public class Field
{
    private String fieldName;

    public Field(String fieldName, String type, ClassReader classReader) {
        this.fieldName = fieldName;
        this.dataType = new DataType(type, classReader);
        this.classReader = classReader;
    }

    public int getModifiers() { return 1; }

    public String getName() { return fieldName; }

    public DataType getType() { return dataType; }

    public DataType getGenericType() { return dataType; }

    public String toGenericString() { return fieldName; }

    public ClassReader getDeclaringClass() { return classReader; }
    private DataType dataType;
    private ClassReader classReader;
}
```

Figure 19 - Field class

```

public class Parameter {
    public DataType type;
    public String variableName;
}

```

Figure 20 - Parameter class

```

public class Method {

    public int lineNo;
    public String methodName;
    public DataType returnType;
    public List<Parameter> parameters = new ArrayList<>();
    public CPPASTBinaryExpression ifCondition;
    public CPPASTSwitchStatement switchCondition;
    public ClassReader classReader;

    public int getModifiers() { return 0; }

    public DataType[] getGenericParameterTypes() {...}

    public String getName() {...}

    public DataType[] getParameterTypes() {
        return getTypeParameters();
    }

    public ClassReader getDeclaringClass() {
        return classReader;
    }

    public DataType[] getTypeParameters() {...}

    public DataType getGenericReturnType() { return returnType; }

    public DataType getReturnType() { return returnType; }

    public String toGenericString() { return getName(); }

    public Parameter[] getParameters() { return parameters.toArray(new Parameter[0]); }
}

```

Figure 21 - Method class

```

public class Constructor {

    private List<ClassReader.Parameter> parameters;
    private ClassReader classReader;
    private String constructorName;

    public Constructor(String constructorName, List<ClassReader.Parameter> parameters, ClassReader classReader) {
        this.constructorName = constructorName;
        this.parameters = parameters;
        this.classReader = classReader;
    }

    public DataType[] getGenericParameterTypes() {
        ArrayList<DataType> dataTypes = new ArrayList<>();
        for (ClassReader.Parameter parameter : parameters) {
            dataTypes.add(parameter.type);
        }
        return dataTypes.toArray(new DataType[0]);
    }

    public DataType[] getParameterTypes() { return getGenericParameterTypes(); }

    public ClassReader getClassReader() { return classReader; }

    public String getName() { return constructorName; }

    public int getModifiers() { return 1; }

    public int getParameterCount() { return parameters.size(); }

    public ClassReader.Parameter[] getParameters() { return parameters.toArray(new ClassReader.Parameter[0]); }
}

```

Figure 22 - Constructor class

4.5.2 Fitness function evaluator module

The main responsibility of the fitness function evaluator module is to calculate the branch distance in the function under test when the function consists of branches. The following table in figure 26 summarizes the computation of branch distances. First branch conditions are summarized into fitness functions and then this module evaluates fitness functions starts with random values [14] [28]. The value generated by the fitness function is taken as the feedback to generate the next set of data. And this process will take place until all branches are covered. To realize the execution code path and to generate asserts accordingly without creating duplicate asserts when a random test generator is selected tool needs to calculate the branch distance for the given input. This module is only needed when the random test generator is

enabled since by default random generator is used in the tool this module will be enabled in the module chain by default.

Below figure explains the functionality of the fitness function evaluator module:

```

Public int getDiscountedPrice(int price)
{
    if (price < 100)
    {
        return price*.95;
    }
    else if (price >= 100 && price < 500)
    {
        return price*.90;
    }
    else if (price >= 500 && price < 1000)
    {
        return price*.80;
    }
    Return price;
}

```

Figure 24–Sample function for fitness calculation

	Decision Type	Branch Distance
1	A < B	A – B
2	A <= B	A – B
3	A > B	B – A
4	A >= B	B – A
5	A == B	Abs (A-B)
6	A != B	Abs (A-B)
7	A && B	Min (A,B)
8	A B	A + B

Figure 23–Branch Distance computation

Fitness Function	Branch Condition
F(n) = Price – 100	Price < 100
F(n) = Min(100 – Price, Price – 500)	Price >=100 && price < 500
F(n) = Min(500 – Price, Price – 1000)	Price >= 500 && price < 1000

Figure 25 - Derived Fitness Function for Figure 27

As shown in figure 28, fitness functions will be evaluated starting with random values. If the fitness function returns a negative value, it means the code will take the true branch. Likewise, if the fitness function value is positive, code will take the false path. For function in figure 27, it is required to create 6 test cases. Hence based on the result return by the fitness function for

the input random value, taking as the feedback, another input value will be generated for the negated path.

After successful iterations, values generated for function methods will be passed on to the next module in the module chain for test case generation.

4.5.3 Analyzer

First of all analyzer module recompiles the class it receives from the fitness function evaluator module because as explained in the above section log trace is needed to infer the execution path. This object is going to be used to derive outputs for the given inputs.

As mentioned in the above chapter there are mainly two algorithms that can be used to generate input data. Based on the algorithm configured analyzer module starts to generate input data.

At the initialization phase of the tool, an object of the respective algorithm will be created based on the configuration and will be assigned to the module. Analyzer module is considered the core of the test generation tool as it does most of the core work. Once the previous module task is completed analyzer module calls the generate function of the algorithm. How each algorithm works is described in the above subsection.

When the algorithm computation is completed, this module output a list of test case blocks that are ready for the developer to execute.

The following diagram shows a sample unit test block.

```
173 TEST(SuiteTest, test17) {  
174  
175     int int0 = (-4164);  
176     int int1 = 769;  
177     MathUtils* class0 = new MathUtils(int0, int1);  
178     int int2 = 0;  
179     int int3 = 0;  
180     int subtraction_Val_ = class0->subtraction(int2, int3);  
181     ASSERT_EQ(0, subtraction_Val_);  
182 }
```

Figure 26 - Generated unit test block

At this stage even though the test body is available still the unit test class is not generated. This will then be passed on to the code generation module.

4.5.4 Code generator

This is the module that creates the actual unit test class. The pre-condition of this class is a list of unit test bodies. This class does not validate the body of the class and this module assumes that unit test blocks that are generated are with correct syntax.

This module gets this list of unit test functions and writes those functions to a file according to the gtest framework, along with the list of test bodies this module inserts the list of headers that are required to execute the unit test class and also the main method which is required to execute the program.

The following diagram shows the snippet of the generated test code.

The screenshot shows a terminal window on the left and a code editor on the right. The terminal window displays the output of the 'll' command, listing files and their sizes. The code editor shows the content of 'ESTest.cpp', which includes headers for 'TestFile.cpp' and 'gtest/gtest.h', and defines three test suites: 'test0', 'test1', and 'test3'. Each test suite contains a set of test cases with various mathematical operations and assertions.

```

shyabith@shyabith-VirtualBox:~/Documents/testGenerator$ ll
total 136
drwxr-xr-x 3 shyabith shyabith 4096 Oct 3 22:46 src/
drwxrwxr-x 4 shyabith shyabith 4096 Oct 9 17:44 target/
-rw-rw-r-- 1 shyabith shyabith 647 Oct 11 14:46 TestFile.cpp
-rw-rw-r-- 1 shyabith shyabith 674 Oct 7 09:41 TestFile.h
-rw-r--r-- 1 shyabith shyabith 423 Oct 3 22:46 testGenerator.tml
shyabith@shyabith-VirtualBox:~/Documents/testGenerator$ clear

shyabith@shyabith-VirtualBox:~/Documents/testGenerator$ ll
total 136
drwxr-xr-x 15 shyabith shyabith 4096 Oct 11 14:49 ./
drwxr-xr-x 5 shyabith shyabith 4096 Oct 3 23:10 ../
drwxr-xr-x 5 shyabith shyabith 4096 Oct 9 17:44 client/
-rw-rw-r-- 1 shyabith shyabith 4276 Oct 11 14:49 ESTest.cpp
drwxr-xr-x 8 shyabith shyabith 4096 Oct 16 12:19 .git/
-rw-r--r-- 1 shyabith shyabith 47 Oct 3 22:46 .gitignore
drwxr-xr-x 2 shyabith shyabith 4096 Oct 11 14:47 .idea/
drwxr-xr-x 2 shyabith shyabith 4096 Oct 3 22:46 javacpp/
drwxrwxr-x 3 shyabith shyabith 4096 Oct 4 10:56 lib/
drwxr-xr-x 2 shyabith shyabith 4096 Oct 16 12:33 libccp/
drwxrwxr-x 2 shyabith shyabith 4096 Oct 16 12:35 linux-x86_64/
drwxr-xr-x 1 shyabith shyabith 290 Oct 3 22:46 Makefile*
drwxr-xr-x 5 shyabith shyabith 4096 Oct 9 17:44 master/
-rw-rw-r-- 1 shyabith shyabith 1090 Oct 16 12:35 MathUtilsClzz.class
-rw-rw-r-- 1 shyabith shyabith 1108 Oct 16 12:35 MathUtilsClzz.java
-rw-rw-r-- 1 shyabith shyabith 776 Oct 16 12:35 'MathUtilsClzz$MathUtils.class'
-rw-rw-r-- 1 shyabith shyabith 31126 Oct 3 22:46 pom.xml
-rw-rw-r-- 1 shyabith shyabith 639 Oct 5 23:59 'RectangleClzz$Rectangle.class'
drwxrwxr-x 4 shyabith shyabith 4096 Oct 16 12:35 Results/
drwxr-xr-x 5 shyabith shyabith 4096 Oct 9 17:44 runtime/
drwxr-xr-x 2 shyabith shyabith 4096 Oct 3 22:46 shaded/
drwxr-xr-x 3 shyabith shyabith 4096 Oct 3 22:46 src/
drwxrwxr-x 4 shyabith shyabith 4096 Oct 9 17:44 target/
-rw-rw-r-- 1 shyabith shyabith 647 Oct 11 14:46 TestFile.cpp
-rw-rw-r-- 1 shyabith shyabith 674 Oct 7 09:41 TestFile.h
-rw-r--r-- 1 shyabith shyabith 423 Oct 3 22:46 testGenerator.tml
shyabith@shyabith-VirtualBox:~/Documents/testGenerator$ gedit TestFile.cpp
shyabith@shyabith-VirtualBox:~/Documents/testGenerator$ gedit ESTest.cpp

```

```

#include "TestFile.cpp"
#include "gtest/gtest.h"

TEST(SuiteTest, test0) {
    int int0 = 0;
    int int1 = 0;
    MathUtils* class0 = new MathUtils(int0, int1);
    int int2 = 0;
    int factorial__Val__ = class0->factorial(int2);
    ASSERT_EQ(1, factorial__Val__);
}

TEST(SuiteTest, test1) {
    int int0 = 0;
    int int1 = 0;
    MathUtils* class0 = new MathUtils(int0, int1);
    int int2 = 1314;
    int int3 = (-1717);
    int multiplication__Val__ = class0->multiplication(int2, int3);
    ASSERT_EQ(-2256138, multiplication__Val__);
}

TEST(SuiteTest, test2) {
    int int0 = (-2976);
    int int1 = (-3303);
    MathUtils* class0 = new MathUtils(int0, int1);
    int int2 = 0;
    int int3 = 0;
    int setWidth__Val__ = class0->setWidth(int2, int3);
    ASSERT_EQ(1, setWidth__Val__);
}

TEST(SuiteTest, test3) {

```

Figure 27 - Generated Test file

4.5.5 Report generator

The responsibility of this module is to generate the execution report which user can use to analyze the code coverage. This is not a mandatory module which means this module gets executed only when the user gives the parameter required to execute the code block in this module.

This module links with Cmake and once the code generator module completes its work, using the CMake report generator module executes the unit test class. The sample CMake file can be found in the appendix section of the report. This module also links with lcov and when the unit test class gets executed it creates the lcov report.

4.6 Parameters

Following is a sample execution with parameters:

```
java -jar {binary_name} -projectCP{absolute_path_to_the_class} -class {class_name} -  
Ddebug=true -Dreport=true
```

TestGenerator.jar is the binary name.

- Binary name – Specifies the binary name with the absolute path of the binary location
- projectCP–This tag specifies the path variable. To locate the class file, the tool needs the absolute path of the class.
- class –This variable specifies the class name.
- debug - Debugging options can be enabled if the developer wishes to debug the program. It helps a developer to have an insight about the test data and test class generation process.
- report - Reporting module in the module chain will only get executed if the report option is set to true. If this variable is not specified or set false, then the tool will bypass this module.
- goal_oriented – Enables the goal-oriented data generation algorithm

CHAPTER 5

EVALUATION

5 EVALUATION

5.1 Overview

In this section, the research carried out will be evaluated and based on the generated results the validity and the accuracy of the research will be justified. The implementation component of this research project is the unit test code generator. Algorithms along with modules discussed in previous chapters have been implemented.

5.2 Evaluation of the Tool

The main objective of this tool is to generate test cases with higher code coverage. And the tool will be evaluated based on the following criteria's [7]:

- Effectiveness
- Efficiency

To evaluate the effectiveness of the tool, the tool generated lcov report will be used and with the lcov report observations can be made and paths that are covered with the tool and paths that are not covered by the tool can be identified and ultimately a conclusion can be made about the effectiveness of the tool. The higher the code coverage the better the effectiveness of the tool.

To evaluate the efficiency of the tool generated log files by the tool will be used. If the time is taken as the parameter to evaluate the efficiency of the tool, the expectation is that the tool to generate results within a justifiable time limit. Timeout reaches within the tool is not expected during the execution of tool and test generation.

For the purpose of evaluation, multiple C++ implementations will use (C++ classes) and the above-mentioned factors will be evaluated against each class with the two algorithms that are discussed in the above chapters.

5.2.1 Evaluation of feedback driven unit test generator

Following class consist of adding elements to an std vector and to a map.

```
Plugins supporting *.h files found.
1  #include <iostream>
2  #include <vector>
3  #include <map>
4  #include "Util.h"
5
6  using namespace std;
7
8  class ClassA {
9
10     private:
11         int x;
12         int y;
13         int z;
14         std::vector<int> m_vec;
15         std::map<int, Student*> m_map;
16
17     public:
18         ClassA();
19         int setX(int x1) { x = x1; return x; }
20         int setY(int y1) { y = y1; return y; }
21         int getZ() { return z; }
22         int generate();
23         int insert(int i);
24         int getValue(int i);
25         int insertToMap(int i);
26         int getValueFromMap(int i);
27     };
```

Figure 28 - Header Class

The below image depicts the implementation of the above header. If the edge cases are not handled properly the program has the potential to crash. Take the getValue function as an

example. If the requirement is to retrieve a value from the vector for a given index and if the method returns the value without checking vector boundaries, the program is susceptible to crash.

```
1 #include "TestFile.h"
2
3 ClassA::ClassA() : x(0), y(0), z(0) {}
4
5 int ClassA::generate() {
6     for (int i =0; i < x;++i) {
7         z += i;
8     }
9     if (z > 10) {
10        return z*2;
11    } else {
12        return z;
13    }
14 }
15 int ClassA::insert(int i)
16 {...}
17 int ClassA::getValue(int i){
18     if (m_vec.size() > i) {
19         return m_vec.at(i);
20     } else {
21         return -1;
22     }
23 }
24
25
26
27
28 int ClassA::insertToMap(int i) {
29
30     auto it = m_map.find(i);
31     if (it != m_map.end()) {
32         return -1;
33     } else {
34         Student* st = new Student();
35         st->id = i;
36         st->name = "Test";
37         m_map.insert(std::pair<int, Student*> (i, st));
38     }
39 }
40
41 int ClassA::getValueFromMap(int i) {
42     auto it = m_map.find(i);
43     if (it != m_map.end()) {
44         return (it->second)->id;
45     } else { return -1; }
46 }
```

Figure 29 - Implementation class with std data structures

Below image shows the code coverage report generated by the tool for the above class and the random value generator used for the experiment.

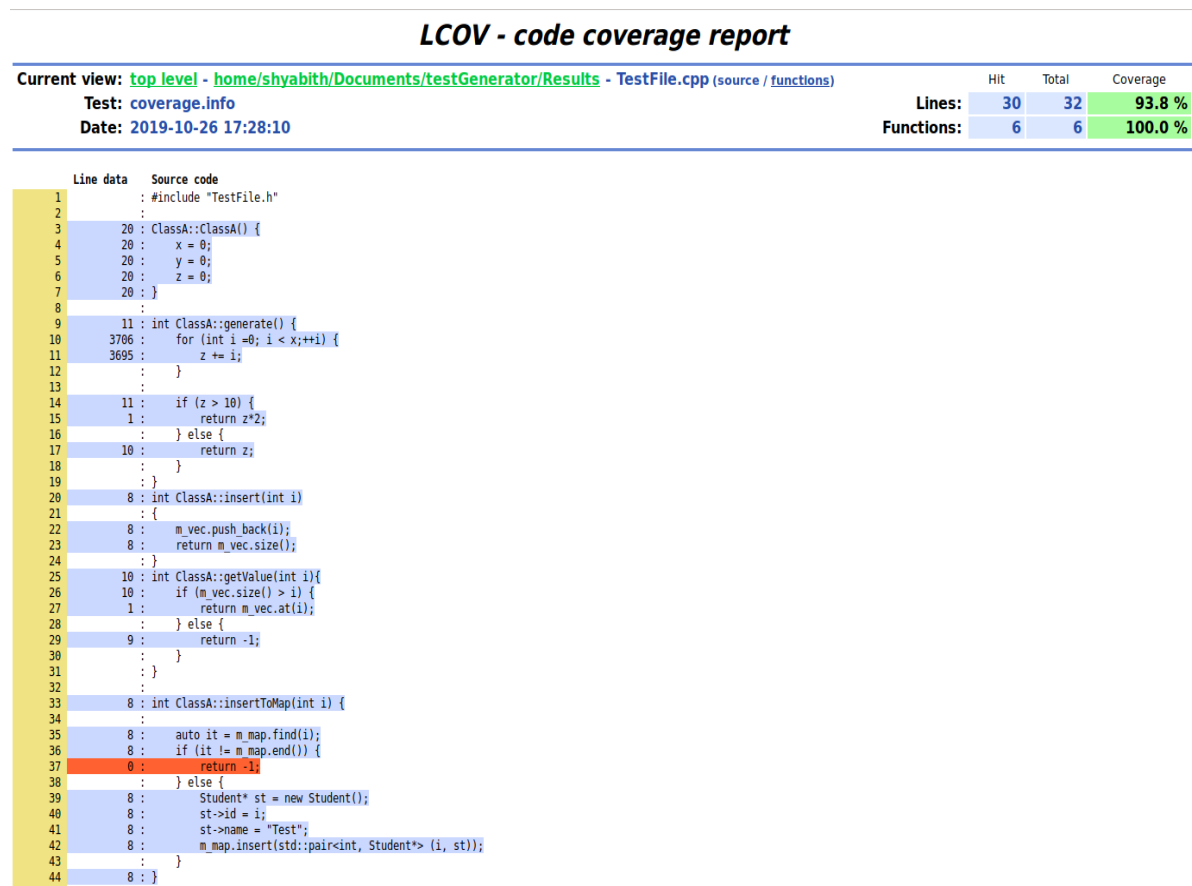


Figure 30 - LCOV report for the data structure class

As you can see in the above graph, the Feedback Driven Test generator was able to produce approximately 94 percent of code coverage for the above class. Since data is being generated randomly there is a chance that the tool misses a branch as evident in the above image. However, with feedback directed mechanism it is possible to minimize this limitation.

```

shyabith@shyabith-VirtualBox:~/Documents/testGenerator$ java -jar master/target/master-1.0-SNAPSHOT.jar -projectCP . -class TestFile.cpp -Dreport=true
* Start Time: 2019/10/27 15:49:02
* Going to generate test cases for class: TestFile.cpp
echo "Creating shared lib"
Creating shared lib
mkdir -p ./libccp
g++ -Wall -fPIC -o ./libccp/Test.o -c TestFile.cpp
g++ -Wall -shared -o ./libccp/libTest.so ./libccp/Test.o

* Starting client
* Connecting to master process on port 14388
* Analyzing classpath:
- .
* Finished analyzing classpath
* Generating tests for class TestFile.cpp
* Setting up search algorithm for whole suite generation
* Using seed 1572171544114
* Starting evolution

* Search finished after 6s and 0 generations, 1347 statements, best individual has fitness: 0.0
* Writing JUnit test case 'cppESTest' to .
* Done!

* Computation finished at: 2019/10/27 15:50:37
shyabith@shyabith-VirtualBox:~/Documents/testGenerator$

```

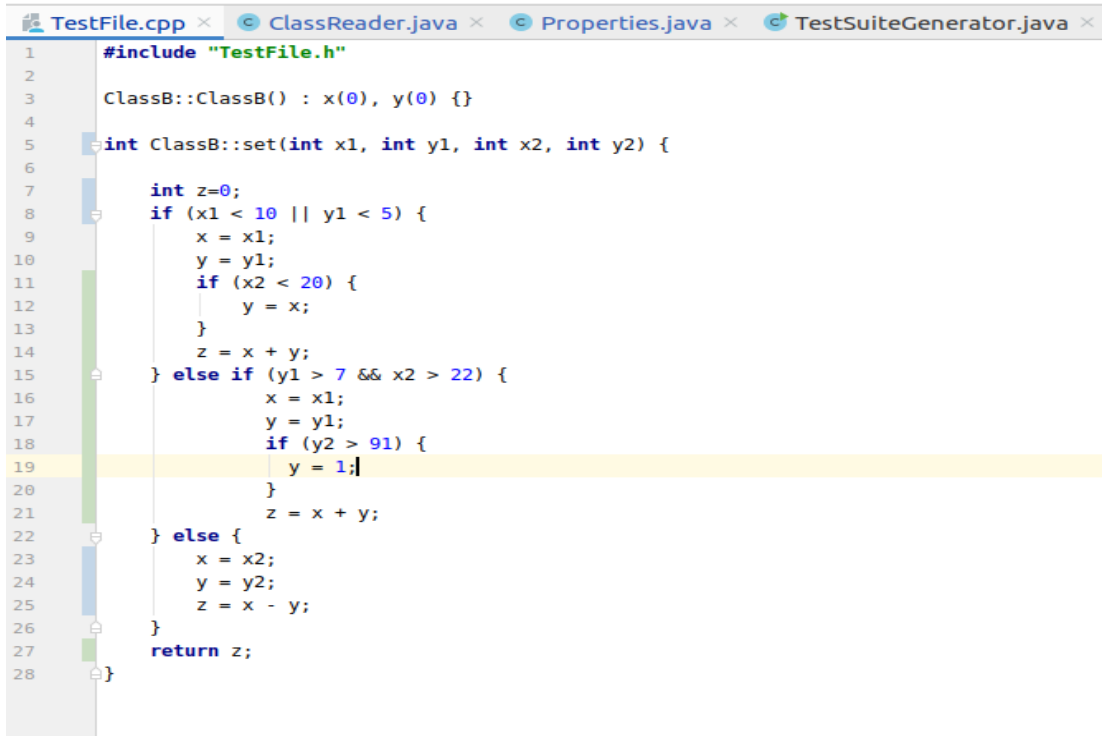
Figure 31 - Time analysis for the data generation

The above diagram shows the time taken to generate test results. As shown in the above graph the tool has received the action to perform the unit test generation at 15.49.02 p.m. and tool has finished the data generation, the test execution and the report generation at 15.50.37 p.m. And according to the image the tool has taken roughly about 1 minute and 30 seconds to generate test data.

When thoroughly analyzed it was identified that the reason for taking 1 minute time to generate test cases, is mainly due to the fact that the tool composes, compiles and runs intermediate classes hence makes the test generation process slower. Usually, the compilation process and the other mentioned tasks consume a lot of time as it involves operations such as file I/O.

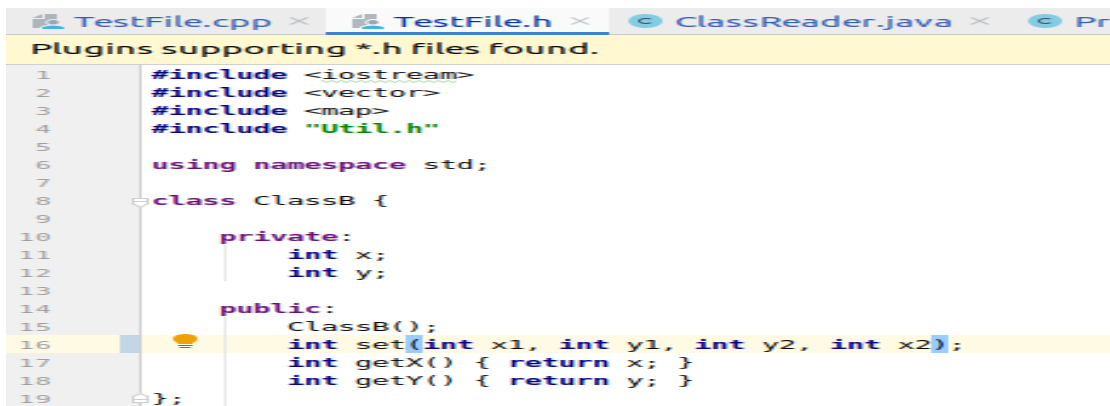
5.2.1 Evaluation of goal oriented test generator

To evaluate the performance and for the comparison purpose, the following class in figure 35 will be used for this section. It is shown in the below section when there are multiple branches in the function even with feedback directed mechanism there is a chance that the tool misses a branch.



```
1 #include "TestFile.h"
2
3 ClassB::ClassB() : x(0), y(0) {}
4
5
6 int ClassB::set(int x1, int y1, int x2, int y2) {
7     int z=0;
8     if (x1 < 10 || y1 < 5) {
9         x = x1;
10        y = y1;
11        if (x2 < 20) {
12            y = x;
13        }
14        z = x + y;
15    } else if (y1 > 7 && x2 > 22) {
16        x = x1;
17        y = y1;
18        if (y2 > 91) {
19            y = 1;
20        }
21        z = x + y;
22    } else {
23        x = x2;
24        y = y2;
25        z = x - y;
26    }
27    return z;
28 }
```

Figure 32–Class I



```
TestFile.cpp x TestFile.h x ClassReader.java x Pr
Plugins supporting *.h files found.
1 #include <iostream>
2 #include <vector>
3 #include <map>
4 #include "Util.h"
5
6 using namespace std;
7
8 class ClassB {
9
10     private:
11         int x;
12         int y;
13
14     public:
15         ClassB();
16         int set(int x1, int y1, int y2, int x2);
17         int getX() { return x; }
18         int getY() { return y; }
19     };
```

Figure 33 - Header file for the above class

LCOV - code coverage report

Current view: top level - home/shyabith/Documents/testGenerator/Results - TestFile.cpp (source / functions)	Hit	Total	Coverage
Test: coverage.info	Lines: 19	19	100.0 %
Date: 2019-11-11 10:43:24	Functions: 2	2	100.0 %

Line data	Source code
1	: #include "TestFile.h"
2	:
3	20 : ClassB::ClassB() : x(0), y(0) {}
4	:
5	53 : int ClassB::set(int x1, int y1, int x2, int y2) {
6	:
7	53 : int z=0;
8	53 : if (x1 < 10 y1 < 5) {
9	22 : x = x1;
10	22 : y = y1;
11	22 : if (x2 < 20) {
12	6 : y = x;
13	: }
14	22 : z = x + y;
15	31 : } else if (y1 > 7 && x2 > 22) {
16	17 : x = x1;
17	17 : y = y1;
18	17 : if (y2 > 91) {
19	17 : y = 1;
20	: }
21	17 : z = x + y;
22	: } else {
23	14 : x = x2;
24	14 : y = y2;
25	14 : z = x - y;
26	: }
27	53 : return z;
28	: }

Generated by: [LCOV version 1.13](#)

Figure 34 - Coverage report for class in figure 30

As shown in the above diagram when the algorithm is switched to goal-oriented test generation we can achieve a greater coverage goal. For the above class, we achieved 100% code coverage and tool generated data within 1 minute and 2 seconds


```
Terminal: Local x Local (2) x Local (3) x Local (4) x +
shyabith@shyabith-VirtualBox:~/Documents/testGenerator$ java -jar master/target/master-1.0-SNAPSHOT.jar -projectCP . -class TestFile.cpp -Dreport=true -Denablegoal=true -Ddebug=true
* Start Time: 2019/11/11 10:41:13
* Going to generate test cases for class: TestFile.cpp
echo "Creating shared lib"
Creating shared lib
mkdir -p ./libccp
g++ -Wall -fPIC -o ./libccp/Test.o -c TestFile.cpp
g++ -Wall -shared -o ./libccp/libTest.so ./libccp/Test.o

* Waiting for remote debugger to connect on port 1044...
* Starting client
* Connecting to master process on port 15186
* Analyzing classpath:
.
.
* Finished analyzing classpath
* Generating tests for class TestFile.cpp
* Setting up search algorithm for whole suite generation
* Using seed 1573449088773
* Starting evolution

* Search finished after 6s and 0 generations, 4030 statements, best individual has fitness: 0.0
* Writing JUnit test case 'cppESTest' to .
* Done!

* Computation finished at: 2019/11/11 10:43:25
shyabith@shyabith-VirtualBox:~/Documents/testGenerator$
```

Figure 35 - Time measures for Class I with Goal Oriented Test data

However, there can be scenarios where the goal which is desired to achieve is not attainable due to logical errors hence in cases like this tool will not be able to cover those branches. Below is an example:

```
Public int getDiscountedPrice(int price, int itemCount)
{
    if (price < 100 || itemCount>5)
    {
        if (itemCount == 5) {
            return price*itemCount*0.95;
        }
    }
    Return price*itemCount*0.9;
}
```

Figure 36 - Unachievable branch statements

In the above case inner if block which resides inside the outer block will never achieve as two blocks consist of contradicting conditions.

5.3 Algorithm Comparison

```

1  #include "TestFile.h"
2
3  ClassB::ClassB() : x(0), y(0) {}
4
5  int ClassB::set(int x1, int y1, int x2, int y2) {
6
7      int z=0;
8      if (x1 < 10 || y1 < 5) {
9          x = x1;
10         y = y1;
11         if (x2 < 20) {
12             y = x;
13         }
14         z = x + y;
15     } else if (y1 > 7 && x2 > 22) {
16         x = x1;
17         y = y1;
18         if (y2 > 91) {
19             y = 1;
20         }
21         z = x + y;
22     } else {
23         x = x2;
24         y = y2;
25         z = x - y;
26     }
27     return z;
28 }

```

Figure 37 - Class A

LCOV - code coverage report

Current view: top level - home/shyahith/Documents/testGenerator/Results - TestFile.cpp (source / functions)			Hit	Total	Coverage
Test: coverage.info			Lines: 19	19	100.0%
Date: 2019-11-11 10:43:24			Functions: 2	2	100.0%

Line data	Source code
1	: #include "TestFile.h"
2	:
3	28 : ClassB::ClassB() : x(0), y(0) {}
4	:
5	53 : int ClassB::set(int x1, int y1, int x2, int y2) {
6	:
7	53 : int z=0;
8	53 : if (x1 < 10 y1 < 5) {
9	22 : x = x1;
10	22 : y = y1;
11	22 : if (x2 < 20) {
12	6 : y = x;
13	:
14	22 : z = x + y;
15	31 : } else if (y1 > 7 && x2 > 22) {
16	17 : x = x1;
17	17 : y = y1;
18	17 : if (y2 > 91) {
19	17 : y = 1;
20	:
21	17 : z = x + y;
22	:
23	23 : } else {
24	14 : x = x2;
25	14 : y = y2;
26	14 : z = x - y;
27	53 : return z;
28	:

Generated by LCOV version 1.13

Figure 38 - Random Generator Result for Class A

LCOV - code coverage report

Current view: top level - home/shyahith/Documents/testGenerator/Results - TestFile.cpp (source / functions)			Hit	Total	Coverage
Test: coverage.info			Lines: 19	19	100.0%
Date: 2019-11-11 10:43:24			Functions: 2	2	100.0%

Line data	Source code
1	: #include "TestFile.h"
2	:
3	28 : ClassB::ClassB() : x(0), y(0) {}
4	:
5	53 : int ClassB::set(int x1, int y1, int x2, int y2) {
6	:
7	53 : int z=0;
8	53 : if (x1 < 10 y1 < 5) {
9	22 : x = x1;
10	22 : y = y1;
11	22 : if (x2 < 20) {
12	6 : y = x;
13	:
14	22 : z = x + y;
15	31 : } else if (y1 > 7 && x2 > 22) {
16	17 : x = x1;
17	17 : y = y1;
18	17 : if (y2 > 91) {
19	17 : y = 1;
20	:
21	17 : z = x + y;
22	:
23	23 : } else {
24	14 : x = x2;
25	14 : y = y2;
26	14 : z = x - y;
27	53 : return z;
28	:

Generated by LCOV version 1.13

Figure 39 - Goal Oriented Result for Class A

```

TestFile.cpp x TestFile.h x ClassReader.java x Properties.java x
5  int ClassB::set(int x1, int y1, int x2, int y2) {
6
7      int z=0;
8
9      if (x1 < y2) {
10         for (int i = x1; i < y2; ++i) {
11             z = z + i;
12         }
13     }
14     if (y1 < x2) {
15         for (int j = y1; j < x2; j++) {
16             z = z - j;
17         }
18     }
19
20     if (y1 + y2 < x1) {
21         z = 0;
22     } else if (x1 > y2 - y1) {
23         z = z - 10;
24     } else if (x1 + x2 > y1 + y2) {
25         z = z + 10;
26     } else if (x2 < y2) {
27         z = z + 1;
28     }
29     if (x2 > y2) {
30         z = 2;
31     }
32     return z;
33 }

```

Figure 40 - Class B

LCOV - code coverage report

Current view: top level - home/shyabith/Documents/testGenerator/Results - TestFile.cpp (source / functions)		Hit	Total	Coverage
Test: coverage.info		Lines: 20	20	100.0 %
Date: 2020-01-10 19:39:36		Functions: 2	2	100.0 %

Line data	Source code
1	1 : #include "TestFile.h"
2	2 :
3	3 : ClassB(ClassB() : x(0), y(0))
4	4 :
5	5 : int ClassB::set(int x1, int y1, int x2, int y2) {
6	6 :
7	7 : int z=0;
8	8 :
9	9 : if (x1 < y2) {
10	10 : for (int i = x1; i < y2; ++i) {
11	11 : z = z + i;
12	12 : }
13	13 : }
14	14 : if (y1 < x2) {
15	15 : for (int j = y1; j < x2; j++) {
16	16 : z = z - j;
17	17 : }
18	18 : }
19	19 :
20	20 : if (y1 + y2 < x1) {
21	21 : z = 0;
22	22 : } else if (x1 > y2 - y1) {
23	23 : z = z - 10;
24	24 : } else if (x1 + x2 > y1 + y2) {
25	25 : z = z + 10;
26	26 : } else if (x2 < y2) {
27	27 : z = z + 1;
28	28 : }
29	29 : if (x2 > y2) {
30	30 : z = 2;
31	31 : }
32	32 : return z;
33	33 : }

Figure 42 - Random Generator Result for Class B

LCOV - code coverage report

Current view: top level - home/shyabith/Documents/testGenerator/Results - TestFile.cpp (source / functions)		Hit	Total	Coverage
Test: coverage.info		Lines: 20	20	100.0 %
Date: 2020-01-10 19:39:36		Functions: 2	2	100.0 %

Line data	Source code
1	1 : #include "TestFile.h"
2	2 :
3	3 : ClassB(ClassB() : x(0), y(0))
4	4 :
5	5 : int ClassB::set(int x1, int y1, int x2, int y2) {
6	6 :
7	7 : int z=0;
8	8 :
9	9 : if (x1 < y2) {
10	10 : for (int i = x1; i < y2; ++i) {
11	11 : z = z + i;
12	12 : }
13	13 : }
14	14 : if (y1 < x2) {
15	15 : for (int j = y1; j < x2; j++) {
16	16 : z = z - j;
17	17 : }
18	18 : }
19	19 :
20	20 : if (y1 + y2 < x1) {
21	21 : z = 0;
22	22 : } else if (x1 > y2 - y1) {
23	23 : z = z - 10;
24	24 : } else if (x1 + x2 > y1 + y2) {
25	25 : z = z + 10;
26	26 : } else if (x2 < y2) {
27	27 : z = z + 1;
28	28 : }
29	29 : if (x2 > y2) {
30	30 : z = 2;
31	31 : }
32	32 : return z;
33	33 : }

Figure 41 - Goal Oriented Result for Class B

```

TestFile.cpp x
1 #include "TestFile.h"
2
3 MathUtils::MathUtils(int a, int b) {
4     width = a;
5     height = b;
6 }
7
8 int MathUtils::sum(int x, int y) {
9     return x + y;
10 }
11
12 int MathUtils::factorial(int x) {
13     if(x > 1) {
14         return x * this->factorial(x-1);
15     }
16     else {
17         return 1;
18     }
19 }
20
21 int MathUtils::substraction(int x, int y) {
22     return x - y;
23 }
24
25 int MathUtils::multiplication(int x, int y) {
26     return x * y;
27 }
28
29 int MathUtils::setWidth(int a, int b)
30 {
31     if (a < b)
32     {
33         width = a;
34         height = b;
35     }
36     else
37     {
38         height = a;
39         width = b;
40     }
41     return 1;
42 }
43
44 int MathUtils::getWidth() { return width; }

```

Figure 43 - Class C

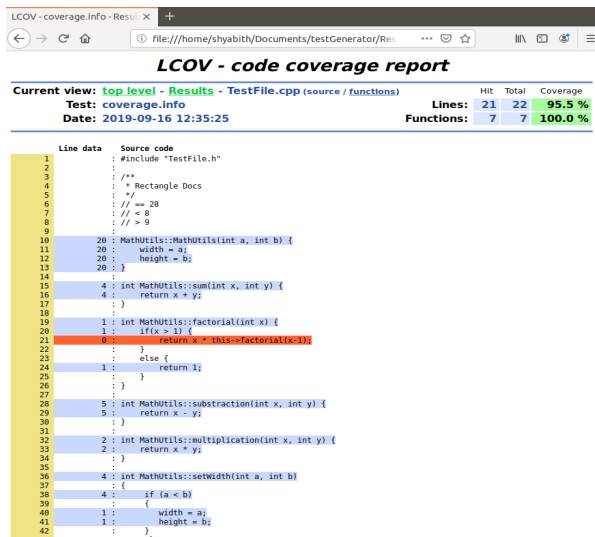


Figure 45 - Random Generator Result for Class C



Figure 44 - Goal Oriented Result for Class C

```

DateTime::DateTime() {
}

int DateTime::isValidDate(int year, int month, int date) {
    if(year >= 1 && month <= 12 && month >= 1 && date <= 31 && date >= 1) {
        return true;
    }
    return false;
}

int DateTime::isValidateDateTime(int year, int month, int date, int hour, int min, int seconds) {
    if(year >= 1 && month <= 12 && month >= 1 && date <= 31 && date >= 1) {
        if (hour > 0 && hour <=23 && min >= 0 && min <= 59 && seconds >= 0 && seconds <= 59) {
            return true;
        }
    }
    return false;
}

```

Figure 48 - Class D



Figure 47 - Random Generator Result for Class D

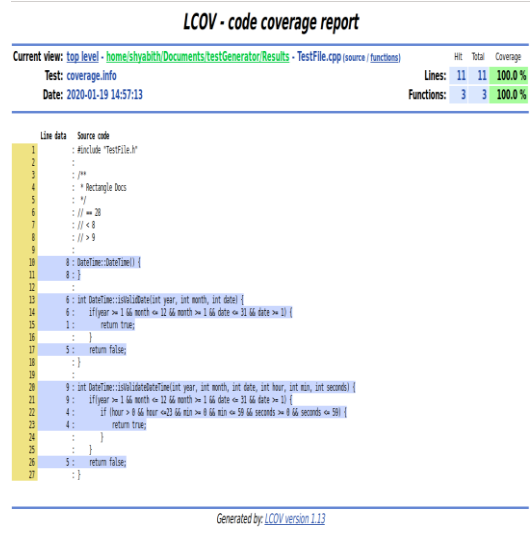


Figure 46 - Goal Oriented Result for Class D

```

Bill::Bill() {
}

int Bill::purchases(int total, int price, int quantity) {
    if (total < 0) {
        total = 0;
    }
    if (price <= 0 && quantity <= 0) {
        total += price*quantity;
    }
    return total;
}

int Bill::calculateTax(int total) {
    return total*0.15;
}

int Bill::getTotalBill(int total) {
    if (total < 100) {
        return total+calculateTax(total);
    } else if (total >= 100 && total < 1000) {
        return total+calculateTax(total)/2;
    } else {
        return total+calculateTax(total)/4;
    }
}

```

Figure 49 - Class E

```

9 :
10 : 9 : Bill::Bill() {
11 :
12 : 9 : }
13 :
14 : 13 : int Bill::purchases(int total, int price, int quantity) {
15 : 13 :     if (total < 0) {
16 : 6 :         total = 0;
17 :
18 : 13 :     }
19 : 7 :     if (price <= 0 && quantity <= 0) {
20 :         total += price*quantity;
21 :
22 : 13 :     }
23 :     return total;
24 : }
25 :
26 : 14 : int Bill::calculateTax(int total) {
27 : 14 :     return total*0.15;
28 : }
29 :
30 : 7 : int Bill::getTotalBill(int total) {
31 :     if (total < 100) {
32 :         return total+calculateTax(total);
33 :     } else if (total >= 100 && total < 1000) {
34 :         return total+calculateTax(total)/2;
35 :     } else {
36 :         return total+calculateTax(total)/4;
37 :     }
38 : }

```

Figure 50 - Random Generator Result for Class E

```

10 : 25 : Bill::Bill() {
11 :
12 : 25 : }
13 :
14 : 69 : int Bill::purchases(int total, int price, int quantity) {
15 : 69 :     if (total < 0) {
16 : 29 :         total = 0;
17 :
18 : 69 :     }
19 : 37 :     if (price <= 0 && quantity <= 0) {
20 :         total += price*quantity;
21 :
22 : 69 :     }
23 :     return total;
24 : }
25 :
26 : 8 : int Bill::calculateTax(int total) {
27 : 8 :     return total*0.15;
28 : }
29 :
30 : 6 : int Bill::getTotalBill(int total) {
31 :     if (total < 100) {
32 :         return total+calculateTax(total);
33 :     } else if (total >= 100 && total < 1000) {
34 :         return total+calculateTax(total)/2;
35 :     } else {
36 :         return total+calculateTax(total)/4;
37 :     }
38 : }

```

Figure 51 - Goal Oriented Result for Class E

```

void DataPublisher::purchases(int a, int b) {
    if (a <= 0 && b <= 0) {
        return;
    } else if (a < 100 && b < 100) {
        l.push_back(PTI(a,b));
    } else if (a > 100 && b > 100 && a < 500 && b < 200) {
        l.push_back(PTI(a*2,b));
    } else if (b > 300 && b < 1000) {
        l.push_back(PTI(a*3,b));
    } else {
        l.push_back(PTI(a*4,b));
    }
}

int DataPublisher::deduction() {
    int acc = accumulate(l.begin(), l.end(), 0, [](auto &a, auto &b) { return a;});
    if (acc > 100) {
        return acc*0.15;
    } else {
        return acc;
    }
}

int DataPublisher::finalValue() {
    return accumulate(l.begin(), l.end(), 0, [](auto &a, auto &b) { return a;}) - deduction();
}

```

Figure 53 - Class F

Line data	Source code
1	: #include "TestFile.h"
2	: #include <numeric>
3	: #include <utility>
4	:
5	13: DataPublisher::DataPublisher() {
6	:
7	13: }
8	:
9	39: void DataPublisher::purchases(int a, int b) {
10	39: if (a <= 0 && b <= 0) {
11	3: return;
12	36: } else if (a < 100 && b < 100) {
13	11: l.push_back(PTI(a,b));
14	25: } else if (a > 100 && b > 100 && a < 500 && b < 200) {
15	12: l.push_back(PTI(a*2,b));
16	13: } else if (b > 300 && b < 1000) {
17	6: l.push_back(PTI(a*3,b));
18	7: } else {
19	7: l.push_back(PTI(a*4,b));
20	:
21	:
22	:
23	24: int DataPublisher::deduction() {
24	53: int acc = accumulate(l.begin(), l.end(), 0, [](auto &a, auto &b) { return a;});
25	24: if (acc > 100) {
26	6: return acc*0.15;
27	:
28	24: } else {
29	24: return acc;
30	:
31	:
32	10: int DataPublisher::finalValue() {
33	:
34	24: return accumulate(l.begin(), l.end(), 0, [](auto &a, auto &b) { return a;}) - deduction();
35	:

Figure 52 - Random Generator Result for Class F

Line data	Source code
1	: #include "TestFile.h"
2	: #include <numeric>
3	: #include <utility>
4	:
5	13: DataPublisher::DataPublisher() {
6	:
7	13: }
8	:
9	39: void DataPublisher::purchases(int a, int b) {
10	39: if (a <= 0 && b <= 0) {
11	3: return;
12	36: } else if (a < 100 && b < 100) {
13	11: l.push_back(PTI(a,b));
14	25: } else if (a > 100 && b > 100 && a < 500 && b < 200) {
15	12: l.push_back(PTI(a*2,b));
16	13: } else if (b > 300 && b < 1000) {
17	6: l.push_back(PTI(a*3,b));
18	7: } else {
19	7: l.push_back(PTI(a*4,b));
20	:
21	:
22	:
23	24: int DataPublisher::deduction() {
24	53: int acc = accumulate(l.begin(), l.end(), 0, [](auto &a, auto &b) { return a;});
25	24: if (acc > 100) {
26	6: return acc*0.15;
27	:
28	24: } else {
29	24: return acc;
30	:
31	:
32	10: int DataPublisher::finalValue() {
33	:
34	24: return accumulate(l.begin(), l.end(), 0, [](auto &a, auto &b) { return a;}) - deduction();
35	:

Figure 54 - Goal Oriented Result for Class F

5.3.1 Comparison of code coverage

Test Function	Random Test Generation based on feedback	Goal-oriented test generation
Class A	100%	100%
Class B	100%	100%
Class C	95.5%	100%
Class D	100%	100%
Class E	93.8%	93.8%
Class F	94.7%	94.7%

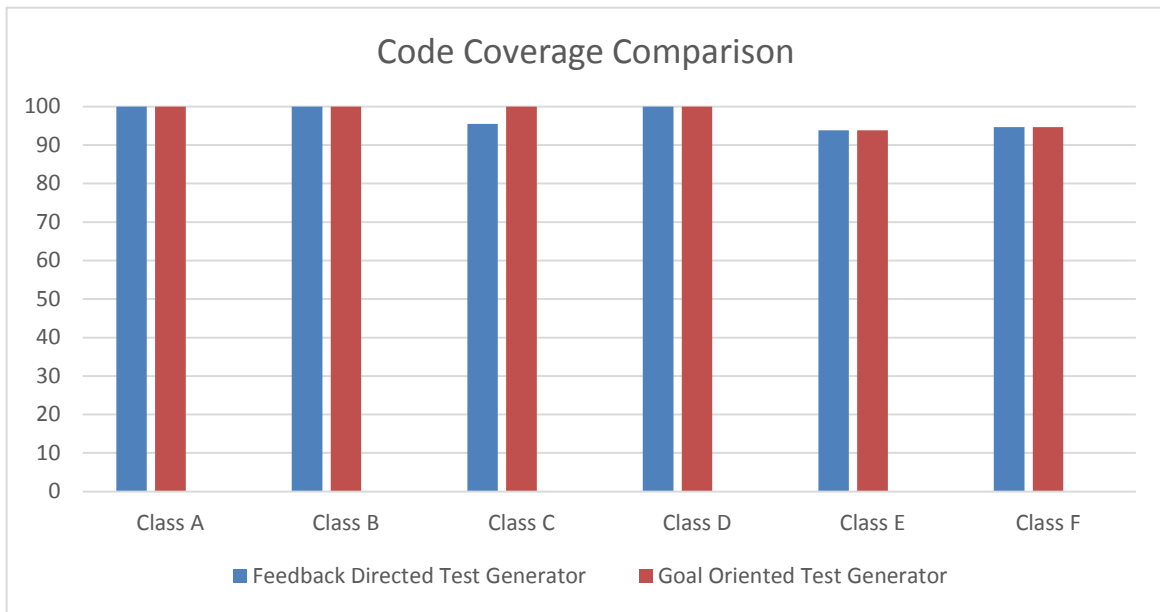


Figure 55 - Code Coverage Comparison Table

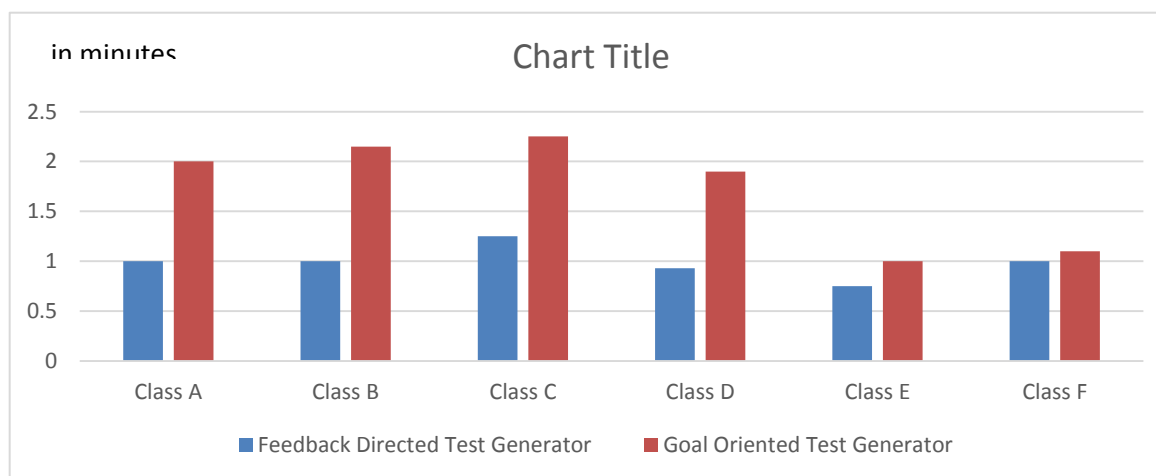
As shown in the above diagrams, both test data generators show promising results as both approaches successfully achieved code coverage of above 90% in all cases. However theoretically in goal oriented value generator, 100% code coverage is attainable with parameter fine-tuning and improving tool implementation but with feedback driven value generator we won't be able to achieve 100% code coverage for certain conditions. In both approaches, a predefined timeout value is set to exit the test data generation process. Also,

method calls are taken randomly by the tool to test hence the tool has no control over the meaning of method calls. Eg: Manipulation of member variables are not taken into consideration when generating test results.

5.3.2 Comparison of time

Test Function	Random Test Generation based on feedback	Goal oriented test generation
Class A	1 min	2 min 02 seconds
Class B	1 min	2 min 12 seconds
Class C	1 min 25 seconds	2 min 24 seconds
Class D	57 seconds	1 min 49 seconds
Class E	46 seconds	1 min
Class F	1 min	1 min 7 seconds

Figure 56 - Time Analysis Table



Feedback directed test generation method specifically targets the branches that needs to be tested, and generates value in constant time in most cases as a reason compared to goal-oriented test generation method it has taken lesser time to produce results. The main reason for taking more time to generate results in goal oriented test generator is goal oriented value generator uses binary search algorithm to generate input values and it can be costly when it comes to finding the value required, hence this approach has taken more time to produce final

coverage reports. However, goal-oriented test generation always has a better chance of finding the solution in most cases compared to feedback driven value generation as feedback driven value generator use random values to obtain actual input values.

In instances where the branch checks for an equivalent value; by using feedback directed value generator it is highly unlikely that we find the solution that satisfy the condition, however with binary search it is guaranteed that tool will always converge into the final solution after some iterations in such cases.

5.3.3 Proof of concept

Following is the pseudo code of the binary search:

```
Procedure binary_search (value, x, condition, l, u)

    if value condition x
        EXIT: x found

    set m = 1 + (u - 1) / 2

    if m < x
        set l = m + 1

    if m > x
        set u = m - 1

end procedure
```

Figure 57 - Pseudo code for search logic

Goal Oriented Data Generator is based on divide and conquer algorithm, and it initializes the search logic with a negative value, and proceed with binary search logic. As depicted in the above image it disregard a section of values if the chosen value does not satisfy the specified condition and recursively search algorithm tries to locate the value disregarding a value range. And as a result the logic will reduce the search space by half in each recursion.

Value range we select is (-int_max = lower, int_max = upper). If value selected satisfy the condition, logic will return the selected value, if the value < x then we change the range to

lower+upper/2 make assign it as new lower. Else if value > x then we change the range and assign the new upper as lower+upper/2 and this logic will be executed until the condition specified is satisfied.

Case I:

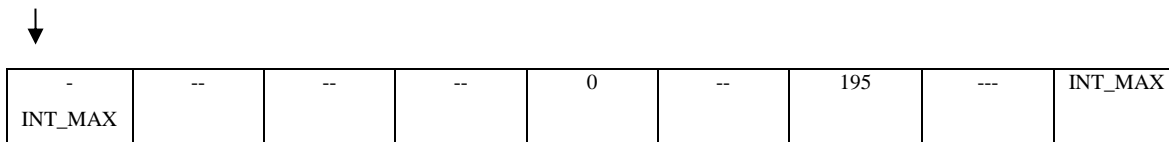
```

If(x == 195) {
    // do something
}

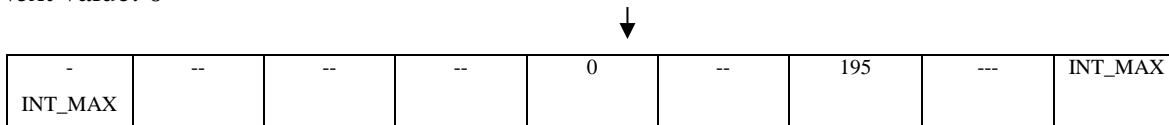
```

Figure 58 - Example for proof of concept case 1

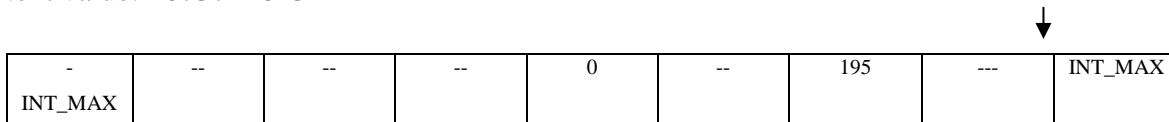
Start value: -INT_MAX



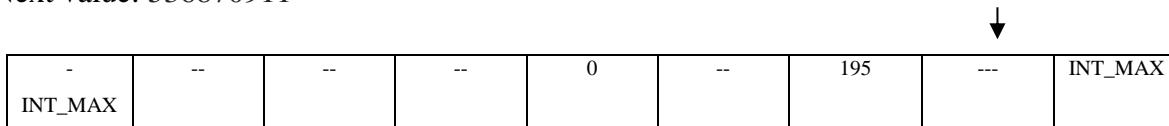
Next value: 0



Next value: 1073741823



Next value: 536870911



The above mentioned search logic in figure-60, will be executed until the final result which is 195 is found. However the limitation as described in above section is the time. Given the fact that time is unlimited in cases like this relevant input value can always be found with this approach. However with feedback driven value generator it is not feasible as values are generated randomly.

Case II:

```
If (x < 200) {  
    //do something  
}
```

Figure 59 - Example for proof of concept case II

Goal Oriented Data Generator, starts from the negative range, do the binary search.

Start value: -INT_MAX



-	--	--	--	0	--	195	---	INT_MAX
INT_MAX								

In the first iteration itself the condition satisfies therefore exist the searching algorithm.

Case III:

```
If (x > 200) {  
    //do something  
}
```

Figure 60 - Proof of concept case III

Goal Oriented Data Generator, starts from the negative range, do the binary search.

Start value: -INT_MAX



-	--	--	--	0	--	195	---	INT_MAX
INT_MAX								

Next value: 0



-	--	--	--	0	--	195	---	INT_MAX
INT_MAX								

Next value: 1073741823

By the second iteration the condition satisfies therefore exist the searching algorithm with value which is denoted last.

5.3.4 Line coverage

Both test data generators show equal performance in terms of code coverage and time for data generation completion, Hence we can conclude that line coverage does not have any impact on the code generator we choose.

5.3.5 Branch coverage

Number of Branches	Feedback Directed Test Generator		Goal-Oriented Test Generator	
	Coverage	Time	Coverage	Time
1	100%	8 seconds	100%	14 seconds
2	100%	8 seconds	100%	14 seconds
3	100%	10 seconds	100%	22 seconds
4	100%	10 seconds	100%	21 seconds
5	100%	11 seconds	100%	24 seconds
6	100%	11 seconds	100%	35 seconds
7	100%	11 seconds	100%	56 seconds
8	100%	13 seconds	100%	1 min 20 seconds

As per the above table, both test data generators show 100% code coverage for the selected data set, however, it is clear when the number of branches increases Goal Oriented Test Generator degrades its performance in terms of time taken for completion.

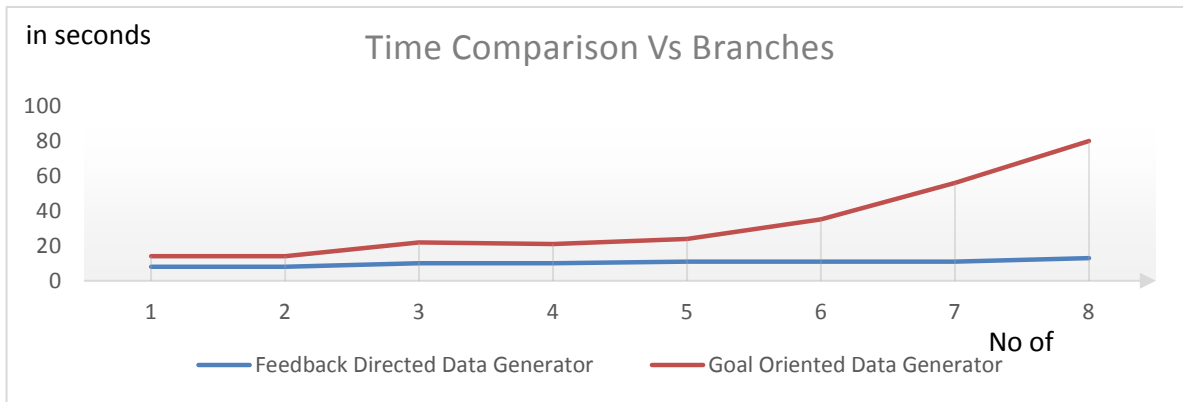


Figure 61 - Time Comparison Vs Branches

However, we cannot come to a conclusion to select one approach over the other. If the user prefers to see results in a quick time Feedback Driven Test Generation should be the choice, and if the user needs more accurate test cases Goal Oriented Test Generation should be the choice.

CHAPTER 6

CONCLUSION

6 CONCLUSION

6.1 Summary

Software is needed in almost all industries, in every business and the dependency with software grows every day. With the growth of software dependency, it is required that the software be secure and reliable. But there are many cases where businesses/industries fail due to software failures. Software testing is widely considered as the best way to ensure and improve the reliability of software [8].

The C++ language is a low-level programming language and applications that have low latency, high-performance requirements, and applications such as mission-critical solutions usually implement using low-level languages such as C/C++. Due to the data encapsulation mechanisms, convenient abstraction, software developers prefer C++ over C language. C++ has the following language features:

- Function and operator overloading
- Constructors and destructors
- Classes with multiple levels of inheritance, virtual functions
- Templates, exceptions
- Functors
- Standard libraries (STL and BOOST).

However, these features also complicate the process of implementing C++ testing programs. Furthermore, due to the mixing of object-oriented programming concepts on top of C code, it makes test implementation even harder. Manual unit test implementation is a time-consuming process hence there is an industrial need to automate test generation for C++ application.

Code structural coverage is an important metric in unit testing. Hence it is expected that every single line of code is executed at least once during the testing. The unit test generation techniques discussed in the above section ultimately narrowed down to finding test inputs to execute a specific statement in the code. And the final goal is to generate a test suite that gives a higher code coverage.

There are many unit test development frameworks for the test-driven development approach, however as described in the above chapters in cases where TDD approach is not practical due to time and budget constraints, however still there should be a way to measure the code quality when TDD approach is not taken. Hence the sole purpose of this research is to cater to the above requirement and propose the implemented solution as a replacement of test-driven development based on the result.

There have been many attempts to develop a practical solution to generate unit tests to achieve higher software code coverage and find potential code errors that were previously hidden. However often in all these approaches, manual intervention is required where the user needs to mark certain inputs as test inputs. Therefore still there is a growing need for a practical solution for a fully automated unit test generation tool.

Test generation can be viewed as a search problem and for a given program set of well-defined tests, goals can be defined. And for a program an unlimited number of test cases can be generated but when a generalized only finite number of test cases may be valid as coverage goals may be duplicated. Test goals can be generalized into the following [9]:

- Branch coverage – Derive at least one test case where all branches in a given class are covered. In branch coverage, there should be one test case to cover branch TRUE condition and another for branch FALSE condition.
- Line coverage – If the test suite covers the non-commented line of the code at least one time. To ensure this each basic block of code must be reached by the test suite.
- Exception coverage – The goal here is to build a test suite that forces the class under test to throw exceptions. Basically, this coverage criteria forces the program to take negative paths of the program.
- Method coverage – The goal of this criteria is to call every method in the class under test to be executed by the test suite at least one time.
- Output coverage - Test suite satisfies output coverage when each method that is publicly accessible in the class under test, at least one test yields a concrete return value characterized by each abstract value.

There are tools to generate unit tests for popular, high-level languages such as Java, however, for low-level languages such as C++, there are only a few attempts. In this thesis, I explored and developed a demo tool to generate unit tests for applications that are developed in the C++ language.

In this thesis, I researched and developed an effective tool to improve the degree of attainable software automation. This thesis explored methods of effectively derive test cases, the basic method of testing software. I explored methods such as symbolic analysis, constraint solving, dynamic program analysis, etc and applied the ideas derived from these methods to build an effective tool to achieve a higher structural coverage of the code.

6.2 Contribution

The testing framework I propose three data generators; random unit test generation, feedback-driven random unit test generation and goal-oriented test data generation to improved code coverage. The main objective of this research is to implement an effective and efficient test data generation/ unit test generation tool for automated unit testing. Unlike a symbolic driven unit test generation where generated test cases are in bit code form, the proposed tool will generate test cases in C++ language and hence the test file is easily readable. When a C++ class is given as the input to the tool, the tool processes it and provides a unit test class. Developers can use this class to improve the code coverage and improve the meaningfulness in the unit test. The unit test generation process is completely automated and the developer has the option to run the unit test code and generate coverage report. During the code generation, the process code tool executes the class under test multiple times if the function which is testing returns a value, to generate asserts for the function. Many technologies have been used to develop this demo tool like Eclipse CDT parser, gtest, gmock, lcov, gocv and JAVA, JNI. To make the generated unit test file readable, the tool is integrated with the GTest framework which is a widely used unit test framework for C++ programs.

In this research, my objective is to encourage developers to use this tool to generate test cases and as a result, they can save up a lot of time as time no longer needed to be allocated for unit test implementation, and they can focus on improving the code reliability and readability.

6.3 Limitations

The main problem which is addressed in this research is the reachability problem, discover inputs for a given a statement or branch of a program and the starting point is the input generation and the main objective is to discover methods to handle path exploration effectively to achieve the test goal (achieve statement or the branch). When the test goal is presented, the primary analysis of data dependency is carried out to identify the statements that affect the execution of the goal and then propagate the result of the test goal up in the order in order to trigger the execution of the test goal. The path exploration efficiency heavily relies on the results of the data dependency analysis and this leads to lower precision analysis. Also once a test case is derived this test case is executed to obtain the output result to generate asserts. Since compiling and executing the program is time-consuming and the end result is the program takes a long time to generate the test suite.

The main limitation of this tool developed is the correctness of the unit test generated. As stated in the above chapters, the test tool could detect issues that exist in the code, however, the tool cannot detect missing requirements. Once the unit test is generated manual intervention is required and the developer can improve the unit test further if the code generated can be improved and the developer should it a more meaningful test code.

Another limitation is, In order to tool to generate unit tests for a given class or a function, the tool needs to isolate that particular unit from its dependencies. If dependencies have not been properly designed tool will fail to generate unit tests. For example, if a dependency function is implemented without the virtual keyword tool will not be able to generate a mock function for it, hence it will call the actual implementation and this is not the desirable case. Hence during the design phase of the class, some level of attention needs to be given for unit test design as well even though the implementation is not done, meaning C++ implementation needs to comply with unit test standards. However, this is not a limitation in the tool rather a forced rule in unit test design.

6.4 Future work

This main focus of this research project is to implement a tool that generates unit test suite for a given C++ class such a way that at the end of test generation, and upon running the test suite a high code coverage should be attainable. However, as pointed out in the limitation section even though high code coverage can be attainable with this approach still there are questions regarding the validity of the generated result.

As described in the above chapters the ideal way to implement software is with Test Driven Development and we can call it TDD. However, as described TDD is expensive and some companies are not willing to invest a lot for TDD. This is basically a solution or rather I call it a replacement for the TDD approach. However as described in the limitation section, the proposed solution does not cover the entire scope of the TDD. TDD not only captures bugs in the software or application, but this approach can also identify missing features. However, the solution I propose can be extended to fully cover the scope of TDD. The tool can be extended so that it can detect missing requirements.

Reference

- [1] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, “An orchestrated survey of methodologies for automated software test case generation,” Aug. 2013.
- [2] H. Yoshida, S. Tokumoto, M. R. Prasad, I. Ghosh, T. Uehara, “FSX: Fine-grained Incremental Unit Test Generation for C/C++ Programs”, Proceedings of the 25th International Symposium on Software Testing and Analysis ser. ISSTA 2016, pp. 106-117, 2016.
- [3] J. Burnim, K. Sen, “Heuristics for scalable dynamic test generation”, in ASE '08, 2008.
- [4] G. Fraser, A. Arcuri, “Whole test suite generation”, IEEE Trans. Softw. Eng., vol. 39, no. 2, pp. 276-291, Feb. 2013.
- [5] S. J. Galler, B. K. Aichernig, “Survey on test data generation tools”, STTT, vol. 16, no. 6, pp. 727-751, 2014.
- [6] H. Tanida, T. Uehara, G. Li, I. Ghosh, “Automatic unit test generation and execution for javascript program through symbolic execution,” Proceedings of the Ninth International Conference on Software Engineering Advances, pp. 259-265, 2014.
- [7] X. Deng, Robby, J. Hatcliff, Kiasan/KUnit, “Automatic Test Case Generation and Analysis Feedback for Open Object-oriented Systems,” in Proc. of MUTATION 2007, IEEE CS, 2007.
- [8] Tassey, G, “The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology,” RTI Project Number 7007.011, 2002.
- [9] Hong Zhu, Patrick A. V. Hall, John H. R. May, “Software unit test coverage and adequacy,” ACM Computing Surveys, 29(4):366–427, 1997.
- [10] M. Ellims, J. Bridges, D. C. Ince, “The economics of unit testing”, Empir. Softw. Eng., vol. 11, no. 1, pp. 5-31, 2006.

- [11] Paul Ammann, Jeff Offutt, “Introduction to Software Testing,” Cambridge University Press, New York, NY, USA, 2008.
- [12] G. Li, I. Ghosh, and S. P. Rajan, “KLOVER: A symbolic execution and automatic test generation tool for C++ programs,” in CAV, 2011.
- [13] Li, G., Ghosh, I., Rajan, S. P., Gopalakrishnan, G. GKLEE, “Concolic Verification and Test Generation for GPUs,” 2012.
- [14] P. Garg et al., “Feedback-Directed Unit Test Generation for C/C++ Using Concolic Execution”, Proc. 35th Int'l Conf. Software Eng. (ICSE 13), pp. 132-141, 2013.
- [15] Jacob Burnim, Koushik Sen, “Heuristics for Scalable Dynamic Test Generation” in ASE '08, 2008.
- [16] Olivier Crameri, RekhaBachwani, Tim Brecht, Ricardo Bianchini, DejanKostic, Willy Zwaenepoel, “Concolic Execution Driven by Test Suites and Code Modifications,” 2009.
- [17] Y Kim, M Kim, YJ Kim, Y Jang, “Industrial Application of Concolic Testing Approach,” A Case Study on libexif by Using CREST-BV and KLEE, 2012.
- [18] Cristian Cadar, Daniel Dunbar, Dawson Engler, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs”, Proceedings of the USENIX Symposium on Operating System Design and Implementation, 2008.
- [19] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, MattiaVivanti, Ali Muhammad, “Software testing with code-based test generators: data and lessons learned from a case study with an industrial software component,” Software Qual J, vol. 22, no. 2, pp. 311-333, 2014.
- [20] Duc Anh, Nguyen & Ngoc Hung, Pham, “A Test Data Generation Method for C/C++ Projects,” 2017.
- [21] Patrice Godefroid, Nils Klarlund, and Koushik Sen, “DART: Directed Automated Random Testing,” In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05). ACM, New York, NY, USA, 2005.

- [22] Koushik Sen, Darko Marinov, and Gul Agha, “CUTE: A Concolic Unit Testing Engine for C”, In Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13). ACM, New York, NY, USA, 2005.
- [23] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger, “PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis”, In Proceedings of the 5th European Conference on Dependable Computing (EDCC’05). Springer-Verlag, Berlin, Heidelberg, 2005.
- [24] McMinn, Phil, “Search-based software test data generation”, a survey: Research Articles. *Softw. Test*, 2004.
- [25] Runeson, Per. (2006). A Survey of Unit Testing Practices. *IEEE Software*. 23. 10.1109/MS.2006.91.
- [26] Kassab, Mohamad & DeFranco, Joanna & Laplante, Phillip. (2017). Software Testing: The State of the Practice. *IEEE Software*. 34. 46-52. 10.1109/MS.2017.3571582.
- [27] Thummalapenta, Suresh & Xie, Tao & Tillmann, Nikolai & Halleux, Jonathan & Schulte, Wolfram, “MSeqGen: Object-Oriented Unit-Test Generation via Mining Source Code.” 2009.
- [28] Jaygarl, Hojun. “Capture-based Automated Test Input Generation”, 2010.
- [29] Yatoh, Kohsuke & Sakamoto, Kazunori & Ishikawa, Fuyuki & Honiden, Shinichi, “Feedback-Controlled Random Test Generation”, 2015.
- [30] Cseppentő, Lajos & Micskei, Zoltan. “Evaluating code-based test input generator tools. *Software Testing, Verification and Reliability*”, 2017.
- [31] Zhang, Chengyu & Su, Ting & Yan, Yichen & Wu, Ke & Pu, Geguang. “Towards Efficient Data-flow Test Data Generation Using KLEE”, 2018.

Appendix A – Random Test Generation Method

I used following class for testing purposes:

```
TestFile.cpp x
1  #include "TestFile.h"
2
3  MathUtils::MathUtils(int a, int b) {
4      width = a;
5      height = b;
6  }
7
8  int MathUtils::sum(int x, int y) {
9      return x + y;
10 }
11
12 int MathUtils::factorial(int x) {
13     if(x > 1) {
14         return x * this->factorial(x-1);
15     }
16     else {
17         return 1;
18     }
19 }
20
21 int MathUtils::substraction(int x, int y) {
22     return x - y;
23 }
24
25 int MathUtils::multiplication(int x, int y) {
26     return x * y;
27 }
28
29 int MathUtils::setWidth(int a, int b)
30 {
31     if (a < b)
32     {
33         width = a;
34         height = b;
35     }
36     else
37     {
38         height = a;
39         width = b;
40     }
41     return 1;
42 }
43
44 int MathUtils::getWidth() { return width; }
```

Figure A.1 – Sample test class

Tool generated following unit test class, and I have inserted captures of the generated class:

```
TestFile.cpp x ESTest.cpp x
1  #include "TestFile.cpp"
2  #include "gtest/gtest.h"
3
4  TEST(SuiteTest, test0) {
5
6      int int0 = 0;
7      int int1 = 0;
8      MathUtils* class0 = new MathUtils(int0, int1);
9      int int2 = 0;
10     int factorial_Val__ = class0->factorial(int2);
11     ASSERT_EQ(1, factorial_Val__ );
12 }
13
14 TEST(SuiteTest, test1) {
15
16     int int0 = 0;
17     int int1 = 0;
18     MathUtils* class0 = new MathUtils(int0, int1);
19     int int2 = 1314;
20     int int3 = (-1717);
21     int multiplication_Val__ = class0->multiplication(int2, int3);
22     ASSERT_EQ(-2256138, multiplication_Val__);
23 }
24
25 TEST(SuiteTest, test2) {
26
27     int int0 = (-2976);
28     int int1 = (-3303);
29     MathUtils* class0 = new MathUtils(int0, int1);
30     int int2 = 0;
31     int int3 = 0;
32     int setWidth_Val__ = class0->setWidth(int2, int3);
33     ASSERT_EQ(1, setWidth_Val__);
34 }
35
36 TEST(SuiteTest, test3) {
37
38     int int0 = 3862;
39     int int1 = 1088;
40     MathUtils* class0 = new MathUtils(int0, int1);
41     int int2 = 0;
42     int int3 = 0;
43     int sum_Val__ = class0->sum(int2, int3);
44     ASSERT_EQ(0, sum_Val__);
```

Figure A.1 – Sample test class

Figure A.2 – Generated Unit test - I

```
TestFile.cpp x  EStest.cpp x
47 TEST(SuiteTest, test4) {
48
49     int int0 = 2230;
50     int int1 = 1123;
51     MathUtils* class0 = new MathUtils(int0, int1);
52     int int2 = 1287;
53     int int3 = (-1225);
54     int subtraction_Val__ = class0->subtraction(int2, int3);
55     ASSERT_EQ(2512, subtraction_Val__);
56 }
57
58 TEST(SuiteTest, test6) {
59
60     int int0 = (-640);
61     int int1 = 2224;
62     MathUtils* class0 = new MathUtils(int0, int1);
63     int int2 = 0;
64     int int3 = 15;
65     int sum_Val__ = class0->sum(int2, int3);
66     ASSERT_EQ(15, sum_Val__);
67 }
68
69 TEST(SuiteTest, test7) {
70
71     int int0 = 2901;
72     int int1 = (-237);
73     MathUtils* class0 = new MathUtils(int0, int1);
74     int int2 = (-52);
75     int int3 = 0;
76     int setWidth_Val__ = class0->setWidth(int2, int3);
77     ASSERT_EQ(1, setWidth_Val__);
78 }
79
80 TEST(SuiteTest, test8) {
81
82     int int0 = 0;
83     int int1 = 303;
84     MathUtils* class0 = new MathUtils(int0, int1);
85     int getWidth_Val__ = class0->getWidth();
86     ASSERT_EQ(0, getWidth_Val__);
87 }
```

Figure A.3 – Generated Unit test - II

```
TestFile.cpp × ESTest.cpp ×
164     int int0 = (-1855);
165     int int1 = 0;
166     MathUtils* class0 = new MathUtils(int0, int1);
167     int int2 = 0;
168     int int3 = 0;
169     int setWidth_Val__ = class0->setWidth(int2, int3);
170     ASSERT_EQ(1, setWidth_Val__);
171 }
172
173 TEST(SuiteTest, test17) {
174
175     int int0 = (-4164);
176     int int1 = 769;
177     MathUtils* class0 = new MathUtils(int0, int1);
178     int int2 = 0;
179     int int3 = 0;
180     int subtraction_Val__ = class0->subtraction(int2, int3);
181     ASSERT_EQ(0, subtraction_Val__);
182 }
183
184 TEST(SuiteTest, test18) {
185
186     int int0 = 0;
187     int int1 = 0;
188     MathUtils* class0 = new MathUtils(int0, int1);
189     int area_Val__ = class0->area();
190     ASSERT_EQ(0, area_Val__);
191 }
192
193 TEST(SuiteTest, test19) {
194
195     int int0 = 0;
196     int int1 = 0;
197     MathUtils* class0 = new MathUtils(int0, int1);
198     int int2 = 0;
199     int int3 = 0;
200     int sum_Val__ = class0->sum(int2, int3);
201     ASSERT_EQ(0, sum_Val__);
202 }
203
204 int main(int argc, char** argv) {
205     testing::InitGoogleTest(&argc, argv);
206     return RUN_ALL_TESTS();
207 }
```

Figure A.4 – Generated Unit test - III

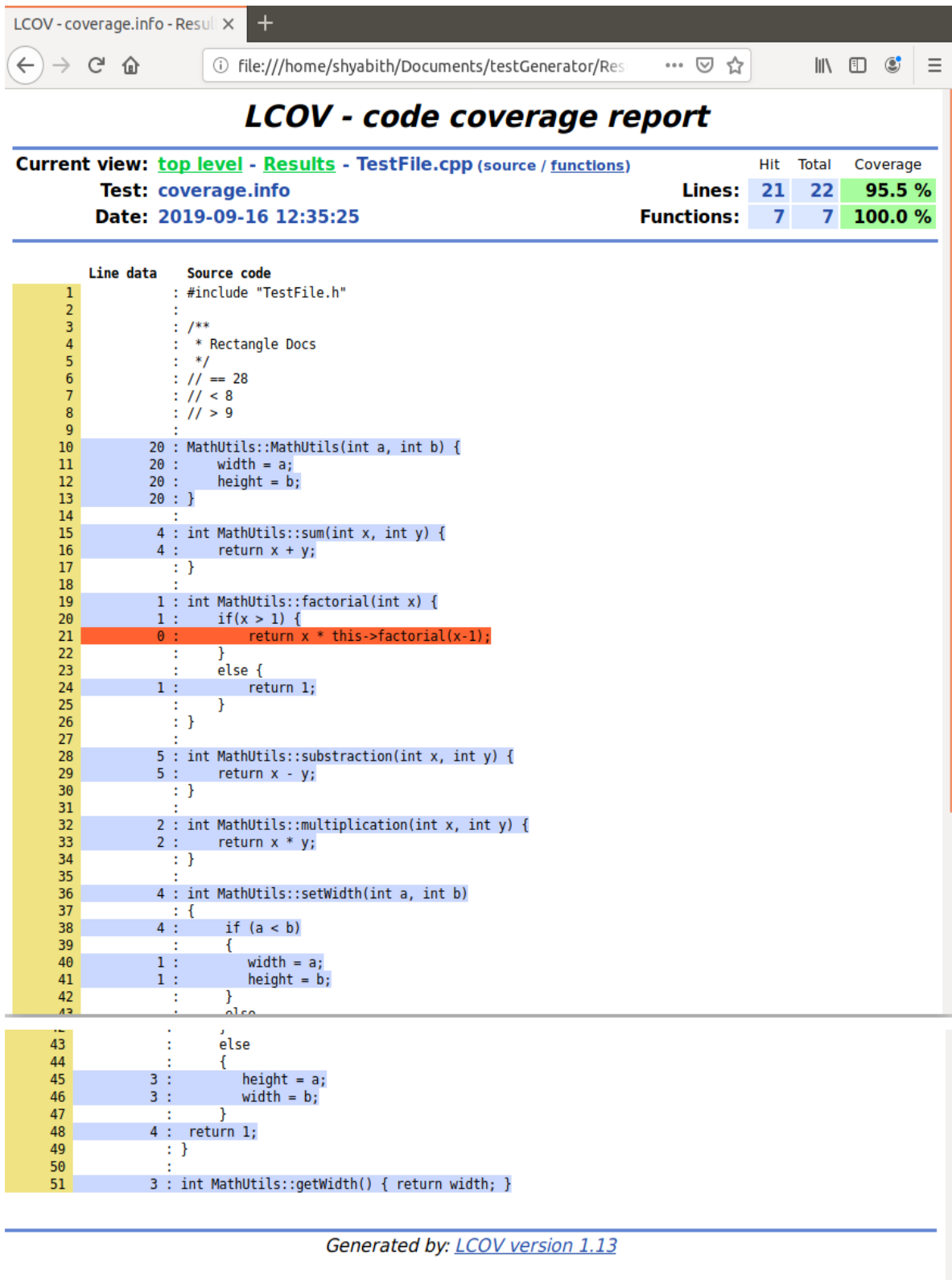


Figure A.5 – Generated LCOV Report

CMake is used to compile the C++ unit test class created, and also used for compilation of code in the intermediate stages as well:

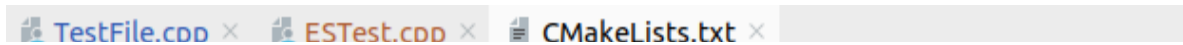


Figure A.6 – CMake file use to compile the class

```
2
3 find_program( GCOV_PATH gcov )
4 find_program( LCOV_PATH NAMES lcov lcov.bat lcov.exe lcov.perl)
5
6 # Locate GTest
7 find_package(GTest REQUIRED)
8 include_directories(${GTEST_INCLUDE_DIRS})
9
10 #include( cmake/CodeCoverage.cmake )
11 SET(CMAKE_CXX_FLAGS "-g -O0 --coverage -fprofile-arcs -ftest-coverage")
12 SET(CMAKE_C_FLAGS "-g -O0 --coverage -fprofile-arcs -ftest-coverage")
13
14 # Link runTests with what we want to test and the GTest and pthread library
15 add_executable(runTests ESTest.cpp)
16 target_link_libraries(runTests ${GTEST_LIBRARIES} pthread)
17
18
19
```

Class generated with JNI by the tool to execute the C++ code:

```
TestFile.cpp x ESTest.cpp x CMakeLists.txt x MathUtilsClzz.java x
1  import ...
4
5  @Properties(
6      value=@Platform(include={"TestFile.h"},
7                          linkpath = {"~/home/shyabith/Documents/testGenerator/libccp"},
8                          link="Test"),
9      target="MathUtilsClzz"
10 )
11 public class MathUtilsClzz {
12     public static class MathUtils extends Pointer {
13         static {
14             Loader.load();
15         }
16         public MathUtils(int x, int y) { allocate(x, y); }
17         private native void allocate(int x, int y);
18         public native @ByVal int sum(int x, int y);
19         public native @ByVal int substraction(int x, int y);
20         public native @ByVal int factorial(int x);
21         public native @ByVal int multiplication(int x, int y);
22         public native @ByVal int area();
23         public native @ByVal int getWidth();
24         public native @ByVal int setWidth(int a, int b);
25     }
26 }
27
28
29 public static void main(String[] args) {
30     int int0 = 0;
31     int int1 = 0;
32     MathUtils class0 = new MathUtils(int0, int1);
33     int int2 = 0;
34     int int3 = 0;
35     int sum_Val_ = class0.sum(int2, int3);
36     System.out.println("###sum_Val_ "+ sum_Val_);
37
38 }
39 }
```

Figure A.7 – Intermediate generated JAVA class with JNI