

CRG: Condition Rank Generator for Program Repair

*Submitted in partial fulfilment of the requirements
of the degree of
Bachelor of Technology
By*

Jaswant Meghwal 177124
Neeraj Kumar Gond 177141
Nikhil Kumar 177142

Under the Guidance of

Dr. Sangharatna Godbole
Assistant Professor
Department of CSE
NIT Warangal



Department of Computer Science and Engineering

National Institute of Technology

Warangal

2020-21

APPROVAL SHEET

The project work entitled “CRG: Condition Rank Generator for Program Repair” by Jaswant Meghwal (1777124), Neeraj Kumar Gond (177141), and Nikhil Kumar (177142) is approved for the degree of Bachelor of Technology in Computer Science and Engineering.

EXAMINERS

SUPERVISOR

Dr. Sangharatna
Godbole
Assistant Professor
Department of Computer Science and Engineering

CHAIRMAN

Prof. P Radha Krishna
Department of Computer Science and Engineering

Date :

Place :

DECLARATION

We declare that this written submission represents our ideas in our own words and where other's ideas or words have been included, we have adequately cited and referenced the sources. We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in our submission. We understand that any violation of the above will be a cause for disciplinary action by the institute and can also evoke penal action from the sources which have thus not been properly cited or from where proper permission has not been taken when needed.

Jaswant Meghwal
177124

Neeraj Kumar Gond
177141

Nikhil Kumar
177142

Date:

NATIONAL INSTITUTE OF TECHNOLOGY

WARANGAL-506004

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the project work entitled "CRG: Condition Rank Generator for Program Repair" is a bonafide record of work carried out by "by Jaswant Meghwal (1777124), Neeraj Kumar Gond (177141), and Nikhil Kumar (177142)", submitted to the faculty of "Computer Science and Engineering Department", in partial fulfillment the requirements for the award of the degree of Bachelor of Technology in "Computer Science and Engineering" at National Institute of Technology, Warangal during the academic year 2020-21.

Dr. Sangharatna Godbole

Project Guide

Prof. P. Radha Krishna HOD of CSE Department of NIT Warangal

ABSTRACT

Automated program repair holds the potential to significantly reduce software maintenance effort and cost. However, recent studies have shown that it often produces low-quality patches that repair some but break other functionality. We hypothesize that producing patches by replacing likely faulty regions of code with semantically-similar code fragments, and doing so at a higher level of granularity than prior approaches can better capture abstraction and the intended specification, and can improve repair quality.

SOSRepair, an automated program repair technique that uses semantic code search to replace candidate buggy code regions with behaviorally-similar (but not identical) code written by humans. SOSRepair is the first such technique to scale to real-world defects in real-world systems. Research in fault localization can significantly improve the quality of program repair techniques.

Software fault localization, the act of identifying the locations of faults in a program, is widely recognized to be one of the most tedious, time consuming, and expensive yet equally critical activities in program debugging. Due to the increasing scale and complexity of software today, manually locating faults when failures occur is rapidly becoming infeasible, and consequently, there is a strong demand for techniques that can guide software developers to the locations of faults in a program with minimal human intervention. This demand in turn has fueled the proposal and development of a broad spectrum of fault localization techniques, each of which aims to streamline the fault localization process and make it more effective by attacking the problem in a unique way.

We catalog and provide a comprehensive overview of such techniques and discuss key issues and concerns that are pertinent to software fault localization as a whole.

We first chose Line Code Coverage as a basis to identify probable faulty statements in the code. Later we analyzed that Line Code Coverage was unable to differentiate between a conditional statement and a simple statement i.e., statement coverage cannot distinguish the code separated by logical operators from the rest of the statement. It gives full statement coverage of the code without exposing the bug in it. So, we decided to choose Condition Code Coverage as basis to implement Fault Localization (FL). The main idea of this coverage is to expose bug by exercising as many paths or conditions through the code as possible since bugs are often sensitive to branches and conditions.

CONTENTS

ABSTRACT	6
CONTENTS	8
CHAPTER 1	10
Introduction	10
1.1 Overview of Program Repair:	10
1.2 Overview of Automated Program Repair:	10
1.2.1 Advantages of Automated Program Repair:	10
1.2.2 Problems with existing Automated Program Repair:	11
1.3 Overview of SOS Repair:	12
1.3.1 What is SOS Repair?	12
1.3.2 Advantages of SOS Repair:	12
1.4 Overview of Fault Localization:	12
1.4.1 Importance of Fault Localization	13
1.4.2 Methods of Fault Localization	13
1.5 Problem Statement	14
1.6 Objectives	14
CHAPTER 2	15
Related Work	15
2.1 Historical Spectrum based Fault Localization	15
2.2 Exploiting count spectra for Bayesian Fault Localization	15
2.3 Combining Spectrum-Based Fault Loc. & Statistical Debugging: An Empirical Study	16
2.4 AI for the Win: Improving Spectrum-based Fault Localization	16
2.5 Slice-based statistical Fault Localization	17

2.6 BP NEURAL NETWORK BASED EFFECTIVE FAULT LOCALIZATION	18
2.7 Spectrum-based Fault Localization Using Machine Learning	18
2.8 Fault Localization Analysis Based on Deep Neural Network	19
2.9 A Lightweight Fault Localization Approach based on XGBoost	19
CHAPTER 3	21
Design	21
3.1 Architecture of SOS Repair:	21
3.2 Profile Construction: Architecture of Fault Localization	22
3.2.1 Line Code Coverage-based FL	23
3.2.2 Condition Code Coverage-based FL	25
CHAPTER 4	27
Implementation Details	27
4.1 Line/Statement Code Coverage based FL	27
4.2 Condition Code Coverage based FL	32
Chapter 5	38
Experimental Results and Observations	38
5.1 Line/Statement Code Coverage based FL Results	38
5.2 Condition Code Coverage based FL Results	40
CHAPTER 6	42
Conclusion	42
CHAPTER 7	43
References	43

CHAPTER 1

Introduction

1.1 Overview of Program Repair:

A program becomes buggy when it gives wrong output to at least one valid input or it crashes because of exceptions. In such cases it needs to be repaired to fulfil the requirement of goal that we want to achieve with that program. The technique of exposing and fixing bugs in the program is called Program Repair. Program Repair can be done:

1. Manually (By Human) – Technically debugging by Human
2. Automatically (By another special Program) – Automated Program Repair technique

1.2 Overview of Automated Program Repair:

Automated Program Repair is a rapidly growing research area in computer science. There are numerous tools and techniques that have been developed to automatically find and fix bugs in source code. This topic is directly related to Quality Engineering and improving the quality of software by reducing the cost and effort of fixing bugs.

The goal of automated program repair techniques is to take a program and a suite of tests, some of which that program passes and some of which it fails, and to produce a patch that makes the program pass all the tests in that suite.

1.2.1 Advantages of Automated Program Repair:

There are many advantages of automated program repair in practice. The debugging process is a very time consuming and expensive activity in the software development process, annually costing hundreds of billions of dollars globally. Automatically repairing bugs can greatly reduce this cost and help save companies

money and time. A study at UVA analyzed open-source C projects and determined their tool GenProg could fix bugs for just under \$8 each. Additionally, even though focus on testing and quality is increasing, the total number of bugs in software is also increasing rapidly. Top tech companies such as Microsoft, Google, Mozilla, Facebook, Twitter, Paypal, and more have hosted bug bounty programs which pay bug hunters to report bugs for them. This has its own pros and cons, but automatically detecting and fixing bugs can help companies find vulnerabilities and make their software more secure.

1.2.2 Problems with existing Automated Program Repair:

The main disadvantage of automated program repair scaling. The research and tools listed at have shown automated program repair is effective, however the studies are performed on smaller code bases and student programming assignments. The technology isn't there yet to automatically repair larger projects due to their size and complexity. For patch generation the search space for creating a fix can be infinite, which is much harder to account for enterprise software with thousands of lines of code (or more and other dependencies).

Technique based on test suite weak specification to find patches can overfit to particular test suite (patches passes all the test cases in the test suite, but fails to fix underlying problems i.e. held-out secondary independent test cases)

Overfitting problem

```
1 largest = 0 ;
2 if( a>b )
3 {
4     largest = a ;
5 }
6
7 if( b > a )
8 {
9     largest = b ;
10 }
```

For the above code -

Passes test suits (a, b, largest) (3,2,3), (1,5,5), (6,6,6)

But fails secondary test suite (7, 12, 12), (-3, -10, -3), (4, 4, 6) i.e. overfits.

Higher-granularity (code-snippets), semantic-based changes can improve quality. But never scales to large programs. So go with SOS Repair which scales with large, real world programs (>2000KLOC). Earlier approach scales to only with max 25LOC.

1.3 Overview of SOS Repair:

SOS Repair, an automated program repair technique that uses semantic code search to replace candidate buggy code regions with behaviorally-similar (but not identical) code written by humans. SOSRepair is the first such technique to scale to real-world defects in real-world systems.

1.3.1 What is SOS Repair?

SOS Repair is a novel technique that uses input-output-based semantic code search to automatically find and contextualize patches to fix real-world defects. SOSRepair locates likely buggy code regions, identifies similarly-behaving fragments of human-written code, and then changes the context of those fragments to fit the buggy context and replace the buggy code

1.3.2 Advantages of SOS Repair:

On a subset of the many bugs benchmark of such defects, SOS Repair produces patches for 22 (34%) of the 65 defects, including 3, 5, and 6 defects for which previous state-of-the-art techniques Angelix, Prophet, and GenProg do not, respectively. On these 22 defects, SOSRepair produces more patches (9, 41%) that pass all independent tests than the prior. SOSRepair, a novel technique that uses input-output-based semantic code search to automatically find and contextualize patches to fix real-world defects. SOSRepair locates likely buggy code regions, identifies similarly-behaving fragments of human-written code, and then changes the context of those fragments to fit the buggy context and replace the buggy code.

1.4 Overview of Fault Localization:

In general, FAULT LOCALIZATION is the act of tracing and locating of bugs, if any, present in the software. Here we intend to locate bugs in the codes based on some fixed

criteria. Fault localization precision is a key factor in SOS Repair's success. Manually improving fault localization allows SOSRepair to patch 23 (35%) defects, of which 16 (70%) pass all independent tests.

1.4.1 Importance of Fault Localization:

- Software fault localization is one of the most expensive, tedious and time-consuming activities.
- High demand for Automatic Fault Localization techniques.
- Guide programmers to the locations of faults, with minimal human intervention.
- This demand has led to the proposal and development of various methods

1.4.2 Methods of Fault Localization:

- Traditional Method
 - Analyse Memory Dump –
 - Analyzing the memory dump helps to locate bugs.
 - Problem - Analysis of a tremendous amount of data.
 - Using Print Statements –
 - Inserting the print statements around suspicious code.
 - Problem:
 - Need of good understanding of code and which test case corresponds to which line of code.
 - Print statements neither should be too few nor too many.

Both the methods need human intervention at each step.

- Modern Method

Modern method has two major phases

1. Phase-1: Identify suspicious code that may contain bugs.

2. Phase-2: Programmer need to examine if the identified code actually has bug or not.

We focus on the Phase-1 of Modern Method based on the following idea:

- Suspicious code is prioritized based on its likelihood (in case of Line Code Coverage) and fitness value (in case of Condition Code Coverage) of containing bugs.
- Code with a higher priority should be examined before code with a lower priority.

1.5 Problem Statement

To implement the Modern Fault Localization using Line Code Coverage and Condition Code Coverage technique.

1.6 Objectives

1. To identify potentially faulty statements by mining both passing and failing execution of a faulty program – Line Code Coverage
2. To identify the potentially faulty predicate by finding the reachability of atomic conditions – Condition Code Coverage
3. To formulate the fitness score for ranking predicate.
4. To rank the faulty statements based on Statement coverage using likelihood score.
5. To rank the faulty predicate based on Condition coverage using fitness score.

CHAPTER 2

Related Work

2.1 Historical Spectrum based Fault Localization

First, version histories record how bugs are introduced to software projects and this information reflects the root cause of bugs directly. Second, the evolution histories of code can help differentiate those suspicious code entities ranked in tie by SBFL. Intuitions are also inspired by the observations on debugging practices from large open source projects and industry. Based on the intuitions, they propose a novel technique HSFL (historical spectrum based fault localization). Specifically, HSFL identifies bug-inducing commits from the version history in the first step. It then constructs historical spectrum (denoted as Histrum) based on bug-inducing commits, which is another dimension of spectrum orthogonal to the coverage based spectrum used in SBFL. HSFL finally ranks the suspicious code elements based on our proposed Histrum and the conventional spectrum. HSFL outperforms the state-of-the-art SBFL techniques significantly on the Defects4J benchmark. Specifically, it locates and ranks the buggy statement at Top-1 for 77:8% more bugs as compared with SBFL, and 33:9% more bugs at Top-5. Besides, for the metrics MAP and MRR, HSFL achieves an average improvement of 28:3% and 40:8% over all bugs, respectively. Moreover, HSFL can also outperform other six families of fault localization techniques, and our proposed Histrum model can be integrated with different families of techniques and boost their performance.

2.2 Exploiting count spectra for Bayesian fault localization

Aim: In this paper, they study the impact of exploiting component execution frequency on the diagnostic quality. **Method:** they present a reasoning-based SFL approach, dubbed Zoltar-C, that exploits not only component involvement but also their frequency, using an approximate, Bayesian approach to compute the probabilities of the diagnostic candidates. Zoltar-C is evaluated and compared to other well-known, low-cost techniques (such as

Tarantula) using a set of programs available from the Software Infrastructure Repository. **Results:** Results show that, although theoretically Zoltar-C can be of added value, exploiting component frequency does not improve diagnostic accuracy on average. **Conclusions:** The major reason for this unexpected result is the highly biased sample of passing and failing tests provided with the programs under analysis. In particular, the ratio between passing and failing runs, which has a major impact on the probability computations, does not correspond to the false negative (failure) rates associated with the actually injected faults.

2.3 Combining Spectrum-Based Fault Localization and Statistical Debugging: An Empirical Study

In this paper they perform a systematical empirical study on the combination of SBFL and SD. We first build a unified model of the two techniques, and systematically explore four types of variations, different predicates, different risk evaluation formulas, different granularities of data collection, and different methods of combining suspicious scores. Their study leads to several findings. First, most of the effectiveness of the combined approach contributed by a simple type of predicates: branch conditions. Second, the risk evaluation formulas of SBFL significantly outperform that of SD. Third, fine grained data collection significantly outperforms coarse-grained data collection with a little extra execution overhead. Fourth, a linear combination of SBFL and SD predicates outperforms both individual approaches. According to their empirical study, we propose a new fault localization approach, PREDFL (Predicate-based Fault Localization), with the best configuration for each dimension under the unified model. Then, we explore its complementarity to existing techniques by integrating PREDFL with a state-of-the-art fault localization framework. The experimental results show that PREDFL can further improve the effectiveness of state-of-the-art fault localization techniques. More concretely, integrating PREDFL results in an up to 20.8% improvement w.r.t the faults successfully located at Top-1, which reveals that PREDFL complements existing techniques.

2.4 AI for the Win: Improving Spectrum-based Fault Localization

In this paper, they discussed the beneficial use of AI techniques in the domain of

automated debugging. In particular, we showed how to improve spectrum-based fault localization. All of the mentioned improvements aim in providing a better ranking of fault candidates in order to further reduce the number of statements to be examined during debugging. All of the presented approaches make use of ideas originating from model-based diagnosis. In the empirical section, we showed that Deputo performs best. However, since Deputo does not scale to large programs, Barinel and Sendys are good alternatives. As a consequence, we strongly recommend the use of AI techniques for automated debugging. Moreover, we discussed open challenges of automated debugging and outlined possible solutions.

they presented three variants of the well-known spectrum-based fault localization technique that are enhanced by using methods from Artificial Intelligence. Each of the three combined approaches outperforms the underlying basic method concerning diagnostic accuracy. Hence, the presented approaches support the hypothesis that combining techniques from different areas is beneficial. In addition to the introduction of these techniques, we perform an empirical evaluation, discuss open challenges of debugging and outline possible solutions.

2.5 Slice-based statistical Fault Localization

Recent techniques for fault localization statistically analyze coverage information of a set of test runs to measure the correlations between program entities and program failures. However, coverage information cannot identify those program entities whose execution affects the output and therefore weakens the aforementioned correlations. This paper proposes a slice-based statistical fault localization approach to address this problem. Their approach utilizes program slices of a set of test runs to capture the influence of a program entity's execution on the output, and uses statistical analysis to measure the suspiciousness of each program entity being faulty. In addition, this paper presents a new slicing approach called approximate dynamic backward slice to balance the size and accuracy of a slice, and applies this slice to our statistical approach. They use two standard benchmarks and three real-life UNIX utility programs as our subjects, and compare our approach with a sufficient number of fault localization techniques. The experimental results show that our approach can significantly improve the effectiveness of fault localization.

2.6 BP NEURAL NETWORK-BASED EFFECTIVE FAULT LOCALIZATION

They proposed the use of a back-propagation (BP) neural network, a machine learning model which has been successfully applied to software risk analysis, cost prediction, and reliability estimation, to help programmers effectively locate program faults. A BP neural network is suitable for learning the input-output relationship from a set of data, such as the inputs and the corresponding outputs of a program. We first train a BP neural network with the coverage data (statement coverage in our case) and the execution result (success or failure) collected from executing a program, and then we use the trained network to compute the suspiciousness of each executable statement, in terms of its likelihood of containing faults. Suspicious code is ranked in descending order based on its suspiciousness. Programmers will examine such code from the top of the rank to identify faults. Four case studies on different programs (the Siemens suite, the Unix suite, grep and gzip) are conducted. Their results suggest that a BP neural network-based fault localization method is effective in locating program faults.

2.7 Spectrum-based Fault Localization Using Machine Learning

In this thesis, they proposed a debugging approach that is a hybrid of SBFL and Static Analysis to provide extra information to SBFL about programs under test. Program statements are categorized into various types which are given weights based on how likely a category is related to a bug. They evaluate the performance of our technique both for small programs from the Siemens Test Suite (STS) and the larger Space program. Results show that our technique improves the performance of a wide variety of fault localization metrics on single and multiple bug data. Statement type is one of the many program features that can be used to get valuable clues about the location of a bug. Other features could be statement length, nesting depth, cyclomatic complexity etc. However, it is very expensive and thus impractical to plugin each of these features into our proposed technique to find how effective they are in fault localization. They devised a statistical method that can be used to evaluate the importance of a program feature in fault localization without using it in a machine learning method. The similarity of the results obtained by our proposed statistical

method and actual implementation of the feature in machine learning techniques depicts the accuracy of our proposed method in feature-importance identification. Usually, when two or more well-performing techniques are combined, they do not perform better than any of the techniques individually. They combined our hyperbolic class of metrics with statement weightage technique, and the combined technique further improves the performance of the hyperbolic class of metrics. The improvement in performance is statistically significant for many single and multi-bug datasets.

2.8 Fault Localization Analysis Based on Deep Neural Network

We propose a fault localization method based on deep neural network (DNN). This approach is capable of achieving the complex function approximation and attaining distributed representation for input data by learning a deep nonlinear network structure. It also shows a strong capability of learning representation from a small sized training dataset. Our DNN-based model is trained utilizing the coverage data and the results of test cases as input and we further locate the faults by testing the trained model using the virtual test suite.

This paper conducts experiments on the Siemens suite and Space program. The results demonstrate that our DNN-based fault localization technique outperforms other fault localization methods like BPNN, Tarantula, and so forth. This paper proposes a fault localization method based on deep neural network (DNN). With the capability of estimating complicated functions by learning a deep nonlinear network structure and further attaining distributed representation of input data, this method exhibits strong ability to learn representation from minority sample data. Moreover, DNN is one of the deep learning models that has been successfully applied in many other areas of software engineering.

2.9 A Lightweight Fault Localization Approach based on XGBoost

Software fault localization is one of the key activities in software debugging. The program spectrum-based approach is widely used in fault localization. However, lots of program information, for example, the sequence of the execution statement and statement semantics, is missing when such an approach is utilized, which affects the performance. XGBoost is an effective learning algorithm, which can use the

characteristics of the training data to build a classification tree during training. In addition, XGBoost can iteratively adjust the information value of the feature, so that the training process retains the importance information of the feature. This paper proposes applying XGBoost into fault localization utilizing information of program execution behaviors. A novel method called XGB-FL is developed, where the program spectrum information is converted into a coverage matrix to train the XGBoost model. We can get the characteristics of the data through the trained model and the importance of the program statement in the classification process. This is also the basis for judging whether the statement is likely to contain a fault. Nine representative data sets have been chosen to evaluate the performance of XGB-FL. The experimental results show that XGB-FL can generally deliver a higher performance in fault localization than those baseline techniques, in terms of precision and efficiency.

CHAPTER 3

Design and Architecture

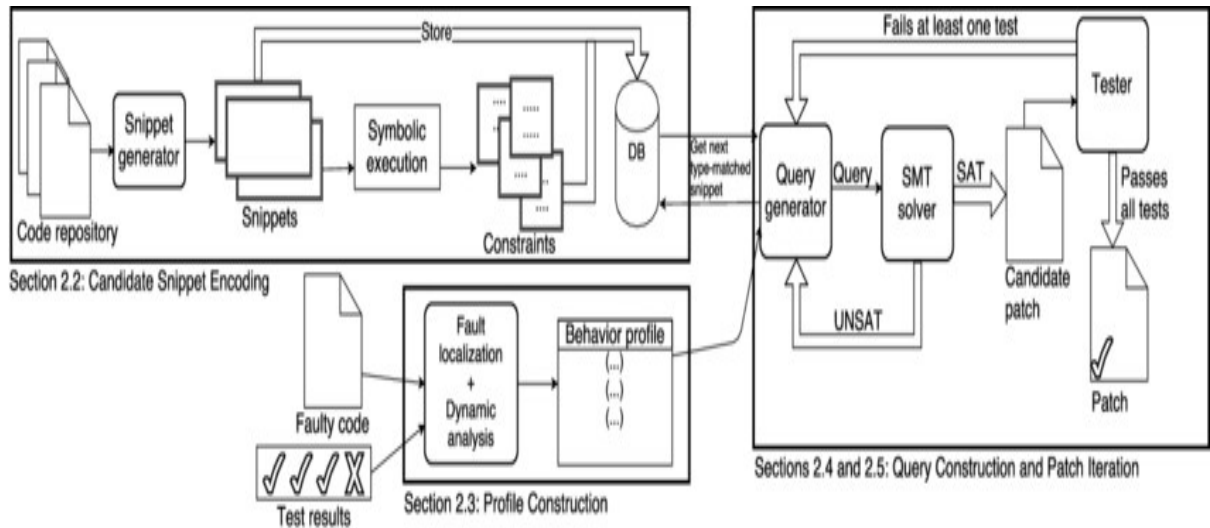


Figure 3.1: Overall Architecture Diagram of SOS Repair

3.1 Architecture of SOS Repair

Given a program and a set of test cases capturing correct and buggy behavior, SOSRepair generates patches by searching over a database of snippets of human-written code. semantic search looks for code based on a specification of desired and undesired behavior. SOSRepair uses test cases to construct a behavioral profile of a potentially buggy code region. SOSRepair then searches over a database of snippets for one that implements the inferred desired behavior, adapts a matching snippet to the buggy code's context, and patches the program by replacing the buggy region with patch code, inserting patch code, or deleting the buggy region. Finally, SOSRepair validates the patched program by executing its held-out secondary test cases.

- Uses symbolic execution to produce static behavioral approximations of a set of candidate bug repair snippets

- Constructs a dynamic profile of potentially-buggy code regions, which serve as inferred input-output specifications of desired behavior
- Constructs an SMT query to identify candidate semantic repairs to be transformed into patches and validated
- Iteratively attempts to produce a patch until timeout occurs.

1) Candidate Snippet Encoding:

- Codes are split into snippets of desired granularity (e.g., 3-7 lines).
- Symbolically execute the snippets to capture their semantics to generate constraints and store the results on a database.
- Find inputs and outputs to these snippets of code and passes to symbolic execution which generates constraints represents semantics of code.

2) Profile construction:

- Need to capture what buggy code is supposed to do.
- Find faulty regions using automated/manual (location which developer uses to fix program) in SOS Repair fault localization as fault localization has significant impact on patch quality.
- Use the test cases to create profile constraints of inputs and outputs to the buggy snippet. (Dynamic Analysis).

3) Query construction:

- First select semantically similar snippet from our database that matches the I/O types of buggy snippets i.e., candidate repair snippet.
- Using profile and snippet constraints we need to create and run a query to see if the snippet can be used as a patch.
- For query generation, select valid mapping between two sets of variables i.e. profile constraint and snippet constraint.
- SOS Repair uses I/O component-based program synthesis to find a single valid mapping and a single query.
- Now, we have snippet, profile and mapping constraints.

4) Give this query to SMT SOLVER(Z3):

- If returns UNSAT, candidate snippet can't be used as patches to buggy program.
- If returned SAT then return valid mapping between two sets of variables, move to next step patch generation.

5) Patch Generation:

- Given a snippet of code and valid mapping, generate patch for defect.
- We run held-out test suite with patched program. If passes all we return it else we look for new candidate snippet.

3.2 Profile Construction: Architecture of Fault Localization

Need to capture what buggy code is supposed to do and find faulty regions using automated/manual (location which developer uses to fix program) in SOS Repair fault localization as fault localization has significant impact on patch quality. Use the test cases to create profile constraints of inputs and outputs to the buggy snippet. (Dynamic Analysis).

3.2.1 Line Code Coverage-based

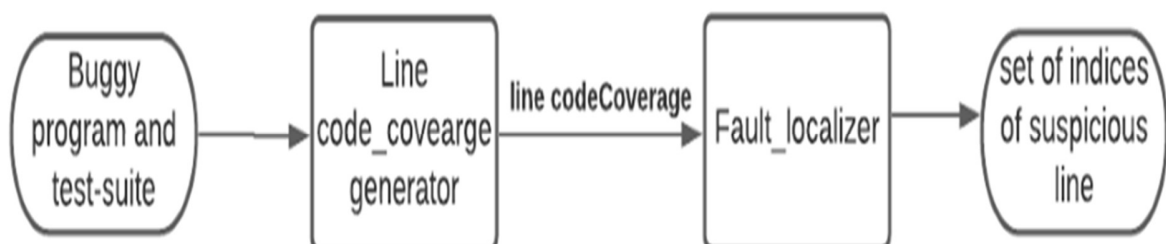


Fig: High Level component diagram of FL based on Line/Statement coverage

Description of each component in the above architecture:

➤ Buggy Program and Test-Suite:

We have python small programs each one having bug in some of the lines/statements.

Test-suite basically contains test cases corresponding to each buggy program.

- First line of each test case having input arguments.
- Second line contains the target output i.e., correct output.

➤ Line/Statement code coverage Generator:

Statement coverage is a code coverage metric that tells you whether the flow of control reached every executable statement of source code at least once. It takes a program file and corresponding test-suite as inputs and for each testcase of test-suite, it executes buggy program and store output in a file. Each observed output is compared with target output already stored in test suite. It uses dictionary data structure to store pair test case and result as key-value pair. It is required output is set of lines executed by a particular testcase and result of its execution.

➤ Fault Localizer:

It takes input from output of line code coverage i.e. set of executed lines by a particular testcase and calculate the following for each statement:

- Count of total pass and fail test case.
- Count of pass test cases that executed the given statement.
- Count of fail test cases that executed the given statement.
- Compute **pass percentage** and **fail percentage**.
- Compute **Suspicious score** which is equivalent to fail percentage divided by sum of pass and fail percentage.

- It generates output as set of indices of suspicious line ranked based on suspicious score.

3.2.2 Condition Code Coverage-based

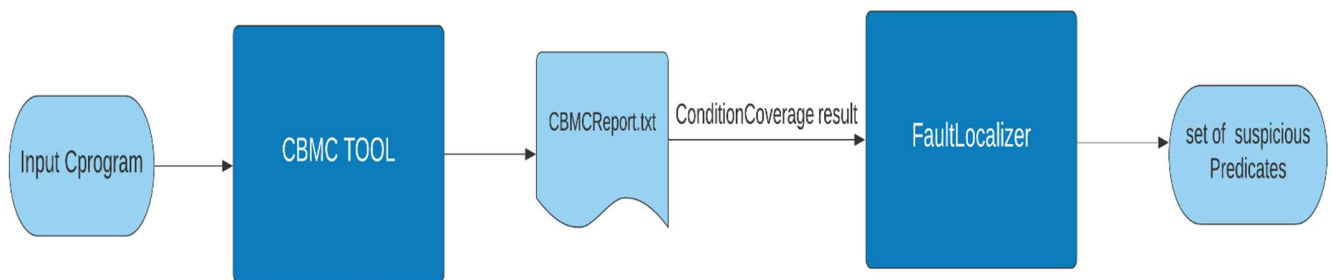


Fig: High Level component diagram of FL based on condition code coverage

Description of each component in the above architecture:

➤ CBMC TOOL

CBMC is a Bounded Model Checker for C and C++ programs. It supports C version variants C89, C99, most of C11 and most compiler extensions provided by gcc and Visual Studio. CBMC verifies memory safety (which includes array out of bounds checks and checks for the safe use of pointers), checks for exceptions, checks for various variants of undefined behavior, and user-specified assertions. Verification is performed by unwinding the loops in the program and passing the resulting equation to a decision procedure. Here, in our Project we use CBMC to generate the condition coverage of the code.

Results generated by CBMC:

**** coverage results:**

```
[main.coverage.1] file problem10.c line 1254 function main condition 'input != 5' false: SATISFIED
[main.coverage.2] file problem10.c line 1254 function main condition 'input != 5' true: SATISFIED
[main.coverage.3] file problem10.c line 1254 function main condition 'input != 4' false: SATISFIED
[main.coverage.4] file problem10.c line 1254 function main condition 'input != 4' true: SATISFIED
[main.coverage.5] file problem10.c line 1254 function main condition 'input != 3' false: SATISFIED
[main.coverage.6] file problem10.c line 1254 function main condition 'input != 3' true: SATISFIED
[main.coverage.7] file problem10.c line 1254 function main condition 'input != 2' false: SATISFIED
[main.coverage.8] file problem10.c line 1254 function main condition 'input != 2' true: SATISFIED
[main.coverage.9] file problem10.c line 1254 function main condition 'input != 1' false: SATISFIED
[main.coverage.10] file problem10.c line 1254 function main condition 'input != 1' true: SATISFIED
```

Above result shows the coverage generated, by applying CBMC to the C program which is having line number, function name, various atomic conditions in each predicate and shows SATISFIED or FAILED for true and false branch of each atomic condition.

➤ Fault Localizer

It takes condition coverage report file and compute several predicate properties like **Count of atomic condition, operator score, reachability score, line no, target suspicious score, observed suspicious score, fitness score**. After that it gives rank to each predicate ordered based on decreasing fitness score. In case of tie breaker condition of rank, it applies 3 level of filtration to find rank i.e. it first checks reachability score then operator score and at last atom count.

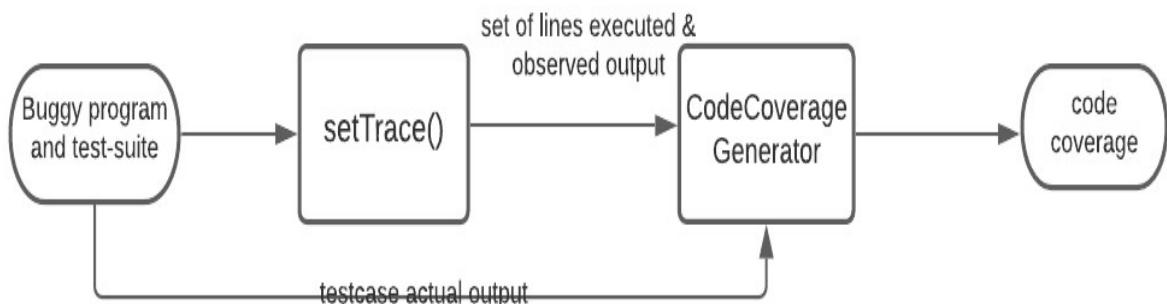
CHAPTER 4

Implementation Details

We have used Python programming language for implementing the work. We've used some of the open-source library functionalities e.g. python's csv, file input, line cache, os, collections, sys, sys.settrace(), exec (), etc.

Fault Localization based on Line/Statement code coverage:

4.1 Line/Statement Code Coverage-based FL



Line Code Coverage ALGORITHM:

INPUTS: inputs.csv file containing buggy program and its test-Suite file name

OUTPUT: Multiple output.txt files containing code-coverage for each buggy program

- For each buggy program, test-suite file in inputs.csv, do
 - Initialize empty output file
- For each testcase in test-suite, do
 1. Call set trace fun from system library which takes trace function as

argument.

2. Call exec fun with buggy program's function and testcase as input argument.
 3. Compare exec function output with actual testcase output.
 4. Append the result(pass/fail) and lines executed by exec function, in output file.
- Store output file.

The Algorithm takes input as set of buggy programs and test-suites to generate multiple output files containing code coverage of each testcase in test-suite of a particular program.

We are using a set trace function from system library to trace the lines being executed while running the program for a particular testcase, and return set of lines being executed.

Expected results: For each program –

```
138 #Trace each buggy program
139 for p in range(len(testFileNames)):
140     #these are file names
141     testFileName=testFileNames[p]
142     testSuiteFileName=testSuiteFileNames[p]
143     funName=os.path.splitext(testFileName)[0]
144     outputFile="output"+str(p+1)+".txt"
145     lines=[]
146     fileLen=fileLength(testFileName,lines)#get file len and append all line object to lines list
147     testCases=fillTestCases(testSuiteFileName)
148     exec(open(testFileName).read())
149
150     suspiciousScore={} #key:lineNo val:suspiciousScore
151     ranks={} #key:lineNo val:rank
152     testToLines = {} ## current testcase contains which set of lines
153     lineToTests={} ## current line is executed in how many testcase.
154     resultOfTestCase={} ## key:testCaseNo val:result
155     totalPassedTestCase=0
156     totalFailedTestCase=0
157
158     fileWriter=open(outputFile,'w')
159     i=0
160     for tup in testCases:
161         i+=1
162         testCase=tup[0]
163         actualOutput=tup[1]
164         testToLines[testCase]=[]
165         sys.settrace(trace)
166         exec('observedOutput=%s (* (testCase))' %funName)
```

```

167     result=""
168     if actualOutput==observedOutput:
169         result="passed"
170         totalPassedTestCase+=1
171     else:
172         result="failed"
173         totalFailedTestCase+=1
174     resultOfTestCase[i]=result;
175     fileWriter.write("TestCase"+str(i)+": "+str(testCase)+"    Actual Output : "+str(actualOutput)+"\n")
176     fileWriter.write("Observed Output : "+str(observedOutput)+"    TestResult : "+str(result)+"\n")
177     fileWriter.write("Lines Covered: "+str(list(dict.fromkeys(testToLines[testCase])))+"\n")
178     fileWriter.write("-----\n")
179

```

- `sys.settrace (trace)` registers the traceback to the Python interpreter. A traceback is basically the information that is returned when an event happens in the code.
 - `def trace (frame, event, arguments):`
 - returns: Reference to the local trace function which then returns reference to itself.
- frame: current stack frame
event: string {'call', 'line', 'return', 'exception'}
arguments: depends on the event type

```

46 def trace(frame, event, arg):
47     code=frame.f_code
48     functionName=code.co_name
49     if event == "line":
50         lineNo = frame.f_lineno
51         if lineNo<=fileLen:
52             testToLines[testCase].append(lineNo)
53             if lineNo in lineToTests.keys():
54                 lineToTests[lineNo].append(i)
55             else:
56                 lineToTests[lineNo]=[i]
57     return trace

```

We implemented the one fault localization technique based on line or statement code coverage such that an automated Python program can be used to identify bugs in a given

python script. In our implementation we computed the suspiciousness score of a line of code's likelihood for bug using the formula given in the paper i.e.

$$suspiciousness(e) = \frac{\frac{failed(e)}{total\ failed}}{\frac{passed(e)}{total\ passed} + \frac{failed(e)}{total\ failed}}$$

A dictionary data structure was used to cater the ranks of each of line of code using "key" as line number and its "value" is the ranking based on suspiciousness score. The suspiciousness score for each line was calculated and sorted such that the line with highest score was given a rank of 1 and should be evaluated first and foremost for bug checking. If that statement is found to not be faulty, then the statement with the next rank is evaluated.

To test which lines were hit by the code, we used Python's settrace() method from sys library which takes three arguments - the stack frame; event; and an argument frame which is the current stack frame. An event simply returns a 'call' or 'line' string etcetera based on which the function decides what to do such as in case of a 'call' string the trace function is invoked and returns a reference to the local trace function to be used in that scope. So whenever any line is hit, it is reported by the tracing function which we could catch in our dictionary to store key value pairs of statements hit against the particular test case that was being run. We preferred settrace method over the python library coverage.py since we could directly use the coverage information given by settrace method inside our script, whereas for coverage it would have been an indirect method to first run on the command line and then feed the results to our script.

A basic arithmetic function then used the values stored in this dictionary data structure to evaluate the suspiciousness score of each line while also taking into account whether that particular test failed or passed i.e., by measuring the number of passed test cases hit and the number of failed test cases hit by each line of code then pushing these as arguments to the suspiciousness function to calculate each line of code's score. The score may vary from 0 to 1 (1 being the most suspicious).

We wrote unit tests against three simple python programs against which our program was evaluated. Python's unit test library was used to test the programs and generate test data.

For example, we wrote a `middle()` function that takes three numbers as input and should return the median of the three values as output. Our program successfully determined line 7 as the most suspicious statement where the bug was expected to be found and present. We also used Python's hypothesis library to compute test cases which fail the program. This was particularly helpful because we were able to derive the test cases which would not meet the assertion and hence saved our time from manually figuring out the cases. We're attaching the `generateFailMid.py` file which generates a failing test case for `mid` function.

The ranking algorithm simply takes the suspiciousness values of all the lines of code and sorts them in descending order based on their suspiciousness score. Also, much effort was spent on automating our tool such that any python program with arbitrary number of arguments and arbitrary number of test cases can be processed by it.

```
22 class Line:
23     def __init__(self,suspiciousScore=0.0, rank=0, text= "", lineNo=0):
24         self.suspiciousScore =suspiciousScore
25         self.rank = rank
26         self.text = text
27         self.lineNo = lineNo
28
29     def setSuspiciousScore(self,suspiciousScore):
30         self.suspiciousScore =suspiciousScore
31
32     def setRank(self, rank):
33         self.rank = rank
34
35     def setText(self, text):
36         self.text = text
37
38     def setLineNo(self,lineNo):
39         self.lineNo = lineNo
40
41     def __repr__(self):
42         return str(self.lineNo)+" "+str(self.suspiciousScore)+" "+str(self.rank)+" "+self.text;
43     def __gt__(self, line2):
44         return self.rank > line2.rank
```



```

91 def getPassPercentage(lineNo, TestCases, totalPassedTestCase, resultOfTestCase) :
92     passed=0
93     for tcNo in TestCases:
94         if resultOfTestCase[tcNo]=="passed":
95             passed+=1
96     return (passed*100)/totalPassedTestCase
97 def getFailPercentage(lineNo, TestCases, totalFailedTestCase, resultOfTestCase) :
98     failed=0
99     for tcNo in TestCases:
100         if resultOfTestCase[tcNo]=="failed":
101             failed+=1
102     return (failed*100)/totalFailedTestCase
103 .
104 def assignSuspiciousScore(suspiciousScore, k, passPercentage, failPercentage) :
105     ss=failPercentage/(failPercentage+passPercentage)
106     suspiciousScore[k]=ss
107
108 def findRanks(ranks, suspiciousScore) :
109     prev=-1
110     rank=0
111     for k in suspiciousScore:
112         if suspiciousScore[k]!=prev:
113             rank+=1
114         prev=suspiciousScore[k]
115     ranks[k]=rank

```

4.2 Condition Code Coverage-based FL

In this technique of Fault localization, we first generate condition code coverage report file by executing CBMC program on our input C program. we then parse that report file to extract information like predicate's line no, atomic conditions, atom count, satisfiability of true and false branch of each atomic condition in each predicate. We computed atom count, operator score, reachability score and target suspicious score and observed suspicious score and finally fitness score of each predicate in given C program.

We formulated the following formula used in our condition coverage-based implementation:

1. **Atom count:** It is the number of atomic conditions in each predicate.

Atom count(p) := Number of atomic conditions in predicate p

e.g.: if (a == 32 && (b == 1 && c == 32))

Here atom count is 3 (Number of atomic conditions are 3).

2. **Operator score:**

$$OperatorScore(p) := 10 * mixOpCount(p) + 5 * (andOpCount(p) + orOpCount(p))$$

$$\bullet mixOpCount := \begin{cases} 1 & \text{if mix operator present,} \\ 0 & \text{otherwise} \end{cases}$$

$$\bullet andOpCount(p) := \text{count of AND operator}$$

$$\bullet orOpCount(p) := \text{count of OR operator}$$

e.g.: if (a == 32 && (b == 1 && c == 32))

$$Operator\ score = (10 * 1) + 5 * (1 + 1) = 20$$

Here, number of AND operator i.e., andOpCount(p) and OR operator i.e., orOpCount(p) in a particular predicate p. Value of mixOpCount will be set to 1 if both type of AND and OR operators are present in p, otherwise will be set to 0. Value 10 and 5 are given as weight to each count respectively. Since, presence of mix operator leads to more vulnerabilities to predicate than either AND or OR, so is given more weight 10.

3. Reachability score:

Let i be the i th atomic condition of predicate p

$$rScore(i) = \begin{cases} 3 & \text{if both true and false branch are satisfied,} \\ 2 & \text{if either branch is satisfied,} \\ 1 & \text{otherwise} \end{cases}$$

$$ReachabilityScore(p) = \sum_{i=1}^{atom\ count(p)} rScore(i)$$

In this formula, we calculated rScore of each atomic condition in each predicate. RScore takes the value 1, 2, 3 as weights based on reachability of each atomic condition. Bugs are often sensitives to branch and conditions and exercising as many paths as possible increases chance of exposing bug in a program. So based on these facts we assign weight 3, if both true and false branch of an atomic condition is covered or satisfied, weight 2, if either of two is satisfied and 1 if no branch satisfied or both branches are dead let's say. We then calculate reachability score of predicate p by summing all rScore values

of their atomic conditions.

e.g. :

```
[errorCheck.coverage.1] file problem10.c line 205 function errorCheck condition `a1933271548 == 3' false:
SATISFIED

[errorCheck.coverage.2] file problem10.c line 205 function errorCheck condition `a1933271548 == 3' true:
SATISFIED

[errorCheck.coverage.3] file problem10.c line 205 function errorCheck condition `a1554992028 == 9' false: FAILED

[errorCheck.coverage.4] file problem10.c line 205 function errorCheck condition `a1554992028 == 9' true:
SATISFIED

[errorCheck.coverage.5] file problem10.c line 205 function errorCheck condition `a1551570219 == 34' false:
FAILED

[errorCheck.coverage.6] file problem10.c line 205 function errorCheck condition `a1551570219 == 34' true:
FAILED

rScore(1) = 3 ; rScore(2) = 2 ; rScore(3) = 1

Reachability Score(p) = 3 + 2 + 1 = 6
```

4. Fitness score:

$$STarget(TargetSuspiciousScore)(p) := \frac{Atom\ count(p)}{3} + \frac{Operator\ Score(p)}{2} + \frac{3 * Atom\ count(p)}{1}$$

$$SObserve(ObserveSuspiciousScore)(p) := \frac{Atom\ count(p)}{3} + \frac{Operator\ Score(p)}{2} + \frac{Reachability\ Score(p)}{1}$$

$$Fitness(p) := \frac{SObserve(p)}{STarget(p)}$$

$$0 \leq Fitness(p) \leq 1$$

```

e.g. : p => if(a1933271548 == 3 || (a1554992028 == 9 && a1551570219 == 34))

[errorCheck.coverage.1] file problem10.c line 205 function errorCheck condition `a1933271548 == 3` false:
SATISFIED

[errorCheck.coverage.2] file problem10.c line 205 function errorCheck condition `a1933271548 == 3` true:
SATISFIED

[errorCheck.coverage.3] file problem10.c line 205 function errorCheck condition `a1554992028 == 9` false: FAILED

[errorCheck.coverage.4] file problem10.c line 205 function errorCheck condition `a1554992028 == 9` true:
SATISFIED

[errorCheck.coverage.5] file problem10.c line 205 function errorCheck condition `a1551570219 == 34` false:
FAILED

[errorCheck.coverage.6] file problem10.c line 205 function errorCheck condition `a1551570219 == 34` true:
FAILED

rScore(1) = 3 ; rScore(2) = 2 ; rScore(3) = 1

Reachability Score(p) = 3 + 2 + 1 = 6

Atom count(p) = 3

Operator Score(p) = 20

STarget(p) = 3/3 + 20/2 + (3^3)/1 = 20

SObserved(p) = 3/3 + 20/2 + 6/1 = 17

Fitness(p) = 0.85

```

In this formula, we calculated Target suspicious score which is maximum suspicious score of a predicate in the worst case i.e., when all branches of each atomic conditions are satisfied. This is the case when a predicate can be maximum vulnerable. Then we calculated Observed suspicious score which is found from actual reachability score of each atomic condition in a predicate. In both above formulae, we have given higher priority to reachability score and medium priority to operator score and lowest priority to atom count by dividing it by 1, 2, and 3 respectively. The very fact is reachability indirectly depend on atom count and bugs are often sensitives to branch and conditions and exercising as many paths as possible increases chance of exposing bug in a program. Also, there is more chance of committing mistakes in case when mix operators are present hence given second priority. We then calculated fitness score of a predicate p as ratio of observed suspicious score and target suspicious score. So, it takes value in the range between 0 and 1. Based on decreasing order of fitness value, we sort all predicates in a given C program and ranked them accordingly. In order to break tie, we used three level of filtration described above.

```

80 def fetchPredicatesInCoverageReport(CBMCFile):
81     predicates=[]
82     with open(CBMCFile,"r") as file:
83         flag=0
84         prevLineNo=-1
85         while True:
86             line1=file.readline().strip()
87             try:
88                 if len(line1)==0:
89                     if flag==0:
90                         flag=1
91                         file.readline()
92                     else:
93                         flag=0
94                         predicates.append(predicate)
95                         break
96                 elif flag==1:
97                     line2=file.readline().strip()
98                     line1Content=line1.split(' ')
99                     line2Content=line2.split(' ')
100                     line1No=int(line1Content[4])
101                     line2No=int(line2Content[4])
102
103                     if line1No!=prevLineNo:
104                         if prevLineNo!=-1:
105                             predicates.append(predicate)
106                             predicate=Predicate(0,0,0,0,0,0,"",line1No,[],0,0,0,0)
107                             prevLineNo=line1No
108
109                             trueBranch=0
110                             falseBranch=0
111                             if line1Content[12]=="SATISFIED":
112                                 trueBranch=1
113                             if line2Content[12]=="SATISFIED":
114                                 falseBranch=1
115                             atom=Atom(line1Content[8]+line1Content[9]+line1Content[10],trueBranch,falseBranch)
116                             predicate.atomCount+=1;
117                             predicate.atoms.append(atom)
118             except ValueError:
119                 break
120     return predicates

```

```

129 def getOperatorScore (andOpCount,orOpCount,mixOpCount):
130     return 10*mixOpCount+5*andOpCount+5*orOpCount
131
132 def findOperatorScoreOfEachPredicate (predicates,lineContent):
133     for predicate in predicates:
134         text=lineContent[predicate.lineNo]
135         predicate.text=text
136         andOpCount=text.count("&&")
137         orOpCount=text.count("||")
138         mixOpCount=1 if (andOpCount>0 and orOpCount>0) else 0
139         predicate.operatorScore=getOperatorScore (andOpCount,orOpCount,mixOpCount)
140
141
142 def findReachabilityScoreOfEachPredicate (predicates):
143     for predicate in predicates:
144         rScore=0
145         for atom in predicate.atoms:
146             if atom.trueBranch==1 and atom.falseBranch==1:
147                 rScore+=3
148             elif atom.trueBranch==1 or atom.falseBranch==1:
149                 rScore+=2
150             else:
151                 rScore+=1
152         predicate.reachabilityScore=rScore
153
154 def getSTarget (atomCount,operatorScore):
155     return atomCount/3.0+operatorScore/2.0+(3*atomCount)/1.0
156
157 def findSTargetOfEachPredicate (predicates):
158     for predicate in predicates:
159         predicate.STarget=getSTarget (predicate.atomCount,predicate.operatorScore)
160
161
162 def getSObserve (atomCount,operatorScore,reachabilityScore):
163     return atomCount/3.0+operatorScore/2.0+reachabilityScore/1.0
164
165 def findSObserveOfEachPredicate (predicates):
166     for predicate in predicates:
167         predicate.SObserve=getSObserve (predicate.atomCount,predicate.operatorScore,predicate.reachabilityScore)
168
169 def getFitness (SObserve,STarget):
170     return SObserve/STarget
171
172 def findFitnessOfEachPredicate (predicates):
173     for predicate in predicates:
174         predicate.fitness=getFitness (predicate.SObserve,predicate.STarget)
175
176 def myCompareFn (predicate1,predicate2):
177     if predicate1.fitness<predicate2.fitness:
178         return -1
179     elif predicate1.fitness>predicate2.fitness:
180         return 1
181     elif predicate1.reachabilityScore<predicate2.reachabilityScore:
182         return -1
183     elif predicate1.reachabilityScore>predicate2.reachabilityScore:
184         return 1
185     elif predicate1.operatorScore<predicate2.operatorScore:
186         return -1
187     elif predicate1.operatorScore>predicate2.operatorScore:
188         return 1
189     elif predicate1.atomCount<predicate2.atomCount:
190         return -1
191     elif predicate1.atomCount>predicate2.atomCount:
192         return 1
193     return 0

```

Chapter 5

Experimental Results and Observations

5.1 Line/Statement Code Coverage-based FL Results

An example:

Input: Buggy program (middle.py), Test-suite (testSuiteMid.txt)
Note: Bug in line number 7

1	def middle(x,y,z):	1 1 1
2	m = z	1
3	if (y<z):	10 7 1000
4	if(x<y):	10
5	m = y	1700 5678 1111
6	elif (x<z):	1700
7	m = y	12 17 19
8	else:	17
9	if(x>y):	28 199 77
10	m = y	77
11	elif (x>z):	199 12 6666
12	m = x	199
13	return m	1666 178 99

178
1222 776 3000
1222
55 55 55
55
277 123 11
123
12 13 14

Intermediate Output: Code-coverage (Output1.txt)

```

TestCase1: (1, 1, 1)   Actual Output : 1
Observed Output : 1   TestResult : passed
Lines Covered: [1, 2, 3, 9, 11, 13]

-----

TestCase2: (10, 7, 1000)   Actual Output : 10
Observed Output : 7   TestResult : failed
Lines Covered: [1, 2, 3, 4, 6, 7, 13]

-----

TestCase3: (1700, 5678, 1111)   Actual Output : 1700
Observed Output : 1700   TestResult : passed
Lines Covered: [1, 2, 3, 9, 11, 12, 13]

-----

TestCase4: (12, 17, 19)   Actual Output : 17
Observed Output : 17   TestResult : passed
Lines Covered: [1, 2, 3, 4, 5, 13]

-----

TestCase5: (28, 199, 77)   Actual Output : 77
Observed Output : 77   TestResult : passed
Lines Covered: [1, 2, 3, 9, 11, 13]

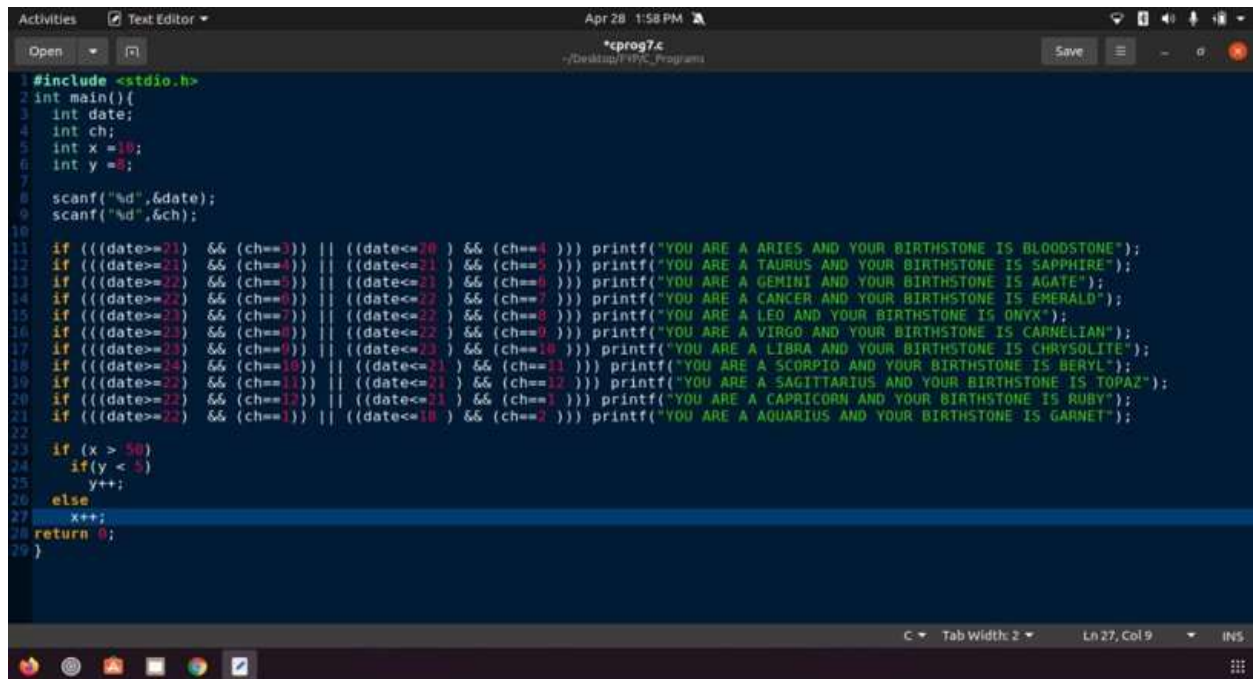
```

Final Output: Set of suspicious statements ranked based on their suspicious-score

LineNo	Suspicious_Score	Rank	Line of code
6	1.0	1	elif (x<z):
7	1.0	1	m = y
4	0.8	2	if(x<y):
1	0.5	3	def middle(x,y,z):
2	0.5	3	m = z
3	0.5	3	if (y<z):
13	0.5	3	return m
5	0.0	4	m = y
8	0.0	4	else:
9	0.0	4	if(x>y):
10	0.0	4	m = y
11	0.0	4	elif (x>z):
12	0.0	4	m = x

5.2 Condition Code Coverage-based FL Results

Sample Input:



```
1 #include <stdio.h>
2 int main(){
3     int date;
4     int ch;
5     int x =10;
6     int y =8;
7
8     scanf("%d",&date);
9     scanf("%d",&ch);
10
11     if (((date>=21) && (ch==3)) || ((date<=20) && (ch==4))) printf("YOU ARE A ARIES AND YOUR BIRTHSTONE IS BLOODSTONE");
12     if (((date>=21) && (ch==4)) || ((date<=21) && (ch==5))) printf("YOU ARE A TAURUS AND YOUR BIRTHSTONE IS SAPPHIRE");
13     if (((date>=22) && (ch==5)) || ((date<=21) && (ch==6))) printf("YOU ARE A GEMINI AND YOUR BIRTHSTONE IS AGATE");
14     if (((date>=22) && (ch==6)) || ((date<=22) && (ch==7))) printf("YOU ARE A CANCER AND YOUR BIRTHSTONE IS EMERALD");
15     if (((date>=23) && (ch==7)) || ((date<=22) && (ch==8))) printf("YOU ARE A LEO AND YOUR BIRTHSTONE IS ONYX");
16     if (((date>=23) && (ch==8)) || ((date<=22) && (ch==9))) printf("YOU ARE A VIRGO AND YOUR BIRTHSTONE IS CARNELIAN");
17     if (((date>=23) && (ch==9)) || ((date<=23) && (ch==10))) printf("YOU ARE A LIBRA AND YOUR BIRTHSTONE IS CHRYSOLITE");
18     if (((date>=24) && (ch==10)) || ((date<=21) && (ch==11))) printf("YOU ARE A SCORPIO AND YOUR BIRTHSTONE IS BERYL");
19     if (((date>=22) && (ch==11)) || ((date<=21) && (ch==12))) printf("YOU ARE A SAGITTARIUS AND YOUR BIRTHSTONE IS TOPAZ");
20     if (((date>=22) && (ch==12)) || ((date<=21) && (ch==1))) printf("YOU ARE A CAPRICORN AND YOUR BIRTHSTONE IS RUBY");
21     if (((date>=22) && (ch==1)) || ((date<=10) && (ch==2))) printf("YOU ARE A AQUARIUS AND YOUR BIRTHSTONE IS GARNET");
22
23     if (x > 50)
24         if(y < 5)
25             y++;
26     else
27         x++;
28     return 0;
29 }
```

Intermediate output: Condition Coverage Report


```

155 ** coverage results:
156 [main.coverage.1] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 13 function main condition 'date >= 21' false: SATISFIED
157 [main.coverage.2] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 13 function main condition 'date >= 21' true: SATISFIED
158 [main.coverage.3] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 13 function main condition 'ch == 4' false: SATISFIED
159 [main.coverage.4] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 13 function main condition 'ch == 4' true: SATISFIED
160 [main.coverage.5] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 13 function main condition 'ch == 3' false: SATISFIED
161 [main.coverage.6] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 13 function main condition 'ch == 3' true: SATISFIED
162 [main.coverage.7] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 13 function main condition 'date <= 20' false: SATISFIED
163 [main.coverage.8] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 13 function main condition 'date <= 20' true: SATISFIED
164 [main.coverage.9] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 15 function main condition 'date >= 21' false: SATISFIED
165 [main.coverage.10] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 15 function main condition 'date >= 21' true: SATISFIED
166 [main.coverage.11] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 15 function main condition 'ch == 5' false: SATISFIED
167 [main.coverage.12] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 15 function main condition 'ch == 5' true: SATISFIED
168 [main.coverage.13] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 15 function main condition 'ch == 4' false: SATISFIED
169 [main.coverage.14] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 15 function main condition 'ch == 4' true: SATISFIED
170 [main.coverage.15] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 15 function main condition 'date <= 21' false: SATISFIED
171 [main.coverage.16] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 15 function main condition 'date <= 21' true: SATISFIED
172 [main.coverage.17] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 17 function main condition 'date >= 22' false: SATISFIED
173 [main.coverage.18] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 17 function main condition 'date >= 22' true: SATISFIED
174 [main.coverage.19] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 17 function main condition 'ch == 6' false: SATISFIED
175 [main.coverage.20] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 17 function main condition 'ch == 6' true: SATISFIED
176 [main.coverage.21] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 17 function main condition 'ch == 5' false: SATISFIED
177 [main.coverage.22] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 17 function main condition 'ch == 5' true: SATISFIED
178 [main.coverage.23] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 17 function main condition 'date <= 21' false: SATISFIED
179 [main.coverage.24] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 17 function main condition 'date <= 21' true: SATISFIED
180 [main.coverage.25] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 19 function main condition 'date >= 22' false: SATISFIED
181 [main.coverage.26] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 19 function main condition 'date >= 22' true: SATISFIED
182 [main.coverage.27] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 19 function main condition 'ch == 7' false: SATISFIED
183 [main.coverage.28] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 19 function main condition 'ch == 7' true: SATISFIED
184 [main.coverage.29] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 19 function main condition 'ch == 6' false: SATISFIED
185 [main.coverage.30] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 19 function main condition 'ch == 6' true: SATISFIED
186 [main.coverage.31] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 19 function main condition 'date <= 22' false: SATISFIED
187 [main.coverage.32] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 19 function main condition 'date <= 22' true: SATISFIED
188 [main.coverage.33] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 21 function main condition 'date >= 23' false: SATISFIED
189 [main.coverage.34] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 21 function main condition 'date >= 23' true: SATISFIED
190 [main.coverage.35] file /home/neeraj/Desktop/FYP/C Programs/cprog7.c line 21 function main condition 'ch == 8' false: SATISFIED

```

Final output: Set of suspicious predicates ranked based on their fitness score

LineNo	Rank	Fitness	S0bserved	STarget	ReachabilityScore	OperatorScore	AtomCount	Line of code	
3	13	1	1.0	25.8333	25.8333	12	25	4	if (((date>=21) && (ch==3)) ((date<=20) && (ch==4)))
4	15	1	1.0	25.8333	25.8333	12	25	4	if (((date>=21) && (ch==4)) ((date<=21) && (ch==5)))
5	17	1	1.0	25.8333	25.8333	12	25	4	if (((date>=22) && (ch==5)) ((date<=21) && (ch==6)))
6	19	1	1.0	25.8333	25.8333	12	25	4	if (((date>=22) && (ch==6)) ((date<=22) && (ch==7)))
7	21	1	1.0	25.8333	25.8333	12	25	4	if (((date>=23) && (ch==7)) ((date<=22) && (ch==8)))
8	23	1	1.0	25.8333	25.8333	12	25	4	if (((date>=23) && (ch==8)) ((date<=22) && (ch==9)))
9	25	1	1.0	25.8333	25.8333	12	25	4	if (((date>=23) && (ch==9)) ((date<=23) && (ch==10)))
10	27	1	1.0	25.8333	25.8333	12	25	4	if (((date>=24) && (ch==10)) ((date<=21) && (ch==11)))
11	29	1	1.0	25.8333	25.8333	12	25	4	if (((date>=22) && (ch==11)) ((date<=21) && (ch==12)))
12	31	1	1.0	25.8333	25.8333	12	25	4	if (((date>=22) && (ch==12)) ((date<=21) && (ch==1)))
13	33	1	1.0	25.8333	25.8333	12	25	4	if (((date>=22) && (ch==1)) ((date<=18) && (ch==2)))
14	36	2	0.7	2.3333	3.3333	2	0	1	if (x > 50)
15	38	3	0.4	1.3333	3.3333	1	0	1	if(y < 5)

CHAPTER 6

Conclusion

Software fault localization, the act of identifying the locations of faults in a program, is widely recognized to be one of the most tedious, time consuming, and expensive yet equally critical activities in program debugging. Due to the increasing scale and complexity of software today, manually locating faults when failures occur is rapidly becoming infeasible, and consequently, there is a strong demand for techniques that can guide software developers to the locations of faults in a program with minimal human intervention. This demand in turn has fueled the proposal and development of a broad spectrum of fault localization techniques, each of which aims to streamline the fault localization process and make it more effective by attacking the problem in a unique way.

We first chose Line Code Coverage as a basis to identify probable faulty statements in the code. Later we analyzed that Line Code Coverage was unable to differentiate between a conditional statement and a simple statement i.e., statement coverage cannot distinguish the code separated by logical operators from the rest of the statement. It gives full statement coverage of the code without exposing the bug in it. So, we decided to choose Condition Code Coverage as basis to implement Fault Localization (FL). The main idea of this coverage is to expose bug by exercising as many paths or conditions through the code as possible since bugs are often sensitive to branches and conditions.

CHAPTER 7

References

1. Afzal, Afsoon, Manish Motwani, Kathryn Stolee, Yuriy Brun, and Claire Le Goues. "SOS Repair: Expressive semantic search for real-world program repair." *IEEE Transactions on Software Engineering* (2019).
<http://www.spideruci.org/papers/jones05.pdf>
2. Jones, James A., and Mary Jean Harrold. "Empirical evaluation of the tarantula automatic fault-localization technique." In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 273-282. 2005.
3. Github Link : <https://github.com/Neeraj921721/FYP.git>