Comprehensive Testing Tool for Automatic Test Suite Generation and Prioritization
5.1 Introduction

Test cases and test data generation are essential aspects of the software testing. Traditionally, test cases are used to perform the testing of developed software. There is a phase in the system development life cycle for testing. It is testing phase where test engineers focus on testing software to know whether it works as per functional requirements of the system. There has been considerable research on test case generation automatically. Towards this end a single goal was used to generate test cases. However, test suite generation with multiple objectives in mind will improve code coverage besides enhancing the quality and lifespan of the Software Under Test (SUT). The generated test suites are not independent, but have inter-dependency. Moreover, some part of the SUT can have high complexity with inter-component dependency. In such cases, it is essential to have dependency structures to understand the dynamics of dependencies. Then it is possible to prioritize test cases. In the literature, an approach was found recently which was manual with respect to discovering dependency structures. Haidry and Miller [76] discovered the dependency structures manually. However, they did not automate the extraction of dependency structures among the test suites that can help in effective prioritization of functional test suites.

The essence of research presented in this chapter is related to the automatic test suite generation with mutation testing, the automatic discovery of dependency structures for test case prioritization and a methodology for test case prioritization. Thus the tool is capable of generating whole test suits besides prioritizing them. The empirical study with 20 case studies revealed this fact.

5.2 Background

It is a very important fact that testing is inevitable in SDLC as it can improve the quality of SUT. There are many testing approaches to uncover bugs in software.
However, unit testing is widely used in the software industry for discovering bugs [36]. Each test case was written for testing purpose should have logic to test a particular functionality of SUT and the test data that is used to test the functionality. Many techniques came into existence for the automatic generation of test cases in the real world. As a matter of fact, it is a challenging job to generate test cases for SUT automatically. It is also not easy to generate test data for test cases. Search based test data generation approaches were found in [23] and [81]. Later, there is change in the approach for ensuring high code coverage. Nevertheless, there is a problem in finding the expected results which is termed generally as the test oracle problem. Thus it is well known that the test oracles are to be taken care of while performing the automatic test case generation.

As found in the literature for many years it was a practice followed to generate test case for each coverage goal and then club all the test cases into a single test suite. This approach has some problems in terms of size of test cases. A single test suite has huge number of test cases. This problem can be solved by using number of coverage goals at a time and generate a test suite which is representative of all coverage goals. This approach has got significant importance in the research circles. Many researchers explored the concept of producing representative test suits. Their concept is also known as whole test suite generation.

5.2.1 Need for the Whole Test Suite Generation

When a single goal is considered for the test case generation, it results in many test cases. Out of them, some test cases can be avoided when multiple test goals are considered and a test suite is generated which is representative in nature. In fact, some branches are infeasible and some branches are considered for coverage. Things can be considered for the whole test suite generation. It is an open problem to address the issues of coverage of all branches in the SUT and ensure that the test cases are optimal. The researcher focused on the usage of Genetic Algorithm (GA) for automatic generation of test suite that can cater to many coverage goals in this chapter. Thus the generated test suite can be used in mutation testing which will help in evaluating test cases. Introduction of bugs in the source code intentional by making small change in the code is known as creation of a mutation. When such mutations are used, it is possible to evaluate the functionality of the test cases. The mutation testing
is also known as fault based testing which is best used to strengthen test cases by validating them.

Generating individual test cases and then combining them into a test suite is the traditional approach. This can be changed in order to have the whole test suite generation. The generated test suite is the representative of many coverage goals. The fitness function of GA is used to have the best test cases and all test cases generated at a time are kept in the whole test suite. The usage of GA could optimize the test suite generation process. The GA technique as usual starts with an initial population and then follows an iterative approach till the test suite is generated.

5.2.2 Test Case Prioritization

A test suite contains a collection of test cases. These test cases are used to test functionalities of SUT and detect faults. When test cases are executed in arbitrary order, there are certain issues. Specifically, one issue is that the order followed may not be able to discover more bugs. The rationale behind this is that some functions in object oriented programming will dependent on other functions. Unless the dependent functions are called prior to other function, it is not possible to discover faults accurately. Therefore, it is essential to prioritize test cases in order to achieve best results. Towards this end, Chapter 4 covered the process of automatic discovery of dependency structures that are used to prioritize test cases. Thus the hypothesis conceived in the literature that is execution of test cases in a particular order can have its impact on the number of faults discovered. This is based on the fact that the functions in SUT can have dependencies and the dependencies are considered while testing such functions.

Recently Haidry and Miller [76] found the utility of prioritization of test cases. Their research focused on discovering dependency structures in SUT manually and measures such as DSP height and DSP volume are used in order to make prioritization decisions. It is possible to uncover more bugs in the software under test when methods are tested in prioritized order. The rate of fault detection is thus improved significantly. The sample dependency structure that shows both open and closed dependencies can be found in Figure 4.3.1 of Chapter 4.
Open dependency refers to the fact that a test case is executed after another test case but may not be immediately. Whereas the closed dependency indicates that a test case needs to be executed immediately after another test case on which it depends. It is possible to have optimal results when open and closed dependencies are utilized. Further, the usage of DSP height and DSP volume increase the fault detection rate when they are used for generating test cases after prioritization.

5.2.3 Dependency Structures

As mentioned earlier, there are direct and indirect dependencies. While computing DSP volume or DSP height, the two kinds of dependencies are considered. Both DSP height and DSP volume are known as graph measures that are used to ensure that the quality of test case prioritization is high. They are used to achieve a best ordering of test cases. The experiments on dependency structures were presented in the previous chapter. The methodology used for the automatic discovery of dependency structures can be found in Chapter 4. It is found that dependency structures when discovered automatically can reduce time and effort. The manual discovery of the same takes more time and effort. Therefore, the research carried out in this thesis is a significant step forward in discovering dependency structures automatically. To the knowledge of the researcher, it is for the first time to have dependency structures discovered automatically.

The applications that are considered for experiments subjected to the discovery of dependency structures using the proposed method explored in Chapter 4. Then the measures computed from the dependency structures are used to prioritize test cases in this way the utility of the dependency structures is proved to be useful. Moreover, dependency structures are obtained using reflection API that is best used to know the runtime details of an application. The SUT is used to know the dependencies and then to be use in specifying best order of execution of test cases.

5.2.4 Findings of Literature Review

Extensive study on literature is made and the same is reviewed. The literature review threw light into different aspects of software testing including automatic generation of test cases, test suits, regression testing, concurrency testing, testing robustness of Java applications using JCrasher, semi-automated unit test case
generation, automatic test suite generation from decision table, automatic unit test generation using Testful tool that makes use of evolutionary algorithms, integration testing with AgitarOne, test case generation using JML-JUnit tool, automatic test case generation using TestGen4J, test-driven software testing approaches, search based testing approaches, bad smell detection techniques, whole test suite generation, test suite prioritization, and discovery of dependency structures. It is understood that there has been significant work on the testing and automated test case generation from the review of literature. However, it is felt that there was little research on automatic discovery of dependency structures and test case prioritization which are important to have high coverage of SUT besides ensuring that the fault detection rate is increased.

5.3 Testing Tool for Automatic Test Suite Generation, Prioritization

A comprehensive tool is built that facilitates automatic testing of object oriented applications. The tool includes features such as automatic test suite generation, automatic discovery of dependency structures and prioritization of test cases. The following sub sections provide insights of the underlying features of the tool. This work is an extension to the author’s previous work [13], [14], and [15].

5.3.1 Automatic Test Suite Generation

GA is one of the candidates that can be used to generate test suites. The author employed GA in order to prepare representative test suite. The generated test suite can reflect high coverage and reduce the number of test cases to be generated. The generated test cases are subjected to mutation testing. Mutation is nothing but a bug intentionally introduced into the source code. Mutations can improve the evaluation of test cases. GA takes randomly generated initial population. It makes use of mutation and crossover operators iteratively in order to achieve this. Therefore, GA is known as an evolutionary algorithm that employs a fitness function to generate optimal solutions. The test suites are generated for select applications that were built using the object oriented programming paradigm. The generated test suite is considered to have many test cases denoted as \( t = \{s_1, s_2, s_3, \ldots, s_n\} \). In the traditional approach test cases are generated as part of the test suite by using individual test goals. But such an approach does not provide optimal test cases as the generated test cases may not be optimal and also the number of test cases or the size of the test cases might be different. The whole
test suite generation that considers a number of test goals at a time can help reduce the number of test cases needed. However, the test cases thus generated ensures full branch coverage.

Fitness function is used in the approach in order to ensure that the test cases are optimally generated by keeping branch coverage with respect to different test branches. The proposed methodology used for test case prioritization is shown in Figure 4.7.1 of Chapter 4. It shows the general framework that guide the generation of test cases. It has modules like generating mutations, chromosome operations, an assertions process and the JUnit Test Case besides test evaluation with mutants. Mutations are used iteratively for chromosome operations. In order to generate test cases JUnit is used. The assertion process is used as part of the test case generation. The test cases are evaluated and some mutants are killed. The final test suite is generated. Java code is subjected to mutations. Different types of Java operators are used to generate mutations. The mutation operations are also used in order to have operations of different kinds. The operators are: replace variable, replace constant, replace comparison operator, replace bitwise operator, delete call, replace arithmetic operator, insert unary operator, and delete filled.

### 5.3.2 Automatic Discovery of Dependency Structures and Test Case Prioritization

The tool implemented supports the whole test suite generation, mutation testing, the automatic discovery of dependency structures and the test case prioritization. The implementation is based on the methodology illustrated in Figure 4.7.1 of Chapter 4. Here the methodology is discussed from the tool implementation point of view. The tool is implemented with provision for selection of SUTs, generating execution traces, method discovery, and test case prioritization. It also has provision for mutation testing.

The execution traces plays a vital role in finding dependency structures from the SUTs. The automatic discovery of dependency structures is implemented using reflection API. The process of discovering methods in the object oriented programming is done using such API which is in java.lang.reflect package. Different classes are used to realize the framework. From SUT the runtime methods can be
discovered and the traces can be used to determine the correct flow of execution. The execution traces are subjected to known the dependencies and the method discovery approach is used to know the methods in Object Oriented Programming. The call processing and method discovery can help in the prioritization of test cases. The dependency structures, and the test suite that has been generated, using GA as explored in Chapter 3, are used for the test case prioritization module of the tool in order to prioritize test cases.

5.4 Test Case Prioritization (TCP) Algorithm

The test case prioritization algorithm explored in Chapter 4 is described here from the tool implementation point of view. The algorithm is implemented using reflection and I/O API provided by Java programming language. It takes generated test suite, SUT and execution traces as input. The output of the algorithm is the prioritized test cases. The first reflection API is used to discover methods from SUT. Then the algorithm makes use of the execution traces and the underlying Meta data in order to know the prioritization of test cases. The test cases in the test suite are reordered to have the execution flow in an orderly fashion. When an arbitrary order is used to execute test cases, the results revealed that less number of bugs is discovered. However, then the test cases are ordered properly using the dependency structures. The execution of the test cases is done based on the dependencies. The tool is capable of discovering more by executing test cases in an orderly fashion.

Prioritization of test cases is the process of arranging the test cases in a logical manner. It is useful because it helps in better utilization of expensive resources like, disk space and defects can be found earlier. Also it reduces the time required to execution of the whole process. Over the past few years many prioritization techniques have been elevated. But prioritization using dependencies in model based testing can give better results as comparison. Testing is a very important part of the software development life cycle, so it must be performed as efficiently as possible.

Various techniques have been provided by the researchers to make testing effective. As test case optimization, test case selection, test case prioritization is used to reduce the time and increase the rate of fault detection. A model based testing has also been introduced in order to detect defects in the earlier stages. It is believed that
the developer will also get benefit by earlier detection of faults, because they can fix them if these defects are known earlier which will ultimately reduce the cost, effort and time.

There are various techniques which can be used to prioritize test cases like:

a) History based: Prioritization of test cases is done by using the prior knowledge.

b) Knowledge based: Test cases are prioritize by extracting the knowledge from human beings.

c) Model based: Test cases are prioritize by taking any system model into account.

In this research work, prioritization of test cases with their dependencies and model based testing mechanisms are also performed. Detecting the defects and providing information about these defects to the developer is done at an early stage. It will also reduce the time, as the faults are detected an early phases.

The prioritization results revealed that there is an improvement in finding faults. The fault detection ratio is improved with automatic test case prioritization. The reason behind this is that the order in which test cases are executed is based on the prioritization measures obtained from dependency structures.

5.5 Tool Implementation Details

A tool is implemented using Java platform. The user interface is created using Swing API taken from javax.swing package. Java.io package is used to have disk I/O operations. The input files are .java files that are part of real world case study projects. The tool supports the underlying algorithms and methodologies to discover dependencies and also the generation of the whole test suite with mutation testing in order to improve the accuracy or quality of test cases. The test case prioritization improves the rate of detection of faults.
As shown in Figure 5.5.1, the proposed system has provision for both mutation generation and viewing mutations. Different operators are used as explained in Chapter 3. The operators are associated with specific lines of the code shown in the left pane of the window.

Figure 5.5.2 – List of chromosomes for given Java class and method
As shown in Figure 5.5.2, it is evident that the AOIS operator is associated with the given piece of code. The chromosome can have different mutations that can be used to improve the quality of test cases.

As shown in Figure 5.5.3, it is evident that the given Java class is analyzed and methods are analyzed for dependencies. Reflection API and the proposed methodology is used to achieve this. The dependency structures and the underlying details of them are used to prioritize test cases. More details are found in the preceding sections in this chapter.

As shown in Figure 5.5.4, it is evident that the given Java class is analyzed and methods are analyzed for dependencies. Reflection API and the proposed
methodology are used to achieve this. The dependency structures and the underlying details of them are used to prioritize test cases.

5.6 Experimental Results

The tool was tested with 20 real time applications. The details of the applications in terms of the number of classes, number of branches and LOC are as shown in Figure 5.6.1. The tool demonstrates the proposed methodology and discovers dependency structures for the given program. The tool can distinguish between open and closed dependencies as described earlier in this chapter. The tool supports both automatic test suite generation and test case prioritization with the automated discovery of dependency structures.

![Figure 5.6.1 – Details of case studies](image)

As shown in, Figure 5.6.1 the case studies considered for experiments include Colt, Commons CLI, Commons Codec, Commons Collections, Commons Math, Commons Primitives, Google Collections, Industrial Case Study, Java Collections, JDom, JGraphT, Joda Time, NanoXML, Numerical Case Study, Java Regular Expressions, String Case Study, GNU Trove, Xmlenc, XML Object Model and Java ZIP Utils.
The experiments are made with the proposed approach and the similar approach EvoSuite besides a single objective approach. Case studies are represented by horizontal axis while the vertical axis represents the count of number of the lines of code, number of branches, and number of classes. These statistics provide enough details of the case studies that are used for testing. Out of them LOC provides important information with respect to the lines of code present in the source code of SUTs. The number of branches can help in analyzing the branch coverage related things in the experiments. The number of classes can help assess the relative size of the SUTs.

TRO has the highest number of lines of code that is 24297. The next application which has highest LOC is CMAA with 23881 and the third highest LOC application is XOM with 23814. The application with highest number of branches XOM with 11794 branches and the application with the least number of branches is NCS with 209 branches. The application with the highest number of classes available is CCO with 421 classes while the application which exhibits the least number of classes is NXM with only 1 class.

![Figure 5.6.2 - Comparison of the proposed approach with other approaches](image)

As shown in, Figure 5.6.2, it is evident that the proposed approach is compared with other approaches. The proposed approach is better than the single objective approach. This is because the generating test suite with multiple objectives
in mind can reduce the number of test cases besides being able to cover all branches.

With respect to average branch coverage Evosuite and the proposed method outperform a single branch approach. The horizontal axis represents the number of infeasible targets while the vertical axis represents average number of the missed infeasible targets. The average number of missed infeasible targets is more when a single goal approach is used. When the Evosuite and Proposed are used, the average number of missed infeasible targets is very less.

As mentioned earlier, the rationale behind this is that the single goal approach will not be able to discover infeasible targets effectively as it does not use multiple coverage goals. When multiple coverage goals are targeted, it is possible to have a cohesive performance on finding the infeasible targets as much as possible. This is the main difference between a single approach and the other approaches like Evosuite and proposed. Infeasible targets cannot have meaningful test cases and corresponding inputs thus wasting time and effort.

Sometimes coverage goals are infeasible as there will be no inputs to make use of the goals in the practical work. Finding infeasible branches can help reduce the time and effort. Therefore, targeting infeasible goals proved to be a futile exercise. Thus it is essential to have the report of infeasible targets so that budget for the testing tasks can be estimated accurately. Improper allocation of the testing budget is a problem with infeasible targets. For each target testing time can be allocated and providing a realistic time, that is, not possible in the absence of facts on the infeasible targets. When the budget is allocated for infeasible targets, it becomes a problem for the budget allocation to the genuine targets that can get inputs and can discover faults when executed with proper prioritization.

When the infeasible targets are identified the allocated testing budget will be fully utilized. It is understood with experiments that infeasible branches when used in the testing process can waste time. Due to the reason of infeasible branches it is possible that the testing tools many not be able to cover 100% coverage for all of the classes in the given SUT. When there is no test that can test coverage goal, such goal is deemed to be infeasible. Sometimes dead code detection can reveal infeasible goals.
Figure 5.6.3 – Average branch coverage comparison for all case studies

As shown in Figure 5.6.3, the average branch coverage of the proposed system and Evosuite perform better than the single branch strategy. The reason behind this strategy is that when test suite is generated with multiple objectives in mind, the generated test cases will be less besides being able to cover all possible branches. In horizontal axis case studies and in vertical axis the average branch coverage is presented.

Figure 5.6.3(a) shows the results of branch coverage comparison for COL, CCL, CCD, CCO and CMA applications. For all of them single branch coverage has shown less coverage while the Evosuite and proposed system shows high performance. With respect to CCL Evosuite exhibited the least performance while the single goal and proposed shows the improved performance in increasing order. Similar kind of trend is with COL application. For CCD and CMA applications both Evosuite and proposed approach shows similar performance. Similar performance is exhibited for a single approach with respect to three applications such as CCD, CCO and CMA.
Figure 5.6.3 (b) shows the results of branch coverage comparison for CPR, GCO, ICO, JCO and JDO applications. For all of them single branch coverage has shown less coverage except CPR and ICO while the Evosuite and proposed approach shows high performance. With respect to JDO Evosuite exhibited least performance, while the single goal and proposed system improves performance. Similar kind of trend is with JCO and GCO applications. For CPR application both Evosuite and single approach shows similar performance while the proposed approach shows high performance. Similar trend is shown by ICO application.

Figure 5.6.3 (c) shows the results of branch coverage comparison for JGR, JTI, NXM and REG applications. For all of them single branch coverage has shown less coverage except JGR and NXM while the Evosuite and proposed approach shows high performance. With respect to JGR Evosuite exhibited least performance, while the single goal and proposed approach shows improved performance in increasing order. For NXM application, both Evosuite and a single approach showed similar performance while the proposed approach shows high performance.

Figure 5.6.3 (d) shows the results of branch coverage comparison for SCS, TRO, XEN, XOMM and ZIP applications. For all of them single branch coverage has shown less coverage except ZIP, XOM and XEN while the Evosuite and proposed approach shows high performance. With respect to ZIP Evosuite exhibited least performance, while the single goal and proposed approach shows improved performance in increasing order. For XEN and XOM applications both Evosuite and single approach shows similar performance while the proposed approach shows high performance. In case of SCS and TRO Evosuite and the proposed approach shows similar performance but higher than that of a single approach.
As shown in Figure 5.6.4, the average test suite length of the proposed system and Evosuite perform better than the single branch strategy. The reason behind this strategy is that when test suite is generated with multiple objectives in mind, the generated test cases will be less besides being able to cover all possible branches. In horizontal axis case studies and in vertical axis the average length is presented.

Figure 5.6.4 (a) shows the results of the average test suite length comparison for COL, CCL, CCD, CCO and CMA applications. For all of them single approach has shown less average length while the Evosuite and proposed approach shows high average length. With respect to CMA proposed approach and Evosuite exhibited similar performance. For CMA application both Evosuite and proposed approach shows similar performance while single approach showed less performance. In case of CCD Evosuite shows highest average length.

Figure 5.6.4 (b) shows the results of average test suite length for CPR, GCO, ICO, JCO and JDO applications. For all of them single approach has shown less
average length while the Evosuite and proposed approach shows high average length except GCO and CPR. With respect to CMA proposed approach and Evosuite exhibited similar performance. Evosuite shows highest average length for ICO, JCO and JDO. The average test suite length exhibited by the proposed approach is higher than single approach for all applications. For CPR the average test suite length exhibited by proposed approach is highest.

Figure 5.6.4 (c) shows the results of average test suite length for JGR, JTI and NXM applications. For all of them single approach has shown high average length while the Evosuite and proposed approach shown less average length. With respect to JGR the Evosuite approach exhibited high average length when compared with that of proposed. With respect to JTI and NXM proposed the approach exhibited higher average length when compared with that of Evosuite.

Figure 5.6.4 (d) shows the results of average test suite length for SCS, TRO, XEN and XOM applications. For all of them, a single approach has shown less average length while the Evosuite and proposed approach shown high average length. With respect to TRO and XEN proposed approach and Evosuite exhibited similar performance. Evosuite shows the highest average length for SCS.

![Figure 5.6.5 - Case study applications used for test case prioritization](image-url)
As shown in Figure 5.6.5, case study applications and their statistics in terms of lines of code, number of functions and number of dependencies are presented. The horizontal axis represents the case study SUTs while the vertical axis represents count indicating lines of code, number of functions and number of dependencies. The lines of code refer to the number of lines of code in SUT. The number of functions refers to the count of functions in the given SUT and the number of dependencies refers to the dependencies exist in the given SUT. Case studies were selected with diversity in lines of code, functions and dependencies. Thus it is possible to have conclusions with respect to test case prioritization. These case studies are used to apply the proposed test case prioritization approach with the automated discovery of dependencies. The results revealed that the prioritization can help increase the number of faults detected. This is the ultimate aim of the proposed tool in this chapter.

As shown in Figure 5.6.6, it is evident that the percentage of test suites executed is presented in horizontal axis while the vertical axis presents percentage of faults detected. The percentage of faults detected is gradually increased as the test cases in the test suite are executed. As the percentage of execution of test cases is increases that is reflected in the increase in number of the faults detection. In either cases presented in Figure 5.6.6 (a) and Figure 5.6.6 (b) the trend remains same. However, the fault detection rate is increased when the test cases are executed in the prioritized order. The rate of fault detection is high in other words with respect to the
test cases executed in a pre-defined order. When the test cases are not executed in the prioritized order (executed sequentially like \( t_1, t_2, t_3 \) and \( t_4 \)) the fault detection rate is comparatively less. This is very important observation in this thesis, and it is linked to the hypothesis claim.

**Test case prioritization can have its impact on the fault detection.** This is the important hypothesis that has been tested with the proposed tool and underlying methodologies. As the results revealed, the prioritized test cases were able to detect more faults. Thus the hypothesis has been tested and proved to be positive. The output is reflected in Figure 5.6.6 where it is visible that the need for ordering test cases. In other words, it shows how important to run test cases in a prioritized order. The prioritization order can help in improving the discovery of more faults. This is observed when test cases are prioritized and tested with the tool implemented. For instance the test cases named \( t_1, t_2, t_3 \), and \( t_4 \) are executed in the same order. When the same set of test cases with different order such as \( t_4, t_1, t_2, \) and \( t_3 \) executed, it exhibits less number of faults. Thus it is proved that test case prioritization can help improve the rate of fault detection.

### 5.7 Hypothesis Testing

**Test case prioritization have its impact on the fault detection:** This is the important hypothesis that has been tested with the proposed tool and underlying methodologies. As the results revealed, the prioritized test cases were able to detect more faults. Thus the hypothesis has been tested and proved to be positive. Test case prioritization has its utility in improving the rate of fault detection in SUT. As test suite contains many test cases, the order of their execution plays an important role in increasing the rate of fault detection which can provide early feedback to the development team so as to help them to improve the quality of the software. Therefore, it is very useful to prioritize test cases that will lead to the increase in the rate of fault detection.

Experiments are made with the 20 case studies considered. Specific results of two approaches used for test case prioritization are compared. The comparison is made with the case study SUT, namely CMA in terms the number of faults detected. The numbers of faults detected are compared with the automatic discovery of
dependency structures in the proposed system and the manual identification of dependency structures. The empirical results revealed that the proposed methodology, to discover dependency structures automatically for test case prioritization, shows comparable performance when compared with dependency structures are manually identified and prioritization performed in [8]. The increase in the fault detection is due to the fact that the test case prioritization can help in running test cases in proper order that will reflect the increase in fault detection ratio. The evidence of the fact that the test case prioritization can improve the fault detection ratio is found in Figure 4.8.1 of previous chapter.

5.8 Evaluation of Objectives

There are specific research findings that are mapped to the objectives of the study.

- The first objective is to explore algorithms that generate test suites which are representative of all coverage goals besides reducing the size of test cases.
- The second objective is to propose and implement a methodology to discover dependencies from SUT automatically for enhancing the power of prioritization of test cases.
- The third objective is to propose a methodology for test case prioritization based on the dependencies that have been discovered.

The research was made based on three objectives. The research results are discussed in the subsequent sections of this chapter reveal that the whole test suite generation and mutation testing achieved significant and comparable performance. In the same fashion, the automatic discovery of dependencies with different case studies revealed that the proposed methodology was sound in producing both open and closed dependencies that are the basis for test case prioritization.

Finally, the test case prioritization has been made with the 20 case studies by exploiting the automatic discovery of dependency structures. The results revealed that the prioritization of test cases has improved the number of faults detected. The fault detection ration is significantly improved with the proposed system. The empirical results obtained from the tool built show ample evidence of the research findings.
Moreover, they are evaluated with different case studies and the ground truths provided by human experts.

5.9 Summary

In this chapter the concept of generation of representative test suite which ensures complete code coverage was studied. Coverage criteria play an important role in automatic test case generation. Traditional approaches targeted one particular coverage goal. From the recent experiments in software testing it is evident that the optimization of whole test suite generation is far better than the traditional method of targeting one coverage goal. Genetic algorithms have been applied successfully to generate unit tests for testing object oriented software. GA is one of the search based algorithms widely used to generate test cases. In this chapter GA for generating representative test suites and mutation testing is applied. A tool is built to demonstrate the proof of concept. Mutation testing became easy with generation of test suite that covers all goals.

Test case prioritization has its utility in improving the rate of fault detection in SUT. As test suite contains many test cases, the order of their execution plays an important role in increasing the rate of fault detection which can provide early feedback to development team so as to help them to improve the quality of the software. Therefore it is very useful to prioritize test cases that will lead to the increase in the rate of fault detection.

In this work a novel mechanism is proposed to discover dependency structures from SUT automatically and use them for prioritization of test cases. This work is very closer to that of Haidry and Miller. But they did not automate the discovery of dependency structures. Dependencies are of two types namely direct and indirect. Both types are considered in this work. This methodology was implemented as a test case prioritization tool and performed an empirical study on both real and non-trivial Java programs. Integration of the whole test suite generation and test case prioritization into a single application was done. The empirical results reveal that the automatic discovery of dependency structures can help in complete automation of test case prioritization.