Chapter-2

Background and Related work
BACKGROUND AND RELATED WORK

2.1 Introduction

This chapter reviews the literature on prior works on software testing. It throws light on various aspects of software testing, including the automated generation of test scripts from formal test specifications, class level regression testing, detection of concurrent execution, semi-automated generation of test cases, and the automatic generation of test suites from decision tables and so on. It also reviews tools pertaining to software testing such as TestFul, JAMTester, JML-JUnit tool, JUB and TestGen4J. There were many recent developments in software testing methodologies like test-driven testing approaches, search based testing approaches, saving effort through bad smell detection, the whole test suite generation, and test suite prioritization which are reviewed in this chapter.

2.2 The Automatic Generation of Test Scripts from Formal Test Specifications

Marc [55] studied formal test specifications and TSL (Test Specification Language) processor to build scripts which are self-contained and executable. He proposed a system that was used to test even commercial software with the help of the TSL processor in the production environment. The author used an approach known as Category-Partition Method (CPM) for analyzing various functionalities of SUT. In this methodology, test specifications are created and then compiled in order to obtain scripts used for testing SUT. The test environments depend on test specifications. Such environments are used to run test cases with sample values assigned to variables. The robustness of these test cases is also established. The partition method used for analyzing available test specifications yielded positive results. The TSL can also be integrated with formal test specifications. The tests are generated with the help of the partition method taking inputs and environment conditions.

2.3 Class-Level Regression Testing

Rothermel et al and Hong et al [28] proposed two testing techniques, namely white-box and black-box testing respectively. When these techniques are used
together may help in cross-level regression testing. Regression testing is one of the
testing methods where it is possible to know whether changes are made as per
functional requirements without having the impact on other parts of SUT. The
combination of those approaches can have synergic effects in the results. The
integrated approach that combines white-box and black-box testing techniques,
building blocks, seamlessly can improve performance. According to Chen et al [33]
the effective integration of the two testing approaches can provide tests that are
required for class level regression testing. It was tested and demonstrable with their
generator TACCLE (Testing At the Class and Cluster Levels).

2.4 Dynamic Detection of Concurrent Execution of Object-Oriented Software

TACCLE depicted in the past segment is skilled of the testing article arranged
items; it needs in testing in the technique level and does not handle circumstances
which are non-deterministic and simultaneous. Huo Yan Chen et al [34] proposed this
approach, which can be utilized to test protest situated programming frameworks at
the strategy level with joining capacity to handle both non-deterministic and
simultaneous situations. This is tuned in to Java's prevalent multithreading abilities.
The proposed approach utilizes a couple of simultaneous operations to test
concurrency on Object Oriented System. This plan powerfully distinguishes and test
simultaneousness. The scheme is based in specifications known as OBJSA-net
(Object Superposed Automata) and CLOWN (CLass Orientation With Nets). These
determinations are utilized as part of enormous ventures, for example, ENEL
(National Entity for Electricity), an Italian power company. This plan can distinguish
stops if there should be an occurrence of Java's multithreaded applications. This
scheme can detect deadlocks in case of Java's multithreaded applications. For this
reason it develops LCSD (Lock-Calling-Suspend Diagrams) and CHD (Calling
Hierarchy Diagrams) from the projects being tested. This technique depends on white
box testing, static in nature and can't discover find deadlocks in case of dynamic
binding. It relies on upon replay techniques with a specific end goal to run test cases
created for the proposed scheme. It works for Java as well as projects composed any
language that supports concurrency.
2.5 JCRASHER: An Automatic Robustness Tester for JAVA

Csallner [18] indicated the heartiness of Java programming can be tried with help of JCrasher. The tool can test at Java classes, their constructors, methods, type of data and also it test the methods with sample data generated randomly. It can identify bugs in expressly bringing on the class under the test to a crash which brings about an undeclared exemption at runtime. Finish mechanization is the upside of this device regardless of its confinement in producing restricted test information haphazardly. Assessment of JCrasher can help in determining its upsides and downsides and its contributions for future executions of experiment generators. It has been planned with an accentuation and its components relating to adaptability are viewed as in view of the life cycle of projects. For testing day-to-day developments with regression testing, JCrasher can be utilized. JCrasher accompanies some exceptional components that are not with other experiment generators.

JCrasher considers techniques to be tested and their parameter-spaces before and builds test cases arbitrarily. For choosing whether a special case ought to be dealt with as infringement of code's preconditions in JCrasher inputs or a program bug, JCrasher characterizes heuristics. This has importance as Java has no arrangement for technique preconditions. Tests in JCrashers keep running on "Clean Slate" each time despite the fact that some different past tests change static factors. This is finished in JCrasher in two ways. One way is to utilize distinctive class loader to persuade the new code to be tested. Another way is to reconstruct the byte code of the class to be tested on the fly. The test files created in JCrasher are compatible with JUnit. This can be incorporated with IDE resembles Eclipse and the tests can be future coordinated with regression test suites. To repeat one of the kind elements of JCrasher: the ability to re-initialize static variables to have clean slate in testing functionalities and a test that is useful can be integrated in regression test suite permanently. These two features are not with its predecessors.

2.6 A Semi-Automatic Generator for Unit Testing Code Files Based on JUNIT

Huo Yan Chen [33] said the Object-Oriented software testing is more complex and troublesome when contrasted and procedure-oriented software. Chen [17]
recommended that unlike alternate tools that spotlight at the class level or cluster level, it concentrates on strategy level unit testing. XP (Extreme Programming) where the test-driven classes are defined, become popular in recent times. The test case code is frequently made even before the advancement of the source code. Its theory is "code a bit, test a bit, code a bit, and test a little". The experiment era with programmed and self-loader highlights has turned out to be critical. For making and running unit tests JUnit can be utilized. The experiments are Java classes with the practically comparative sum as that of the source class. So as to create classes in JUnit, the class must resemble testXXX while the technique is testMethodName() for a given class.

This tool generates test cases with the help of reflection and also analyzing JUnit rules with the semi-automated feature. The author felt that the tool generates the test cases semi-automatically. By using JUnit and reflection technique, it proposes a unit testing framework. As it is semi-automatic some part such as testing data and assertions might be written by testers manually. However, it is able to reduce the burden on testers.

As per Chen [17] the accompanying are the means took after for Cheng-Hui for the real semi-automated test case generation. Take a folder with Java source files as input and obtain all .java files, excluding interface, into a vector for further processing. Each element in vector should have a source code of java class. Then Process all the elements of vector in order to generate test case class definitions for each class in the source code. The main() method, which is an entry point to run all the tests, is defined finally. The code makes use of run() method of TestRunner class of JUnit API. It is as shown below.

```java
public class TestEntry
{
    public static void main(String args[])
    {
        JUnit.textui.TestRunner.run(AllUnitTest.suite());
    }
}
```
The main drawback of this approach is, for the given java source code, it generates unit test cases partially and the remaining test cases are written manually. Hence it is not fully automated test case generation. It is a semi-automated test case generation tool which needs further improvement.

2.7 The Automatic Generation of Test Suites from Decision Table

Mamata Sharma et al [54], proposed a novel framework which takes decision trees and generates the automated test suites. Different procedures and tools were utilized to demonstrate the feasibility and practicality of decision trees. Their framework has been tested for Object-Oriented systems with the JUnit. Here, the authors used decision tables for the generating test cases automatically. Testing procedures like Boundary Value Testing and Equivalence Class Testing are alternate techniques from which this choice decision table-based testing advanced. The accompanying figure shows the flow in the form of activity diagram.

![Figure 2.7.1- Shows activities present in decision table-based testing technique](image)

As shown in Fig. 2.7.1, after generating decision tree, its logic takes into a spreadsheet. Then the conditions and activities are extracted from the decision tree. Afterwards, the test data is generated and after that the generation of test cases and test suites are occurred. Presently the test suites keep running in utilizing the JUnit as part of the request to view the test comes about. From the generated equivalence classes and their variable’s data type, test data is generated. The output is sent to the
XML file. The XML record contains rules required. It is parsed utilizing XML parser, for example, DOM/SAX.

2.8 TESTFUL: Automatic Unit-Test Generation for JAVA Classes

Luciano Baresi and Matteo Miraz exhibited an eclipse plug-in that could produce tests automatically for Java classes. They named the module, "TestFul" [19]. Search- based testing technique [56] is the reason for this module. This testing strategy is both at class and method levels. It is another approach for the semi-automated unit test case generation. The features are as given below

- A semi-automatic approach which gets information from the client and perform its job effectively
- It makes utilization of a novel algorithm known as a hybrid "multi-objective evolutionary algorithm"
- TestFul's test rendering depends on the operations indicated below.

  ➔ Assigning a primitive value such as byte, int, char, short, long, float, double and Boolean. (assigning)

  ➔ Creating an object for invoking constructor and passing required arguments in the form of variables and then storing the object in a variable. (object creation)

  ➔ Invoking business logic methods and passing relevant arguments. If methods return, the value is captured. (the method invocation)

- The TestFul's approach is to make utilization of an evolutionary algorithm and a procedure known as hill climb and search based technique for getting optimal test cases. As it relies on both class level and method level, it makes testing robust [57].
2.9 AGITARONE

AGITARONE is a comprehensive tool for the integrated unit testing Java applications. Automatic JUnit test case generation is the fundamental component of AGITARONE. The consequence of this tool can maintain a strategic distance from manually written tests or supplement them in a big way. A unique tool, name "Agitator" is part of "AGITARONE". The AGITARONE is equipped for the testing Java program with an extensive variety of conceivable information sources and watches its conduct. The results of AGITARONE are as a short rundown of perceptions that help the engineer for the future survey. Basic code errors and code rules infringement are likewise detected in Agitator. Such guidelines are modified to suit the project in hand and customer requirements. The special component of AGITARONE is that it can rapidly condense the test results in the form of a dashboard (Agitar Technologies). This will permit the user to comprehend the outcomes just on a look. The outcomes or project outline contains over time, coverage, usage, complexity, etc. Identification of ownership of classes is also done in AgitarOne.

Aldrich [4] indicated, it has the accompanying advantages. When it contrasted with human techniques that can cover un-craved conduct, the tool can do it all the more effectively. The aftereffects of exploratory outcomes uncovered that the tool has recognized legitimate valid observations and assertions from that of invalid ones. It tosses light into recognizable proof of code, conduct and contrast and its proposed conduct. It can diminish the burden of software maintenance up to 50%. It additionally has the accompanying restrictions. The tool is not equipped for producing 100% test code for all classes and methods. So it is not proficient generating input information for a portion of strategies.

2.10 JAM Tester

As per Witry [83] it is a framework which incorporates two tools. The primary tool is implied for instructors and consequently known as "teacher tool" while the second tool is implied for students and thus known as a student tool. The teacher tool serves to instructors for reviewing while the student tool serves to understudies in testing their own particular code. The teacher tool helps in totally testing techniques
for a Java class of each student. In few moments evaluating can be given to students with the assistance of the teacher tool. It additionally incorporates an element known as JPlag that consequently checks an unoriginality of student's code. A student can without much of a stretch produce test case for his venture that is set of Java classes with the assistance of the student. The code scope highlight of the student tool is that the student can see the part of the code in his program that can't be practiced with the test suite made on the student tool. Besides, the student tool is made in a manner that can be fused with the student's most esteemed IDEs like JCreator and BlueJ. Strangely the tool additionally has its own particular fundamental IDE on the off chance that you would prefer not to utilize whatever other IDE.

2.11 JML-JUnit tool

Patrick [61] indicated that outline by-differentiation based files are utilized as a part of the tool named "JML-JUnit". The files are indicated source as post and preconditions. Generation of the JUnit test code is the reason for those ideas. JML (Java Modeling Language) is utilized to indicate the legally binding data required for the tool. Inputs given to this tool are qualities for all parameters of every method and relating object occurrences. Afterwards, test cases are executed for the tool for all items, parameters, and techniques. However, as Tan and Edwards discovered, this tool has a disadvantage as it is not compelling in the case of fault detection with respect to mutation score.

The JUB (JUnit test case Builder)

There is another computerized case era tool, name JUnit Test Case Builder (JUB). It makes utilization of while-box testing. Object oriented aspects like overloaded methods and constructors are considered in this tool. While producing test strategies, the source code, it places exception handling and method signatures correctly.

2.12 TestGen4J

Another tool, the name "TestGen4J" takes Java records as information and automatically creates test cases. Its fundamental concentration is to have boundary value testing. It makes utilization of arguments supplied to methods and applies
boundary value testing. It utilizes a setup record (metadata) where boundary condition is specified for different data types. This XML record is a piece of its rules engine. JTestCse is utilized to separate test information from test code. The created test code is put in the hierarchical fashion. One test suite is created for the whole code. This test suite is responsible for summoning individual class’ test suites. Test Methods identified with a class is kept in test suite of an individual class. According to the progression of the test code, the test information is additionally put progressively in XML arrange for each strategy. The XML record contains the information required for all unit test instances of all techniques. For repeating through experiments relating to every technique JTestCse is utilized (TestGen4J).

The semi-automated test case generator device is not completely automated and the TestGen4J has executed a testing system known as boundary value analysis and no endeavor is made in classification strategy. This is the inspiration driving the proposed tool. The constraints of both the tools are overcome in the proposed tool as it is completely automated. However, certain presumptions are made and the execution of class partitioning technique. The proposed tool additionally makes utilization of Java beans naming conventions as the fields can be populated and recovered with the assistance of “accessors” (getters) and “mutators” (setters). As portrayed before, the automatic test case generation needs reflection API gave for Java, Java Beans naming the conventions and parameter partitioning (Category partitioning and boundary value analysis) as parameters are to be created with the assistance of test information for testing unit test cases with each single conceivable esteem including limit conditions. The next chapter is given for the devoted for the description of needed aspects of the end goal of for all intents and purposes actualizing a tool for the automated generation of test cases.

2.13 Cyclic Dependencies and Quality of Software

Ostrand [58] focused on the concept of cyclic dependencies in a software product which affects the quality of SUT. Experiments were made with help of the OO metrics to unearth cyclic dependencies and their relation with error-proneness of SUT. The empirical results revealed that more defects are found when the degree of cyclic components is more in the SUT. To overcome this problem it is essential to
refractor the components for better quality of software. There was another observation in the research that the complexity of software can possibly add to the error-proneness. When cyclic groups are studied, high degrees of defective classes are found. For example, the study of Apache Camel revealed the fact that there are 82% defects at the package level and 90% at the class level.

2.14 Test-Driven Software Testing Approaches

Carlos Pacheco et al [16] focused on the test driven software testing approaches. The impact of Test Driven Development (TDD) is known for the quality of code and productivity. The size of a task given to a developer has its impact on the quality of software when TDD approach is followed. Researchers revealed that TDD has the positive influence on the quality of SUT. To prove this meta-analysis techniques were employed. Another side of the coin is that the productivity of TDD could not be concluded and it needs further research.

Other researchers, Beck Kent [10] proposed two methods for testing software. They are known as code inspection and TDD. The former is more useful for reducing defects in the SUT. Though it is proved to be expensive, it could reduce defects. The latter has no significant impact on the productivity. Therefore, as far as the two approaches are concerned, code inspection is more effective when compared with TDD.

2.15 Search Based Testing Approaches

Meta-heuristic algorithms were explored in [3] for search based testing. The well-known tool AUSTIN (Augmented Search-Based Testing) incorporated those algorithms. This tool is responsible for generating structural data in order to cover every possible branch in SUT. Search-based approaches coupled with pseudo-symmetric execution was employed to achieve the results. The algorithm used for this named “Hill Climbing” which is a local search algorithm. It is also known as an algorithm for the neighborhood search where search space randomly was chosen. The tool was built for performing testing on C applications. Thus it is capable of filling the gap between industry and research pertaining to search based testing for applications built in C language. Many search based methods are compared in [70]. They are
known as Random Search (RS), Genetic Algorithm (GA), Hill Climbing (HC) and Evolutionary Algorithm (EA). Under various test sequences, these algorithms were tested for their performance. The experimental results showed that their performance was based on SUT. However, a proven fact is that the length of test sequence can influence the degree of structural coverage of the software being tested. Automated test data generation was explored in [52] along with Search-Based Software Testing (SBST). Fitness function is used in evolutionary algorithms that will help in guiding the convergence with respect to test cases among all the searching algorithm.

2.16 Saving Effort through Bad Smell Detection

SUT has potential problems that are known as the bad smell. Liu et al [32] explored the concept of bad smell detection of software testing for solving potential issues in source code. In order to detect bad smells, these researchers proposed a methodology with a sequence of steps for achieving intended results. The bad smells which are identified and named as feature envy, a long method, long parameter list, useless class, large class, useless method, useless field, duplicate code and so on. Pair wise resolution sequences are built in order to know the latent and potential relationships among bad smells. Thus the pair wise resolution sequences help in identifying relationships and dependencies to have accurate assessment of bad smells that commonly occur in source code. To know, how the bad smell detection can be employed in software testing. Thus it is possible to generate a report that has comprehensive information about bad smells which will help developers to take corrective steps.

2.17 Whole Test Suite Generation

Fraser and Arcuri [26] studied the test suite generation and issues with a single goal or multiple goal coverage. They built a new approach using GA for coverage of all goals and also ensure that the size of test suits is small. They built a tool known as EVOSUITE for the whole test suite generation which will be representative of all possible branches in SUT. \( T = \{ t_1, t_2, t_3, \ldots \} \) is considered as a solution for test suite where \( t_1, t_2, \) and \( t_3 \) are programs. In similar fashion, a test case is considered as \( t = \{ s_1, s_2, s_3, \ldots \} \). The length of the test suite is nothing but the sum of the length of every test case which is part of the test suite. The same is denoted as follows.
In the equation, the letter ‘t’ represents four possible types of statements. They are assignment statement, the field statement, the constructor statement and a primitive statement. Primitives include Boolean, string, enumeration and numeric values. Constructor statements are used to construct objects in OOP. The field statements are nothing but the member variables in a class that is also known as instance variables. The assignment statements are the instructions that actually assign values to variables. Branch coverage is given importance in building fitness function that acts as the criterion for the test. The fitness value is best used to measure the closeness of the test suite which can provide complete structural coverage. Their solution also has a mechanism for bloat control which refers the system from generating longer test cases. The genetic operators used for achieving the whole test suite generation are test suite mutation and test suite crossover [26].

The first generation is initialized with random test cases. This is essential to start with GA. The tool uses JUnit in order to produce possible test suites for Java source code given to it. Another concept named byte code instrumentation is used for obtaining required additional information. At the same time, security issues are dealt with using the security manager. Single branch strategy is compared with the whole test suite generation [20]. The experimental results showed that high performance is registered with the whole test suite generation when compared to the results against a single branch strategy. Many conclusions were made in their experiments. They include high coverage is possible with the whole test suite generation; GA kind of evolutionary algorithms yielded better results; the proposed solution is better even when path coverage is compared with other tools like CUTE [48] and DART [59].

2.18 Test Suite Prioritization

Test suites can help to detect faults in SUT. Provided this goal is achieved, there are many issues with it. For instance, test suites when executed in particular sequence can provide chances to unearth more faults. It does mean that test suite prioritization can be used to optimize testing results or to uncover more hidden faults. In order to achieve this, it is possible to find the dependency structure that can be used

\[ \text{length}(T) = \sum_{t \in T} l_t. \]
to priorities test-suites. The functional test suites, when subjected to prioritization, can give effective test results that can help developers to rectify problems in SUT.

Haidry and Miller [76] focused on the process of test suite prioritization. They used a hypothesis "dependencies among test cases can have their influence on the rate of fault detection". Thus the test case prioritization is given importance. It is a process of ensuring that the test cases are executed in proper sequence in order to achieve the high rate of fault detection. The rate of fault detection is measured using the number of faults detected. As some tests should occur before other tests, it is essential to prioritize test cases so as to achieve optimal results [76]. Sample dependency structure can be visualized in Figure 2.18.1.

![Figure 2.18.1 - Sample dependency structure](image)

As shown in Figure 2.18.1, it is evident that the root nodes are independent of other nodes and they do not have dependencies. Dependencies are of two types namely direct and indirect. For example, in Figure.2.18.1 D6 is a direct dependent of D3 and indirectly dependent on I1. Yet in another classification, dependency is of two types namely open dependency and closed dependency. Open dependency is the fact that a test case is executed before another one but need not be necessarily just immediately before the test case. Closed dependency is opposite to it whereas a test case needs to be executed immediately before the other test case. The combinations of open and closed dependencies are also possible for optimal results. Two measures are used to measure dependencies, which are known as DSP height and DSP volume. Dependency Structure Prioritization volume refers to the count of dependencies whereas the DSP height indicates the depth in dependency levels. While computing DSP volume Direct and indirect dependencies are considered. On the other hand, the height of all test paths is considered for computing DSP height. Two graph measures
are used for the best ordering of test cases for optimal results. The above Experiments are made with open dependencies and closed dependencies. Many real-time projects were considered for the above experiments. Bash is recorded to have the highest dependencies and CRM1 and CRM2 recorded the lowest out of them. Many SUTs were tested with prioritization of test cases. The experiments are useful to know the fault detection rate when dependency structures used for prioritization. A measure used in [63] is known as the Average Percentage of Faults Detected (APFD) for fault detection. The APFD value is more in the rate of detection of faults in SUT. All SUTs are tested with APFD measures under open and closed dependencies. Many DSP prioritization methods were considered and some other methods that do not use DSP measures can also be used in the process. The experimental results showed that DSP prioritization methods achieved higher results whereas non-DSP prioritization methods could not achieve the high rate of fault detection. The empirical results proved the fact that DSP measures were able to increase the rate of fault detection for any given SUT. As explored in [76] there are many test case prioritization techniques like the model-based [20], history-based [23] and knowledge-based [88]. The model-based prioritization technique uses the model of the system, history-based uses, past execution cycles, and knowledge-based uses for the purpose of task test case prioritization human knowledge.

2.19 Existing Works on Test Case Prioritization

According to Elbaum et al [71] considerable research is done in the last ten years on the test case prioritization. Different techniques came into existence. Test case prioritization is an important technique that is essential in order to increase fault rate detection. It is true with respect to regression testing as well. Ordering test cases can help to improve the quality of testing. The highest priority test cases are executed first and then the lowest priority based on certain criterion. Rothermel et al [24] stressed nine prioritization techniques out of them; four were related to coverage of branches or statements, whereas the remaining is related to the ability to detect faults. They denied giving any importance to prioritization, additional statement coverage prioritization, and total statement coverage prioritization, prioritization based on fault-exposing-potential (FEP), total FEP prioritization, additional branch coverage prioritization, total branch coverage prioritization, optimal prioritization and random...
prioritization. The empirical study on these techniques proved that prioritization techniques enhanced the rate of fault detection.

Elbaum et al [74] proposed many prioritization techniques in a space program for the increasing fault discovery rate. The methods employed include additional fault index prioritization, additional function coverage prioritization, additional statement coverage prioritization, and the random ordering. Both faults and cost severity are considered for the prioritization approach. The study made with artifacts of the Mozilla project revealed that the test case prioritization techniques were useful. They also compared to the severity of discovered faults. The experiments were made using both linear and exponential models. In the linear model, the severity increases when the severity of discovered faults are increased. In the exponential model, there is an exponential growth of severity.

Kim and Porter [43] provided an important insight on test case prioritization. According to them, there are some situations where execution of the complete test suite is not possible due to some constraints. In such cases, it was possible to have execution of prior testing to obtain valuable insights pertaining to test case prioritization. Based on this hypothesis they proposed a new technique that makes use of history information in order to ascertain the probability of finding bugs for test cases and prioritize them accordingly. Their proposed technique made use of history-based criteria that helped in making decisions pertaining to prioritization of test cases. They made experiments with prioritization of regression test cases and found the usefulness of prioritization technique.

Leon et al [21] proposed a technique known as distribution based filtration for test case prioritization. They prioritized test cases based on the profiles of test cases available. They used a metric known as dissimilarity metric in order to create test case profiles. The dissimilarity metric produces a real number that can reveal the degree of dissimilarity between any two profiles. Thus the test cases were grouped into different categories based on dissimilarity values. Redundant test cases are identified and eliminated in this process. The experiments include test suite minimization, cluster filtering and additional coverage for minimizing test cases in a test suite.
Do et al [35] used code coverage based conditions for test case prioritization. They are used JUnit as the environment for empirical study. The prioritized test cases actually improve the number of faults that are detected. The experiments proved that with the help of the random prioritization technique resulted in higher APFD value when compared to the untreated ordering. These studies revealed that there are new test cases that showed the higher probability of detecting faults. Thus the untreated ordering can be exploited in the random prioritization.

Srijanath et al [30] focused on requirement-related factors for prioritization. They proposed a framework known as PORT. The factors used in the framework include fault-proneness, implementation complexity, volatility and customer assigned priority. As the factors are subjective in nature the outcome is influenced on the perceptions of developers or customers. The framework does not cater the specific regression testing that can be used for prioritization.

Qu et al [86] explored the history of execution of test cases in order to reuse them in black box testing. They used a matrix known as relation matrix in order to move the test cases for prioritization. They developed mechanisms for creating and updating matrix based results of test cases and showed detected faults. Bryce and Memon[64] revealed that the Event-Driven Software systems were also subjected to prioritization techniques. They used a‘t-way’ interaction coverage concept for testing the cases. They provided interactions between events.

Sampath et al [78] proposed a prioritization technique for web based applications. The criteria used for the technique includes parameters and their values, systematic coverage, request sequences and their appearance, and test lengths. Srivastava [63] explored a test case prioritization technique where average faults were computed per minute. The effectiveness of the algorithm is measured in terms of APFD (Average Percentage of Faults Detected) which is computed using the weighted average.

Malhotra et al [66] focused on the selection technique for the regression test. It has the provision for choosing the number of test cases for the generating test suite. It is used to exploit the fault detection when the code is modified. They validated the
utility of the techniques for two case studies. Their technique reduced the resources and cost for regression testing.

Acharya et al [1] proposed a test case prioritization technique in the environment known as Component-Based Software Development (CBSD). They used Object Interaction Graph (OIG) to compute the objective function. It is very useful to find out which component needs to be executed first in case of component-based systems. This knowledge helps to prioritize test cases.

Singh et al [46] specified that the test strategy needs to use certain variables in order to know the impact on the programs. Therefore, they opined that test cases should be prioritized for best results. Later they proposed a prioritization approach based on variables in the changed statements of the source code. They elevated their technique with the help of two paths, namely DC and DU with respect to data flow testing. The results revealed that the test case prioritization technique reduced the size of test suite and coverage that is not much improved.

Baurdy et al [8] explored test cases to know their defect revealing power. They used artificial intelligence (AI) and improved the efficiency of test cases with the help of the .NET component to performance testing. And also they studied the usage of GA for fault detection and for improving the efficiency of test cases. They compared their approach and proved Bacteriological GA was better.

Berndt et al [12] proposed a technique for optimizing test cases with the help of GA and fitness function and made a comparative study between the relative and absolute fitness functions. They revealed the concepts pertaining to GA for organizing fast research through relative and absolute fitness functions. They utilized three factors like severity, proximity, and novelty for fitness functions. They further visualized the fossil records for finding the search behavior of GA.

Walcott & Cupper [51] also used prioritization of test cases using GA. The aim was to keep test cases in a proper order so as to improve their efficiency in finding faults in the system which used many metrics to evaluate their work. They include random prioritization, static metrics, and structurally-based adequacy criteria. Static metrics and test adequacy criteria were used to generate fitness functions required for GA. The measure used is the Average Percentage of Faults Detected with
cost-consciousness (APFDC) for evaluating the effectiveness of ordering of test cases with respect to prioritization of regression test cases with the time-constrained environment.

Walcott et al [47] proposed a test suite generation method known as time-aware prioritization technique and compared the technique to other techniques that employed reverse ordering and the initial ordering. The results revealed that their approach showed high APFD than other approaches. Later, Zheng et al [89] focused on the usage of meta-heuristic algorithms and evolutionary algorithms like GA and hill climbing for prioritizing test cases. They exploited search algorithms in order to achieve this. Their main focus was on the increasing the rate of fault detection with the proposed prioritization mechanism.

2.20 More on Test Suite, Dependency Structure, and Prioritization

This section reviews more relevant literature on test suite generation and test case prioritization besides automatic extraction of dependency structures.

2.20.1 Whole Test Suite Generation

This section reviews prior works conducted on the automated test suite generation and related aspects. Genetic algorithms are widely used for generating test cases automatically. Rodolph [25] analyzed convergence properties pertaining to genetic algorithms. Arcuri and Briand [5] proposed adaptive random testing for enhancing random testing.


Inkumsah and Xie [45] integrated evolutionary testing and symbolic execution for improving structural testing of SUT. Islam and Csallaner [53] proposed a technique for generating mock classes and test cases in support of coding with respect to interfaces. Arcuri [6] explored on the length of test cases for structural coverage of
SUT. With respect to robustness testing Csallner et al. [18] explored JCrasher for testing robustness testing of Java applications.

Malburg and Fraser [39] proposed a hybrid approach using constraint and search-based testing to test software. Fraser and Zeller [27] focused on the mutation-driven generation of test cases and test oracles. Arcuri and Fraser [26] also found it useful to have parameter tuning with respect to search based software engineering. Many researchers contributed to the test suite generation, mutation testing and automated testing of SUT.

2.20.2 The Automatic Generation of Dependency Structures

Ryser and Glinz [40] acquainted dependency charts with represents for the conditions on situations and utilize these to drive testing. Initial, a natural language representation is utilized to map scenario descriptions of framework necessities. Scenarios are refined to state diagrams with pre and post conditions and information ranges with data values are utilized to take out ambiguities in the given code. The dependency structures clear up among scenarios through graphical diagram representations. On the premise of these state diagrams, test cases are separated on crossing all the interior ways of reliance graphs.

Kim and Bae [37] contended that there are predominantly two sorts of components in a framework essential and accidental. Accidental elements are dependent on the fundamental elements to make levels in a framework. They characterized a methodical way to deal with adjusting the components as fundamental and coincidental as per their conditions. Incidental elements change more frequently than the generally static fundamental elements; along these lines, Kim and Bae asserted that in overseeing legitimately adjustments in unplanned elements successfully, the framework testing exertion can be decreased to guarantee dependability. Experiment prioritization is not considered in the above methodologies.

2.20.3 Test Case Prioritization

Several prioritization techniques are outlined that related to the researcher’s approach about this section. These techniques are divided into three categories:
history-based-these techniques use information about the test suite from previous execution cycles to determine test priorities; the knowledge-based-these techniques are used human knowledge to determine test priorities, and model-based- these techniques are used as a model of the system to determine test priorities.

2.20.3.1 History-Based Prioritization

Rothermel et al [23] stressed that in utilizing execution data gained as a part of past trials to define test case the priority for enhancing the rate of fault detection during regression testing. They characterize numerous techniques to organize test cases on the premise of this data, including prioritizing on the basis of code coverage, prioritizing on the basis of the probability of finding faults, prioritizing on the basis of the code not tested before and prioritizing untested code. A greedy algorithm chooses the test with the highest coverage (called total coverage), or the test with the highest coverage on the code not secured in all tests that have as of now been organized (called additional coverage), until the sum total of what tests have been booked. These systems indicate adequacy at expanding the APFD of test suites.

A theory is viewed as that prioritizing on the premise of code or function coverage that likes the researchers dependency structure based approach on the grounds that for executing parts of the software product that cooperate with different parts, it is probably going to build code scope. Be that as it may, reliance structures contain no data about the code or capacities that are executed.

Future work will decide how firmly related these are. Li et al [36] exhibited a few non-greedy algorithms for organizing tests, including hill climbing algorithms and genetic algorithms. Their examination is in light of Rothermel et al note that greedy algorithms don't organize tests ideally [23]. Li et al contrast these non-greedy algorithms with Rothermel et al greedy algorithms and found that the greedy algorithms accomplished the best fault detection rate; genetic algorithms also achieved a high fault detection rate, at an increased computational cost.

Jeffrey and Gupta [19] built up an approach on the number of paths secured amid trial and named these influencing paths relevant slices. The secured number of relevant slices is given the high need of that test case. The relevant slices must be accomplished through trials. Wong et al [85] exhibit a test suite minimization and
prioritization system for regression testing in view of changes in the source code from past runs. Finding the announcements in that source code that have changed and utilizing execution follows from the past trial, Wong et al positioned test cases in view of the quantity of progress proclamations that are executed. Their outcomes demonstrate that the approach is more practical than not utilizing minimization or prioritization.

Li [36] introduced a code-scope based prioritization method. Li utilized a stretched out a type of dominator examination to rank the need of squares of program code. A square B1 commands another piece B2 if a test execution can't experience piece B2 without experiencing B1. A dominator diagram is gotten from the connections characterized between code squares. Every hub is then positioned in light of the quantity of obstructs that are executed if that piece is executed, and test cases are organized in light of this esteem. In spite of the fact that scope data is required, Li utilized typical execution to figure out which tests execute which squares. Later Wong [81], Li et al utilized this system to consequently produce test suites, including priority.

2.20.3.2 Knowledge-Based Prioritization

Ma and Zhao [88] presented a prioritization technique in view of program structure analysis to enhance the viability of discovering severe defects. They initially positioned the modules on the premise of a prioritization index called Testing Importance. The Testing Importance of Module (TIM) is a mix of fault proneness and the significance of a module. All separate respective test cases are organized on the premise of their TIM positioning after every module has been ranked. All Their trial comes about demonstrate a change in “severe” fault detection when contrasted with faults detected with the untreated and random prioritization.

Krishna Moorthi and Sahaaya [65] exhibited a strategy for test case prioritization in view of software requirements. Test cases are ranked through six factors—customer priority, requirements change, implementation complexity, completeness, traceability, and fault impact. System level test cases can be prioritized through this technique. Experimental results show the change in severe fault detection when contrasted with random prioritization.
Srikanth et al [29] acquainted a technique comparative with the method of Krishna Moorthi and Sahaaya, with various factors. They ranked test cases in view of the requirements volatility, customer priority, implementation complexity, and the fault proneness of the requirements. Each one of their Experiments demonstrated that the technique expands the likelihood of identifying extreme faults, and the customer priority contributes most toward this.

Tonella et al [60] exhibited the case-based positioning (CBR) framework, which is a machine-learning-based way to deal with prioritization. It misuses officially existing user knowledge with respect to the program and test cases to decide the test case priority on looking at two test cases. In this way, the relative priority is accomplished. User knowledge assembled after some time is inserted in the prioritization records iteratively to persistently refine the CBR framework. The issue with Tonella's approach is that an expansive number of pair wise comparisons should be made. For even a little test suite, the quantity of correlations is high, which requires effort in the interest of the tester.

To moderate this, Yoo et al [77] presented a clustering method for test case prioritization which diminishes the amount pair wise comparisons required. Clusters of tests are made in light of their closeness for instance, tests that cover comparative statements of the program are clustered together and the tester is asked for to do a couple insightful blends on a single test from every cluster, with the presumption that the single tests are illustrative of the whole cluster. Yoo et al additionally ranked the need for tests inside clusters utilizing statement coverage, and interleave tests between clusters to touch the base at an organized request. Their experimental outcomes exhibited that their technique can outflank standard statement-based coverage.

Qu et al [85] characterized a prioritization technique in light of a precise Combinatorial Interaction Testing (CIT) for picking subsets of test case combinations to control the combinatorial explosion issue. The test engineer first weights the decision (uncombined tests) in every class utilizing some knowledge that how the tests identified with the framework, for example, complexity or code coverage to prioritize tests. The weight of combining it with each other decision (the combinations) gives a communication advantage to every decision. Test cases are
organized in view of this collaboration advantage and result from a straightforward match prioritized methodology increment fault identification rate in regression testing contrasted and all combination, and a standard branch-based strategy accomplished the best rates of fault detection overall.

Basanier et al [22] depicted an approach for prioritizing test sequences produced from UML models. Their strategy creates main trees, in which each level of a tree represents an alternate integration level, and models the progressive conditions between these levels. A test engineer positions the weight of nodes in trees with weights in the range $\frac{1}{4}, 0, 1$, where the weight represents the significance of the test. The weight of all children of a node must aggregate to 1. At every integration level, the need for a test is the result of all nodes from the root node to the node representing that test.

Jiang et al [41] presented an adaptive random testing (ART) technique for test prioritization. Which is categorized as knowledge based because knowledge for the input domain is required; in any case, it is automatable. Jiang et al used a greedy algorithm for prioritization which arranges the accompanying test that is farthest a long way from all tests that have started now been sorted out. The partition measure relies on upon ART procedures, which plan to spread the decision of sporadic tests as impartially as could be permitted over the data range. These results demonstrate that ART prioritization fulfills a higher accuse distinguishing, proof rate than unpredictable prioritization and total extension prioritization, however, that additional degree prioritization finishes a higher accuse an area rate than ART prioritization. Regardless, ART prioritization is particularly less computationally expensive than additional extension techniques.

2.20.3.3 Model-Based Prioritization

Kundu et al [20] has created the test case generation and prioritization strategy for object-oriented projects that is maybe the most firmly identified with our work. This strategy picks UML sequence diagrams as system models and makes an interpretation of them into sequence diagrams, which are graphs that model all communications between the system and the client. Test cases are created for taking
all basic paths in the sequence graph, and producing the test case for every basic path represents a single scenario in the system.

Every edge in the sequence graph is given two weights: 1) a message weight, which is the quantity of messages between the edge nodes and, 2) an edge weight, which is the quantity of ways in the sequence diagram that go through this edge. Utilizing these weights, three prioritization measurements for the paths are characterized: 1) the sum the message weights of all edges in a path, 2) the weighted average edge weight of all edges in a path, and 3) the average of the message weight multiplied to the edge weight for all edges in a path. Kundu et al do not consider the measures, for example, APFD in their assessment; however, results about show that their techniques can expand the rate of code coverage, which is probably going to increase APFD. The result of Kundu et al technique is similar to researchers in that interactions between different system components are given a higher priority in the system. However, the researcher's technique can be applied to any set of test cases with dependencies, so it is not linked to any particular type of system modeling tools or languages. In addition, fine-grained test suites are maintained, whereas Kundu prioritizes entire test sequences.

2.21 Summary

This chapter covers the automated generation of tests from different viewpoints and generation of test data for helping software testing. It throws light into various tools in existence for testing software besides recent developments in software testing methodologies. The essence of this chapter, it is possible to generate test cases automatically instead of manually crafting them to avoid wastage of time and development effort. Other insights include that whole test suite generation provides smaller test suites when compared with a single branch in the generation of test cases; test case prioritization helps in improving the fault detection rate; dependency structures help in test case prioritization.

Essence test cases and test data generation are essential aspects of software testing. Test suite generation with multiple objectives in mind will improve code coverage besides enhancing the quality and lifespan of the SUT. The generated test suites are not independent, but have inter- dependency. The Automatic discovery of
dependencies and prioritizing them based on the dependencies is the problem to be
addressed. Test suite generation, prioritization and testing operations make a testing
tool, comprehensive and very useful for testing real world software products.