Chapter- 4

Automatic Discovery of Dependency Structures for Test Case Prioritization
4.1 Introduction

Software testing test cases are used to discover bugs from SUT. The test cases are executed generally in arbitrary order towards this end. However, from the literature it is understood that the test cases when executed in particular order, it is possible to increase the fault detection ratio. This hypothesis is tested in the next chapter. This chapter focuses on an automatic discovery of dependency structures. It is possible to prioritize test cases when dependency structures are known well. The actual prioritization of test cases is deferred to the next chapter. Dependency structures in any SUT are two types as illustrated in the section 4.2 of this chapter. They are open dependencies and closed dependencies. It is found that there were many methods that make use of dependency structures obtained manually in the literature. This is the first time to have a mechanism for an automatic discovery of dependency structures from SUT to the knowledge of the researcher.

When dependency structures are obtained from the SUT, they can provide details of the open and closed dependencies. However, it is a challenging job to discover the dependencies programmatically without human intervention. The author illustrated the possibilities of discovering dependency structures automatically by using different means such as reflection API and execution traces. The author has proposed a methodology that automates the process of extraction of dependency structures from the test cases that will result in the increase the rate of fault detection in order to discover dependency structures systematically. Thus the number of bugs uncovered from the SUT is improved. This leads to the improvement of quality of software. The important outcomes of this research are observed in this chapter as follows.

- Research Methodology used for an automatic discovery of dependency structures and then utilizing them for prioritizing test cases.
- An algorithm proposed for automatic prioritization of test cases.
- An evaluation methodology for validating the work done.
4.2 Background

Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design and coding. Testing spends almost 50% of software system development resources. The goal of software testing is to design a set of minimal number of test cases and test data such that it reveals as many faults as possible. Testing is a tedious task if it is carried out manually. Therefore, an automatic test case generation is given importance. The testing team can employ methods to prioritize test cases so as to increase the fault detection rate. Prioritization of test cases is a process of ordering test cases in order to increase the chances of fault detection and do it as early as possible. When bugs are identified early, it is possible to fix them early and reduce time, cost and effort and lead to on-time delivery of projects. Sometimes test suites are executed for days and weeks. In such cases, it is possible to prioritize them and reduce the time taken. With the wide usage of agile methodologies in the real world, the impact of test case prioritization will be more.

The literature on the test case prioritization showed that the prioritization techniques could improve the rate of fault detection. However, many techniques are not focusing on the functional dependencies between test cases. Functional dependencies are the dependencies pertaining to the interactions and possible relationships among the functionalities of the system under test. Dependencies have the potential impact on the order of execution of functions and the impact on the fault detection as well. For instance, a function A can only be executed if some precondition holds and function B enables this precondition. This kind of dependencies is known as functional dependencies. Most of the time test cases mirror this situation and test cases also inherit dependencies. Thus it is essential to execute some test cases prior to other test cases. Ensuring this thing is exactly known as test case prioritization.

There is a common strategy that can be used to handle test case dependencies. The strategy is to group the fine-grained ones into coarse-grained ones. It means that two fine-grained tests with dependencies can be merged into one coarse-grained test. It is a naive approach which does not work in all cases. For instance, it may execute a test twice causing unnecessary testing activity. Redundancy of test cases is another
problem that needs to be avoided. In the case of a large number of test cases it is essential to avoid duplicate test cases that kill the time of effort of test engineers.

Many existing methods on test case prioritization tried to include functional dependencies they lack algorithms that can be used to compute the sequencing test cases in order to preserve the order of execution. Rothermel et al [23] proposed a metric that considers the code coverage feature. A test case that exhibits more code coverage is given high priority. However, it is not able to work fine because when dependencies exist, additional test cases are required to be executed beforehand. When test cases achieve low coverage alternative tests are needed to ensure high code coverage.

4.3 Need for Automatic Discovery of Dependency Structures in SUT

Prioritization of test cases was considered when the researcher requires test cases, having top priority were executed prior than the other test cases at hand. In this method, errors can be found early. The problem of resource lacking can be resolved as the resource can be better utilized. Priorities are assigned according to some principle, but not randomly. And then test cases will run according to their priority. When identifying the defects at earlier, then the deployment of the product will be on time [26]. And the author can detect the faults by prioritizing test cases. This approach can be used for the prioritization in identifying the “functional dependencies”. The scenarios that have more dependencies will generate more defects as well.

Open dependency structure: It can be described as one test case should be executed prior to other the test case anywhere in the program [88]

Closed dependency structure: It can be described as; one test case should be executed just before the other test case means the other test case must follow the first test case. Need for prioritization of test cases: whenever the author inadequate number of resources then it may happen that the test cases which are not so much required to get run would consume the resources which results in lack of resources for the test cases which are urgently required to run. Prioritizing the test cases may avoid this kind of condition. The author can give high priority to the test cases which are urgently required to run and lower priority to those test cases which are less needed to get a run which will lead to better utilization of inadequate resources [88].
prioritization will also reduce the time utilization and will lead to early detection of faults.

Test suites help to detect faults in SUT, but there are many issues. For instance, test suites when executed in particular sequence can provide chances to unearth more faults. It means that test suite prioritization can be used to optimize testing results or to uncover more hidden faults. The hidden faults can be detected through the dependency structures. The functional test suites when subjected to prioritization can give effective test results that can help developers to rectify problems in SUT. Recently Haidry and Miller [76] studied the use of dependency structures for test case prioritization. Their research revealed that dependency structures can be used to prioritize test cases. Moreover, they could prioritize test cases and increase the ratio of fault detection. This is the hypothesis on which the researcher of this thesis focused. When a number of faults detected is more due to prioritization that will improve the quality of SUT. The fault detection ability of the testing procedure is measured in terms of the number of faults detected. There are some functions in object oriented programming that are to be executed prior to other functions. This is the rationale behind the thought of test case prioritization. It is important to have an automatic discovery of dependency structures in order to achieve test case prioritization. Thus the tedious task of manual discovery of dependency structures can be avoided.

As shown in Figure 4.2.1, it is evident that the root nodes are independent of other nodes and they do not have dependencies. Dependencies are two types like direct and indirect. For example, in Figure 4.2.1, $D_6$ is a direct dependent of $D_3$ and

![Figure 4.2.1- Sample dependency structure](image-url)
indirectly dependent on $I_i$. Yet in another classification, dependency is two types like 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, where as 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 metrics are used. They are known as DSP height and DSP volume to measure dependencies.

There are two terms associated with dependency structures. They are known as Dependency Structure Prioritization (DSP) volume and Dependency Structure Prioritization height. The former refers to the count of dependencies while the latter refers to the depth of the dependencies in terms of different levels. Two things are considered while computing the volume. They are known as indirect and direct dependencies. In the same fashion, while computing DSP height, the height of all test paths is taken into consideration. These two measures are utilized in order to have the best prioritization of test cases for achieving optimal results in testing. The researcher conducted experiments with both open and closed dependencies. Many real time projects are obtained from Internet sources for experiments. Out of them particularly CRM1 and CRM2 recorded very less number of dependencies while the Bash project recorded the highest number of dependencies.

The research is carried out with the chosen SUTs to have discovery of dependency structures and also to have prioritization. Prioritization details are not in the scope of this chapter as mentioned earlier. These results revealed that the dependency structures with DSP height and DSP volume measures were useful to achieve prioritization of test cases. It was evident in the empirical results presented in this chapter. The DSP prioritization methods achieve higher fault detection rate. The DSP measures help in improving the number of faults detected in the SUTs. Many test case prioritization techniques were presented in [76]. They are classified into knowledge-based [88], model-based [20] and history-based [23]. In the history-based models, the information about the test suite from previous execution cycles is used to determine the test priorities. Model of the underlying system is used in the model-based technique. Human intelligence is used in the knowledge-based approach to achieve test case prioritization. The researcher of this thesis proposed a methodology
for discovering dependency structures automatically. This methodology has guided the author to do the research.

An algorithm is proposed to make use of discovered dependency structures and prioritize test cases automatically. The functions such as an automatic discovery of dependency structures and also the test case prioritization are evaluated with the empirical study using the tool built to demonstrate the proof of the concept.

4.4 State-of-the-Art on Test Case Prioritization

Code prioritization for testing promises to achieve maximum testing coverage with the least cost. The author presents modern routine to provide hints to achieve best code coverage in this chapter. It includes two parts: extending the traditional dominator analysis method to include global impact of function/method calls. Then it relaxes the "guaranteed" condition of traditional dominator analysis to be "at least" relationship among dominating nodes, which makes dominator calculation much simpler without losing its accuracy. Two versions of code prioritization methods are implemented, the former based on original dominator analysis and the other on relaxed dominator analysis with global view [36].

Software developers save the developed test suites and reuse them later as software evolves. Such test suite reuse, in the form of regression testing, is pervasive in the software industry. Running all of the test cases in a test suite, however, can require a large amount of effort: for example, one of the industrial collaborators reported that for one of its products of about 20,000 lines of code, the entire test suite requires seven weeks to run. In such cases, testers want to order their test cases so that those with the highest priority, according to some criterion, are run earlier than those with lower priority [74].

Test case prioritization techniques have been shown to be beneficial for improving regression-testing activities. The rate of fault detection is improved, thus allowing testers to detect faults earlier in the system-testing phase with prioritization. Most of the prioritization techniques are code coverage-based till today. These techniques may treat all faults equally. The research is based on prior test case prioritization research with two main goals: (1) to improve user perceived software quality in a cost effective way by considering potential defect severity and (2) to
improve the rate of detection of severe faults during system level testing of new code and regression testing of existing code. A value-driven approach is presented to system-level test case prioritization called the Prioritization of Requirements for Test (PORT). PORT prioritizes system test cases based upon four factors: requirements volatility, customer priority, implementation complexity, and fault-proneness of the requirements. A PORT case study on four projects developed by students in advanced graduate software testing class was conducted. The result of this thesis shows that PORT prioritization at the system level improves the rate of detection of severe faults. Additionally, customer priority is shown to be one of the most important prioritization factors contributing to the improved rate of fault detection [29].

Test engineers often possess relevant knowledge about the relative priority of the test cases. However, this knowledge can be hardly expressed in the form of a global ranking or scoring. A test case prioritization technique is proposed that takes advantage of user knowledge through a machine learning algorithm, Case-Based Ranking (CBR). CBR elicits just relative priority information from the user, in the form of pair wise test case comparisons in this chapter.

User input is integrated with multiple prioritization indexes in an iterative process that successively refines the test case ordering. Preliminary results of a case study indicate that CBR overcomes previous approaches and for moderate suite size gets very close to the optimal solution [60]. Regression testing is an expensive part of the software maintenance process. Effective regression testing techniques select and order (or prioritize) test cases between successive releases of a program. However, selection and prioritization are dependent on the quality of the initial test suite. An effective and cost-efficient test generation technique is combinatorial interaction testing, CIT, which systematically samples all t-way combinations of input parameters. Research on CIT, to date, has focused on single version software systems. There has been little work that empirically assesses the use of CIT test generation as the basis for selection or prioritization.

The author examined the effectiveness of CIT by considering multiple versions of two software subjects were carried out in this chapter. Our results show that CIT performs well in finding seeded faults when compared with an exhaustive...
test set. Several CIT prioritization techniques were examined and compared them with a re-generation/prioritization technique [85].

Test case prioritization techniques have been empirically proved to be effective in improving the rate of fault detection in regression testing. However, most of previous techniques assume that all the faults have equal severity, which does not meet the practice. In addition, because most of the existing techniques rely on the information gained from previous execution of test cases or source code changes, few of them can be directly applied to non-regression testing. The researcher's objective, for this chapter, is to improve the rate of severe faults detection for both regression and non-regression testing. A novel test case prioritization approach is proposed based on the analysis of the program structure. The key idea of researcher's approach is the evaluation of testing-importance for each module (e.g., method) covered with the test cases. This methodology was implemented as a test case prioritization tool and performed an empirical study on both real and non-trivial Java programs. The experimental result represents that the approach could be a promising solution to improve the rate of severe faults detection.

Pair-wise comparison has been successfully utilized in order to priorities test cases by exploiting the rich, valuable and unique knowledge of the tester. However, the prohibitively large cost of the pair wise comparison method prevents it from being applied to large test suites. A cluster-based test case prioritization technique is introduced in this chapter. By clustering test cases, based on their dynamic runtime behavior, the required number of pair-wise comparisons significantly could be reduced. This approach is evaluated on seven test suites ranging in size from 154 to 1,061 test cases. An empirical study was presented to show that the resulting prioritization is more effective than the existing coverage-based prioritization techniques in terms of rate of fault detection [9].

4.5 Overview of Manual Discovery of Dependency Structures and their Utility

Dependency structures provide useful insights about test cases. There are chances to have functional dependency between functions to be tested. There are functional dependencies between the test cases. It is essential to understand the
dependencies and prioritize test cases in order to achieve good results in this context. Good results do mean that the prioritization of test cases yields in increasing the fault detection rate in SUT. The dependencies are two types namely open and closed dependencies. When a test case needs to be executed immediately before another test case, it is known as closed dependency. Whereas when a test case needs to be executed at any time before another test case is executed, it is known as open dependency. These dependencies are model based and they use underlying dependencies. Fine-grained test suites can be generated by using dependency structures. The fine-grained test cases can help improve the coverage while reducing the number of test cases in a test suite. Haidry and Miller [76] investigated the test case prioritization based on the dependency structures.

They used two measures in order to utilize the dependency structures in order to take well informed decisions on open and closed dependencies. Dependency Structure Prioritization (DSP) volume and DSP height are the two measures. The former refers to the weight of test cases that have more the number of dependents. The rationale for using this dependency measure is to know how close the test cases to each other and their cohesiveness is disclosed. They are likely to have relationships with them and that will reflect in prioritizing test cases. DSP volume is computed through both direct and indirect dependents of a test case. Another coverage measure used is DSP height. DSP height refers to the deepest dependencies that contain in scenarios with depth. Calculating the height of all possible paths, DSP height is computed. Towards this end these authors used depth-first algorithm that is executed on a graph representing dependencies. The dependency structures are represented in the form of graphs and then two measures are computed.

In the proposed approach the discovery of dependency structures is automated. The reason behind this is to reduce the manual process in identifying the dependencies as it is a very tedious task. Moreover, the automated discovery of test cases can provide more accurate results in fining dependencies. As the projects used for testing are usually the bulky automatic discovery is a desirable feature implemented. The following section throws light into the methodology for an automatic discovery of dependency structures.
4.6 Dependency Structure Based Test Case Prioritization

Test case prioritization approach followed in this research is based on dependency structures. It is well understood that software systems are very complex and that results in functions having coupling with respect to interactions between different parts of the system. With this understanding it is conceived that testing the parts of a system that exhibits more coupling and dependent interactions first can lead to a discovery of more faults. Therefore, it is claimed that dependency structures between test cases that are related to the part of the system where the coupling of the interactions is more are closely related and they are to be given priority in execution. It is possible to ensure that those tests will increase the probability of discovering bugs early in the testing phase of SDLC by assigning priority to some test cases. This idea was initially supported by Rothermel et al. [28] in their empirical study. They claimed that test case, concatenation and making them coarse-grained can increase the fault detection rate. This hypothesis is further supported by the author of this thesis.

There are many related techniques to discover functional dependencies among test cases based on dependency structures. The prioritization techniques employed in this research constitute Dependency Structure Prioritization (DSP). The technique related to DSP assign priority based on coverage value. The coverage metric value is used to know the dependencies. Dependency structures are two types. They are known as open dependency structures and closed dependency structures. When such test cases are organized in a tree, there is a possibility that some test cases are executed more than once. Two measures are used based on the following two possibilities to overcome this problem. It is possible to find the number of dependents of a given a test case. It is also possible to find out the longest path of indirect and dependent paths of the given test case. Based on these two points weights are used in order to measure dependencies. Parts in software with more inter-relationships are likely to have more coupling and thus more possibilities of having more faults. Therefore executing such test cases where more dependencies exist should increase the rate of detection of faults.
4.7 Methodology for Automatic Discovery of Dependency Structures

Dependencies can have Open or Closed dependence structures. After observing the dependencies, the test case prioritization is implemented, which leads to the ordering of test cases. The test cases with more dependents will get the highest priority and the high priority test cases are executed first. This will help to better utilization of inadequate resources. Otherwise, the test cases, which are urgent to execute, cannot utilize the resources.

The research on test case prioritization focused on various approaches as found in the previous section. For instance, they are based on execution traces [81], dependency charts that are derived through scenario-based testing [40], test costs and fault severities [71], test suite granularity and its impact on cost-effectiveness on regression testing [28], control techniques, statement level techniques and function level techniques [74], cost prioritization [60], fine granularity and coarse granularity [77], Prioritization of Requirements for Test (PORT) which is a value-driven approach [29], use case based ranking methodology [60], combinatorial interaction testing [85], analysis of program structure [88], adaptive random test case prioritization [41] and clustering test cases [77].

At the time of program execution, the execution traces are recorded. The execution traces information is stored in the form of a text file. Further, the text file is used as one of the input to the methodology.

In the proposed methodology, the execution traces are used to discover dependency structures. Dependency charts are the charts that provide the graphical view of the structural aspects of the source code. The cost of the tests and fault severities were also used in the literature in order to prioritize test cases. Test suite granularity can have its impact on the regression techniques in terms of cost-effectiveness. Control techniques are used to have an explicit comparison of test cases. Statement level and function level techniques are also used to perform the comparison.

In [76], coarse-grained and fine-grained approaches focused on the granularity of the test cases. There were more approaches on the basis of requirements also. Prioritization of requirements for value-driven testing is used for improving quality of
SUT. In Use-Case based ranking methodology, the Use-Cases plays major role and provides ranking for the prioritization. Combinatorial Interaction Testing (CIT) is used to have select test configurations to improve the rate of discovery of faults. Program structure analysis is another technique to know the dependencies and thus prioritize test cases. Adaptive random test case prioritization is an improved form of random test cases which can adapt to the situation based on the coverage. Effective and scalable prioritization was tried using clustering techniques.

Haidry and Miller [76] used dependency structures for test case prioritization recently. The approach used in [76] is improved by discovering dependency structures automatically in this chapter. The architecture of the proposed methodology is shown in Figure 4.7.1.

![Figure 4.7.1 - Proposed methodology](image)

It is evident that the proposed methodology depends on program execution traces and the actual program. In Method Discovery process, the methods are identified through reflection API and generate a list of all available methods. The call processing component is responsible for using traces and having some Meta data associated with calls. This Meta data is used later for test case prioritization. The test case prioritization component is responsible for understanding the Meta data associated with all calls and also considers the test suite. It makes the final and best ordering of test cases. The prioritized test cases are thus produced through the proposed approach. The important modules that are responsible for an automatic
discovery of dependency structures and automatic prioritization of test cases are as follows.

- The method discovery from SUT
- Call Processing from Traces
- Identification of Open and Closed Dependencies
- Making Final Dependency Structures
- Exploiting Dependency Structures for Test Case Prioritization

4.7.1 Method Discovery

The method discovery of the SUT is done by using reflection API provided by the Java programming language. The reflection API is used to know the details of applications being executed in the JVM and even modify its runtime behavior. This is an advanced feature in Java that can be elevated to discover the runtime details of the code being executed. The reflection API has many features such as extensibility features, support for generating class browsers, visualization features, debugging and for preparing testing tools. The reflection API played a very important role in discovering the methods. For generating test cases automatically, it is important to know all the functionalities or public functions of a class that can be tested. It needs to be discovered instead of identifying them manually by looking at the source code. This is the reason the reflection API played a vital role in the successful discovery of dependency structures.

The methodology includes an automatic discovery of methods available in Java classes. For this reflection API is used. Once the methods are identified and listed, the next step is to know the dependencies among them while executing them. This is very difficult to know just with reflection API. Therefore, another method Call Processing is used in the methodology. This is explored in the ensuing sub section.

4.7.2 Call Processing

The execution traces are recorded in order to capture the runtime flow of execution. This is vital for the discovery of dependency structures. Moreover, the methodology takes care of both closed and open dependencies. The traces are processed through iterative approach. The iterative approach makes use of collection
API provided by Java programming language. The collection API is available in `java.util` package. Especially the hash table and its equivalent classes and `ArrayList` are used in order to store intermediary results and finally map the open and closed dependencies. Once dependencies are identified and stored in collections, they are further subjected to verification to know whether the dependency is closed or open using the definitions presented earlier in this chapter.

4.7.3 Test Case Prioritization

The procedure for the generating test suite and the mutation testing were explained in the previous chapter. The test cases available are used in this chapter in order to prioritize them. When test cases are executed in the proper order it is possible to increase the rate of fault detection. This is the motivation for the whole research work presented in this chapter. As a matter of fact, the discovery of hidden faults can improve the quality of SUT. The test cases generated by the algorithm presented in the previous chapter are used in the methodology and the underlying algorithm presented in the ensuring sub section. The test case prioritization algorithm describes the execution of test cases with knowledge gained from the dependency structures. This is termed as test case prioritization.

4.7.4 Test Case Prioritization (TCP) Algorithm

<table>
<thead>
<tr>
<th>Input</th>
<th>Execution traces (<code>ET</code>) and program (<code>P</code>), Test Suite (<code>TS</code>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: Prioritized test cases (<code>PT</code>)</td>
<td></td>
</tr>
<tr>
<td>1. Initialize a vector (<code>M</code>)</td>
<td></td>
</tr>
<tr>
<td>2. Initialize another vector (<code>MM</code>)</td>
<td></td>
</tr>
<tr>
<td>3. Discover methods from <code>P</code> and populate <code>M</code></td>
<td></td>
</tr>
<tr>
<td>4. for each method <code>m</code> in <code>M</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a. scan TS</td>
</tr>
<tr>
<td></td>
<td>b. associate meta data with calls</td>
</tr>
<tr>
<td></td>
<td>c. add method <code>m</code> to the vector <code>MM</code></td>
</tr>
<tr>
<td>5. end for</td>
<td></td>
</tr>
<tr>
<td>6. for each <code>mm</code> in <code>MM</code></td>
<td></td>
</tr>
</tbody>
</table>
a. analyze TS
b. correlate with \( mm \)
c. add the corresponding \( m \) to \( PT \)
7. return \( PT \)

*Listing 4.7.4.1 – Algorithm for test case prioritization*

As can be seen in listing 4.7.4.1, it is evident that the proposed method takes traces, program and test suite as input. It performs the discovery of methods and automatic discovery of dependencies in the form of methods associated with Meta data and finally performs prioritization of test cases in the given test suite. The notations used in the algorithm are as follows.

<table>
<thead>
<tr>
<th>NOTATION</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ET</td>
<td>Execution Traces that are captured while the source code is executed</td>
</tr>
<tr>
<td>P</td>
<td>Program or application for which traces are available</td>
</tr>
<tr>
<td>TS</td>
<td>Test Suite that has been generated from the SUT.</td>
</tr>
<tr>
<td>PT</td>
<td>Prioritized Test cases</td>
</tr>
<tr>
<td>M</td>
<td>It represents a vector that can hold the discovered methods from given program or application</td>
</tr>
<tr>
<td>MM</td>
<td>It represents a vector that can hold the discovered methods from given program or application. However, it also has the associated meta data of methods</td>
</tr>
</tbody>
</table>

*Table 4.7.4.1 – Notations used in the algorithm*

The algorithm discovers the methods from SUT using reflection API as described in earlier in this chapter. Then the methods are subjected to order based on
the dependency structures extracted with the help of the execution traces. The methods are organized and visualized based on the dependencies (both open and closed). The dependency structures are utilized in this algorithm to prioritize test cases and generate the final list of test cases that are to be executed in the order of priority.

4.8 Experimental Results

This section provides details of the tool implemented and experiments made on it. The tool implemented in the previous chapter has been extended to incorporate the functionality of the proposed methodology in this chapter. It provides an intuitive interface with Graphical User Interface (GUI). It is menu-driven and helps the user to perform an automatic test case generation. It also helps in an automatic discovery of dependency structures for prioritization of test cases. However, in this chapter the experiments are limited to an automatic discovery of dependency structures. The tool demonstrates the proof of the concept and discovers dependency structures from the given program. The tool can distinguish between open and closed dependencies as described earlier in this thesis. Inputs and outputs were presented in this section besides the results of experiments.

The implementation of the methodology presented in this chapter, is done through reflection API and java.io provided by Java Programming Language. Open dependency related input program is as shown in listing 4.8.1.

```java
class Open Dependency {
    int a,b,result;
    void inputs(int a,int b)
    {
        this.a=a;
        this.b=b;
        add(a,b);
    }
    void add(int i,int j)
    {
        a=i;
    }
}
```
As shown in listing 4.8.1, the code is written in Java programming language. It is defined in such a way that it can be used to demonstrate the open dependency. This is used for discovering dependency structures automatically.

```java
class OpenDependency {
    int a, b, c;
    void sum(int a, int b) {
        this.a = a;
        this.b = b;
        c = a + b;
        System.out.println("sum=\"" + c);
    }
    void mul(int a, int b) {
        this.a = a;
        this.b = b;
        c = a * b;
        System.out.println("mul=\"" + c);
    }
    void sub(int a, int b) {
```

```java
```
As shown in listing 4.8.2, the code is written in Java programming language. It is defined in such a way that it can be used to demonstrate the closed dependency. This is used for discovering dependency structures automatically.

![Dependency Graph](a) ![Dependency Graph](b)

**Figure 4.8.1 – Visualization of dependencies for Listing 2 (a) and Listing 3 (b)**

As shown in Figure 4.8.1, it is evident that the open and closed dependencies are presented for visualization of dependencies. The GUI application is capable of presenting both kinds of dependencies to end users. These dependencies are discerned automatically, which is one of the main contributions of research. When dependencies
are identified, it is possible to prioritize them. The test cases generated can be rearranged in order to ensure the testing process, which is carried out systematically based on the dependencies. Thus it is possible to identify or unearth more bugs in the presentation.

Here improving fault detection rate considered which is a very important for outcome of the automatic discovery of dependency structures. An automatic discovery of dependency structures is made for the first time. Earlier it was done with the manual identification of dependencies. Now it is automated and demonstrated through the prototype application.

Many experiments proved that the automatic discovery of dependency structures do match with the ground truth and the tool has been extended to prioritize test cases automatically. The following section throws light on the evaluation of the results. The horizontal axis represents the result obtained by the tool and the ground truth, while the vertical axis represents similarity percentage.

4.9 Evaluation

This section presents how the automatic discovery of dependency structures is done correctly. Usually the evaluation was done by human experts. The senior software engineers who were working in different software companies in the testing department were identified. The sample size is 15. The prototype application is given
A human expert discovers the dependency structures manually from the SUT and compared with dependency structures generated by the applications. This process takes a long time as the human experts did this based on the request and in leisure time. Once evaluation results come from each one, the results are merged or averaged. Interestingly, dependency structures produced by the application and the same discovered by human experts manually were matching with 100% accuracy. This is considered to be the performance of the application when compared with the ground truth. The results are presented in Figure 4.8.2. In the results, accuracy comparison is presented between obtained accuracy with this methodology and the ground truth reflected in the manual discovery of dependency structures. Therefore, the accuracy is proved to be 100%. This evaluation is made with several case studies on the prototype application.

4.9.1 Threats to Validity

The measures used in the implementation of the prioritization technique are used for an automatic discovery of dependency structures in SUT. However, while evaluation of the methodology, senior software engineers were involved. They manually discovered dependencies from SUTs so as to evaluate the effectiveness of the proposed system. Here is the possibility of having bias may be unintentional. When the same work is done by other human experts, the measures might show different values. This probability cannot be ignored. Therefore, it is considered the first threat to the validity of our system.

The second threat which is very important is the size of applications. The number of applications considered for experiments and the size of the applications has limitations. This might cause biased conclusions. However, it is believed that the huge number of case studies are applied with different variations helped us to draw conclusions. Another aspect is that the test suites also can be subjected to human bias as test suites obtained by other human experts may be more efficient in increasing the fault detection rate.
4.10 Summary

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