NTNU – Trondheim
Norwegian University of
Science and Technology

Algorithms to identify failure pattern

Bhuwan Krishna Som
Poudel

Master in Information Systems
Submission date: June 2013
Supervisor: Tor Stålhane, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

ACKNOWLEDGEMENT

I would like to thank my supervisor, Professor Tor Stålhane for giving me opportunity to work with him and for helping me by giving feedback on my work time to time. I would also like to express my gratitude towards Dr. Surya Bahadur Kathayat who provided me guidelines in the research process.

ABSTRACT

This project report was written for "Algorithms to Identify Failure Pattern" at NTNU (Norwegian University of Science and Technology), IME (Faculty of Information Technology, Mathematics and Electrical Engineering) and IDI (Department of Computer Science).

In software application, there are three types of failure pattern: point pattern, block pattern and stripe pattern. The purpose of the report is to prepare an algorithm that identifies the pattern in a software application. Only theoretical concept is written in this report. My goal is to compare these algorithms and find the efficient one.

The report is written in the period from February 2013 to June 2013.

CONTENTS

LISTS OF TABLES

LISTS OF FIGURES

1 INTRODUCTION AND PROBLEM STATEMENTS

1.1 Introduction

The dependency on computer system has been steadily increasing and thus, the quality of the systems becomes more and more important. This can be partly done by testing the software part of the systems using appropriate testing mechanism.

Software testing is an activity that helps to assure the quality of software under test, by detecting bugs before serious failures takes place during operation. The inputs that are used for testing are called test cases and those that lead to software failures are called defects or failure causing inputs. Due to limited resources and huge set of possible inputs to the software under test, it is impossible to do exhaustive testing but if we identify how the defects are distributed in the input domain, it will be easier to reveal failures. According to Chen et al. [1,9], the defects or failure causing inputs can be distributed in a software input domain in three different ways. Some defects may be distributed all over the input domain while others make some types of clusters along an area in the software domain. Chen et al. categorized those failure-causing inputs into three categories: point pattern, block pattern and stripe pattern.

Test case selection is a time consuming and costly task in software engineering. Once the type of pattern is identified, we can apply the appropriate testing mechanism for the selection of the test cases in that particular software domain. There are different types of testing mechanisms that can be used for testing and removing the defects and a short description of the testing mechanisms are given in the section State of the art.

1.2 Problem definition

The selection of the most efficient testing method depends on the types of failure pattern in the software system. The problem here is to identify the type of failure pattern before testing the software. In this paper I have discussed three algorithms that can be used to find the failure pattern that exists in a software application. The algorithm first selects a point-containing defect in the input domain and finds the other defects around it. It then categories the types of defects into one of the three failure patterns i.e. point, stripe and block, thus making testing faster and thus, cheaper. Only the simulation of the algorithm is done to wee if it is possible to identify failure patterns that are already inserted for simulation purposes. I made real defects in a software application that is used for this simulation process assuming that the features with defects are clustered in the same way as defects are clustered in code.

For the simulation, we first made the real defects in features and tested with the algorithm to identify a pattern.

1.3 Research Questions:

RQ1- How is textual data transformed to numeric- data?

RQ2 - What is the most efficient strategy for identifying pattern type?

RQ3 – How does the efficient algorithm works with non-numeric data?

RQ3 - What is the best way to determine step size?

The step size is the value of distance by how much we move from one point to another. In our case, it is the distance from one feature to another. This is explained in section step size 4.

2 STATE OF THE ART

The orientation, shape, size and location of the geometric structure of the failure causing inputs in the input program under test is a topic of great interest in software engineering. A number of researchers have looked both theoretically and empirically at the types of failure patterns present in programs. Ammann and Knight [1988] observed that failure causing inputs tended to be clustered together in contiguous regions of the input space with varying cross-section sizes and also observed the existence of continuous failure regions in a missile launch control program [13]. Bishop in 1993 observed that many programs have contiguous areas of failures in some parts of the input domain. Also - according to them (Ammann and Knight, Bishop), the parts of the input domain that contains failure causing inputs are most often contiguous [12]. In the field of testing, researches are mainly focused on finding methods for the improvement of faultfinding effectiveness random testing [12]. Random testing is a testing approach in which inputs are selected randomly.

Regarding the issue of test input generation, two different techniques are possible: either the deterministic way, or the probabilistic way. Deterministic testing is the mostly focused technique in the field of testing, with focus on coverage of either a structural (i.e. white box) model of the software or a functional (i.e. black box) model of the software (see e.g., [Myers 1979, Howden 1987, Beizer 1990, Roper 1992]). As for functional test criteria, emphasis is put on the coverage of black box models of the program provided by the software specifications, whether they are informal or whether they are semi-formal (i.e. graphical).

Concerning the probabilistic generation, the conventional random testing approach involves exercising the software with inputs that are randomly generated according to a uniform distribution over the input domain (see e.g. [DeMillo et al. 1978, Ntafos 1981, Duran and Ntafos 1984]). But the fault revealing power of this approach is questionable: uniform testing is probably the poorest test strategy, since it does not take into account information relative to the target piece of software [7].

Ad Chen et al. [1, 9] observed that most of the faulty programs contain certain type of failures occurring in regular pattern throughout the input domain. According to him, inputs that are close to each other in the domain tend to go through the same path. Thus, in order to find most of the errors, test cases should be spread as much as possible. In the last few years, Chen purposed four approaches for the proper distribution of the test cases [Jonas G. Brustad, 2012].

- Partition Adaptive Random Testing
- Basic Adaptive Random Testing - ART
- Basic Random Testing
- Mirror Adaptive Random Testing – MART

For the explanation of the above different types of random testing, some assumptions are done:

- D: input domain type
- n: number of test cases
- m: number of test that fail
- $F = D/m$

The larger the value of m, the smaller will be the failure rate and the smaller the value of m the larger will be the failure rate.

Probability of failure rate θ is given by

$\theta = 1/F$ and

$F_{rel} = F_{obs} * \theta$

Where F_{rel} is real failure rate and F_{obs} is observed failure rate

Partition Adaptive Random Testing:

In this method test cases are selected by continuously partitioning the input domain. If C is the input domain that extends from (X_{min}, Y_{min}) to (X_{max}, Y_{max}), a point (X_1, Y_1) is randomly selected and from the location of the selected point the domain is divided into four parts (R_1, R_2, R_3, R_4) as shown in the figure 2.1 below.

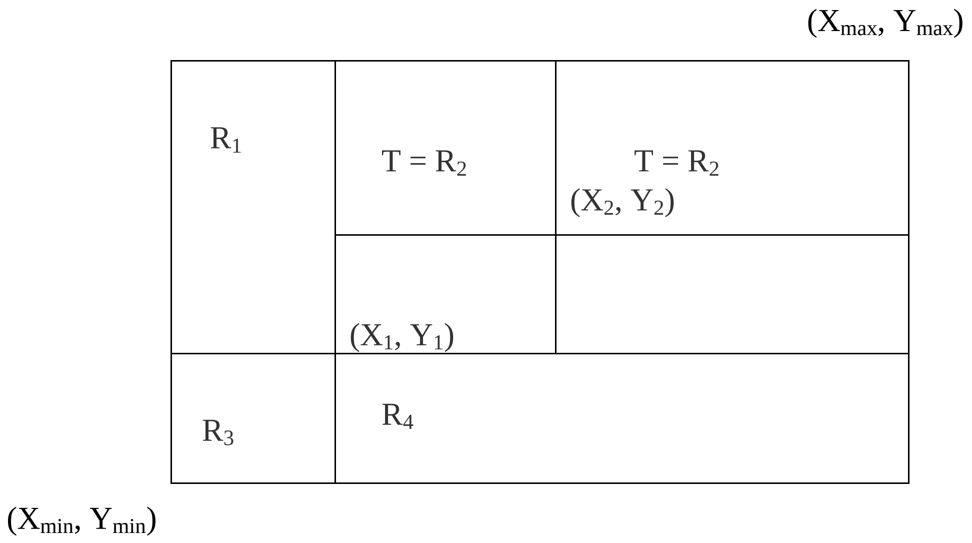


Figure 2.1: Partition Adaptive Random Testing

Among these four partitions, the largest part is selected for the next test case and the processes are repeated until a failure containing point is found.

Basic Adaptive Random Testing:

In Adaptive Random testing, test cases are selected on the basis of previous records. The Euclidean distance is calculated between the previously selected test cases and new test case is selected on the basis of the calculated distance.

For the selection of best data, Jonas G. Brustad [16] found an algorithm.

Maximum distance algorithm:

```
Function select_the_Best_Data(select_set, candidate_set, total_number_of_candidates);
best_distance: = -1.0;
for i: = 1 to otal_number_of_candidates do
candidate:= randomly generate one test data from the program input domain, the test data cannot be in candidate_set nor in selected_set;
candidate_set:= candidate_set + {candidate};
nim_candidate_distance:= Max_Integer;
foreach j in selected_set do
min_candidate_distance:= Minimum (min_candidate_distance, Euclidean_Dostance(j, candidate));
end_foreach
if best_distance < min_candidate_distance) then
        best_data: = candidate;
        best_distance: = min_candidate_distance;
end_if
end_for
return best_data;
end_function
```

Example:

Let T and C are the already executed tests and candidate test set respectively.

$T = \{t_1, t_2\}$ where $t_1 = (1, 1)$, $t_2 = (3, 4)$

$C = \{\}$ where $c_1 = (1, 2)$, $c_2 = (3,4)$

Using the above maximum distance algorithm we get

j = 1 => (c1, t_1) dist = 1.0, (c_1, t_2) dist = 2.8

j = 2 => (c_2, t_1) dist = 2.0, (c_2, t_2) dist = 3.0

Minimum distance, min(dist) = 1.0 and first distance larger than min(dist) is 2.0. Hence next test case is C2 = (3, 1)

Basic Random Testing:

In the Basic Random Testing methodology, every test case is selected randomly.

Mirror Adaptive Random Testing (MART):

In MART, the input domain is divided into several partitions. Adaptive random testing is applied to one of the partitions and duplicate test cases are generated in all the remaining partitions. The more the number of partitions better is the distribution of test cases.

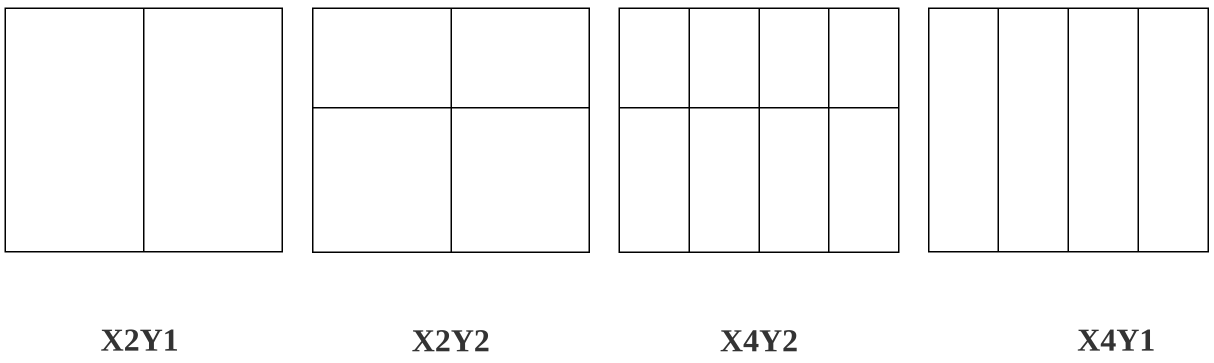


Figure 2.2: Mirror adaptive random testing

X2Y1 : X is bisected, Y is unchanged

X2Y2 : both X and Y are bisected

X4Y2 : X is split into four parts, Y ino two parts.

X4Y1 : X is split into four parts, Y is unchanged.

Adaptive random testing spreads the tests cases by computing the distance between them, while random testing selects test cases without taking care of the previously selected test cases. Thus, it is slower than random testing. Mirror adaptive random testing is a variant of random testing where distance computation is done only in a small portion or the input domain.

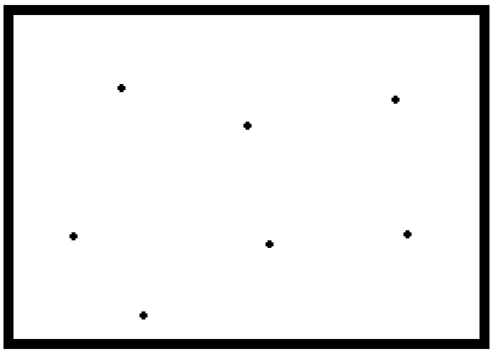
Exclusion Factor (f)

This factor will force the new tests away from the tests that have already been run and depends upon the failure rate and the failure pattern.

According to Chen et al., error patterns are categorized into three categories.

2.1 Point type pattern

In a point pattern, the defects are distributed throughout the input domain. There is not any type of pattern or geometric structure between the defects when the defects are organized in point type patterns. The following figure 2.3 shows how defects are distributed in a point type failure pattern.

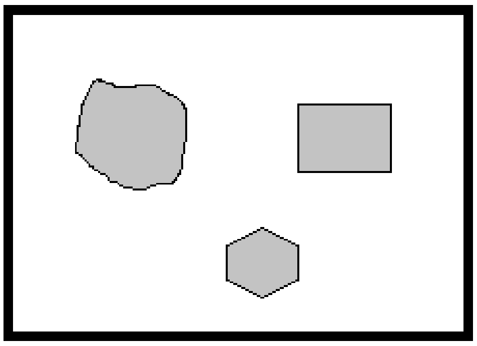


```
INTEGER X, Y, Z
INPUT X, Y
IF(X mod 4 = 0 and Y mod 6 = 0)
THEN
Z = X/2 * Y
        // correct statement is: Z = X / 7 * Y
ELSE
        Z = X * Y;
OUTPUT Z
```

Figure 2.3: Point type pattern

2.2 Block type pattern

If the defects are grouped at one place it is called a block type pattern. In a block pattern, defects are contiguously arranged at one place making a small sub-domain of the input domain in which the structure of the area covered by the defects is of a rectangular type. It does not have to be the shape of a rectangle, but if we draw a rectangle around this area, then all the defects lie inside the rectangle. Some possible shapes of the block type pattern are shown in the figure 2.4 below.

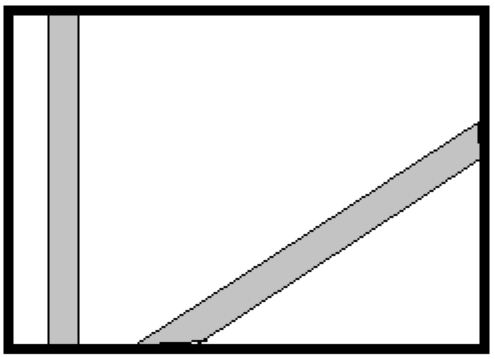


```
INTEGER X, Y, Z
INPUT X, Y
IF(X >= 10 AND Y <=11)
THEN
        Z = X/2 * Y
// correct statement is : Z = X/7 * Y
ELSE
        Z = X * Y
OUTPUT Z
```

Figure 2.4: Block type pattern

2.3 Stripe type pattern

As in block pattern, the defects of the stripe pattern are also clustered in one place but the two ends of the pattern touches the borders of the input domain. The shape of the structure of the area covered by defects is a stripe where the length of the area covered usually is much greater than its breadth. This is shown in the figure 2.4 below.



```
INTEGER X, Y, Z
INPUT X, Y
IF(2 * X – Y > 10 )
// should be if (2 * X – Y > 18)
THEN
Z = X / 2 * Y
ELSE
Z = X * Y
OUTPUT Z
```

Figure 2.5: Stripe type pattern

According to Chen et al., stripe and block patterns of failure occur more frequently than point patterns [2]. White-box testing and black box testing are the two common approaches to test the generated cases. White-box testing considers the structure of the program under test while black-box testing selects the test cases without considering the internal structure and functions of the program under test. In this paper, I have considered only black box testing and white box testing is out of the scope of my thesis.

Black-box testing techniques are the most used tests to test software applications. A person with no information or very little knowledge of the application can do black box testing. Random testing is one of the black-box techniques in which test cases are selected randomly under the assumption of uniform distribution of inputs. Random testing is a commonly used technique by practitioners [4, 6, 8] since it is intuitively simple, easy to implement and can be used to estimate the reliability of the software system. The system's reliability is expressed in terms of probability (described in section 5.2.1).

Reliability (R) = 1 – P(Θ)

where P(Θ) is the probability of finding a defect.

As the cases are randomly selected from the input domain without considering the information of the program under test, generated test cases may be too close. Thus, Adaptive Random Testing (ART) [3, 5] has been proposed to enhance random testing for non-point failure patterns. In ART, computing some distance with the previously selected test cases will spread the test cases evenly. It has greater chances of hitting the non-point failure pattern since it prevents the selection of test cases from the same region as the previously selected test cases.

Only the rate of failure-causing inputs is used in the measurement of effectiveness in random testing studies. For example, the expected number of failures detected and the probability of detecting at least one failure are all defined as functions of the failure rates. However, in a recent study by Chan et al. [14], it has been found that the performance of a partition testing strategy depends not only on the failure rate, but also on the geometric pattern of the failure-causing inputs [15]. This has prompted the researchers to investigate whether the performance of random testing can be improved by taking the patterns of failure-causing inputs into consideration. So they developed adaptive random testing. Their studies show that adaptive random testing outperforms ordinary random testing, as the effectiveness of random testing can be significantly improved without incurring significant overheads.

All research activities within random testing is done on how to select proper test case so that the test cases are distributed to cover the input domain of the software application in the best possible way. This will reduce the probability of missing a defect. I have tried to extend this research. Once a defect is found somewhere in the input domain, there is a high probability of finding other defects near it. I therefore tried to identify the pattern of the defects grouped in one place.

In real applications, all defects are not in integer form but in text and in clicking on icons or form, so it is difficult to apply the algorithms (chapter 4) directly to the text type of input. Thus, those features need to be mapped onto a 2-dimensional space (discussed in chapter 5), from where we can get the numerical data that are suitable for the algorithms that I described in chapter 4.

3 PURPOSE OF TESTING

For any software developer, at some point, there is a pressure to meet the deadline to release the software product. Even if we use the best techniques in software development practices, tools and engineers, we still need to test the product before it is released. Careful consideration should be taken as to the overall impact of a customer finding a bug in the released product. Testing is the first activity done by the programmers to check for bugs in software products before it goes live. It is an important way of assuring the reliability of software and it is the technique used to check the reliability of product and identify the problems that remain. The bugs may be buried deep somewhere in an obscure function in the software product, and if it results in a typo within a seldom used report, the level of the impact is low and the effect is negligible, but if the bug results in the program crashing and loosing data, may be in a traffic control system, the impact will be high and may result in loss of life. Thus, software testing must be performed to find the level of risk and the effect of the bug, prior to its release.

Testing is a powerful tool in ensuring that software development results are aligned with the customer's business objectives and is performed for the following reasons: to

- test the reliability of the product.
- prevent serious failure.
- ensure that software meets the requirement needed to satisfy the customer.
- ensure that the software works with other software and hardware that it needs to work with.

To summarize these points, testing is done for the reasons summed up in the three sections below:

3.1 To assure quality

The quality of the software means fulfilling the requirements of the customers and giving conformation to it. As many software products are used in critical applications, the outcome of a bug can be severe [10]. To improve the quality of software is important since small bug can cause severe problems such as airplane crash, giving wrong direction to space shuttle missions, making loss in trade, and giving wrong directions to the nuclear weapons. Functionality, engineering and adaptability are the three factors that determine the quality of the software [10].

3.2 Validity and verification

During implementation of software, testing is used as a tool in the validation and verification process where we try to verify whether the product works under certain conditions or not. Verification checks whether the product behaves according to the requirements and design specifications or not, while validation determines whether it fulfill the specified requirements [11].

3.3 Reliability estimation

A system, for example the Facebook application, consists of many modules such as login, chat, update as its component, whose individual or combined failure can lead to collapse of the system. By performing testing before the implementation of any software system we can estimate the reliability of the particular system.

4 ALGORITHMS

Once the structure of the pattern is identified in the software domain, it will be easy to apply the appropriate testing mechanism. This saves time and cost in the testing process.

It is easy to distinguish the point type patterns from the others two patterns. After finding one defect in any portion of the software domain, if we find any others around it, then it is either a block or stripe type pattern. Otherwise it is simply a point type pattern. One input domain may contain all three types of patterns as shown in the figure 4.1 below.

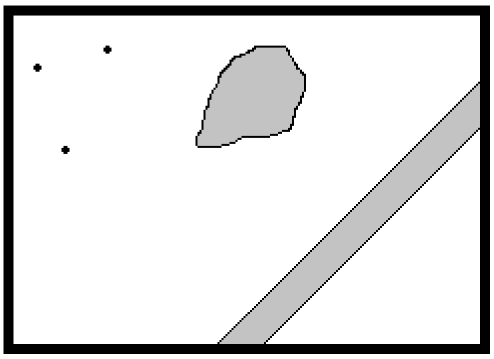


Figure 4.1: All patterns in a single application

To distinguish between these three types of pattern, three different algorithms are taken into consideration for analysis.

- Simple distance computation method
- Circular method
- Heuristic method

In all the three methods, one defect is detected first and then we search for other defects nearby by exploring its surroundings. For this we compute all the neighbors of the first point. In the Heuristic method, only one of the neighbors is taken into consideration whereas in other methods, each and every neighbor is considered. If defects are not found, then the software is said to be error free. This is, however, a rare case.

In the end, all the test cases that contain defect are collected in the array list DF[]. If there is only one test case in DF[], then it is point pattern. Otherwise it is either block pattern or strip pattern. By doing some boundary tests - see section 5.4 - we can see whether it is a block pattern or a strip pattern.

The structure of the failure pattern can also be shown in two dimensional graph by plotting the points of the array DF[].

4.1 Some often used terms:

In this paper for the simulation purpose I have made a software tool which executes all the algorithms discussed in chapter 4. The software tool contains ten modules (see chapter

4.11a). Loginform is the module which authenticates the user with a valid user name and password. Through 'New Defects' module, we can seed defects in the features of all modules. 'MainFrame' module displays three buttons which allows us to test three different algorithms, which are explained in chapter 4. Figure 4.2 below shows the mainframe of the software tool.

While tracing the algorithms the following terms are used:

I have described the system (tool) using UML use case diagrams and sequence diagrams.

4.11a Module: a module is a class file.

I have assumed a module as a part of a software application, which contains a class file. For example we can take login part of any application as one module and update as another module.

4.11b Feature: a feature is a method in a class file, which displays the characteristics of a module.

For example cancel is a feature of a login form, which closes the form.

4.11c Point: a point is a location of feature in a module. P(n,m) represents a nth feature of mth module.

4.12a MainFrame: Is the main frame of the software tool where the user can have access to all the sections of the application as shown in the figure 4.2 below:

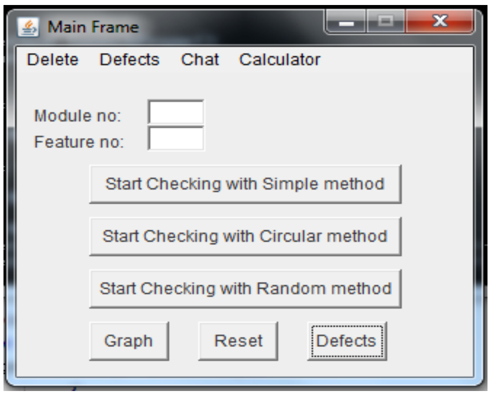

Main Frame
Delete
Defects
Chat
Calculator
Module no:
Feature no:
Start Checking with Simple method
Start Checking with Circular method
Start Checking with Random method
Graph
Reset
Defects

Figure 4.2: MainFrame of the application

4.12b Database table: Is the table in the database where all the records of the feature are stored including the defects – remember: this tool is used to simulate a real software system and we will need to know all the defects in order to run the simulations. When we seed a defect in the application, it is stored in the table "AllDefects" in a database named "database4".

4.13a Continuous points: Those points, which come one after another. If i-1, j-1, i+1 and j+1 are not outside the border, P(i, j), P(i+1, j), P(i+2, j), P(i+3, j) and P(i, j), P(i, j+1), P(i, j+2), are continuous points.

4.13b Dis-continuous points: are points, which are not neighbors. If i-1, j-1, i+1 and j+1 are not outside the border and ‘S’ is the step size , P(i, j), P(i+N, j) and P(i, j), P(i, j+N), are dis-continuous points. Where ‘S’ is not equal to ‘N’

4.14 Random numbers: are generated by using a java library function.

4.15a End Points: are the points at the border lines of the corresponding point.

If we have m numbers of modules and n numbers of features, the end points of a point P(i, j)are :

P(1, j), P(m, j), P(i, n) and P(i, 1) as shown in figure 4.1c.

For the point that is not at the borderline, there are four end points. If we have 10 modules and 12 features, then;

- the end points of the point P(3, 4) are P(3, 1), P(1, 4), P(3, 12) and P(10, 4).
- the end points of the point P(1, 4) are P(1, 1), P(1, 12) and P(10, 4).
- The end points of the point P(1, 1) are P(1, 12) and P(10, 1) .

4.15b Neighbors: are the points around a point.

Assume a point P(i, j). If i-1, j-1, i+1 and j+1 are not outside the border, this point has the following neighboring points:

P(i-1, j), P(i+1, j), P(i, j-1) and P(i, j+1)

For example, P(5, 6), P(7, 6), P(6, 5) and P(6, 7) are the neighbors of the point P(6, 6).

4.16 Conceptual distance between features: The function dist(x,y) measures the distance between two features in a module (see chapter 4.11a), where x and y are the terms used to describe the property of the features. The function dist(x,y) determines the similarity/relation of each pair of terms from the two features which distance is to be calculated by considering the distance of terms in a lexicon, according to lexicon relation of synonymy, hyponymy and hyponymy.

$$\text{dist}(x,y) = \begin{array}{l} 0 \text{ if } x = y \quad \text{or} \quad \text{synonym}(x,y) \\ d \text{ if hyponym } (x,y,d) \ \ \text{nypernym}(x,y,d) \\ \text{infinite} \quad \text{otherwise} \end{array}$$

As an example, consider some features from the block “Login Module”. They are

1. Is the format of the title good?
2. Does the minimize button work?
3. Does the maximize button work?

In these three features, the last two features contain common terms such as ‘button’ and ‘work’ but there is not any common single term for the first feature and the last two features. Thus, we consider feature number 2 and 3 to be related.

4.18 getdetectedNeighbours(p,DT[]): This is a recursive function which takes two parameters p(location of a point) and the array DT[] which stores the detected points only. At the end of the execution of this function it stores all the location of the points it detects to be failure causing inputs around the initial point to an array called DF[].

The ‘d’ and asterisk in the table shows the defects and detected points by the algorithms respectively. “$*_0$” is the initial point selected randomly, “$*_1$” is first detected neighbor containing error and “d” is the tested point that does not contain defect.

4.2 Simple distance computation method

The distance computation methods used here entails an algorithm that access data sets for a pattern of defects, which is initiated by randomly searching point defects. If one is found, the algorithm then tests entries adjacent of the defected point in all four directions; X axis (left and right) and Y axis (up and down). The algorithm tests the datasets in both axes. If an error is found, we will select points continuously in the same direction until a data point without a defect is encountered. If a selected point contains a defect, a point next to it on the same axis (i.e. X-axis) is selected and we test whether it contains an error or not, otherwise neighbors of the initial point on the Y-axis are tested in the same way as those on the X-axis.

For example, if P(3,9) is the initially selected point amd the step size is 1, P(2,9) and P(4,9) are its neighbors along the X-axis. The points P(2,9) and P(4,9) are tested. If any one of them contains a defect, their neighbors on the X-axis are tested, otherwise the neighbors of the initial point i.e. P(3,9) on the Y-axis are tested. Subscript on the asterisk and ‘d’ represents the step to show the process of selecting a point. In this method, the step size is taken as a distance to compute the next test case from the previous one. The following figure 4.3 shows the selection method of test cases in simple distance computation method. Flowchart for simple distance computation method is given in appendix A3.

	1	2	3	4	5	6	7	8	9	10
1			$*_{15}$							
2			$*_{14}$							
3			$*_{13}$							
4			$*_{12}$							
5			$*_{11}$							
6			$*_{10}$							
7			$*_{9}$							
8			$*_{8}$							
9	d_{7}	$*_{6}$	$*_{0}$	$*_{1}$	$*_{2}$	$*_{3}$	$*_{4}$	d_{5}		
10			$*_{16}$							
11			$*_{17}$							
12			$*_{18}$							

Figure 4.3: process showing simple distance computation method with step size 1

Algorithm:

Step1: Locate randomly a point V(p) in the domain(D).

Step 2: Check if it is already detected.

If it is not detected mark point P as detected and put into array DT[]. Otherwise repeat step one.

Step 3: Check if the detected point contains a defect.

```
If it is not a defect
    Return null
If it is a defect
    Mark it as defected and put that point into array defected DF[]
```

Step 4: Take one point from neighbor along X-axis and repeat the process from Step 2.

Step 5: Take one point from the neighbor of the initial point along Y-axis and repeat the process from Step 2.

Pseudo code:

```
Set  DT[] , DF[] and NG[] to  null.
    Declare k as integer and p as point type which contains coordinates
    getdetectedNeighbours(p,DT[])
        {
            Add p to DT[]
            If (DF(p))
                {
                    Add p to DF[]
                    for(k=0;k<2;k++)
                        {
                            if $P_k \notin$ DT[]
                            getdetectedNeighboursinX(k,DT[])
                        }
                        for(k=0;k<2;k++)
                        {
                            if $P_k \notin$ DT[]
                            getdetectedNeighboursY(k,DT[])
                        }
                }
        }
```

4.3 Circular method

In the Circular method, one point is randomly selected and checked in the database table to see if it contains a defect. If the point contains a defect, the neighbors (see chapter 4.15b) of that point are checked and this process is repeated for all the neighbors until we either find a point containing an error or, if the initially selected point does not contain a defect, select another point at random and check this point. For example, let the initially selected point be P(1, 2). According to our database record, the point P(1, 2) does not contain a defect. Thus another point is randomly selected. Let this a point, which contains an error, be P(5, 7). Its neighbors are computed. If we still use step size =1, these are P (4,7), P (6,7), P (5,6) and P (5,8). Since the point P(5, 7) contains a defect, it is inserted in an array list and the same process is repeated for every neighbors, first right and upper one, left and finally lower one as shown in the figure 4.3a.

If P(i, j) is the selected defect containing point and i-1, j-1, i+1 and j+1 are not outside the border, according to this algorithm first we test P(i+1, j) then P(i, j+1) followed by P(i-1, j) and P(i, j-1).

In the end, all the defect-containing points around the initial point (in this case P (5, 7)), are collected in the array list and by performing a boundary test (see chapter 5.4), the pattern of the collected defects is identified. The steps for this procedure are given below. The subscript along with the asterisk or 'd' represents the step of the process. 'd' denotes the defect -free point and '*' denotes a defecting containing point. Flowchart for simple distance computation method is given in appendix A4.

	1	2	3	4	5	6	7	8	9	10
1										
2							d_{12}			
3						d_{13}	$*_{10}$	d_{11}		
4					d_{15}	$*_{14}$	$*_{8}$	d_{9}		
5				d_{17}	$*_{16}$	$*_{5}$	$*_{6}$	d_{7}		
6			d_{18}	$*_{18}$	$*_{2}$	$*_{3}$	d_{4}			
7				d_{20}	$*_{0}$	d_{1}				
8			d_{25}	$*_{24}$	$*_{21}$	$*_{22}$	d_{23}			
9				d_{26}	d_{27}					
10										
11										
12										

Figure 4.4: Process showing the Circular method

Algorithm:

Step1: Locate randomly a point V(p) in the domain(D).

Step 2: Check if it is already detected.

If it is not detected mark point P as detected and put into array DT[]. Otherwise repeat step one.

Step 3: Check if the detected point contains defect.

```
If it is not a defect
    Return null
If it is a defect
    Mark it as a defect and put that point into array defected DF[]
    and  find its four neighbors.
```

Step 4. Repeat the process from Step 2 for all the neighbors.

Pseudo code:

```
Set  DT[] , DF[] and NG[] to  null.
Declare k as integer and p as point type which contains coordinates
getdetectedNeighbours(p,DT[])
        {
            Add p to DT[]
            If (DF(p))
                {
                    Add p to DF[]
                    for(k=0;k<4;k++)
                        {
                            if $P_k \notin$ DT[]
                            getdetectedNeighbours(k,DT[])
                        }
                }
        }
```

4.4 The heuristic method

As the name implies, the Heuristic method is a trial and error method and is risky that it may conclude with a wrong pattern. It avoids complex computations, which is important in a large domain, by testing a maximum of five points. In this method, an initial point is randomly selected and its end points (see chapter 4.15a) are determined and tested. If the tested point contains a defect, it is inserted into array DF[] otherwise it is inserted into array DT[]. If none of the end points contain a defect, we assume that we have a point pattern. If the initially selected point is not an end point and only one point among end points contains a defect, then we assume it is a block pattern. If any two end points contain defect then we assume that it is a strip pattern.

Let us look at example. If P(3,9) is an initially selected point, P(1, 9), P(10, 9) P(3, 1) and P(3, 12) are its end points. The initially selected point P(3, 9) is not an end point. Among four end points, two points (P(3, 1) and P(3,12)) contain defects that is in line X = 1 and Y = 12. Thus, we assume the following figure 4.5 a strip pattern. Flowchart for simple distance computation method is given in appendix A5.

	1	2	3	4	5	6	7	8	9	10
1			$*_{3}$							
2										
3										
4										
5										
6										
7										
8										
9	d_2		$*_{0}$							d_1
10										
11										
12			$*_{4}$							

Figure 4.5: Process showing Heuristic method

Algorithm:

Step1: Locate randomly a point V(p) in the domain(D).

Step 2: Check if it is already detected.

If it is not detected mark point P as detected and put into array DT[]. Otherwise repeat step one.

Step 3: Check if the detected point contains a defect.

```
If it is not a defect
    Return null
If it is a defect
    Mark it as a defect and put that point into array defected DF[]
```

Step 4: Find the four end points of point V(p).

Step 5: Test each end points to see whether it contains defects or not. If the tested point contains defect, put it in DF[] else put it in DT[].

Pseudo code:

```
    Set DT[] , DF[] and NG[] to null.
```

```
Declare k as integer and p as point type which contains coordinates
getDefect(p,DT[])
        {
                Add p to DT[]
                If (DF(p))
                        {
                                Add p to DF[]
                                for(k=0;k<4;k++)
                                        {
                                                if $P_k \notin$ DT[]
                                                testAllEndPoints(k,DT[])
                                        }
                        }
        }
```

5 MAPPING

5.1 The purpose of mapping

In computer science, mapping is a logical connection between two sets of entities. In our case, the entities are: features of a real application, and a two dimensional space. The purpose of the mapping is to show how a real software domain can be logically connected to the two dimensional domain so that it will be possible to represent the position of a defect point.

Determination of a failure pattern starts when the development phase of the application is completed. We assume that the patterns are meaningful- if there is a defect in a single feature on a module (se chapter4.11a), then we will assume that as a point pattern. If the number of defects clustered in a module is greater than one then we assume that as a block pattern. In block pattern if the pattern is distributed from one boundary to another boundary (see chapter 5.4) of the input domain then that is a strip pattern. To start checking for a failure pattern according to the algorithms described in chapter 4, mapping is done from the programmers' point of view to the patter checking algorithm's point of view. In order to perform the mapping, the programmer has to perform the following activities:

- The application is divided into modules (see chapter 4.11a). This is done based on the class files of the software application. If we take the Facebook application as an example, the part that checks the authentication and authorization, "login section" is one class file and is defined to be a module and "logout", "edit profile" and "upload status" are other such modules.
- Each module has several properties, called features (see chapter 4.11b), and consists of one or more lines of codes. The length of the password, characters type of password, visibility of password is the feature related to password and is coded within a single module of code.
- While writing code for features in a program, the features that are related to is coded in one block of code. That is similar features are coded in one block of code. The first work of the developer is to divide the program into modules and write code for each feature in each module. Then the function dist(x,y) cabn be used to compute the similarity between two features (see in section 5.2).

In order to implement the algorithms describes in chapter 4, the features are mapped onto a two dimensional space. The modules are ordered along the x-axis and the features along the y-axis. The numbering starts from 0 as shown in the figure 5.1 below. In my case I have assigned module and feature number from one for the purpose of simplicity to explain. Here feature (1,1) represents the first feature of the first module and the feature (6,12) represents the 12th feature of 6th module.

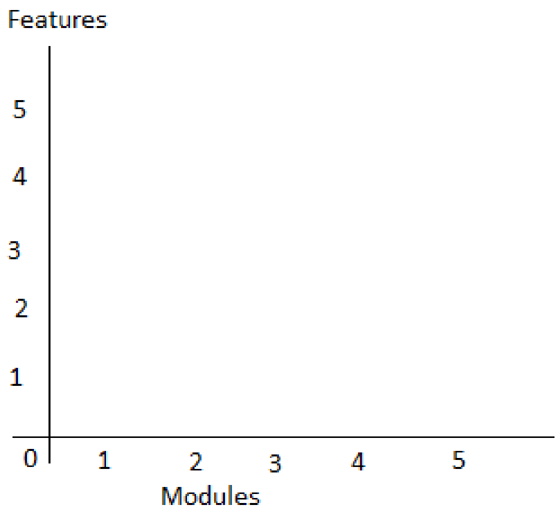


Figure 5.1: Two-dimensional view of application

While mapping to two dimensional space the point P(X, Y) is represented as

P(<module> , <feature>)

When a software application is ready for testing, we first need to generate the feature list. The system analyst, in cooperation with the developer, generates technical features according to the implementation. Figure below is a sequence diagram of mapping process.

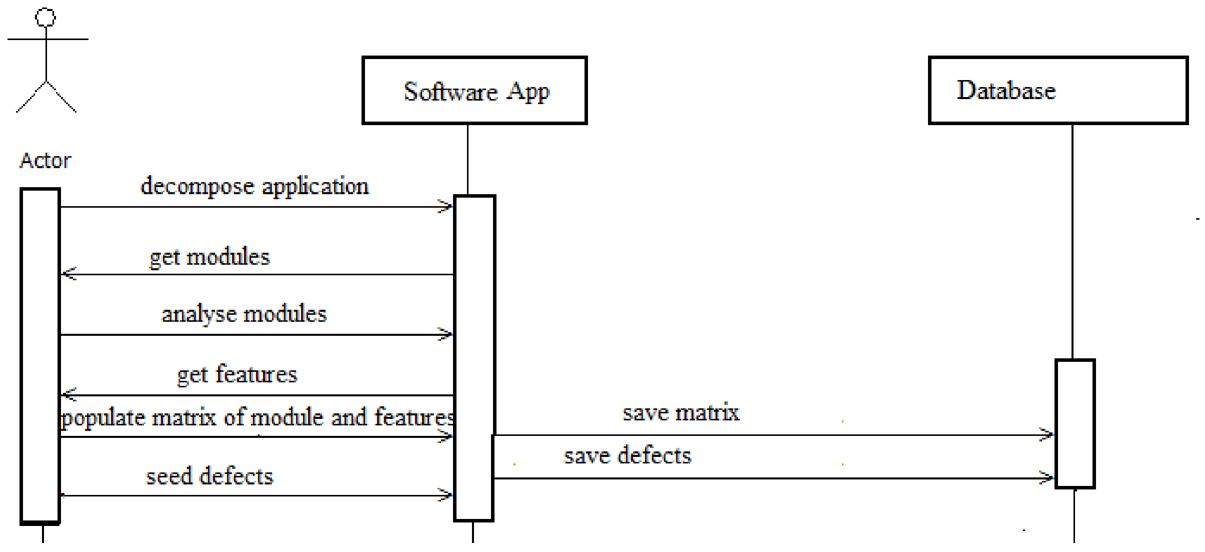
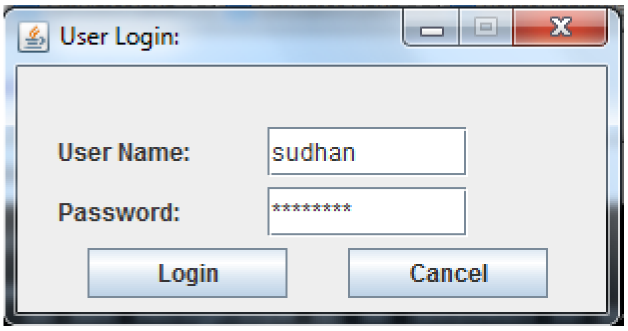


Figure 5.2: Sequence diagram of mapping

I have used the software tool that is used to implement the algorithms (chapter 4), as an example of a software application containing several modules (4.11a) such as Login, Testing, Feature display, Defects display and Feature Insertion. Each module is contained in one class file.

In my software tool LoginForm is a module and is contained in the class file "LoginForm.java". When we execute "LoginForm.java" file, it displays a form as shown in the figure below.


User Login:

User Name: sudhan

Password: ********

Figure 5.3: User Login Form

We need the corresponding class file and the requirements list from the customer to generate a feature list for each module. This will enable us to map the customers' point of view to the programmers' point of view. We can make many features (see 4.11b) from each module. For example, a customer may want a login form to be displayed at the center of the screen. Thus we need a feature for the location of the form. A high number of features increase the complexity of the testing process but also increases the quality of the testing as we explore the application in more detail with higher number of features. The coding of the LoginForm.java is given below.

```
import javax.swing.*;
import java.awt.event.*;
import java.sql.*;
public class LoginForm extends JFrame implements ActionListener
{
        JLabel lbluser,lblpass;
        JTextField txtuser, txtpass;
        //JPasswordField jpf;
        JButton btnlogin;
        public LoginForm()
        {
                setLayout(null);
                lbluser=new JLabel("User Name:");
                lblpass=new JLabel("Password:");
                txtuser=new JTextField(20);
                //jpf=new JPasswordField(20);
                txtpass=new JTextField(20);
                btnlogin=new JButton("Login");
                add(lbluser);
                lbluser.setBounds(20,30,100,25);
                add(txtuser);
```

```
22.             txtuser.setBounds(125,30,100,25);
23.             add(lblpass);
24.             lblpass.setBounds(20,60,100,25);
25.             //add(jpf);
26.             //jpf.setBounds(125,60,100,25);
27.             add(txtpass);
28.             txtpass.setBounds(125,60,100,25);
29.             add(btnlogin);
30.             btnlogin.setBounds(75,90,100,25);
31.             setVisible(true);
32.             setSize(300,150);
33.             setLocation(250,250);
34.             setTitle("User Login:");
35.             setResizable(false);
36.             setDefaultCloseOperation(EXIT_ON_CLOSE);
37.             btnlogin.addActionListener(this);
```

In reality, an error can occur almost everywhere. For example, the size of the form may not be according to the customer's requirements. The location of the form, the color or the font size may be wrong and there may be some functional errors such as the close button may not function properly. Likewise, the login button and cancel button may not behave as required. Based on the required properties we can make a feature list for the module. In our case line number 32 in the code, setSize(300,150); declares the size of the application form. The following are the features that are generated from module LoginForm module.

1. Test the size of the form.
2. Test the title of the form.
3. Test the format of the title.
4. Test whether the minimize button work or not.
5. Test whether the maximize button work or not.
6. Test whether the close button work or not.
7. Test whether the form is resizable or not.
8. Test the position of the cursor.
9. Test the visibility of password characters.
10. Test the function of login button.
11. Test the function of close button.
12. Test the password length.

When generating features from all classes, we must follow the rule that the same types of features should be numbered in the same way in all classes. For example, if we assign number 1, 2, and 3 to the features: close operation, size of form and title of form respectively, the same numbering must also be used for the same features in other classes. Similar features may occur in all the classes. Such as all the modules have some specific title, close button, size, font, color, location etc.

5.2 Related and Similar features:

In the section "Mapping", two words are introduced: 'related' and 'similar'. They are defined as follows:

Related: Two features are related if they are in the

- In the same class file.
  For example, let A be the feature about displaying a button and B be the feature about the boundaries and size of the button. As both the features are about displaying the same button, they are related each other. In the feature list, these types of features are kept closer by assigning a continuous numbers.

Similar:

Two features are similar if they have the same property.

Let A be the feature "display the title of a module" (e.g. LoginForm) and B be the feature "display the title of another module" (e.g. DefectsDisplay form). Since the features A and B both are about displaying the title of a module, they are said to be similar and are assigned the same numbers in a feature list. In my case, feature number two of all modules is about displaying the title of the different modules so they all are similar.

The table below shows some example of numbering of features in three modules. Similar features are assigned the same number and are kept in the same row but different columns and related features are assigned sequential numbers and are kept in the same column.

Table 5.1: Table showing properties of different modules

S.N	**LoginForm**	**FeatureDisplay**	**DefectsDisplay**
1	Size and position of the display form	Size and position of the display form	Size and position of the display form
2	Existence of title	Existence of title	Existence of title
3	Format of title displayed	Format of title displayed	Format of title displayed
4	Function of minimize button	Function of minimize button	Function of minimize button
5	Function of close button	Function of close button	Function of close button
6	Resizability of the form.	Resizability of the form.	Resizability of the form.

The similar features (chapter 5.2) in different modules are assigned the same number. This makes it meaningful to identify strip and block pattern. Feature 'n' in all the modules are similar so they contain same error. To make it clear let us take an example, if the "cancel" button of all the modules does not work, it is strip pattern and the developer can correct this feature in all modules, since they are assigned a same number in all modules. The error may be due to wrong selection of a key word or wrong selection of a library function. For example, the code like this does not work:

setDefaultCloseOperation(EXIT_ON_CLOSE);

The function closes all tabs at the same time but the customer may want to close a particular tab when pressing a close button. The following function works properly:

setDefaultCloseOperation(DISPOSE_ON_CLOSE);

In the ninth semester project report, only features in two dimensions were considered. That is, if we take a feature from a module as a test case, then if it is a defect we move up and down in the two dimensional input space to test the other features of the same class only. Other modules were not taken into consideration. For example, if the sixth feature of module number six is selected, then only the neighbors of the sixth feature in module number six were tested while the moduels that are the neighbors of module six was not taken into consideration. That is, no features of neighbor modules were tested. Now I have extended it to four directions. If a sixth feature of a module is selected, then, sixth feature of other modules (which are similar) are also tested in addition to 7th and 5th features of the same module. To illustrate this, consider the following table:

Table 5.2: Table showing the neighbors of a point

Modules→	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Features										
1	1,1	2,1	3,1	4,1	5,1	6,1	7,1	8,1	9,1	10,1
2	1,2	2,2	3,2	4,2	5,2	6,2	7,2	8,2	9,2	10,2
3	1,3	2,3	3,3	4,3	5,3	6,3	7,3	8,3	9,3	10,3
4	1,4	2,4	3,4	4,4	5,4	6,4	7,4	8,4	9,4	10,4
5	1,5	2,5	3,5	4,5	5,5	***6,5***	7,5	8,5	9,5	10,5
6	1,6	2,6	3,6	4,6	***5,6***	**6,6**	***7,6***	8,6	9,6	10,6
7	1,7	2,7	3,7	4,7	5,7	***6,7***	7,7	8,7	9,7	10,7
8	1,8	2,8	3,8	4,8	5,8	6,8	7,8	8,8	9,8	10,8
9	1,9	2,9	3,9	4,9	5,9	6,9	7,9	8,9	9,9	10,9
10	1,10	2,10	3,10	4,10	5,10	6,10	7,10	8,10	9,10	10,10
11	1,11	2,11	3,11	4,11	5,11	6,11	7,11	8,11	9,11	10,11
12	1,12	2,12	3,12	4,12	5,12	6,12	7,12	8,12	9,12	10,12

In Table 5.2, the sixth feature of module six is selected randomly and as it contains defects, we test the 6th feature of the neighbor module (fourth module and sixth module) and 7th feature and 5th feature of the same module. It is due to that the neighbors of the point (6,6) are (6,5), (6,7), (5,6) and (7,6).

5.3 Rules for mapping

To transform the real input of the software application into a two-dimensional space, the following needs to be done:

1. Select software application
   Decompose the software application into modules.
2. Analyze all features from each module. Modules are generated from the class files.
3. Identify relationships between features (see chapters 6.1 and 4.18).
4. According to the similarity and relativeness, assign different numbers are to the modules and features.
5. Populate test matrix from step 4 as shown in table 5.2.

5.4 Boundary test

A boundary is one of the borders of an input domain. We need the boundary of the input domain to determine the size of the input domain and to identify the pattern types of the failure causing input. In the case of a stripe pattern, the end points (see chapter 4.15a) of the failure region touch two opposite borders. Thus we always have to determine the border of the input domain. In case of a two dimensional space, we specify the number of modules and the numbers of features in terms of X and Y respectively which represent the borders. For example, assume a two dimensional table as below:

Table 5.3: Boundaries of application

	1	2	3	4	5	6	7	8	9	10
1										
2										
3										
4										
5										
6										
7										
8										
9										
10										
11										
12										

This domain is defined by four lines. Thus, these lines are the borders of this two dimensional input domain. There are two vertical and two horizontal lines.

In the example above, the two vertical lines are given by X=1 and X=10 and the two horizontal lines are given by Y=1 and Y=12.

To test whether the collected data is spread from one boundary to another, the boundary test is done for every point containing a defect. If 'm' is the number of modules and 'n' is the number of features then we define borders as:

$X = 1$, $X = m$, $Y = 1$ and $Y = n$

In my example, software tool, there are 10 modules and 12 features in each module(we can make more than 12 features in each module).

Therefore borders for this case are defined as:

X = 1 , X = 10 , Y = 1 and Y = 12

Let us assume that the following are the collected data in the array list DF[].

P(1,2), P(3,2), P(4,2), P(5,2), P(6,2), P(7,2), P(8,2), P(9,2), P(10,2)

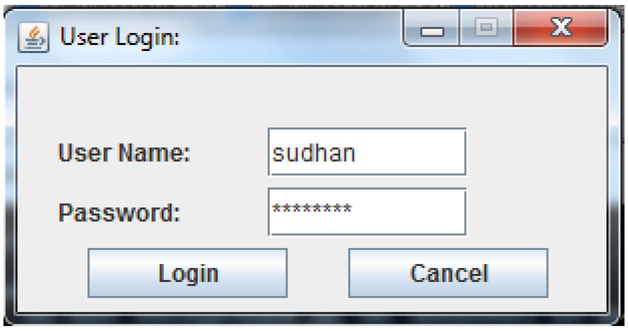
And if we take points P(1,2) and P(10,2), there are two points that satisfies the border condition: X = 1 and X = 10

As there are four border lines, we checked whether they belong to the same border or not. As X = 1 and X = 10 represents different borders, the collected data spread from one border to another and hence we assume that we have a strip pattern.

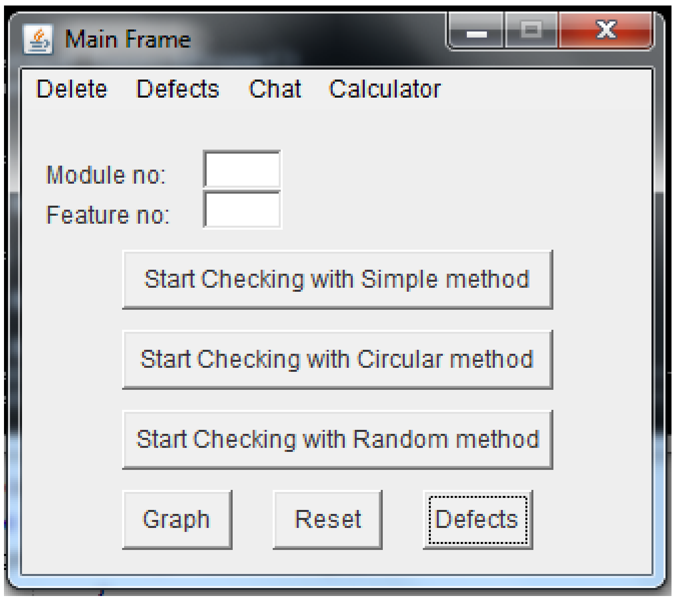
6 SCENARIO DESCRIPTION

For simulation of the algorithms described in chapter four, a real application is needed. I have used a software tool, which implements the three pattern idetification algorithms. In this software tool there are ten modules: LoginForm, MainFrame, NewDefect, AllDefects, Newfeature, FeatureDisplay, Calculator, Chatroom, Showgraph and ShowTestedpoints. Each module is implemented as a of a class file. For example, LoginForm module is implemented in a class file called "LoginForm.java".

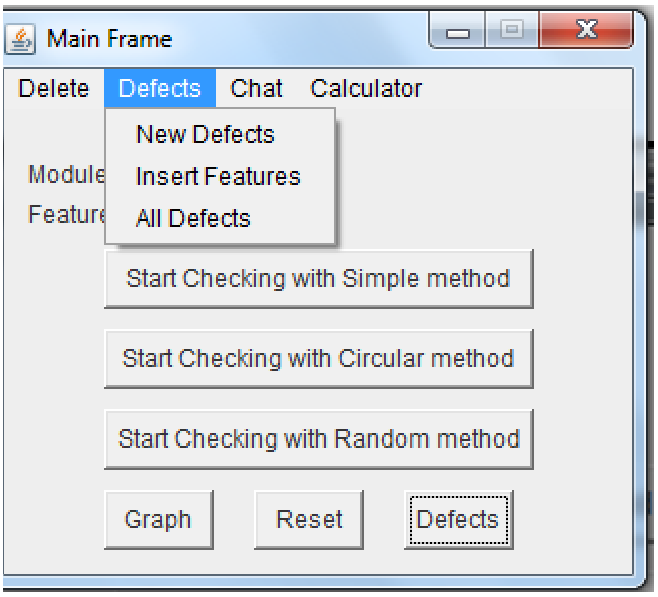
When the software tool is executed, a simple form is displayed as shown in figure 6.1a, where the user has to enter his username and password. When a user is authenticated, he/she is allowed to access other modules of the software tool. A main frame (figure 6.1b) is displayed when the username and password entered by the user are matched. In the main frame, a user can perform several activities (figure 6.1c). He can check the defect pattern of the application by using three different methods, delete all the seeded defects, insert a new feature to the database, chat with other users, use a calculator, and play games. While testing, using the three pattern identification algorithms, we have to insert defects ourselves using the module New Defects. When we click on the menu bar 'Defects' we get three options and the first option displays a form as shown in figure (figure 6.1c). When we select the menu item 'New Defects' a form with all the features is displayed as shown in figure 6.1e. We can seed a defect by clicking on the corresponding feature of a module. The following figures show the modules of the software tool.


User Login:
User Name:
sudhan
Password:
****
Login
Cancel

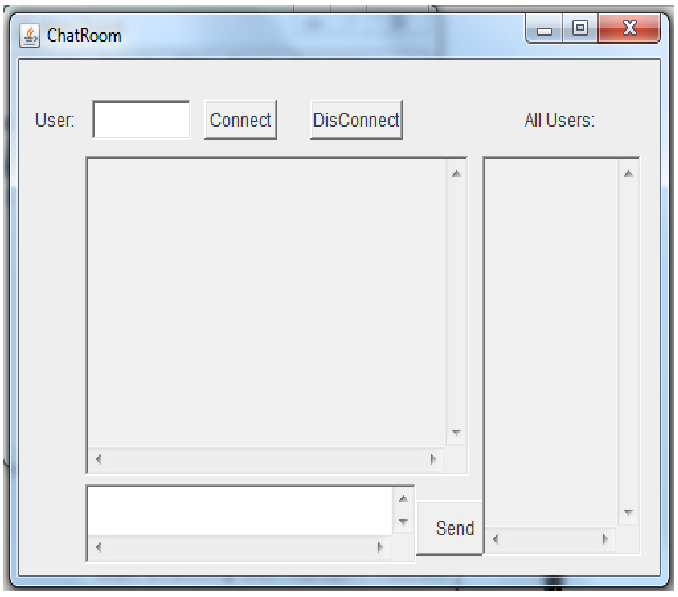
a


Main Frame
Delete
Defects
Chat
Calculator
Module no:
Feature no:
Start Checking with Simple method
Start Checking with Circular method
Start Checking with Random method
Graph
Reset
Defects

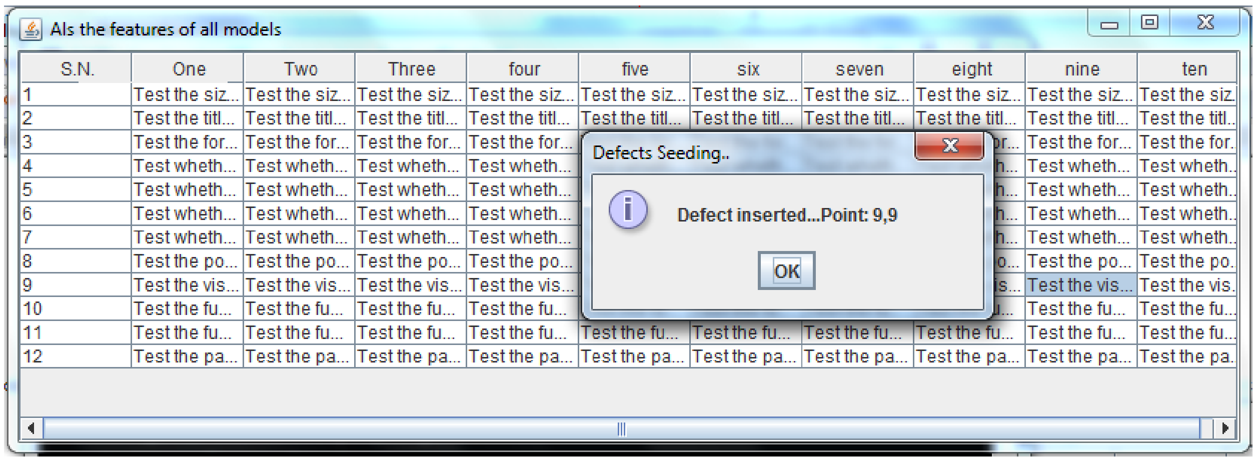
b


Main Frame
Delete
Defects
Chat
Calculator
New Defects
Insert Features
All Defects
Module
Feature
Start Checking with Simple method
Start Checking with Circular method
Start Checking with Random method
Graph
Reset
Defects

c


ChatRoom
User:
Connect
DisConnect
All Users:
Send

d


Als the features of all models

S.N.	One	Two	Three	four	five	six	seven	eight	nine	ten
1	Test the siz...	Test the siz...	Test the siz...	Test the siz...	Test the siz...	Test the siz...	Test the siz...	Test the siz...	Test the siz...	Test the siz.
2	Test the titl...	Test the titl...	Test the titl...	Test the titl...	Test the titl...	Test the titl...	Test the titl...	Test the titl...	Test the titl...	Test the titl.
3	Test the for...	Test the for...	Test the for...	Test the for...				r...	Test the for...	Test the for.
4	Test wheth...	Test wheth...	Test wheth...	Test wheth...				h...	Test wheth...	Test wheth.
5	Test wheth...	Test wheth...	Test wheth...	Test wheth...				h...	Test wheth...	Test wheth.
6	Test wheth...	Test wheth...	Test wheth...	Test wheth...				h...	Test wheth...	Test wheth.
7	Test wheth...	Test wheth...	Test wheth...	Test wheth...				h...	Test wheth...	Test wheth.
8	Test the po...	Test the po...	Test the po...	Test the po...				o...	Test the po...	Test the po.
9	Test the vis...	Test the vis...	Test the vis...	Test the vis...				s...	Test the vis...	Test the vis.
10	Test the fu...	Test the fu...	Test the fu...	Test the fu...				...	Test the fu...	Test the fu...
11	Test the fu...	Test the fu...	Test the fu...	Test the fu...	Test the fu...	Test the fu...	Test the fu...	Test the fu...	Test the fu...	Test the fu...
12	Test the pa...	Test the pa...	Test the pa...	Test the pa...	Test the pa...	Test the pa...	Test the pa...	Test the pa...	Test the pa...	Test the pa.

Defects Seeding..

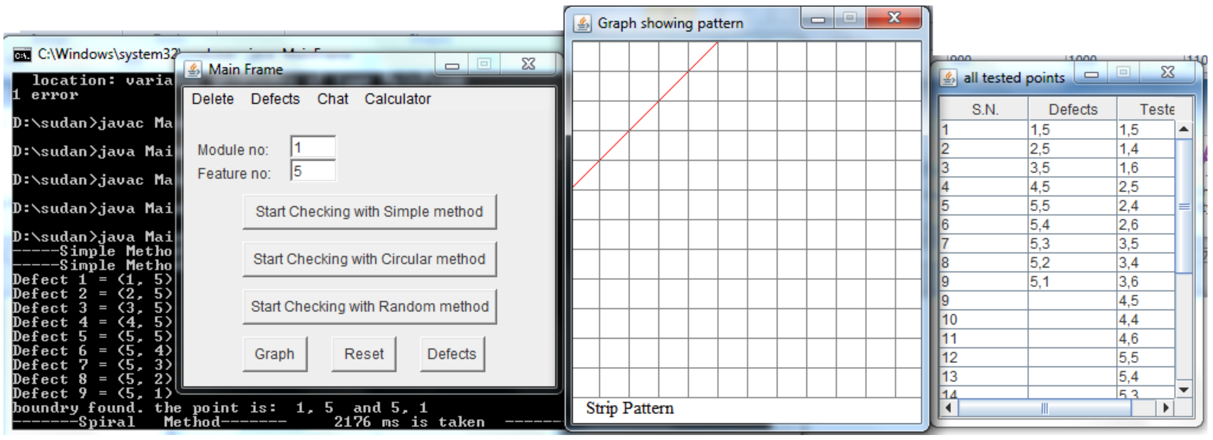
Defect inserted...Point: 9,9

OK

e

Figure 6.1: Different modules of software tool

Figure 6.2 shows a snapshot of simulation process for a point P(1,5) and Table 6.1 shows the all the seeded defect. When we hit "Start with Circular method" all the defects and detected points around point P(1,5) were collected in array lists DF[] and DT[] respectively and when we pressed the button "Graph", the pattern of the defect type was plotted on the graph as shown below.


C:\Windows\system32

```
  location: varia
1 error

D:\sudan>javac Ma

D:\sudan>java Mai

D:\sudan>javac Ma

D:\sudan>java Mai

D:\sudan>java Mai
-----Simple Metho
-----Simple Metho
Defect 1 = (1, 5)
Defect 2 = (2, 5)
Defect 3 = (3, 5)
Defect 4 = (4, 5)
Defect 5 = (5, 5)
Defect 6 = (5, 4)
Defect 7 = (5, 3)
Defect 8 = (5, 2)
Defect 9 = (5, 1)
boundry found. the point is:  1, 5  and 5, 1
-------Spiral   Method-------     2176 ms is taken  ------
```

Main Frame

Module no: 1
Feature no: 5

Delete
Defects
Chat
Calculator
Start Checking with Simple method
Start Checking with Circular method
Start Checking with Random method
Graph
Reset
Defects
Graph showing pattern
Strip Pattern

all tested points

S.N.	Defects	Teste
1	1,5	1,5
2	2,5	1,4
3	3,5	1,6
4	4,5	2,5
5	5,5	2,4
6	5,4	2,6
7	5,3	3,5
8	5,2	3,4
9	5,1	3,6
9		4,5
10		4,4
11		4,6
12		5,5
13		5,4
14		5,3

Figure 6.2: Snapshots showing plotting of pattern

Table 6.1: defects in database table

Modules→ / Features	1	2	3	4	5
1					defect
2					defect
3					defect
4					defect
5	defect	defect	defect	defect	defect

Although we can define as many features as we wish for each module, for the simplicity of the thesis, I have considered only twelve features for each module. As we have to test the properties of the modules, I have set 12 features for the LoginForm Module:

1. The size of the form.
2. The title of the form.
3. The format of the title.
4. Whether the minimize button work or not.
5. Whether the maximize button work or not.
6. Whether the close button work or not.
7. Whether the form is resizable or not.
8. The position of the cursor.
9. The visibility of password characters.
10. The function of login button.
11. The function of close button.
12. The password length.

Through feature number one, we test the size of the form to see whether it is according to the requirement or not. Through feature number two we test to see whether the title of the module exists or not. Similarly, through feature number twelve we test the length of the password is according to the requirement or not.

A use case diagram is a graphic depiction of the interactions among the elements of a user accesses all the modules of the software tool. In some modules (Login, FeatureDisplay, DefectDisplay, DeleteDefects and InsertFeatures) data are retrieved from the database – e.g. in login and defect seeding modules we have to access the database. Username and password given by the user are checked to the database and defects seeded are saved in the database.

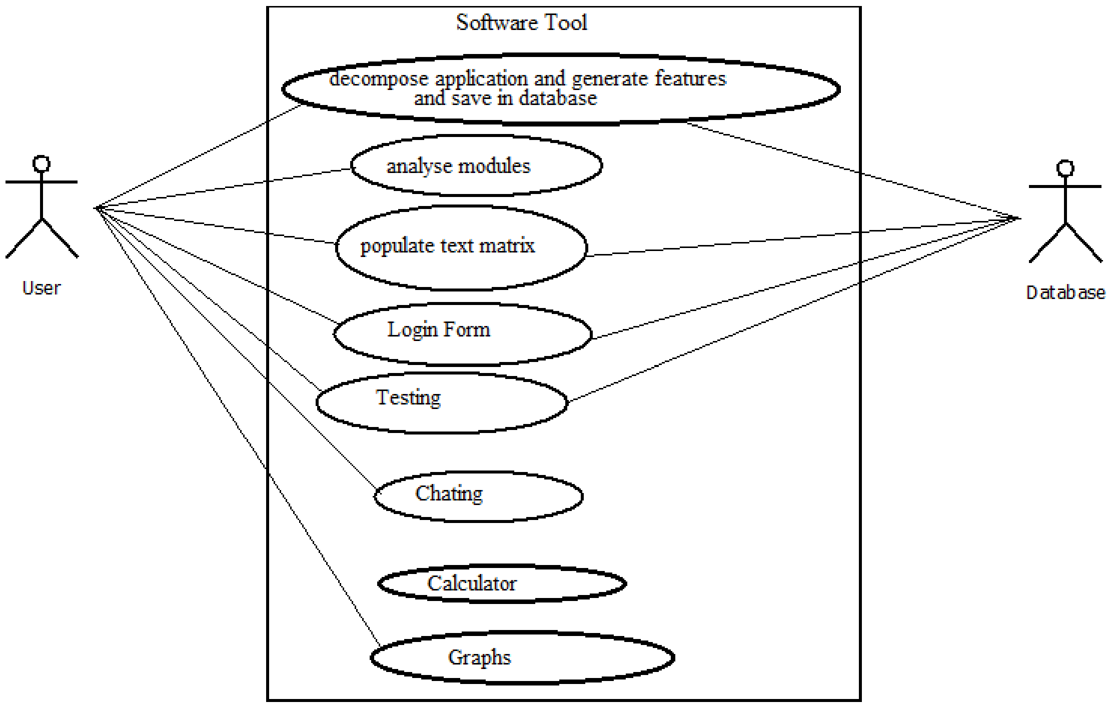


Figure 6.3: Use case diagram of the system

Sequence diagrams are used to create a scenario of events through one or more use cases. The objects associated with this series of events and the interactions between the objects are identified. These interactions are characterized by messages sent between the objects. Figure 6.4, shows a sequence diagram of the software tool I considered as an example.

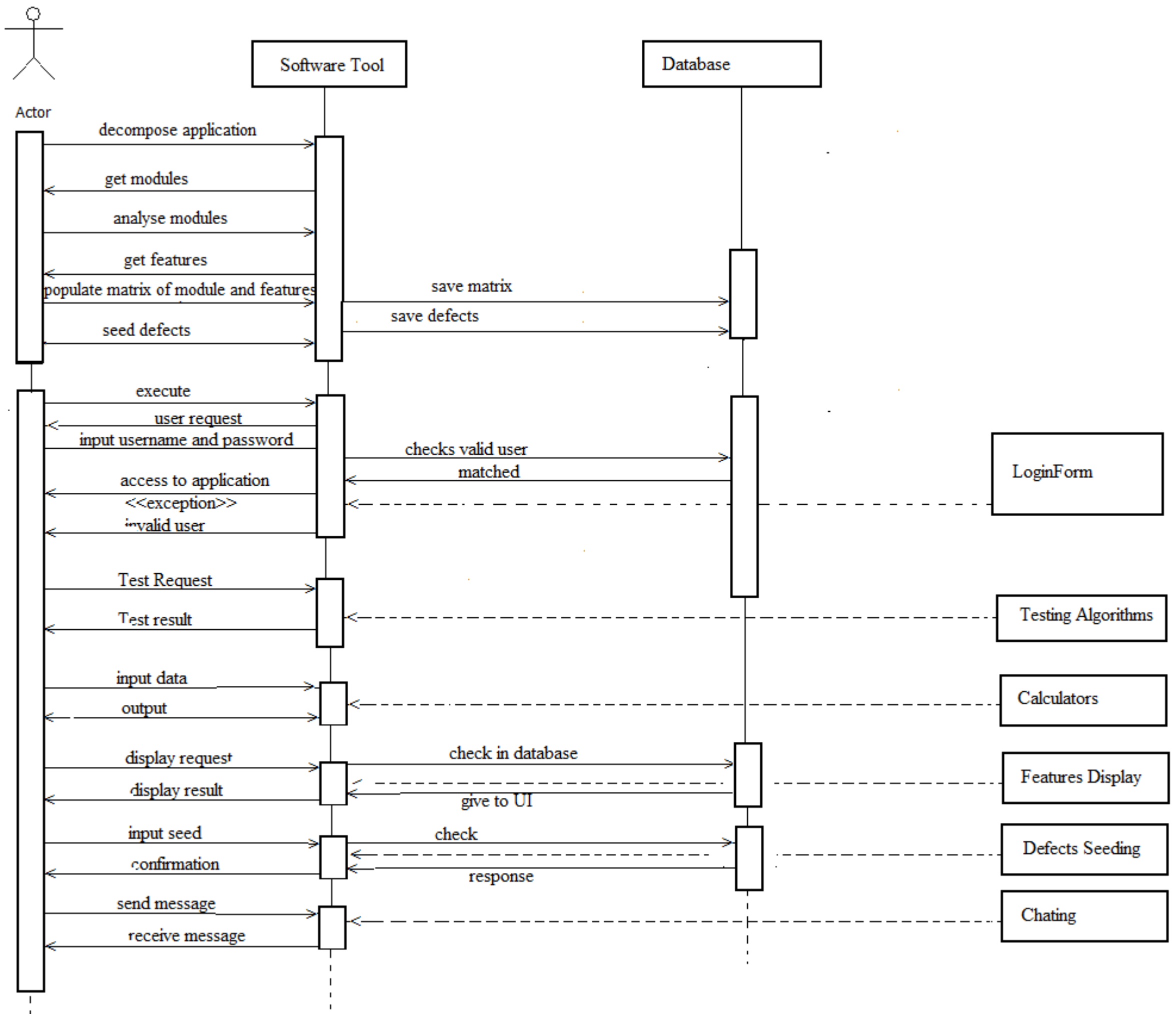


Figure 6.4: Sequence diagram of the system

6.1 Features with modules

Features describe the properties of the module that the customers want in their application. For example, while displaying a login form, a customer may want the password character to be invisible, the location of the form (either at the center or corner), the size of the form, look and feel style and etc. The following list represents the features of the LoginForm in my application.

1. The size of the form.
2. The title of the form.
3. The format of the title.
4. Whether the minimize button work or not.
5. Whether the maximize button work or not.

6. Whether the close button work or not.
7. Whether the form is resizable or not.
8. The position of the cursor.
9. The visibility of password characters.
10. The function of login button.
11. The function of close button.
12. The password length.

Similarly, every module contains features, which are implemented by the developer and through discussions with the customer. For a new application, before it is made, the analyst makes feature lists of all modules based on the customers' requirements and provides them to the developer who develops the application.

6.3 Step Size

In a software application, the input domain is usually large and testing each contiguous point is too expensive. This means that we need to dynamically change the step size for the pattern checking algorithms (in our case we have use 1 unit). We can start with fixing the step size to 1 unit and increase the value of the step size according to the density of the defects in the application or the available resources. If we find defects around a chosen defect, we can increase the step size to 2 units and if more defects are found to 3 units and so on. When we reach an error free feature we can reduce the value of step size until it computes exact size of the defect structure. For example, let us assume that the first defect be P(1,5). We then check both sides of P(1,5) - that is P(1,4) and P(1,3) - and then checks P(1,2) and P(1,6)- If every test case contains a defect then we can increase the step size value to 2. Then from P(1,6) we go to P(1,8) and P(1,9) followed by P(1,10) and P(1,12). If we still find errors in all of them then again we can increase the value of the step size, otherwise we should reduce the step size and go back to the point P(1,10) and we test the point with the reduced step size. This way we find the exact structure of the defect pattern.

In my case, the application is small - only 10 modules and 12 features in each module. There are altogether only 120 points in two-dimensional space. Thus, we do not need to increase the value of the step size. In large programs, however, where there may be more than 100 modules and more than hundred features in each module, the dynamic change of step size is important.

In large systems, we increase the value of step size in two conditions:

1. If the system has defect density high.

   This is done dynamically while running the software tool to test the pattern.

2. If we have few resources available and we need a quicker decision with a greater risk of being wrong, we can fix a larger step size at the beginning manually.

7 EXPERIMENTS

7.1 Defects Seeding

Defect seeding is the process of inserting errors into the program for the software to fail. It is also known as bebugging In defect seeding, a piece of the software is seeded with bugs that are similar to real defects.

The purpose of defect seeding is to find the unseeded defects while finding the seeded defects. It is done by inserting errors into a piece of software or by modifying the code of the program and executing the test set to see how many of the seeded bugs are discovered and how many new real defects are discovered. It's then possible to estimate the number of remaining defects by using some type of mathematical formula.

In our case, defects are seeded and then a test set is run to find the defects and the area covered by those defects. In this way the pattern of the defects can be computed.

An example of how errors are seeded is shown below. We can seed errors in any class of the application. If the class LoginForm is taken as an example, it displays a form when we execute the class file and contains twelve features. We can seed errors in any one of the features or in all the features also. The following are the questions that represent features of class LoginForm class.

1. The size of the form.
2. The title of the form.
3. The format of the title.
4. Whether the minimize button work or not.
5. Whether the maximize button work or not.
6. Whether the close button work or not.
7. Whether the form is resizable or not.
8. The position of the cursor.
9. The visibility of password characters.
10. The function of login button.
11. The function of close button.
12. The password length.

The code for this class file is as follows:

```
import javax.swing.*;
import java.awt.event.*;
import java.sql.*;
public class LoginForm extends JFrame implements ActionListener
{
```

```
JLabel lbluser,lblpass;
JTextField txtuser, txtpass;
//JPasswordField jpf;
JButton btnlogin;
public LoginForm()
{
        setLayout(null);
        lbluser=new JLabel("User Name:");
        lblpass=new JLabel("Password:");
        txtuser=new JTextField(20);
        //jpf=new JPasswordField(20);
        txtpass=new JTextField(20);
        btnlogin=new JButton("Login");
        add(lbluser);
        lbluser.setBounds(20,30,100,25);
        add(txtuser);
        txtuser.setBounds(125,30,100,25);
        add(lblpass);
        lblpass.setBounds(20,60,100,25);
        //add(jpf);
        //jpf.setBounds(125,60,100,25);
        add(txtpass);
        txtpass.setBounds(125,60,100,25);
        add(btnlogin);
        btnlogin.setBounds(75,90,100,25);
        setVisible(true);
        setSize(300,150);
        setLocation(250,250);
            setTitle("User Login:");
        setResizable(false);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        btnlogin.addActionListener(this);
```

Let us choose some features in order to seed defects. Feature number one describes the size and location of the display form. The customer may want the form to be displayed in the center of the screen and if it is in the corner, then it is an error. Feature two describes a proper title displayed on the form. Feature number six and ten describes the function of the close button and Login button respectively.

Let us seed defects to the features one, two, three, four and five. After the defects are seeded, these features will not behave according to the customer’s requirements. These features are coded in lines 32, 34, 36 and 37 respectively in the code snippet below. If we modify the number inside the parenthesis, these features do not behave according to the customer's requirements. After seeding the defects, these line look like below:

32. setSize(900,150); ——— appropriate size to the customer is (300 * 150)

33. setLocation(250,250);——— specifies the location of the form

34. setTitle("User Login///:"); ———— title given should be proper

35. setResizable(false); this function may be missing

36. setDefaultCloseOperation(EXIT_ON_CLOSE); ———— dispose should be written instead of exit

37. btnlogin.addActionListener(this); —— this function may be missing

This part of code is copied from the LoginForm.java along with the line numbers and this type of defect in code are assumed to be inserted in the database table.

7.2 Algorithms Tracing

When simulating the algorithm, some assumptions have to be made. Each feature of each module is arranged according to their similarity as explained in chapter 5.2, and the defects are seeded to see if the algorithm works properly. We keep on selecting and running test cases until a defect free point is found inside the frature. Simulation for all the algorithms is done for all the three types of defect patterns and for this simulation defects are seeded by the user for the testing purpose. The same defect pattern is used for all the algorithms.

7.2.1 Algorithm tracing for Simple distance computation method

In the Simple distance computation method, we first randomly select a point by generating two random numbers. The point is tested to see whether it contains error or not. If it contains error, the neighbors along X-axis are computed and tested first and the neighbors along Y-axis are tested. Otherwise another point is selected randomly. The algorithm used for the Circular method is given below:

Algorithm:

Step1: Locate randomly a point V(p) in the domain(D).

Step 2: Check if it is already detected.

If it is not detected mark point P as detected and put into array DT[]. Otherwise repeat step one.

Step 3: Check if the detected point contains a defect.

```
        If it is not a defect
        Return null
    If it is a defect
```

Mark it as defected and put that point into array defected DF[]

Step 4: Take one point from neighbor along X-axis and repeat the process from Step 2.

Step 5: Take one point from the neighbor of the initial point along Y-axis and repeat the process from Step 2.

7.2.1.1 Point pattern

For simulation purposes a point type pattern generated, first by seeding defect in only one feature in each module. We seeded defect into "LoginForm" and "FeatureDisplay" in feature number 6 and 4 respectively. In the two-dimensional form they are denoted P(1,6) and P(3,4) since the "LoginForm" module is module number 1 and the "FeatureDisplay" module was module number 3. The modules with the seeded defects are shown in the figure below marked as an asterisk.

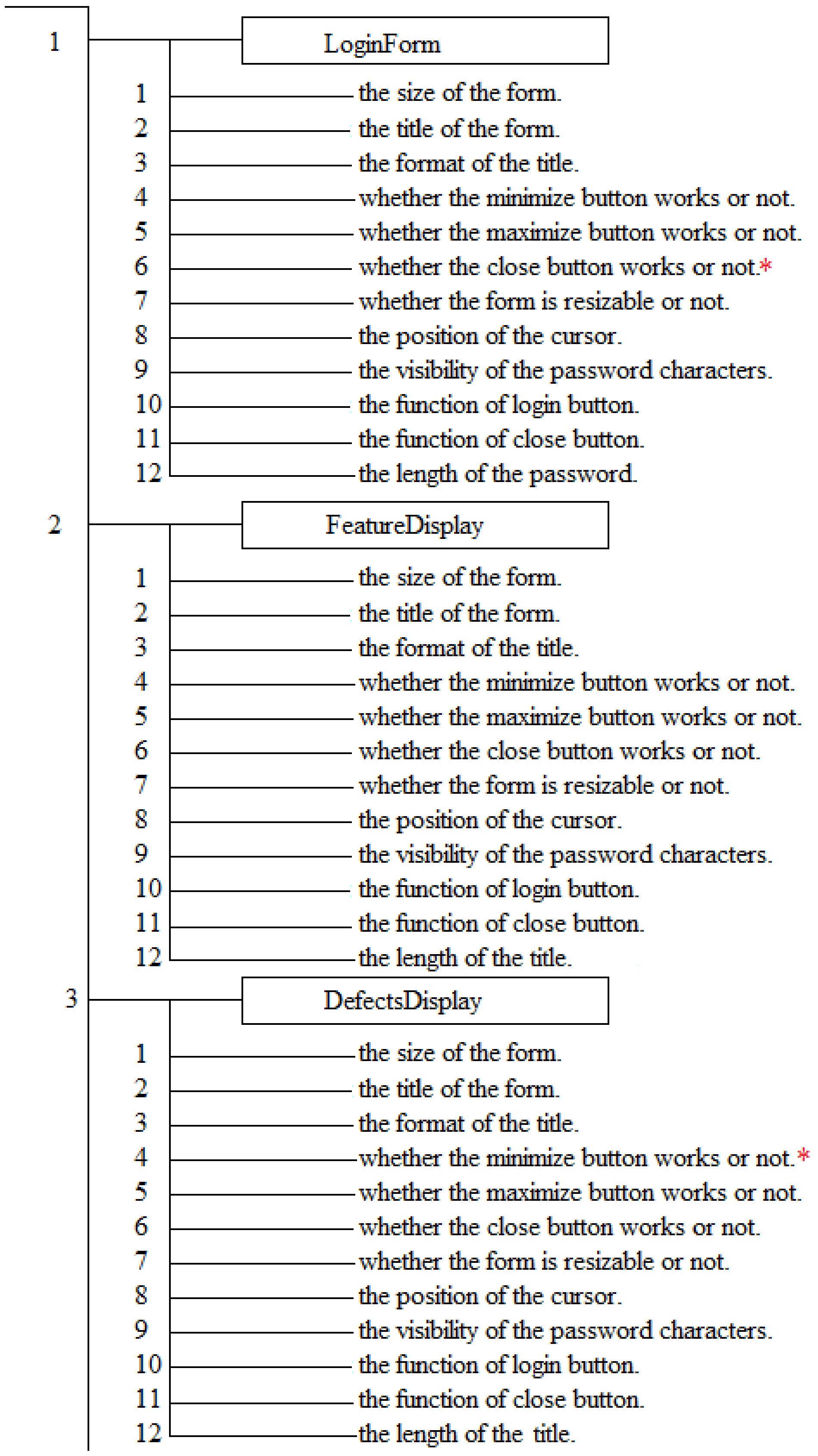
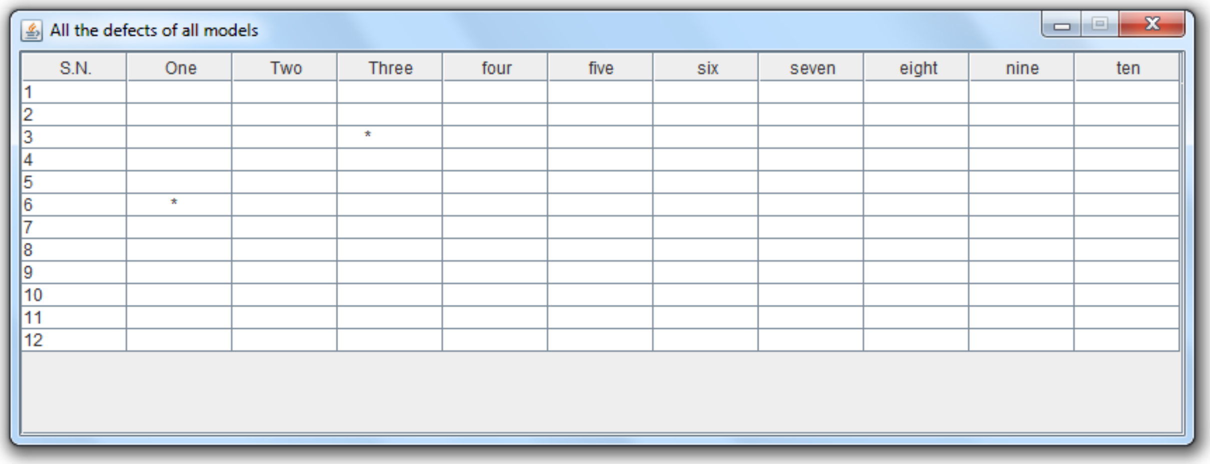

1
LoginForm
1 the size of the form.
2 the title of the form.
3 the format of the title.
4 whether the minimize button works or not.
5 whether the maximize button works or not.
6 whether the close button works or not.*
7 whether the form is resizable or not.
8 the position of the cursor.
9 the visibility of the password characters.
10 the function of login button.
11 the function of close button.
12 the length of the password.
2
FeatureDisplay
1 the size of the form.
2 the title of the form.
3 the format of the title.
4 whether the minimize button works or not.
5 whether the maximize button works or not.
6 whether the close button works or not.
7 whether the form is resizable or not.
8 the position of the cursor.
9 the visibility of the password characters.
10 the function of login button.
11 the function of close button.
12 the length of the title.
3
DefectsDisplay
1 the size of the form.
2 the title of the form.
3 the format of the title.
4 whether the minimize button works or not.*
5 whether the maximize button works or not.
6 whether the close button works or not.
7 whether the form is resizable or not.
8 the position of the cursor.
9 the visibility of the password characters.
10 the function of login button.
11 the function of close button.
12 the length of the title.

Figure 7.1: Point type defect seeding

When the "Start testing with Simple method" button of the main frame was pressed in the software tool, a point was randomly selected and tested to see whether it was an error containing feature or not by matching it with the data kept in table “AllDefects” in the database. In my case, after six trials the point P(1,6) was selected and depicted as a point type pattern since none of its neighbor contains an error. Details of the simulation are shown in appendix A.2.1.


All the defects of all models

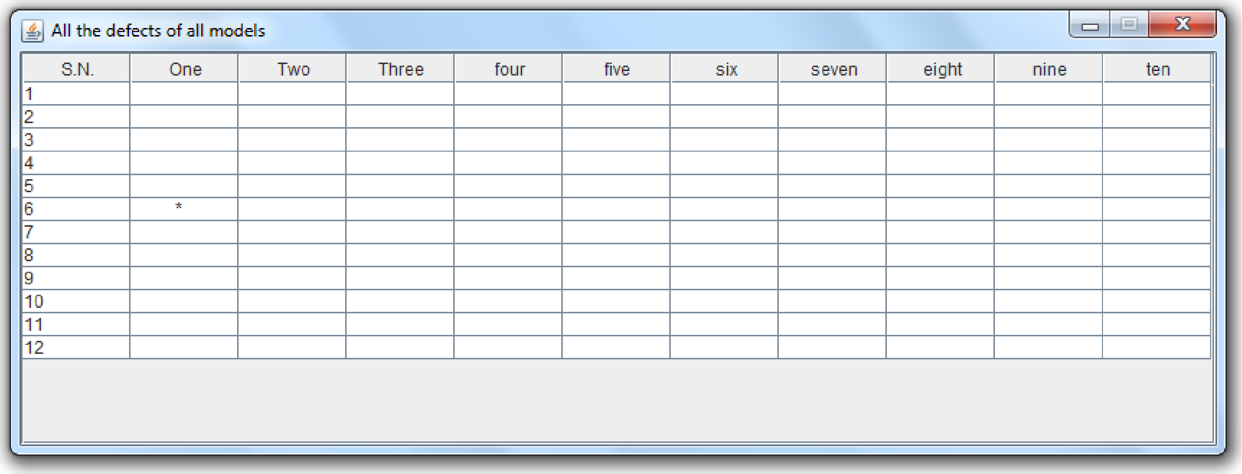
S.N.	One	Two	Three	four	five	six	seven	eight	nine	ten
1										
2										
3			*							
4										
5										
6	*									
7										
8										
9										
10										
11										
12										

Figure 7.2: Point type defect seeding in tabular form

Table 7.1: Point type defect seeding in real application (shows defects in sixth feature of “LoginForm” module and “FeatureDisplay” Module)

Modules→	LoginForm	DefectDisplay	FeatureDisplay	MainFrame						
Features										
1										
2										
3										
4			defect							
5										
6	defect									
7										
8										
9										
10										
11										
12										

Although the process was repeated ten times, the algorithm depicted the defect of only ‘LoginForm’ due to random selection. There was a defect-containing feature in “FeatureDisplay” also, which could not be depicted. This is a drawback of this algorithm. The solution to this drawback is to evenly spread the generation of test cases. An asterisk represents the location of the defect depicted by the algorithm.


All the defects of all models

S.N.	One	Two	Three	four	five	six	seven	eight	nine	ten
1										
2										
3										
4										
5										
6	*									
7										
8										
9										
10										
11										
12										

Figure 7.3: Point type defect detected by the algorithm

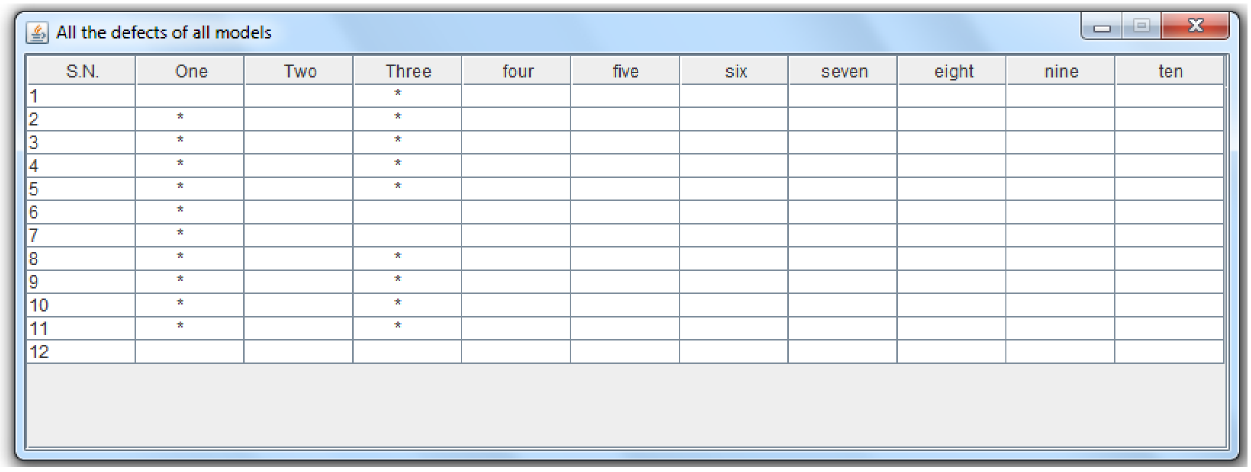
7.2.1.2 *Block pattern*

After simulation of the point pattern, the block type defects were seeded into the application. For this module "one", "two" and "three" were used. These were LoginForm, FeatureDisplay and DefectsDisplay respectively. Eight continuous (see section 4.13a) defects were seeded into module one, ten discontinuous defects (4.13b) were seeded in module "Three" and no defect were seeded in module "two" as shown in the figure 7.4. Defects in the "LoginForm" module were seeded in features two to eleven. In the two-dimensional form they are denoted as P(1,2), P(1,3), P(1,4), P(1,5), P(1,6) P(1,7), P(1,8), P(1,9), P(1,10), and P(1,11). Similarly, some discontinuous defects were seeded in Module "DefectsDisplay" as shown in the figure below.

1. LoginForm
    1. the size of the form.
    2. the title of the form. *
    3. the format of the title. *
    4. whether the minimize button works or not. *
    5. whether the maximize button works or not. *
    6. whether the close button works or not. *
    7. whether the form is resizable or not. *
    8. the position of the cursor. *
    9. the visibility of the password characters. *
    10. the function of login button. *
    11. the function of close button. *
    12. the length of the password.
2. FeatureDisplay
    1. the size of the form.
    2. the title of the form.
    3. the format of the title.
    4. whether the minimize button works or not.
    5. whether the maximize button works or not.
    6. whether the close button works or not.
    7. whether the form is resizable or not.
    8. the position of the cursor.
    9. the visibility of the password characters.
    10. the function of login button.
    11. the function of close button.
    12. the length of the title.
3. DefectsDisplay
    1. the size of the form. *
    2. the title of the form. *
    3. the format of the title. *
    4. whether the minimize button works or not. *
    5. whether the maximize button works or not. *
    6. whether the close button works or not.
    7. whether the form is resizable or not.
    8. the position of the cursor. *
    9. the visibility of the password characters. *
    10. the function of login button. *
    11. the function of close button. *
    12. the length of the title.

Figure 7.4: Block type defect seeding

When the "Start testing with Simple method" button of the main frame was pressed, a point was randomly selected and the tool tests whether it was an error containing feature or not by comparing the point with the data kept in table "AllDefects" in the database. In this case, in the 3rd trial, the point P(1, 6) was selected and in the fifth trial the point P(3, 4) was selected and all the error containing points around these points were collected in the array list and depicted as a block type pattern. Detail of the simulation is kept in appendix A.2.2.


All the defects of all models

S.N.	One	Two	Three	four	five	six	seven	eight	nine	ten
1			*							
2	*		*							
3	*		*							
4	*		*							
5	*		*							
6	*									
7	*									
8	*		*							
9	*		*							
10	*		*							
11	*		*							
12										

Figure 7.5: Block type defect seeding in tabular form

Table 7.2: Block pattern detected by the algorithm

Modules→	LoginForm	DefectDisplay	FeatureDisplay	MainFrame						
Features										
1			defect							
2			defect							
3			defect							
4			defect							
5			defect							
6										
7										
8										
9										
10										
11										
12										

After the completion of five attempts all the detected points and defect containing points were collected in the arrays DT[] and DF[] respectively. By performing boundary test (see chapter 5.4), the pattern type was identified. As the array DF[] did not contain points that contain two boundaries, and there were more than two defects containing points, we assumed that it was a block pattern. The geometric structure of the pattern is shown in the table below:

Table 7.3: Geometric structure of the defects.

Modules→	LoginForm	DefectDisplay	FeatureDisplay	MainFrame						
Features										
1			defect							
2			defect							
3			defect							
4			defect							
5			defect							
6										
7										
8										
9										
10										
11										
12										

There were defects in features number eight to eleven in model 'DefectsDisplay', which were not found by the algorithm. This is a drawback with this algorithm, which can be overcome by distributing the test cases uniformly using Adaptive Random testing methods (discussed in chapter 2).

7.2.1.3 Strip pattern

After simulation of point and block pattern, strip type defects were seeded into the application. For this module "one", "two" and "three" were used. Modules one, two and three were LoginForm, FeatureDisplay and DefectsDisplay respectively. Twelve continuous defects (see section 4.13a) were seeded to module one, ten discontinuous defects (see section 4.13b) were seeded to module "Three" and no defect was seeded in module "two" as shown in figure 7.6. Defects in the "LoginForm" module were seeded into features one to twelve. In the two-dimensional form they are denoted as P(1,2), P(1, 3), P(1, 4), P(1, 5), P(1, 6) P(1, 7), P(1, 8), P(1, 9), P(1, 10), P(1, 11), and P(1, 12). In the same way,Similarly nine discontinuous defects were seeded in Module "DefectsDisplay" as shown in the figure below.

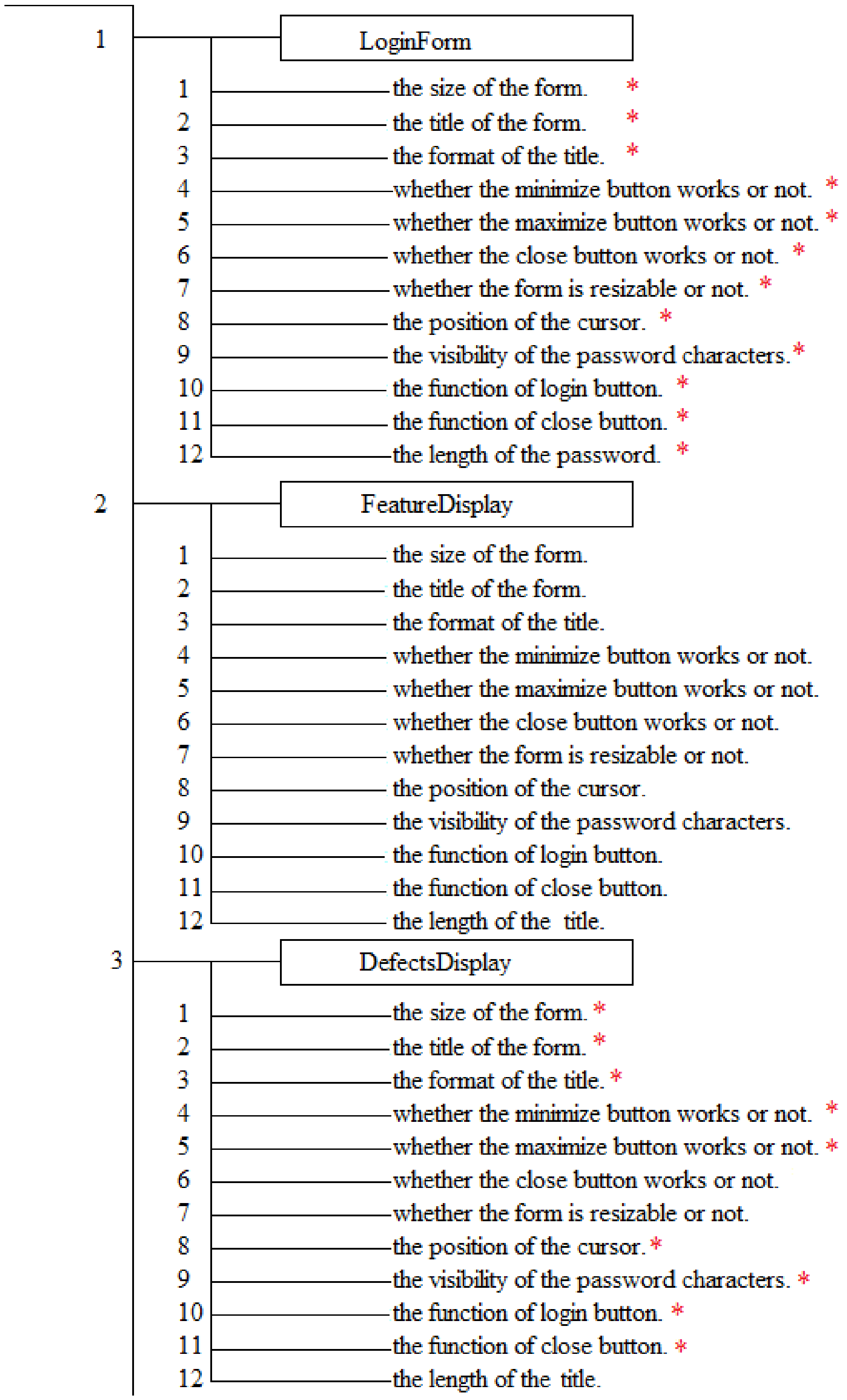
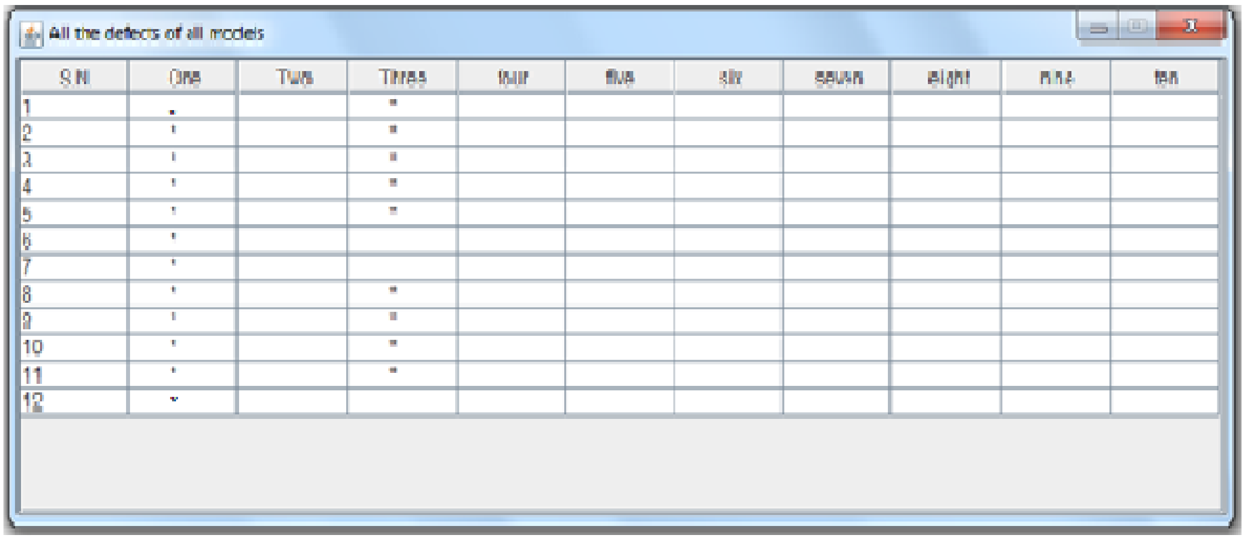

1
LoginForm
1 the size of the form. *
2 the title of the form. *
3 the format of the title. *
4 whether the minimize button works or not. *
5 whether the maximize button works or not. *
6 whether the close button works or not. *
7 whether the form is resizable or not. *
8 the position of the cursor. *
9 the visibility of the password characters. *
10 the function of login button. *
11 the function of close button. *
12 the length of the password. *
2
FeatureDisplay
1 the size of the form.
2 the title of the form.
3 the format of the title.
4 whether the minimize button works or not.
5 whether the maximize button works or not.
6 whether the close button works or not.
7 whether the form is resizable or not.
8 the position of the cursor.
9 the visibility of the password characters.
10 the function of login button.
11 the function of close button.
12 the length of the title.
3
DefectsDisplay
1 the size of the form. *
2 the title of the form. *
3 the format of the title. *
4 whether the minimize button works or not. *
5 whether the maximize button works or not. *
6 whether the close button works or not.
7 whether the form is resizable or not.
8 the position of the cursor. *
9 the visibility of the password characters. *
10 the function of login button. *
11 the function of close button. *
12 the length of the title.

Figure 7.6: Strip type defect seeding

When the "Start testing with Simple method" button of the main frame is pressed, a point is randomly selected each time and tested to see whether it is an error containing feature or not by matching it to the data kept in table "allDefects" in the database. In this case, in the first trial the point P(1, 6) was selected and in the fifth trial the point P(3, 5) was selected and all the error containing points around these points are collected in the array list and depicted as a block type pattern. Detail of the simulation is kept in appendix A.1.2.



All the defects of all models

S.N	One	Two	Three	four	five	six	seven	eight	nine	ten
1	*		*							
2	*		*							
3	*		*							
4	*		*							
5	*		*							
6	*									
7	*									
8	*		*							
9	*		*							
10	*		*							
11	*		*							
12	*									

Figure 7.7: Strip type defect seeding in tabular form

After five experiments, all the detected points and defect containing points are collected in the array lists DT[] and DF[] respectively. By performing a boundary test (see chapter 5.4), the pattern type was identified. As the array DF[] contains points that contain two boundaries (viz. P(1, 1) and P(1, 10)), it is a strip pattern. The geometric structure of the pattern is shown in the table below:

Table 7.4: Strip pattern detected by the algorithm.

Modules→ / Features	LoginForm	DefectDisplay	FeatureDisplay	MainFrame						
1	defect		defect							
2	defect		defect							
3	defect		defect							
4	defect		defect							
5	defect		defect							
6	defect									
7	defect									
8	defect									
9	defect									
10	defect									
11	defect									
12	defect									

The defects in features eight to eleven of module DefectsDisplay are not found. This is a drawback with this algorithm, which can be overcome by uniformly distributing the test cases using Adaptive Random testing methods (discussed in chapter 2).

7.2.2 Algorithm tracing for Circular method

In the Circular method, we first randomly select a point by generating two random numbers. The point is tested to see whether it contains error or not. If it contains an error, all the neighbors are computed and the process is repeated for each neighbors. Otherwise another point is selected randomly. The algorithm used for Circular method is given below:

Algorithm:

```
Step1:  Locate randomly a point V(p) in the domain(D).
Step 2: Check if it is already detected.
If it is not detected mark point P as detected and put into array DT[]. Otherwise repeat step one.
Step 3: Check if the detected point contains defect.

                If it is not a defect
                        Return null
                If it is a defect
                        Mark it as a defect and put that point into array defected DF[]
                        and  find its four neighbors.
Step 4. Repeat the process from Step 2 for all the neighbors.
```

7.2.2.1 Point pattern

For simulation purposes a point type pattern was assumed first by seeding a defect in only one feature in each module. We seeded defects into “LoginForm” and “FeatureDisplay” in features number 6 and 4 respectively. In the two-dimensional form they are denoted P(1, 6) and P(3, 4) since the “LoginForm” module is module number 1 and the “FeatureDisplay” module is module number 3. The modules with the seeded defects are shown in the figure below marked with an asterisk.

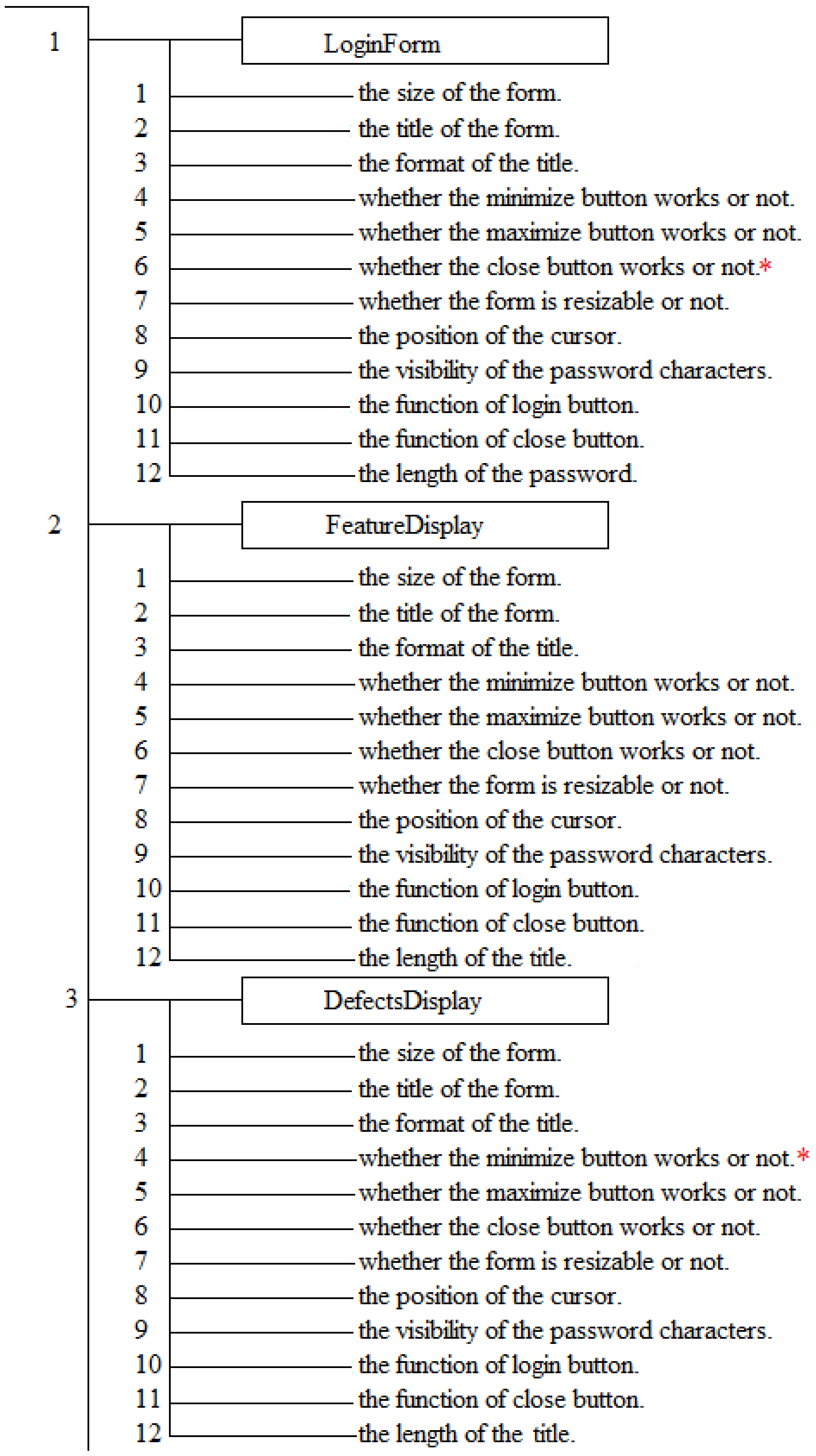
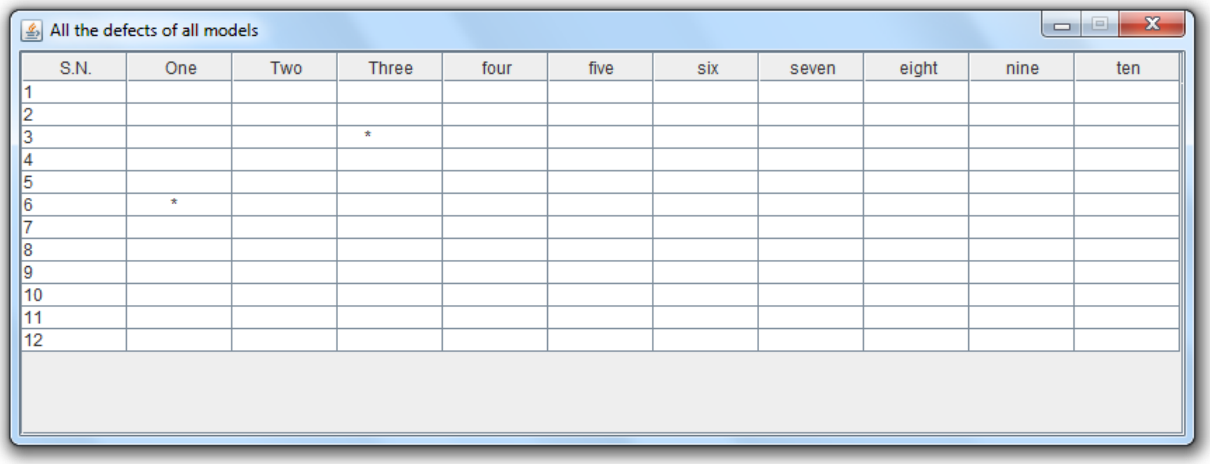

1
LoginForm
1 the size of the form.
2 the title of the form.
3 the format of the title.
4 whether the minimize button works or not.
5 whether the maximize button works or not.
6 whether the close button works or not.*
7 whether the form is resizable or not.
8 the position of the cursor.
9 the visibility of the password characters.
10 the function of login button.
11 the function of close button.
12 the length of the password.
2
FeatureDisplay
1 the size of the form.
2 the title of the form.
3 the format of the title.
4 whether the minimize button works or not.
5 whether the maximize button works or not.
6 whether the close button works or not.
7 whether the form is resizable or not.
8 the position of the cursor.
9 the visibility of the password characters.
10 the function of login button.
11 the function of close button.
12 the length of the title.
3
DefectsDisplay
1 the size of the form.
2 the title of the form.
3 the format of the title.
4 whether the minimize button works or not.*
5 whether the maximize button works or not.
6 whether the close button works or not.
7 whether the form is resizable or not.
8 the position of the cursor.
9 the visibility of the password characters.
10 the function of login button.
11 the function of close button.
12 the length of the title.

Figure 7.8: Point type defect seeding

When the "Start testing with Simple method" button of the main frame was pressed in the software tool, a point was randomly selected and tested to see whether it was an error containing feature or not by matching it with the data kept in table "AllDefects" in the database. In this case, after six trials, the point P(1, 6) was selected and depicted as a point type pattern since none of its neighbor contains an error. Details of the simulation are shown in appendix A.1.1.



All the defects of all models

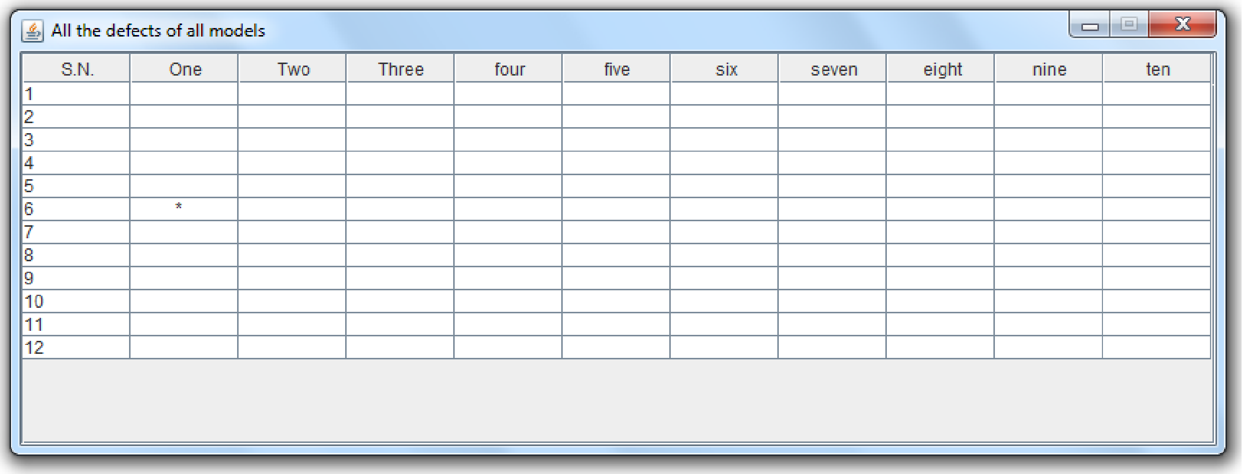
S.N.	One	Two	Three	four	five	six	seven	eight	nine	ten
1										
2										
3			*							
4										
5										
6	*									
7										
8										
9										
10										
11										
12										

Figure 7.9: Point type defect seeding in tabular form

Table 7.5: Point type defect seeding in real application(shows defects in sixth feature of "LoginForm" module and "FeatureDisplay" Module)

Modules→	LoginForm	DefectDisplay	FeatureDisplay	MainFrame						
Features										
1										
2										
3										
4			defect							
5										
6	defect									
7										
8										
9										
10										
11										
12										

Although the process was repeated ten times, the algorithm depicted the defect of only 'LoginForm'. There is one defect-containing feature in "FeatureDisplay", which could not be depicted. This is a drawback of this algorithm. The solution to this drawback is to evenly spread the generation of test cases (see chapter 2). An asterisk represents the location of the defect.



S.N.	One	Two	Three	four	five	six	seven	eight	nine	ten
1										
2										
3										
4										
5										
6	*									
7										
8										
9										
10										
11										
12										

Figure 7.10: Point type defect detected by the algorithm

7.2.2.2 Block pattern

After the simulation of point pattern, block type defects were seeded into the application. For this, module "one", "two" and "three" were use – LoginForm, FeatureDisplay and DefectsDisplay respectively. Eight continuous defects (see sction 4.13a) were seeded to module one, ten discontinuous defects (see section 4.13b) were seeded in module "Three" and no defects were seeded in module "two" as shown in the figure 7.11. Defects in the "LoginForm" module were seeded in features two to eleven. In the two-dimensional form they are denoted as P(1, 2), P(1, 3), P(1, 4), P(1, 5), P(1, 6) P(1, 7), P(1, 8), P(1, 9), P(1, 10), and P(1, 11). In the same way, nine discontinuous features were seeded in Module "DefectsDisplay" as shown in the figure below.

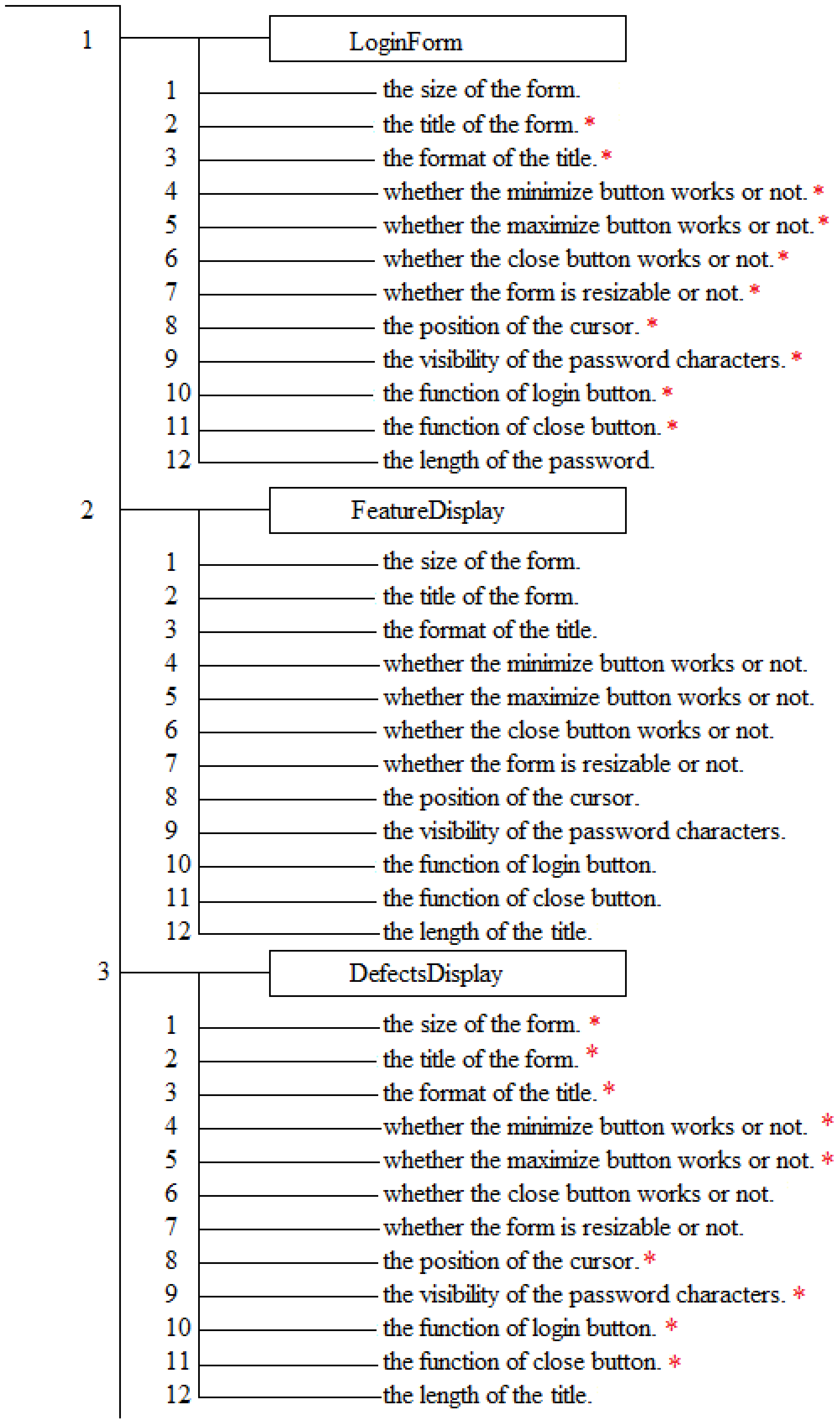
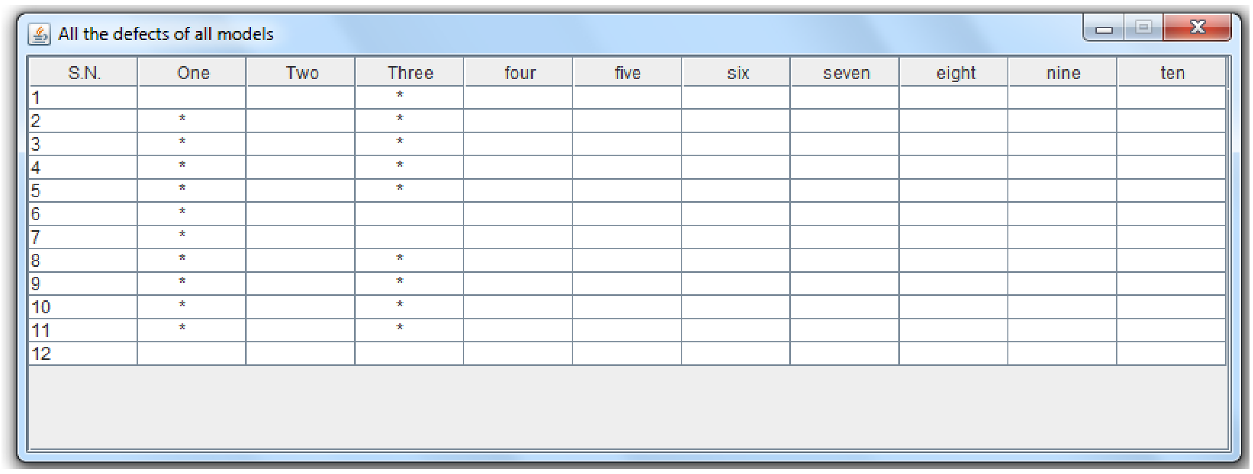

1
LoginForm
1 the size of the form.
2 the title of the form. *
3 the format of the title. *
4 whether the minimize button works or not. *
5 whether the maximize button works or not. *
6 whether the close button works or not. *
7 whether the form is resizable or not. *
8 the position of the cursor. *
9 the visibility of the password characters. *
10 the function of login button. *
11 the function of close button. *
12 the length of the password.
2
FeatureDisplay
1 the size of the form.
2 the title of the form.
3 the format of the title.
4 whether the minimize button works or not.
5 whether the maximize button works or not.
6 whether the close button works or not.
7 whether the form is resizable or not.
8 the position of the cursor.
9 the visibility of the password characters.
10 the function of login button.
11 the function of close button.
12 the length of the title.
3
DefectsDisplay
1 the size of the form. *
2 the title of the form. *
3 the format of the title. *
4 whether the minimize button works or not. *
5 whether the maximize button works or not. *
6 whether the close button works or not.
7 whether the form is resizable or not.
8 the position of the cursor. *
9 the visibility of the password characters. *
10 the function of login button. *
11 the function of close button. *
12 the length of the title.

Figure 7.11: Block type defect seeding

When the "Start testing with Simple method" button of the main frame was pressed in the software tool, a point was randomly selected and the tool tested whether it was an error containing feature or not by comparing to the data kept in table "AllDefects" in the database. In this case, in the 3rd trial, the point P(1, 6) was selected and in the fifth trial the point P(3, 4) was selected and all the error containing points around these points were collected in the array list and depicted as a block type pattern. Detail of the simulation is kept in appendix A.1.2.


All the defects of all models

S.N.	One	Two	Three	four	five	six	seven	eight	nine	ten
1			*							
2	*		*							
3	*		*							
4	*		*							
5	*		*							
6	*									
7	*									
8	*		*							
9	*		*							
10	*		*							
11	*		*							
12										

Figure 7.12: Block type defect seeding in tabular form

Table 7.6: Block pattern detected by the algorithm

Modules→	LoginForm	DefectDisplay	FeatureDisplay	MainFrame						
Features										
1			defect							
2			defect							
3			defect							
4			defect							
5			defect							
6										
7										
8										
9										
10										
11										
12										

After the completion of the five attempts, all the detected points and defect containing points were collected in the arrays DT[] and DF[] respectively. By performing boundary test (see chapter 5.4), the pattern type was identified. As the array DF[] does not contain points that contain two boundaries, and there are more than two defects containing points, we assumed it as block pattern. The geometric structure of the pattern is shown in the table below:

Table 7.7: Geometric structure of the defects

Modules→	LoginForm	DefectDisplay	FeatureDisplay	MainFrame						
Features										
1			defect							
2	defect		defect							
3	defect		defect							
4	defect		defect							
5	defect		defect							
6	defect									
7	defect									
8	defect									
9	defect									
10	defect									
11	defect									
12										

There were defects in features number eight to eleven in model ‘DefectsDisplay’, which were not found by the algorithm. This is a drawback with this algorithm which can be overcome by distributing the test cases uniformly using the Adaptive Random testing methods (see chapter 2).

7.2.2.3 Strip pattern

After simulation of point and block pattern, strip type defects were seeded into the application. For this, module “one”, “two” and “three” are used – LoginForm, FeatureDisplay and DefectsDisplay respectively. Twelve continuous defects (see section 4.13a) were seeded to module one, ten discontinuous defects (see section 4.13b) were seeded to module “Three” and no defect was seeded in module “two” as shown in figure 7.13. Defects in the “LoginForm” module were seeded in features one to twelve. In the two-dimensional form they are denoted as P(1, 2), P(1, 3), P(1, 4), P(1, 5), P(1, 6) P(1, 7), P(1, 8), P(1, 9), P(1, 10), P(1, 11), and P(1, 12). In the same way, nine discontinuous features were seeded in Module “DefectsDisplay” as shown in the figure below.

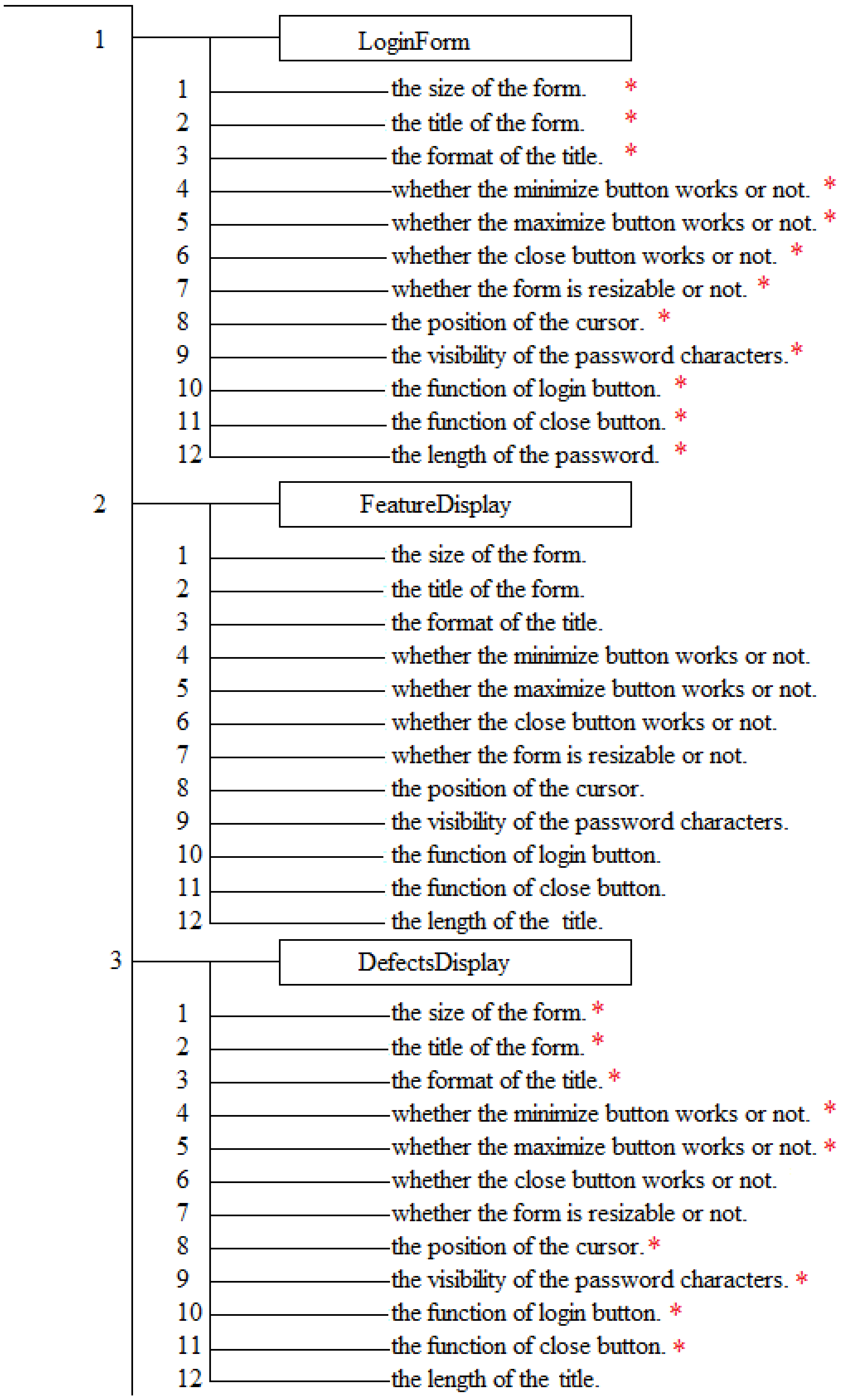
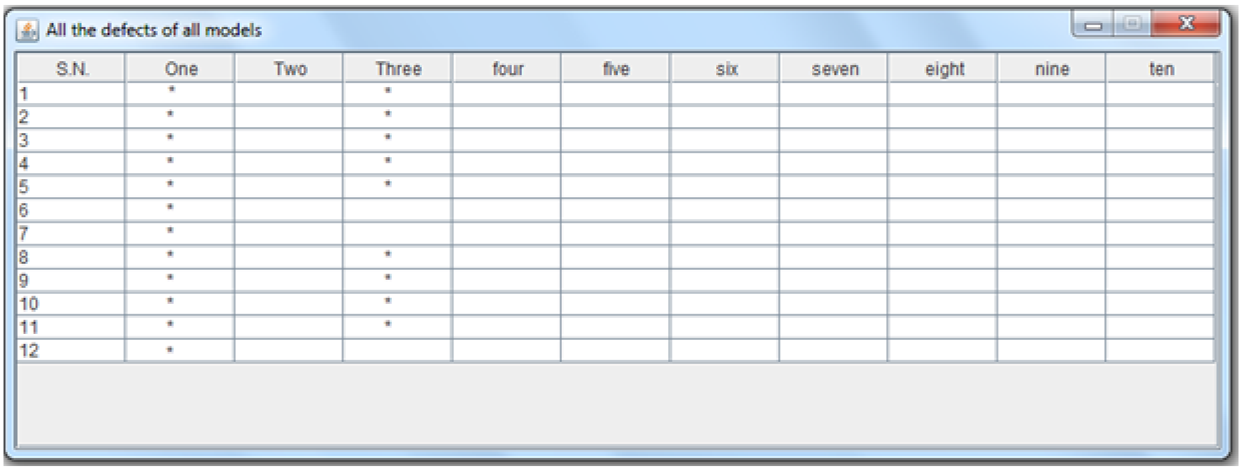

1
LoginForm
1 the size of the form. *
2 the title of the form. *
3 the format of the title. *
4 whether the minimize button works or not. *
5 whether the maximize button works or not. *
6 whether the close button works or not. *
7 whether the form is resizable or not. *
8 the position of the cursor. *
9 the visibility of the password characters. *
10 the function of login button. *
11 the function of close button. *
12 the length of the password. *
2
FeatureDisplay
1 the size of the form.
2 the title of the form.
3 the format of the title.
4 whether the minimize button works or not.
5 whether the maximize button works or not.
6 whether the close button works or not.
7 whether the form is resizable or not.
8 the position of the cursor.
9 the visibility of the password characters.
10 the function of login button.
11 the function of close button.
12 the length of the title.
3
DefectsDisplay
1 the size of the form. *
2 the title of the form. *
3 the format of the title. *
4 whether the minimize button works or not. *
5 whether the maximize button works or not. *
6 whether the close button works or not.
7 whether the form is resizable or not.
8 the position of the cursor. *
9 the visibility of the password characters. *
10 the function of login button. *
11 the function of close button. *
12 the length of the title.

Figure 7.13: Strip type defect seeding

When the "Start testing with Simple method" button of the main frame was pressed, a point was randomly selected each time and we checked whether it was an error containing feature or not by matching it to the data kept in table "AllDefects" in the database. In this case, in the first trial the point P(1, 6) was selected and in the fifth trial the point P(3, 5) was selected and all the error containing points around these points were collected in the array lists. Details of the simulation are shown in appendix A.1.2.


All the defects of all models

S.N.	One	Two	Three	four	five	six	seven	eight	nine	ten
1	*		*							
2	*		*							
3	*		*							
4	*		*							
5	*		*							
6	*									
7	*									
8	*		*							
9	*		*							
10	*		*							
11	*		*							
12	*									

Figure 7.14: Strip type defect seeding in tabular form

After five attempts all the detected points and defect containing points were collected in the array lists DT[] and DF[] respectively. By performing a boundary test (see chapter 5.4), the pattern type was identified. As the array DF[] contains points that contain two boundaries (viz. P(1, 1) and P(1, 10)), it was assumed to be strip pattern. The geometric structure of the pattern is shown in the figure below:

Table 7.8: Strip pattern detected by the algorithm

Modules→	LoginForm	DefectDisplay	FeatureDisplay	MainFrame						
Features										
1	defect	defect	defect							
2	defect		defect							
3	defect		defect							
4	defect		defect							
5	defect		defect							
6	defect									
7	defect									
8	defect									
9	defect									
10	defect									
11	defect									
12	defect									

The defects in features eight to eleven of models DefectsDisplay were not found. This is a drawback with this algorithm which can be overcome by uniformly distributing the test cases using Adaptive Random testing methods (see chapter 2).

7.2.3 Algorithm tracing for Heuristic method

In the Heuristic method, we first randomly select a point by generating random numbers. The point is checked to see whether it contains error or not. If the selected point contains an error all its end points (see 4.15a) are computed and checked to see whether the point contains an error. Otherwise another point is selected randomly. All the tested and defect containing points are put into the arrays DT[] and DF[] respectively and by performing a boundary test, (see section 5.4) the pattern type is identified. The algorithm used for the heuristic method is given below:

Algorithm:

Step1: Locate randomly a point V(p) in the domain(D).

Step 2: Check if it is already detected.

If it is not detected mark point P as detected and put into array DT[]. Otherwise repeat step one.

Step 3: Check if the detected point contains a defect.

```
If it is not a defect
        Return null
If it is a defect
        Mark it as a defect and put that point into array defected DF[]
```

Step 4: Find the four end points of point V(p).

Step 5: Test each end point to see whether it contains defects or not. If the tested point contains defect, put it in DF[] else put it in DT[].

7.2.3.1 Point pattern

For simulation purposes, a point type pattern was generated by first seeding a defect in only one feature in each module. We seeded defects into “LoginForm” and “FeatureDisplay” in feature number 6 and 4 respectively. In the two-dimensional form they are denoted P(1, 6) and P(3, 4) since the “LoginForm” module is noted as module number 1 and the “FeatureDisplay” module is module number 3. The modules with the seeded defects are shown in the figure 7.2.3.1a below, marked with an asterisk.

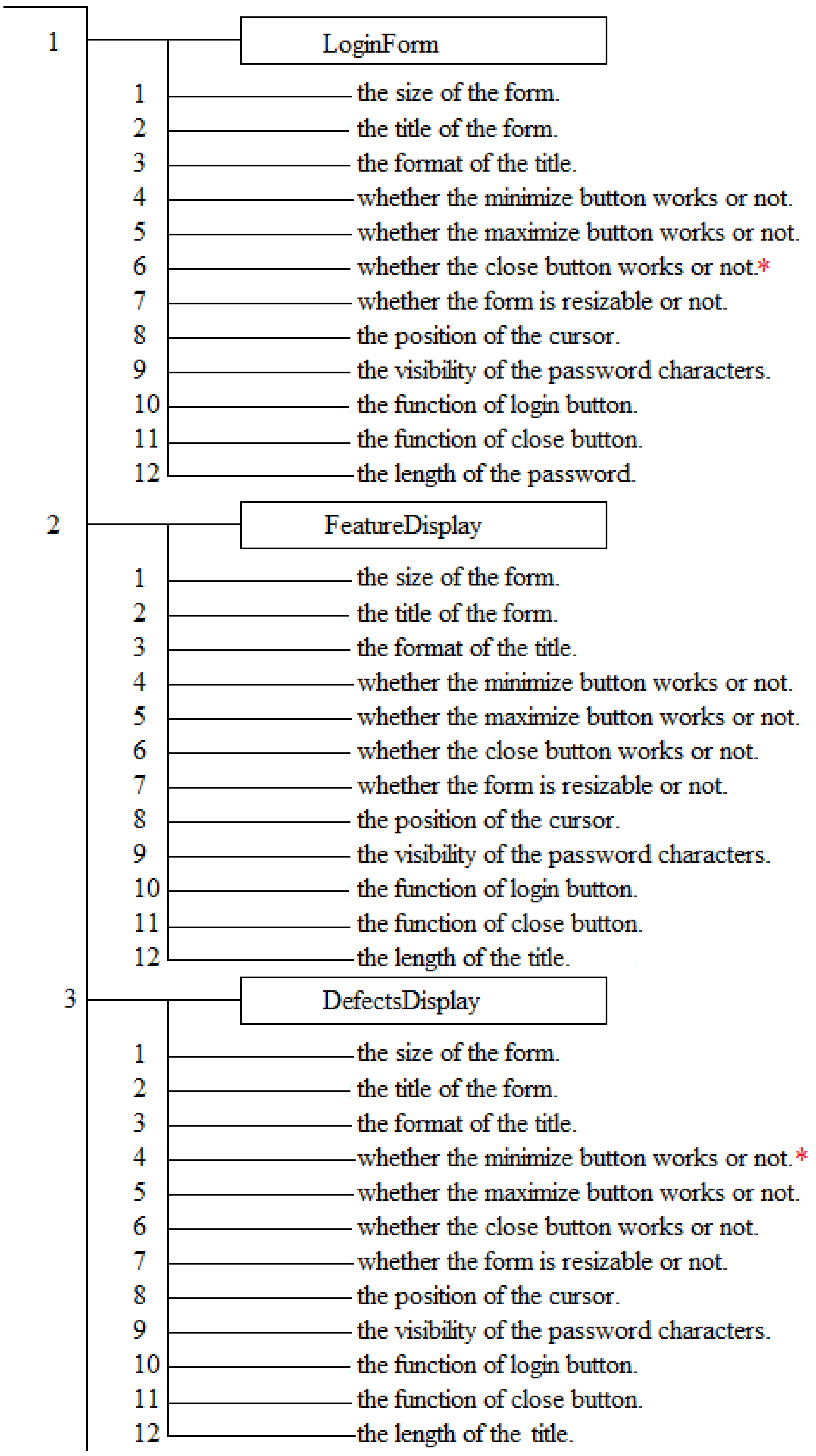
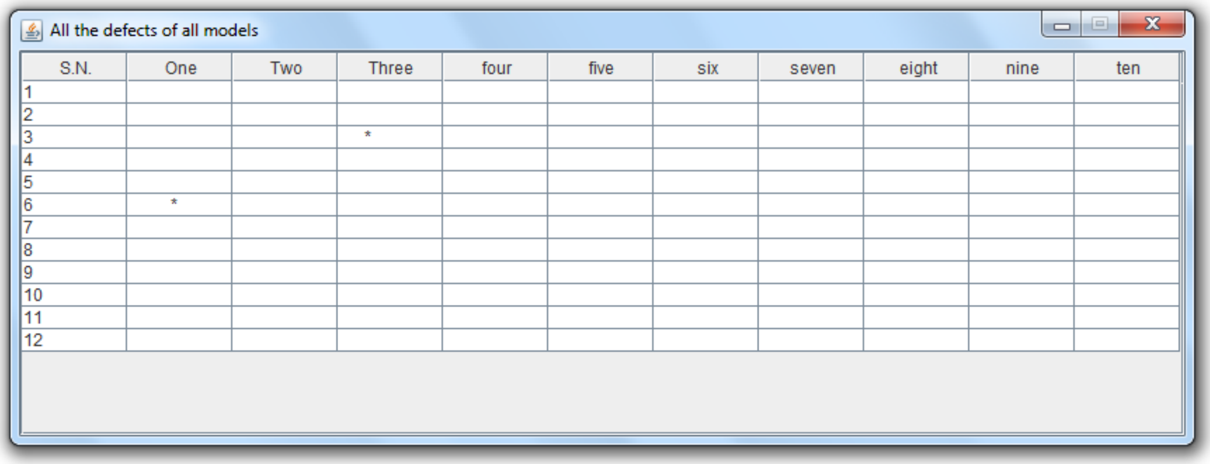

1
LoginForm
1 the size of the form.
2 the title of the form.
3 the format of the title.
4 whether the minimize button works or not.
5 whether the maximize button works or not.
6 whether the close button works or not.*
7 whether the form is resizable or not.
8 the position of the cursor.
9 the visibility of the password characters.
10 the function of login button.
11 the function of close button.
12 the length of the password.
2
FeatureDisplay
1 the size of the form.
2 the title of the form.
3 the format of the title.
4 whether the minimize button works or not.
5 whether the maximize button works or not.
6 whether the close button works or not.
7 whether the form is resizable or not.
8 the position of the cursor.
9 the visibility of the password characters.
10 the function of login button.
11 the function of close button.
12 the length of the title.
3
DefectsDisplay
1 the size of the form.
2 the title of the form.
3 the format of the title.
4 whether the minimize button works or not.*
5 whether the maximize button works or not.
6 whether the close button works or not.
7 whether the form is resizable or not.
8 the position of the cursor.
9 the visibility of the password characters.
10 the function of login button.
11 the function of close button.
12 the length of the title.

Figure 7.15: Point type defect seeding

When the "Start testing with Heuristic method" button of the main frame was pressed in the software tool, a point was randomly selected and we checked whether it was an error containing feature or not by matching it to the data kept in table "AllDefects" in the database. In this case after six trials, the point P(1, 6) was selected and depicted as a point type pattern since none of its end points contains an error. Details of the simulation are shown in appendix A.1.1.


All the defects of all models

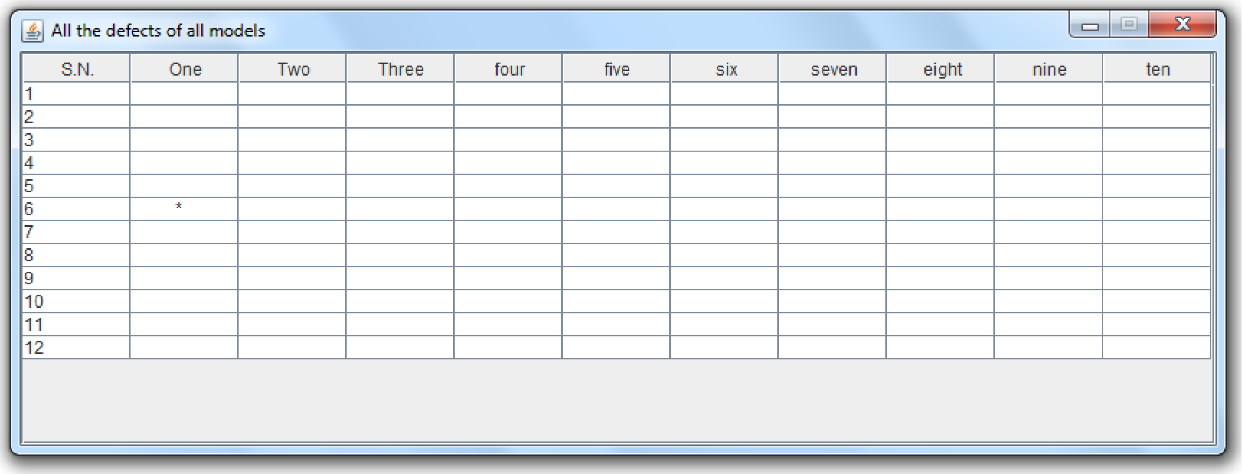
S.N.	One	Two	Three	four	five	six	seven	eight	nine	ten
1										
2										
3			*							
4										
5										
6	*									
7										
8										
9										
10										
11										
12										

Figure 7.16: Point type defect seeding in tabular form

Table 7.9: Point type defect seeding in real application(shows defects in sixth feature of "LoginForm" module and "FeatureDisplay" Module)

Modules →	LoginForm	DefectDisplay	FeatureDisplay	MainFrame						
Features										
1										
2										
3										
4			defect							
5										
6	defect									
7										
8										
9										
10										
11										
12										

Although the process was repeated for ten times, the algorithm depicted the defect of only 'LoginForm'. There was one defect-containing feature in "FeatureDisplay", which could not be depicted. This is a drawback of this algorithm. The solution to this drawback is to evenly spread the generation of test cases. An asterisk represents the location of the defect.



S.N.	One	Two	Three	four	five	six	seven	eight	nine	ten
1										
2										
3										
4										
5										
6	*									
7										
8										
9										
10										
11										
12										

Figure 7.17: Point type defect detected by the algorithm

7.2.3.2 *Block pattern*

After the simulation of the point pattern, block type defects were seeded into the application. For this module "one", "two" and "three" are used – LoginForm, FeatureDisplay and DefectsDisplay respectively. Eight continuous defects (see section 4.13a) are seeded into module one, ten discontinuous defects (see section 4.13b) were seeded in module "Three" and no defect was seeded in module "two" as shown in the figure 7.18. In the "LoginForm" module defects were seeded into features two to eleven. In the two-dimensional form they are denoted as P(1, 2), P(1, 3), P(1, 4), P(1, 5), P(1, 6) P(1, 7), P(1, 8), P(1, 9), P(1, 10), and P(1, 11). In the smae way nine discontinuous features are seeded into Module "DefectsDisplay" as shown in the figure below.

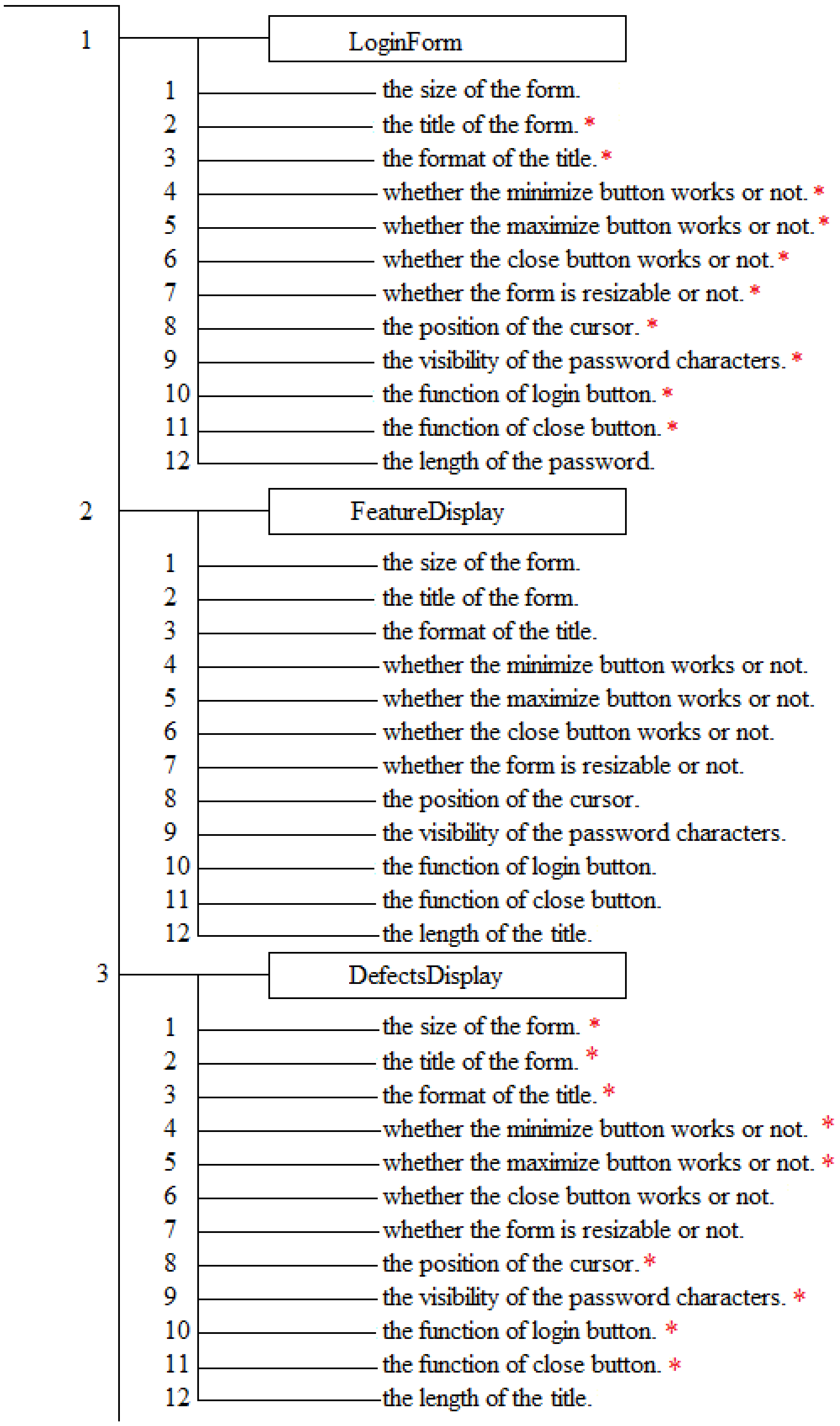
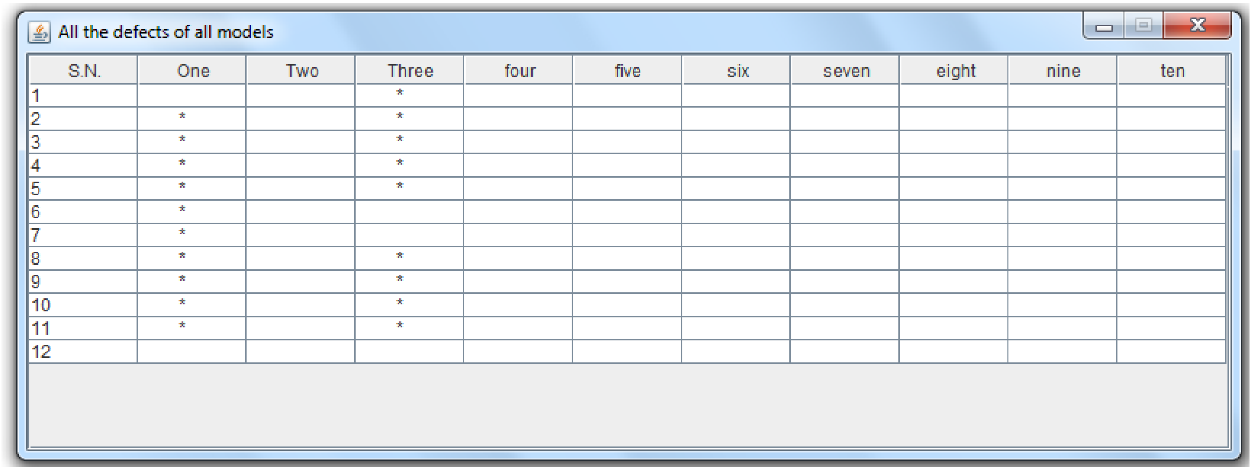

1
LoginForm
1 the size of the form.
2 the title of the form. *
3 the format of the title. *
4 whether the minimize button works or not. *
5 whether the maximize button works or not. *
6 whether the close button works or not. *
7 whether the form is resizable or not. *
8 the position of the cursor. *
9 the visibility of the password characters. *
10 the function of login button. *
11 the function of close button. *
12 the length of the password.
2
FeatureDisplay
1 the size of the form.
2 the title of the form.
3 the format of the title.
4 whether the minimize button works or not.
5 whether the maximize button works or not.
6 whether the close button works or not.
7 whether the form is resizable or not.
8 the position of the cursor.
9 the visibility of the password characters.
10 the function of login button.
11 the function of close button.
12 the length of the title.
3
DefectsDisplay
1 the size of the form. *
2 the title of the form. *
3 the format of the title. *
4 whether the minimize button works or not. *
5 whether the maximize button works or not. *
6 whether the close button works or not.
7 whether the form is resizable or not.
8 the position of the cursor. *
9 the visibility of the password characters. *
10 the function of login button. *
11 the function of close button. *
12 the length of the title.

Figure 7.18: Block type defect seeding

When the "Start testing with Heuristic method" button of the main frame was pressed in the software tool, a point was randomly selected and we tested whether it was an error containing feature or not by comparing it to the data in table "AllDefects" in the database. In this case, in the 3rd trial, the point P(2, 6) was selected and in the fifth trial the point P(3, 4) was selected and all the end points were computed. All the end points of point P(1, 6) were error free , so it was assumed to be a point pattern although it was block pattern. One of the end points of the point P(3, 4), that is P(3, 1), contained an error Thus, the points P(3, 1), P(3, 2), P(3, 3), P(3, 4) and P(3, 5) make up a block pattern. Detail of the simulation is shown in appendix A.1.2.


All the defects of all models

S.N.	One	Two	Three	four	five	six	seven	eight	nine	ten
1			*							
2	*		*							
3	*		*							
4	*		*							
5	*		*							
6	*									
7	*									
8	*		*							
9	*		*							
10	*		*							
11	*		*							
12										

Figure 7.19: Block type defect seeding in tabular form

Table 7.10: Block pattern detected by the algorithm

Modules→	LoginForm	DefectDisplay	FeatureDisplay	MainFrame						
Features										
1			defect							
2			defect							
3			defect							
4			defect							
5			defect							
6										
7										
8										
9										
10										
11										
12										

After the completion of five run, all the detected points and defect containing points were collected in the arrays DT[] and DF[] respectively. By performing boundary test (see chapter 5.4), the pattern type was identified. As the array DF[] does not contain points that contain two boundaries, and there were more than two defect containing points, we assumed that this pattern was a block pattern. The geometric structure of the pattern is shown in the table below:

Table 7.11: Geometric structure of the defects

Modules→	LoginForm	DefectDisplay	FeatureDisplay	MainFrame						
Features										
1			defect							
2			defect							
3			defect							
4			defect							
5			defect							
6	defect									
7										
8										
9										
10										
11										
12										

There were defects in features number eight to eleven in module 'DefectsDisplay', which were not found by the algorithm and also no defects of module one was found. This is a drawback with this algorithm which can be overcome by distributing the test cases uniformly using Adaptive Random testing methods (see chapter 2). According to this algorithm, block pattern whose end points do not contain defect are assumed to be point patterns. In above figure 7.19 points P(1, 2) to P(1, 11) were assumed to be a point pattern although it was a block pattern.

7.2.3.3 Strip pattern

After simulation of point and block pattern, strip pattern defects were seeded into the application. For this module "one", "two" and "three" were used - LoginForm, FeatureDisplay and DefectsDisplay respectively. Twelve continuous defects (see section 4.13a) were seeded into module one, ten discontinuous defects (see section 4.13b) were seeded into module "Three" and no defect was seeded into module "two" as shown in figure 7.20. Defects in the "LoginForm" module were seeded in features one to twelve. In the two-dimensional space they are denoted as P(1, 2), P(1, 3), P(1, 4), P(1, 5), P(1, 6) P(1, 7), P(1, 8), P(1, 9), P(1, 10), P(1, 11), and P(1, 12). In the same way, nine discontinuous features were seeded in Module "DefectsDisplay" as shown in the figure below.

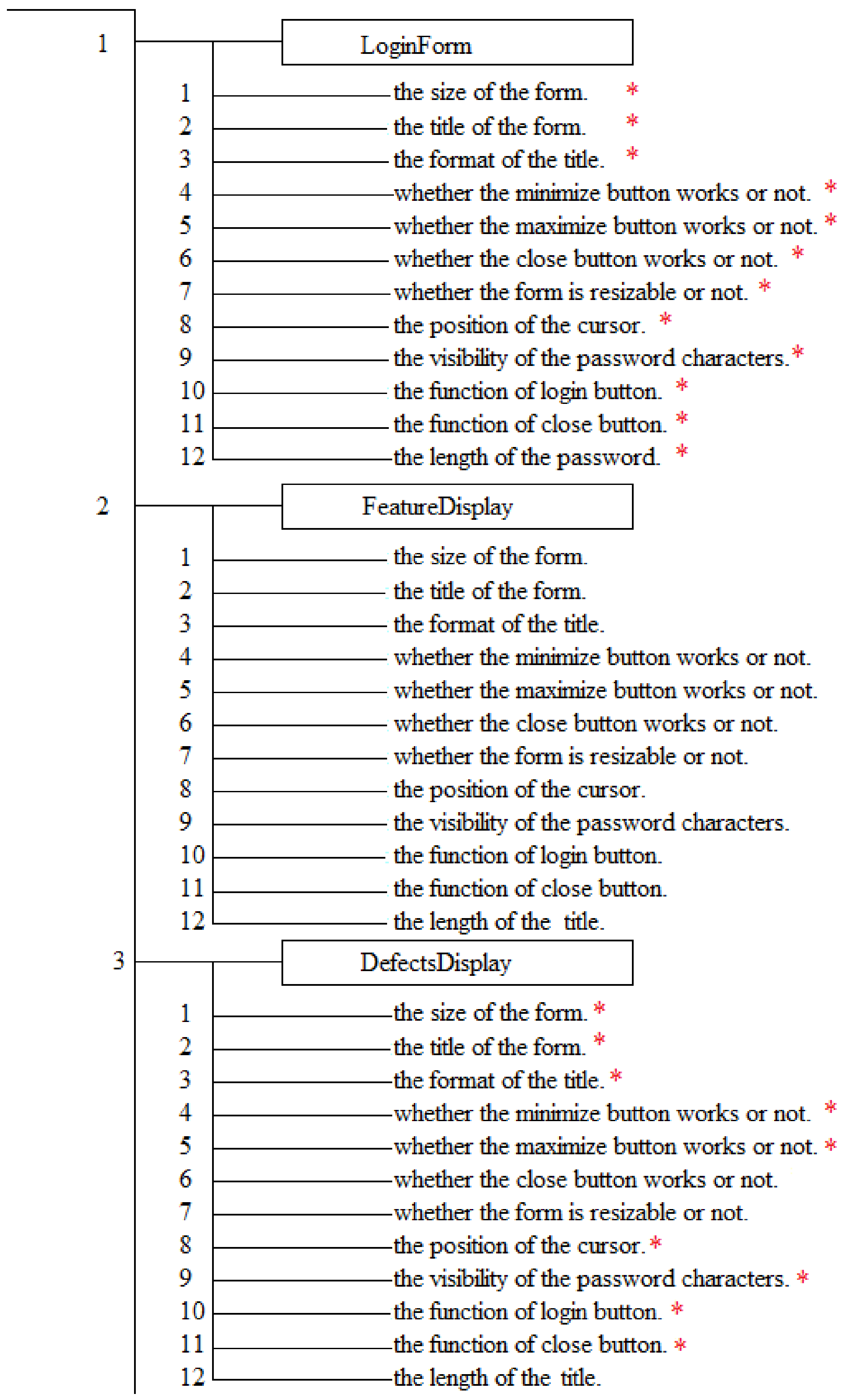
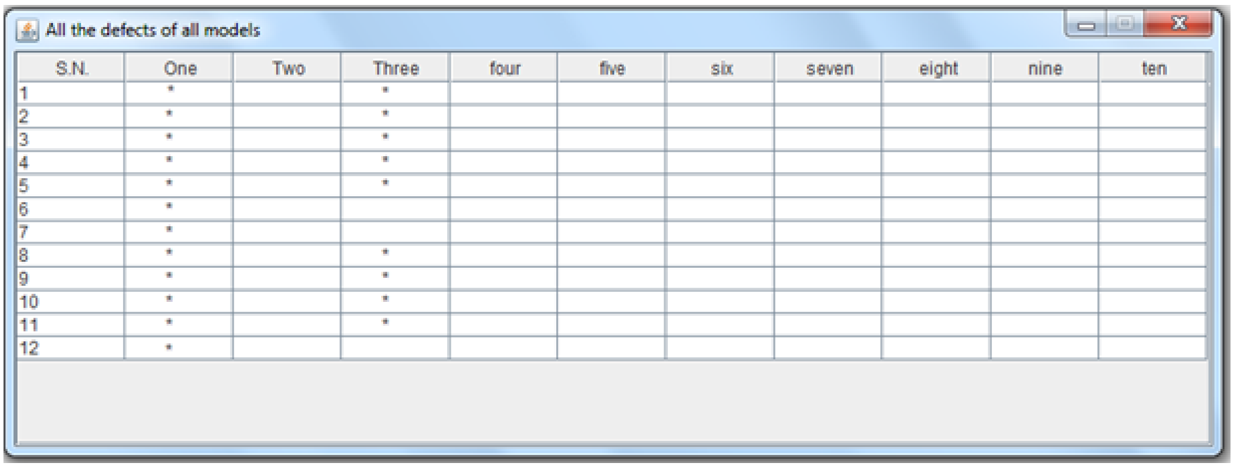

1
LoginForm
1 the size of the form. *
2 the title of the form. *
3 the format of the title. *
4 whether the minimize button works or not. *
5 whether the maximize button works or not. *
6 whether the close button works or not. *
7 whether the form is resizable or not. *
8 the position of the cursor. *
9 the visibility of the password characters. *
10 the function of login button. *
11 the function of close button. *
12 the length of the password. *
2
FeatureDisplay
1 the size of the form.
2 the title of the form.
3 the format of the title.
4 whether the minimize button works or not.
5 whether the maximize button works or not.
6 whether the close button works or not.
7 whether the form is resizable or not.
8 the position of the cursor.
9 the visibility of the password characters.
10 the function of login button.
11 the function of close button.
12 the length of the title.
3
DefectsDisplay
1 the size of the form. *
2 the title of the form. *
3 the format of the title. *
4 whether the minimize button works or not. *
5 whether the maximize button works or not. *
6 whether the close button works or not.
7 whether the form is resizable or not.
8 the position of the cursor. *
9 the visibility of the password characters. *
10 the function of login button. *
11 the function of close button. *
12 the length of the title.

Figure 7.20: Strip type defect seeding

When the "Start testing with Heuristic method" button of the main frame was pressed, a point was randomly selected and checked to see whether it was an error containing feature or not by matching it to the data kept in table "AllDefects" in the database. In this case, in the first trial the point P(1, 6) was selected and in the fifth trial the point P(3,5) was selected and all the error containing points around these points were collected in the array list and depicted as a strip pattern. Details of the simulation are shown in appendix A.1.2.



All the defects of all models

S.N.	One	Two	Three	four	five	six	seven	eight	nine	ten
1	*		*							
2	*		*							
3	*		*							
4	*		*							
5	*		*							
6	*									
7	*									
8	*		*							
9	*		*							
10	*		*							
11	*		*							
12	*									

Figure 7.21: Strip type defect seeding in tabular form

After five attempts all the detected points and defect containing points were collected in the array lists DT[] and DF[] respectively. By performing a boundary test (see chapter 5.4), the pattern type was identified. As the array DF[] contains points that contain two boundaries (viz. P(1, 1) and P(1, 12)), it was assumed to be a strip pattern. The geometric structure of the pattern is shown in the table below:

Table 7.12: Strip pattern detected by the algorithm

Modules→	LoginForm	DefectDisplay	FeatureDisplay	MainFrame						
Features										
1	defect		defect							
2	defect		defect							
3	defect		defect							
4	defect		defect							
5	defect		defect							
6	defect									
7	defect									
8	defect									
9	defect									
10	defect									
11	defect									
12	defect									

The defects in features eight to eleven of models DefectsDisplay were not found. This is a drawback with this algorithm which can be overcome by uniformly distributing the test cases using Adaptive Random testing methods (see chapter 2).

8 ANALYSIS AND DISCUSSION

For analysis and evaluation, we should answer the research questions.

RQ1- How is textual data transformed to numeric- data?

The software application is first divided into modules (see chapter 4.11a) and several features (see chapter 4.11b) and by the process of mapping (chapter 5), textual information of a software application is transformed to numeric data.

RQ2: What is the most efficient strategy for identifying pattern type?

Among the three algorithms, the Circular method explores all the points around a defect containing point while the remaining algorithms do not test all the neighbors of a defect-containing point. Thus, the Circular method is the most efficient and smart method for identifying failure pattern.

RQ3: How does the efficient algorithm works with non-numeric data?

By the process of mapping (see chapter 5), we can transform a real application into a two-dimensional input space and a test matrix as shown in figure 5.1.3. Thus, all the algorithms can use the numeric data, which is generated, in the test matrix.

RQ4: What is the best way to determine the optimal step size?

The value of step size depends on the density of defects in the application and the risk you are willing to take when making a conclusion. If the density of defects is high we can increase the value of step size and small step size for low density defect containing application (discussed in chapter 6.2). If we have few resources available and we need a quicker decision with a greater risk of being wrong, we can fix a larger step size at the beginning manually.

Assuming the working conditions of processor and defect sets constant, all the algorithms were executed ten times for each type of pattern and the average executing time was calculated as shown in table 8.1.

Table 8.1: Execution times of all the algorithms

Method / Patterns	Time in terms of millisecond (ms)		
	Simple method	Heuristic method	Circular Method
Point Patern	142	143	143
Block Pattern	203	142	281
Strip Pattern	219	143	401

In terms of time, the cost for point pattern identification was found to be the same for all the algorithms. For block and strip patterns, however, the Circular method was the most expensive one and the Heuristic method was the cheapest one.

From the table above we can say that Circular method is the most efficient method when it comes to identifying patterns but the most expensive one and the Heuristic method is the fastest and less reliable than other two. When we analyzed the above table 8, we found that:

CA > CS > CH and

(CA > CS + CH)

where we have used

CS = Cost for Simple method

CA = Cost for Circular method

CH = Cost for Heuristic method

If CP is the cost for pattern identification, CE is the cost of the most efficient algorithm and CR is the cost of Random testing, then just to identify the pattern we don't need to explore all the points if it can be identified by two or three end points. The most efficient algorithm explores the defect containing points only. But if we take all the test cases randomly, it takes more time than CP and CE same point may repeat again.

CR > CE > CP

The cost of identifying the pattern is inversely proportional to the step size. If the step size is small, the algorithm uses more time to identify a pattern, as we have to explore more datasets. Thus the cost will be high but if we explore more datasets with small a step size, the result will be more precise. The approximate relation between step size, cost in terms of time and precision of pattern is shown in the diagram below.

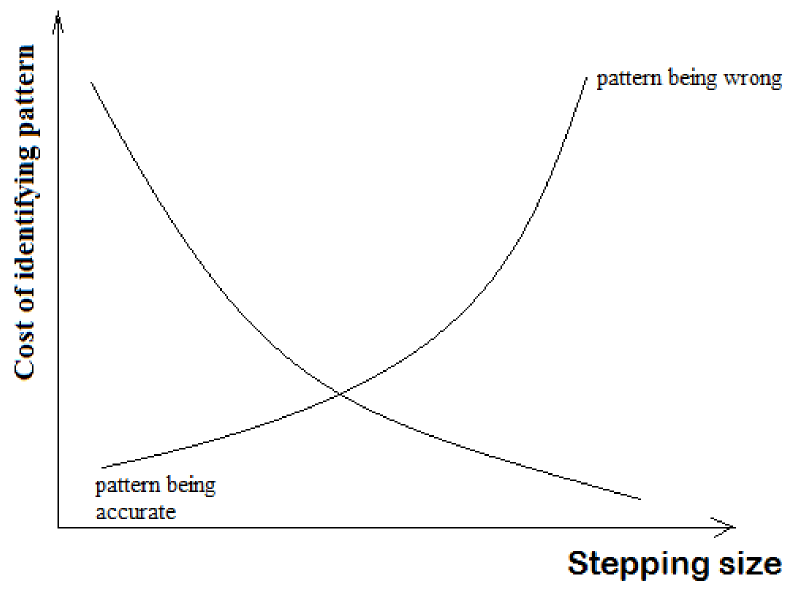


Figure 8.1: Graph showing the effect of step size to the pattern

9 CONCLUSION AND FUTURE WORK

In this report, I have discussed the concept of mapping a real application into two-dimensional space (textual to numeric data mapping). Instead of doing a real testing, I discussed three algorithms and traced them to see if it is possible to identify failure patterns that are already inserted for simulation purposes. The three algorithms are:

- Simple distance computation method
- Circular method
- Heuristic method

In all of these methods, a first defect is selected randomly and the points around it are explored. In the Simple distance computation method all the points on the horizontal axis of the initially selected point are explored first and then all the points of the vertical axis are explored. In the Circular method, all the points around the selected point are explored in a circular way – i.e. right neighbor is tested first and then the top neighbor, left and finally down is explored while in the Heuristic method, as the name implies, a trial-and-error method which is a short cut way of determining the pattern. In this case, only the end points of the initially selected point are tested. In the end all the tested data are collected in an array and boundary test (see chapter 5.4) is performed to identify the pattern of the failure data.

As all the neighbors of the selected point are tested in the Circular method, this is the most reliable and efficient method among the three methods but it is costly in terms of execution time. The Heuristic method, which takes only the end points of a defect containing point, has the same execution time for all patterns but may mistake a strip pattern for a point pattern. Thus, the Heuristic method is the cheapest but least reliable method.

While implementing the algorithms for all types of pattern, we saw that the Circular method is the most expensive one, followed by Simple method and Heuristic method (see table 8.1).

The most important result of this thesis is that we can work on non-numeric data by transforming the real application into two-dimensional numeric data. In the real world, we have to work on real application containing non-numeric data sets but the computer cannot directly recognize the non-numeric data for the computation. For that we divide the application into modules and list their features. For the transformation to two-dimensional input spaces, we insert each module at a point on the x-axis and the corresponding features on the y-axis. We can then map the system's features on to numeric data using the functions explained in chapter 4.18.

Future work:

1. **Step size:**

   We need to find the optimal step size – the lowest possible failure rate at an acceptable cost. It is assumed that fixing the step size to 1 unit gives a right pattern.

For the application having high density of defects, large value of step size may also give a right pattern.

As we took example of small program, there was no need to change the step size. Usually input domains are large for real application with hundreds of modules and thousands of features. In that case dynamic change of step size is important. Thus simulation of step size is kept for future work.

2. Conceptual distance between features (dist(x,y))

Similarity and relatedness between two features can be computed by using the function dist(x,y). For this function, we need more studies of the analysis of lexicon relation of synonymy, hyponymy and hyponymy. Because of limited time, I could not complete this part, which is thus kept for future work.

10 APPENDICES

A.1 Tracing of algorithm for Circular Method

The algorithm for the Circular method is given as:

```
Step1:  Locate randomly a point V(p) in the domain(D).
Step 2: Check if it is already detected.
If it is not detected mark point P as detected and put into array DT[]. Otherwise repeat step one.
Step 3: Check if the detected point contains defect.
        If it is not a defect
        Return null
    If it is a defect
        Mark it as a defect and put that point into array defected DF[]
            and  find its four neighbors.
Step 4. Repeat the process from Step 2 for all the neighbors.
```

A.1.1 Point Pattern

When the "Start Checking with Circular method" button of the mainframe of the software is pressed, it starts generating the tests cases randomly and testing whether it contains defectsor not with the help of the database where all the features are kept along with the seeded defects. In the end, it computes the type of pattern by performing some type of boundary test mechanisms (explained in section 5.2).

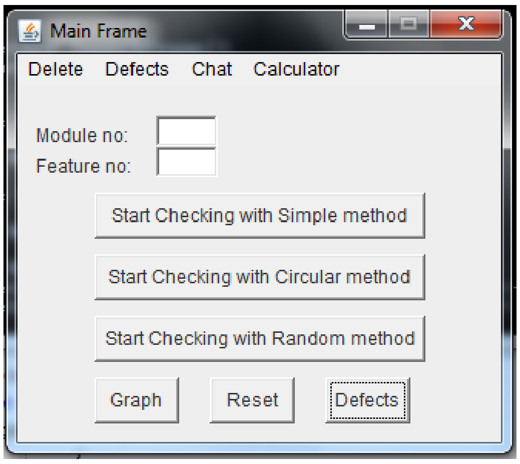

Main Frame
Delete
Defects
Chat
Calculator
Module no:
Feature no:
Start Checking with Simple method
Start Checking with Circular method
Start Checking with Random method
Graph
Reset
Defects

Figure A1: Main frame of the application

For simulation purpose, this algorithm was repeated 10 times, that is the button was pressed 10 times. Before starting simulation for next type pattern, 'Reset' button was pressed to clear the log of DF[] and DT[]. Simulation for point type defect with the Simple distance computation method is given below:

A.1.11 Attempt 1

Step1: The point P(4,6) was selected.

Step2: First we tested to see whether it was already detected or not by matching the point in the array list DT[]. As DT[] is null this time no match is found, therefore it is not detected yet. Then we test whether the feature corresponding to the point P(4,6), contains a defect or not using the database table "allDefects". It is found that this point is bug free. The algorithm is then terminated returning the value of the array list DT[] and DF[]and another attempt is done.

A.1.12 Attempt 2:

Step 1: The point P(9,8) was selected randomly.

Step 2: We tested to see whether the selected point is already tested or not by checking the array list DT[]. As the array list DT[] contains only the point P(4,6), it is found that the selected point is not tested before. With the help of database table we test whether the selected point contains defect or not. This showed that the point is bug free and the algorithm is terminated appending the point P(9,8) to DT[]. Till this attempt, no defects are found. Therefore the array list DF[] which consists only features containing defect are empty.

P(4,6)
P(9,8)

Figure A2: Array List DT[] showing the detected test cases

A.2.13 Attempt 3

Step 1: The point P(3,8) was selected randomly.

Step 2: We tested to see whether the selected point is already tested or not by checking the array list DT[]. As the array list DT[] contains only the point P(4,6) and P(9,8), it is found that the selected point is not tested before. With the help of the database table we tested to see whether it contains a defect or not. Which showed that the point is bug free and the algorithm is terminated appending the point P(3,8) in DT[]. Till this attempt, no defects are found. Therefore the array list DF[] that consists only features containing defect is empty.

P(4,6)
P(9,8)
P(3,8)

Figure A3: Array List DT[] showing the detected test cases A.2.14 Attempt 4

A.2.14 Attempt 4

Step 1: The point P(10,2) was selected randomly.

Step 2: We tested to see whether the selected point is already tested or not by checking the array list DT[]. As the array list DT[] contains only the point P(4,6), P(9,8) and P(3,8), it is found that it is not tested before. With the help of the database table we tested to see whether it contains defect or not. Which showed that the point is bug free and the algorithm is terminated appending the point P(10,2) in DT[]. Till this attempt, no defects are found. Therefore the array list DF[] that consists only features-containing defect is empty.

P(4,6)
P(9,8)
P(3,8),
P(10,2),

Figure A4: Array List DT[] showing the detected test cases

A.2.15 Attempt 5

Step 1: The point P(1,9) was selected randomly.

Step 2: We tested to see whether the selected point is already tested or not by checking the array list DT[]. As the array list DT[] contains only the point P(4,6), P(9,8), P(3,8) and P(10,2), it is found that the selected point is not tested before. With the help of the database table we tested to see whether it contains defect or not. Which showed that the point is bug free and the algorithm is terminated appending the point P(1,9) in DT[]. Till this attempt, no defects are found. Therefore the array list DF[] that consists only features containing defect is empty.

P(4,6)
P(9,8)
P(3,8)
P(10,2)
P(1,9)

Figure A5: Array List DT[] showing the detected test cases

A.2.16 Attempt 6

Step 1: The point P(5,1) was selected randomly.

Step 2: We tested to see whether the selected point is already tested or not by checking the array list DT[]. As the array list DT[] contains only the point P(4,6), P(9,8), P(3,8), P(10,2) and P(1,9), it is found that the point is not tested before. With the help of the database table we tested to see whether it contains defect or not. Which showed that the point is bug free and the algorithm is terminated appending the point P(5,1) in DT[]. Till this attempt, no defects are found. Therefore the array list DF[] that consists only features containing defect is empty.

P(4,6)
P(9,8)
P(3,8)
P(10,2)
P(1,9)
P(5,1)

Figure A6: Array List DT[] showing the detected test cases

A.2.17 Attempt 7

Step 1: The point P(1,6) was selected randomly.

Step 2: We tested to see whether the selected point is already tested or not by checking the array list DT[]. As the array list DT[] contains only the point P(4,6), P(9,8), P(3,8), P(10,2), P(1,9) and P(5,1), it is found that it is not tested before. With the help of the database table we tested to see whether it contains defect or not. This time the point P(1,6) is matched to the point of database table "allDefects". That is feature containing defect is found. As for the algorithm, the point P(1,6) is appended in DT[] and all the neighbors of the point P(1,6) are computed. The neighbors are P(2,6), P(1,5) and P(1,7).

Step 3: The algorithm is repeated for all neighbors of point P(1,6). As none of the neighbors contains defects, the algorithm is terminated putting all the detected points into the array list DT[]. The array lists DT[] and DF[] contains detected and defected points respectively as shown in the diagram below.

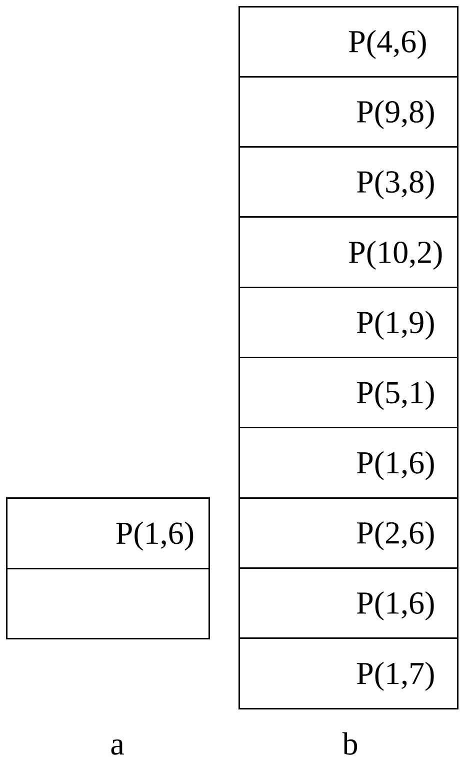


Figure A7: Array Lists DF[] (a) and DT[] (b) showing the detected test cases

A.2.18 Attempt 8

Step 1: The point P(2,6) was selected randomly.

Step 2: We tested to see whether the selected point is already tested or not by checking the array list DT[]. It is found to be detected as the array list DT[] contains the point P(4,6), P(9,8), P(3,8), P(10,2), P(1,9) and P(5,1). Therefore the algorithm was terminated.

A.2.19 Attempt 9

Step 1: The point P(1,5) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. We found it detected as the array list DT[] contained the point P(4,6), P(9,8), P(3,8), P(10,2), P(1,9), P(5,1), P(1,6), P(2,6), P(1,5) and P(1,5). No updates were done in the array lists DT[] and DF[] as the point was already detected. Therefor the algorithm is then terminated.

A.2.110 Attempt 10

Step 1: The point P(3,1) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. As the array list DT[] contained only the point P(4,6), P(9,8), P(3,8), P(10,2), P(1,9), P(5,1), P(1,6), P(2,6), P(1,5) and P(1,5), we found the selected point not tested before. With the help of the database table we tested to see whether it contained defect or not and found that it did not contain a defect. Therefore the algorithm was terminated. The array lists

DT[] and DF[] contains detected and defected points respectively as shown in the diagram below.

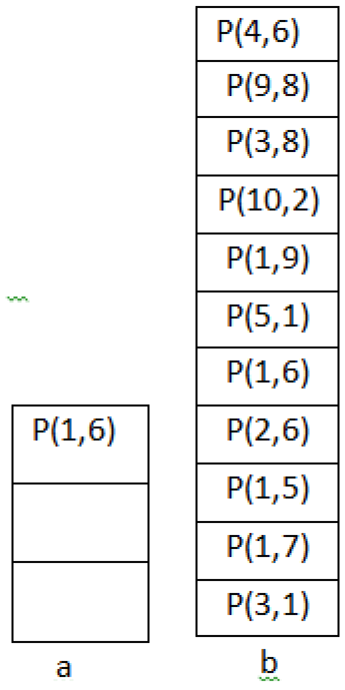
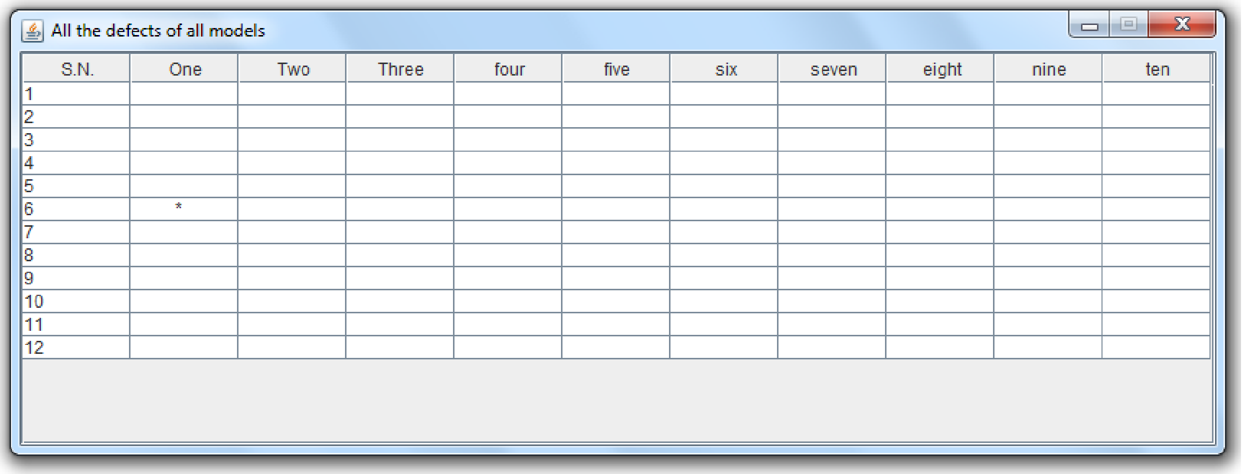


Figure A8: Array Lists DF[] (a) and DT[] (b) showing the detected test cases

In the end, when the required number of trial was completed, the array list with the points having defects was plotted in the graph to see the type of the failure pattern. Plotting was done in the table containing modules and features by assigning asterisk to the corresponding location of the point as shown in the figure below:



S.N.	One	Two	Three	four	five	six	seven	eight	nine	ten
1										
2										
3										
4										
5										
6	*									
7										
8										
9										
10										
11										
12										

Figure A9: Point type defect detected by the algorithm

A.1.2 Block Pattern

A.1.21 Attempt 1:

Step1: The point P(6,6) was selected.

Step2: First we tested to see whether the selected point was already detected or not by matching the point in the array list DT[]. As DT[] was null this time no match was found, therefore it was not detected yet. Then we tested to see whether the feature corresponding to the point P(6,6), contains a defect or not using the database table "allDefects". It was found that this point was bug free. The algorithm was then terminated returning the value of the array list DT[] and DF[]and another attempt was done.

A.1.22 Attempt 2:

Step 1: The point P(9,8) was selected randomly.

Step 2: We tested to see whether the selected point is already tested or not by checking the array list DT[]. As the array list DT[] contains only the point P(6,6), we found that the selected point was not tested before. With the help of database table we tested to see whether the selected point contains defect or not and it was found that the point was bug free and the algorithm was terminated appending the point P(9,8) to DT[]. Till this attempt, no defects are found. Therefore the array list DF[], which consists only features containing defect are empty.

P(6,6)
P(9,8)

Figure A10: Array List DT[] showing the detected test cases

A.2.23 Attempt 3

Step 1: The point P(11,6) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. As the array list DT[] contains only the point P(6,6) and P(9,8), we found that it was not tested before. With the help of the database table we tested to see whether it contains defect or not. This time also the point P(11,6) was not matched to the point of database table "allDefects". As for the algorithm, the point P(11,6) was appended and another point was tested. Till this time the array list containing defects was empty.

P(6,6)
P(9,8)
P(9,8)

Figure A11: Array List DT[] showing the detected test cases

A.2.24 Attempt 4

Step 1: The point P(6,6) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. We found it to be detected as the array list DT[] contained the point P(2,8). Therefore the algorithm was terminated.

A.2.25 Attempt 5

Step 1: The point P(3,4) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. As the array list DT[] does not contain the point P(3,4), we found that it was not tested before and the point was put into DT[]. With the help of the database table we tested to see whether the selected point contains defect or not. This time, the point P(3,4) was matched to the database table "allDefects". That is feature-containing defect was found. As for the algorithm, the point P(3,4) was appended in DF[] and all the neighbors of the point P(3,4) were computed. The neighbors were P(2,4) and P(4,4), P(3,3) and P(3,5).

Step 3: The algorithm was repeated for all neighbors of point P(3,4). Point P(4,4) was taken first, which was not defect containing point then P(3,3) was taken, where we found error and its neighbors were computed and tested to see whether they contain defect or not. For each point step 2 was repeated and finally the defect containing features were collected in array DF[] and detected features are collected in DT[].

P(3,1	…	P(3,5)	P(6,6)	P(9,8)	P(11,6)

a

P(3,1	…	P(3,5)

b

Figure A12: Array List DT[] and DF[]

A.1.3 Strip Pattern

A.2.31 Attempt 1

Step 1: The point P(1,6) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. Initially, as the array list DT[] was empty, we found that it was not tested before. With the help of the database table we tested to whether it contains defect or not. The point P(1,6) was matched to the point of database table "allDefects". That is feature-containing a defect was found. As for the algorithm, the point P(1,6) was appended in DT[] and all the neighbors of the point P(1,6) were computed. The neighbors are P(1,5) and P(1,7).

Step 3: The algorithm was repeated for all neighbors of point P(1,6). Point P(2,6) was tested first, which was not a defect containing point then P(1,5) was tested, where we found error and its neighbors were computed and tested to see whether they contain defect or not. For each point step 2 was repeated and finally the defect containing features were collected in array DF[] and detected features were collected in DT[] as shown in figure A13.

P(1,1)	P(1,2)	P(1,3)	P(1,4)	P(1,5)	P(1,6)	P(1,7)	P(1,8)	P(1,9)	P(1,10)	P(1,11)	P(1,12)								

a

P(1,1)	P(1,2)	P(1,3)	P(1,4)	P(1,5)	P(1,6)	P(1,7)	P(1,8)	P(1,9)	P(1,10)	P(1,11)	P(1,12)	P(2,1)	..	P(2,12)

b

Figure A13: Array List DT[] and DF[]

A.1.22 Attempt 2:

Step 1: The point P(5,5) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. As the array list DT[] did not contain the point P(5,5), the selected point was not tested before. With the help of database table we tested to see whether the selected point contained defect or not and we found that the point was bug free and the algorithm was terminated appending the point P(5,5) to DT[].

A.1.23 Attempt 3:

Step 1: The point P(10,9) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. As the array list DT[] did not contain the point P(10,9), the selected point was not tested before. With the help of database table we tested to see whether the selected point contains defect or not and we found that the point was bug free and the algorithm was terminated appending the point P(10,9) to DT[].

A.1.34 Attempt 4:

Step 1: The point P(1,9) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. As the array list DT[] contained the point P(1,9), the selected point was already tested before. The algorithm was then terminated.

A.2.25 Attempt 5

Step 1: The point P(3,5) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. As the array list DT[] did not contain the point P(3,5), we found that it was not tested before and the point was put into DT[]. With the help of the database table we tested to see whether the selected point contained defect or not. This time also, the point P(3,5) was matched to the point of database table "allDefects". That is feature-containing a defect was found. As for the algorithm, the point P(3,4) was appended in DF[] and all the neighbors of the point P(3,5) were computed. The neighbors were P(2,5) and P(4,5), P(3,4) and P(3,6).

Step 3: The algorithm was repeated for all neighbors of point P(3,4). Point P(4,5) was tested first, which was not defect containing point then P(3,4) was tested, where we found error and its neighbors were computed and tested to see whether they contain defect or not. For each point step 2 was repeated and finally the defect containing features are collected in array DF[] and detected features were collected in DT[] as shown in figure A14.

P(1,1)	P(1,2)	………………	P(1,12)	P(3,1)	P(3,2)	....	P(3,5)

Figure A14: DF[]

A.2 Tracing of algorithm for the Simple Distance Computation Method

The algorithm for the Simple Distance Computation method is given as:

Step1: Locate randomly a point V(p) in the domain(D).

Step 2: Check if it is already detected.

If it is not detected mark point P as detected and put into array DT[]. Otherwise repeat step one.

Step 3: Check if the detected point contains a defect.

```
If it is not a defect
        Return null
If it is a defect
        Mark it as defected and put that point into array defected DF[]
```

Step 4: Take one point from neighbor along X-axis and repeat the process from Step 2.

Step 5: Take one point from the neighbor of the initial point along Y-axis and repeat the process from Step 2.

A.2.1 Point Pattern

When the "Start Checking with Simple method" button of the mainframe of the software was pressed, it started generating the tests cases randomly and testing whether it contains defects or not with the help of the database where all the features were kept along with the seeded defects. In the end, it computed the type of pattern by performing some type of boundary test mechanisms (explained in section 5.2).

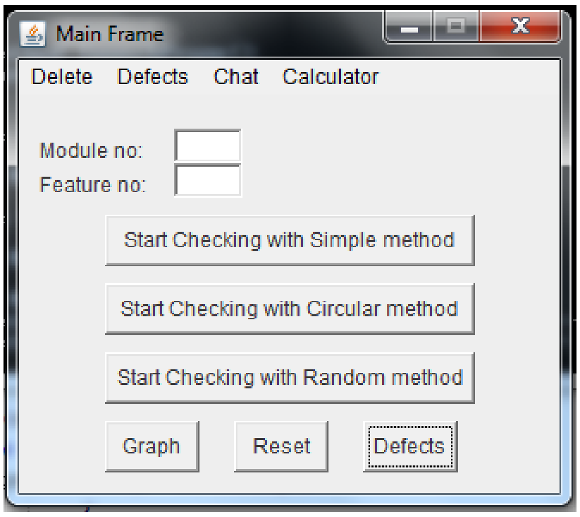

Main Frame
Delete
Defects
Chat
Calculator
Module no:
Feature no:
Start Checking with Simple method
Start Checking with Circular method
Start Checking with Random method
Graph
Reset
Defects

Figure A15: Main frame of the application

For simulation purpose, this algorithm was repeated 10 times, that is the button was pressed 10 times. The result is given below:

A.2.11 Attempt 1:

Step1: The point P(10,6) was selected.

Step2: First we tested to see whether it was already detected or not by matching the point in the array list DT[]. As DT[] was null this time no match was found, therefore it was not detected yet. Then we tested to see whether the feature corresponding to the point P(10,6), contains a defect or not using the database table "allDefects". We found that this point was bug free. The algorithm was then terminated returning the value of the array list DT[] and DF[] and another attempt was done.

A.2.12 Attempt 2:

Step 1: The point P(8,8) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. As the array list DT[] contains only the point P(10,6), we found that the selected point was not tested before. With the help of database table we tested to see whether the selected point contains defect or not. This showed that the point was bug free and the algorithm was terminated appending the point P(8,8) to DT[]. Till this attempt, no defects were found. Therefore the array list DF[] which consists only features containing defect was empty.

P(10,6)
P(8,8)

Figure A16: Array List DT[] showing the detected test cases

A.2.13 Attempt 3

Step 1: The point P(7,2) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. As the array list DT[] contains only the point P(10,6) and P(8,8), we found that the selected point was not tested before. With the help of the database table we tested to see whether it contains a defect or not. Which showed that the point was bug free and the algorithm was terminated appending the point P(7,2) in DT[]. Till this attempt, no defects were found. Therefore the array list DF[] that consists only features containing defect was empty.

P(10,6)
P(8,8)
P(7,2)

FigureA17: Array List DT[] showing the detected test cases A.2.14 Attempt 4

A.2.14 Attempt 4

Step 1: The point P(9,2) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. As the array list DT[] contains only the point P(10,6), P(8,8) and P(7,2), we found that it was not tested before. With the help of the database table we tested to see whether it contains defect or not. Which showed that the point was bug free and the algorithm was terminated appending the point P(9,2) in DT[]. Till this attempt, we did not find any defect. Therefore the array list DF[] that consists only features-containing defect was empty.

P(10,6)
P(8,8)
P(7,2)
P(9,2)

FigureA18: Array List DT[] showing the detected test cases

A.2.15 Attempt 5

Step 1: The point P(1,6) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. As the array list DT[] contains only the point P P(10,6), P(8,8), P(7,2) and P(9,2), we found that the selected point was not tested before. With the help of the database table we tested to see whether it contains defect or not. Which showed that the point contained bug and according to the algorithm, step 4 and step 5 were repeated until we found error containing points along X-axis and Y-axis of the initial point P(1,6).

P(10,6)
P(8,8)
P(7,2)
P(9,2)
P(1,2)

FigureA19: Array List DT[] showing the detected test cases

A.2.2 Block Pattern

A.2.21 Attempt 1:

Step1: The point P(8,3) was selected.

Step2: First we tested to see whether the selected point was already detected or not by matching the point in the array list DT[]. As DT[] was null this time no match was found, therefore it was not detected then. Then we tested to see whether the feature corresponding to the point P(8,3), contained a defect or not using the database table "AllDefects". We found that this point was bug free. The algorithm was then terminated returning the value of the array list DT[] and DF[] and another attempt was done.

A.2.22 Attempt 2:

Step 1: The point P(9,8) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. As the array list DT[] contained only the point P(8,3), we found that the selected point was not tested before. With the help of database table we tested to see whether the selected point contained defect or not and we found that the point was bug free and the algorithm was terminated appending the point P(9,8) to DT[]. Till this attempt, no defects were found. Therefore the array list DF[], which consists only features containing defect are empty.

P(8,3)
P(9,8)

FigureA20: Array List DT[] showing the detected test cases

A.2.23 Attempt 3

Step 1: The point P(7,6) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. As the array list DT[] contains only the point P(8,3) and P(9,8), we found that it was not tested before. With the help of the database table we tested to see whether it contained defect or not. This time also, the point P(7,6) was not matched to the point of database table "AllDefects". As for the algorithm, the point P(7,6) was appended in DT[] and another attempt was done.

P(8,3)
P(9,8)
P(7,6)

FigureA21: DT[]

A.2.24 Attempt 4

Step 1: The point P(3,1) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. We found it to be not detected as the array list DT[] did not contain the point P(2,8). Then the point P(3, 1) was checked in the database table containing defects. As we found the point P(3,1) in 'AllDefects', it was a point containing a defect. So it was inserted into the array DF[] and its neighbors along X- axis were explored but no neighbor in X-axis contained a defect. Then the neigher of P(3, 1) along Y- axis were explored where we found P(3, 2), P(3, 3), P(3, 14) and P(3, 5) were containing error. Thus these points were inserted into the array DF[] and terminated the algorithm.

A.2.25 Attempt 5

Step 1: The point P(3,3) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. We found it to be detected as the array list DT[] contains the point P(3,3). Therefore the algorithm was terminated

P(3,1)	..	P(3,5)

a

P(3,1)	..	P(3,5)	P(9,8)	P(9,8)	P(7,6)

b

Figure A22: DF[] and DT[]

A.2.3 Strip Pattern

A.2.21 Attempt 1:

Step1: The point P(8,3) was selected.

Step2: First we tested to see whether the selected point was already detected or not by matching the point in the array list DT[]. As DT[] was null this time no match was found, therefore it was not detected. Then we tested to see whether the feature corresponding to the point P(8,3), contains a defect or not using the database table "allDefects". We found that this point was bug free. The algorithm was then terminated returning the value of the array list DT[] and DF[] and another attempt was done.

A.2.22 Attempt 2:

Step 1: The point P(9,8) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. As the array list DT[] contained only the point P(8,3), we found that the selected point was not tested before. With the help of database table we tested to see whether the selected point contains defect or not and we found that the point was bug free and the algorithm was terminated appending the point P(9,8) to DT[]. Till this attempt, no defects were found. Therefore the array list DF[], which consists only features containing defect was empty.

P(8,3)
P(9,8)

Figure A23: Array List DT[] showing the detected test cases

A.2.23 Attempt 3

Step 1: The point P(1,6) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. As the array list DT[] contains only the point P(8,3) and P(9,8), we found that it was not tested before. With the help of the database table we tested to see whether it contains defect or not. This time, the point P(1,6) was matched to the point of database table "allDefects". That is feature-containing defect was found. As for the algorithm, the point P(1,6) is appended in DT[] and the neighbors along the X – axis and Y- axis of the point P(1,6) are computed and tested.

Step 3: Along X-axis, we kept on testing the next point along X-axis if we found the detected point as a defect. After completing testing along X-axis the same process was repeated on Y-axis.

A.2.24 Attempt 4

Step 1: The point P(2,6) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. We found it to be detected as the array list DT[] contains the point P(2,8). Therefor the algorithm was terminated.

A.2.25 Attempt 5

Step 1: The point P(3,4) was selected randomly.

Step 2: We tested to see whether the selected point was already tested or not by checking the array list DT[]. As the array list DT[] did not contain the point P(3,4), we found that it was not tested before and the point was put into DT[]. With the help of the database table we tested to see whether the selected point contains defect or not. This time also, the point P(3,4) was matched to the point of database table "allDefects". That is feature-containing defect was found. As for the algorithm, the point P(3,4) was appended in DF[] and [] and the neighbors along the X – axis and Y- axis of the point P(3,4) are computed and tested.

Step 3: Along X-axis, we kept on testing the next point along X-axis if we found the detected point as a defect. After completing testing along X-axis the same process was repeated on Y-axis. Following figures (a) and (b) shows the defects containing points and detected points.

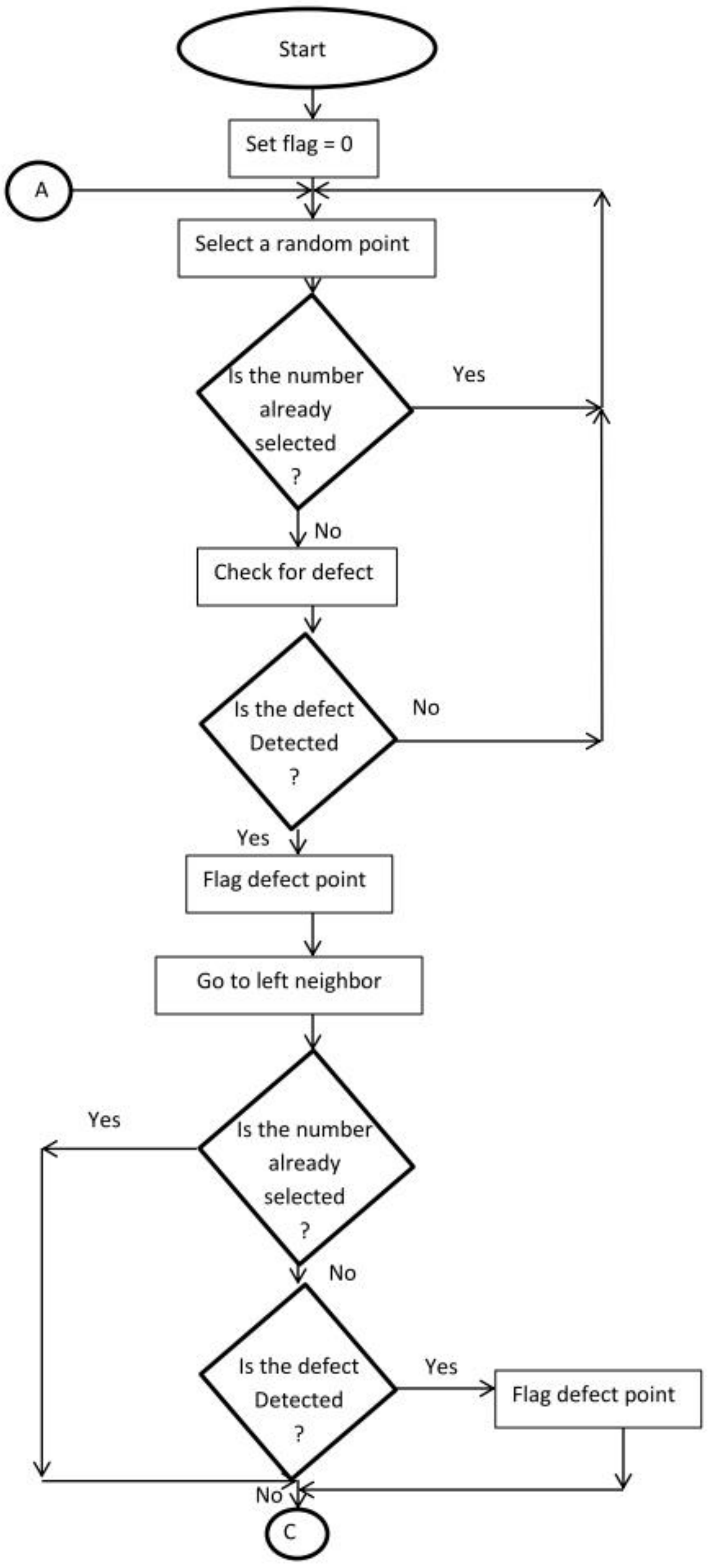
P(1,2)	…….	P(1,11)	P(3,1)	P(3,2)	....	P(3,5)

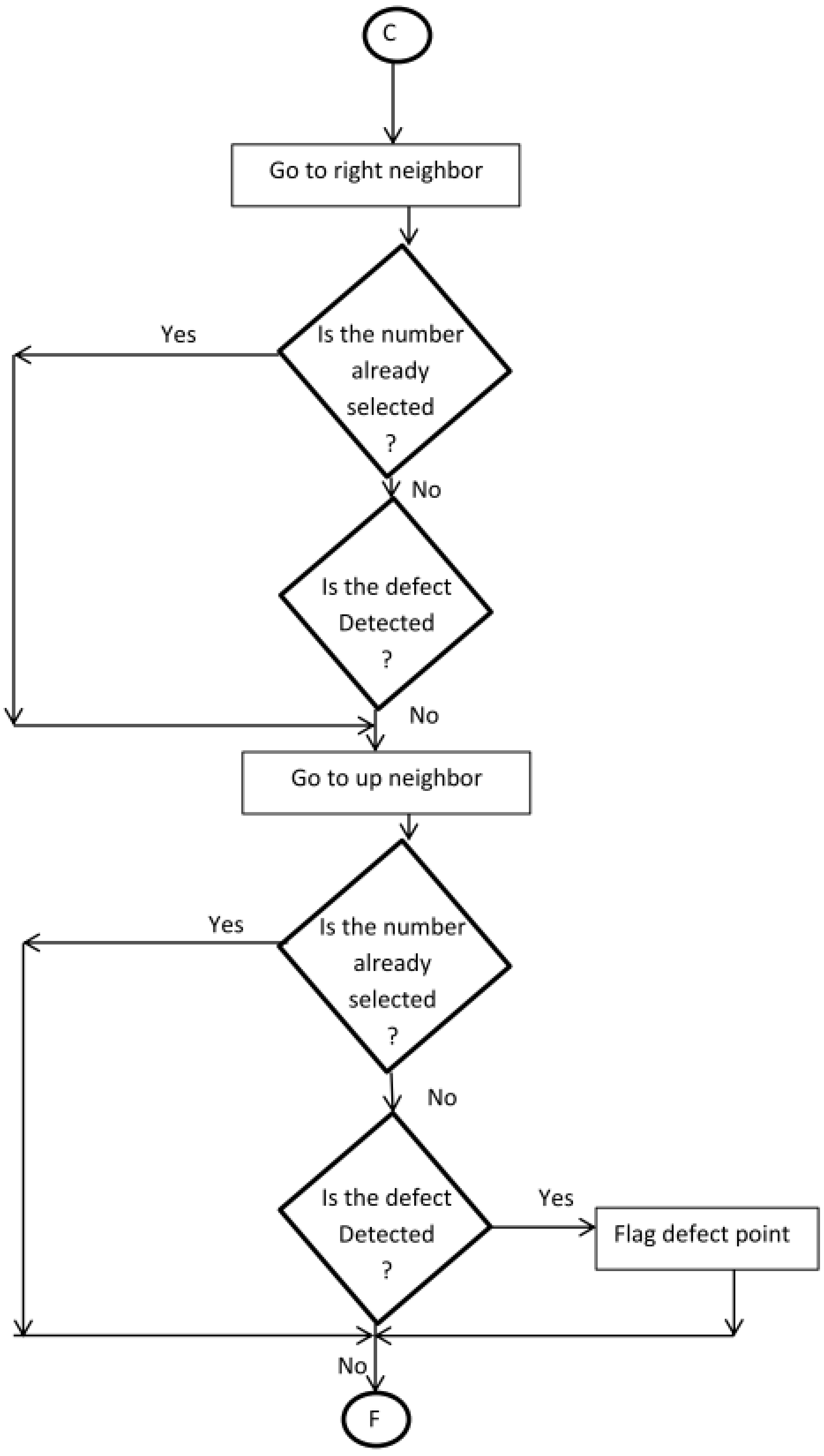
Figure A24: DF[]

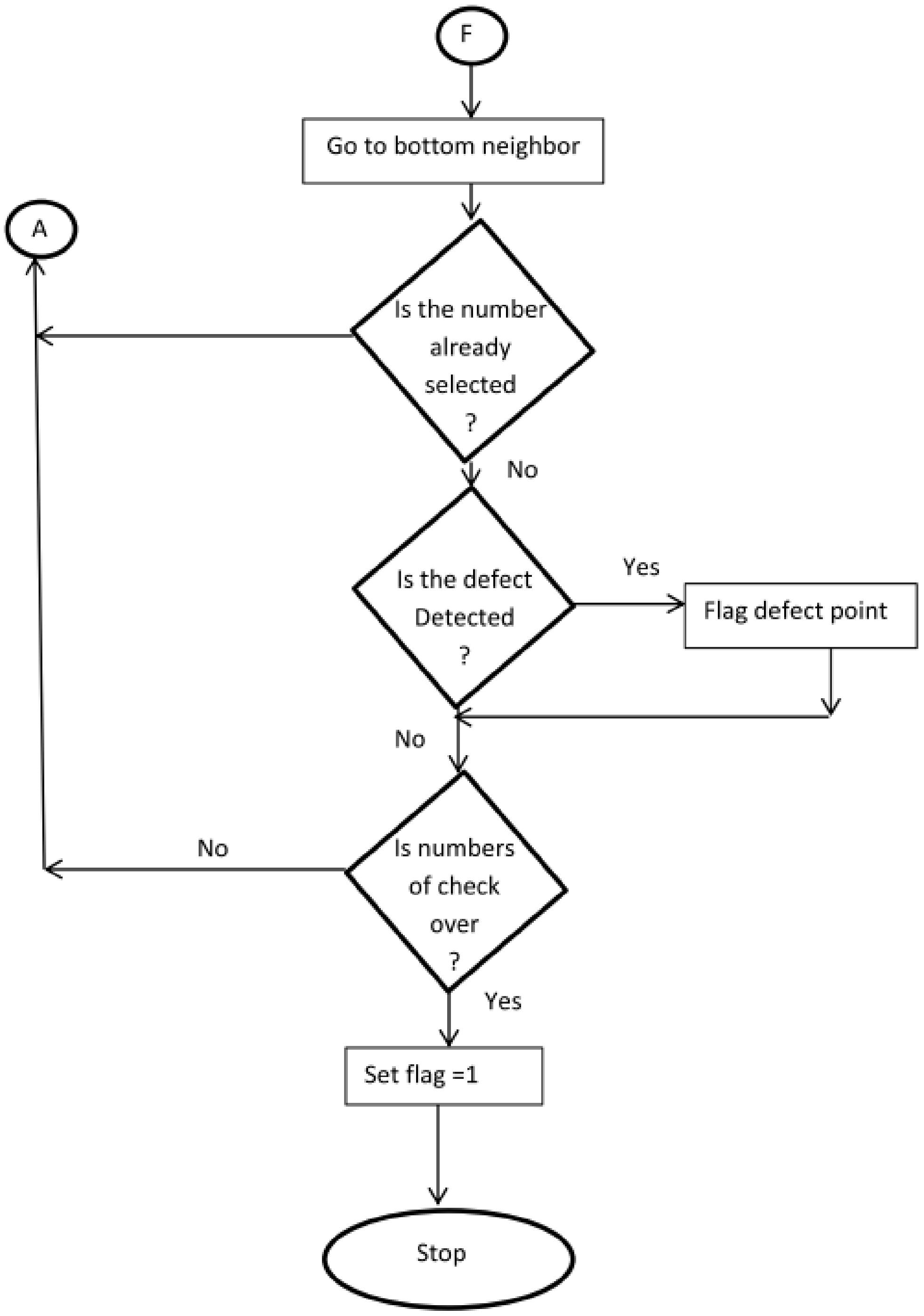
P(1,2)	.	P(1,11)	P(3,1)	P(3,2)	....	P(3,5)	P(4,5)	P(3,1)	P(3,2)	….….	P(3,5)

Figure A25: DT[]

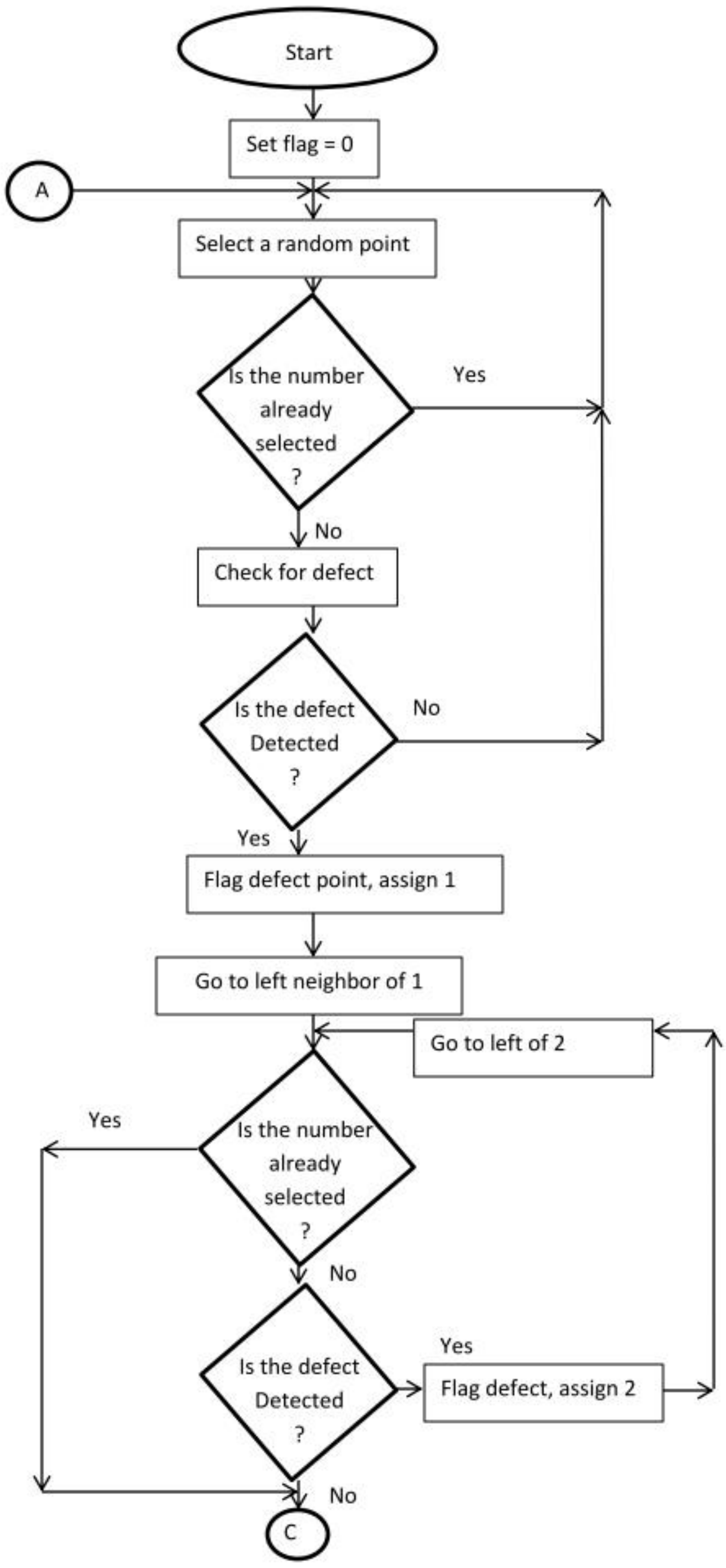
A.3 Flowchart of the Simple Distance Computation Method

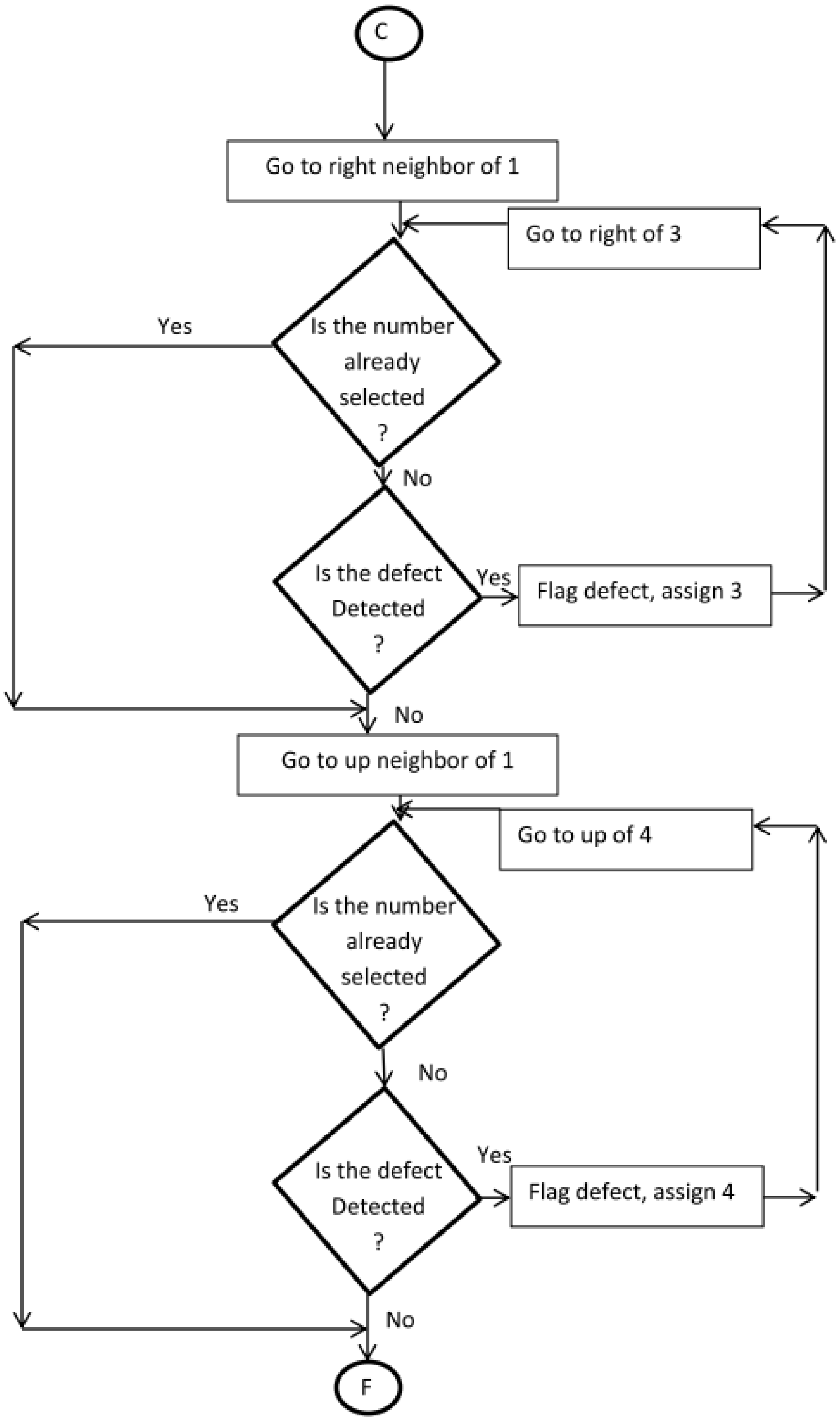


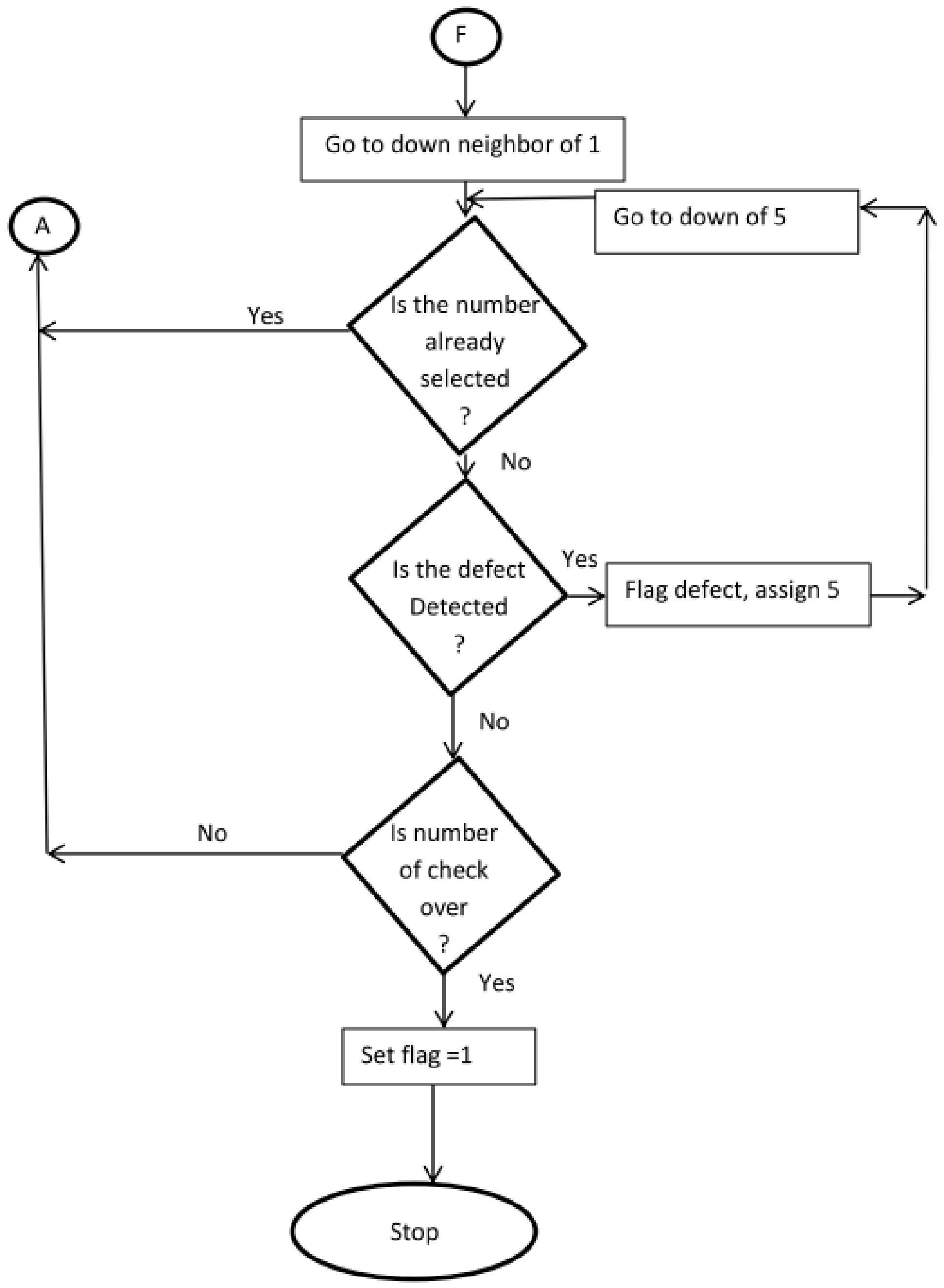

C
Go to right neighbor
Is the number already selected ?
Yes
No
Is the defect Detected ?
No
Go to up neighbor
Is the number already selected ?
Yes
No
Is the defect Detected ?
Yes
Flag defect point
No
F


F
Go to bottom neighbor
A
Is the number already selected ?
No
Is the defect Detected ?
Yes
Flag defect point
No
No
Is numbers of check over ?
Yes
Set flag =1
Stop

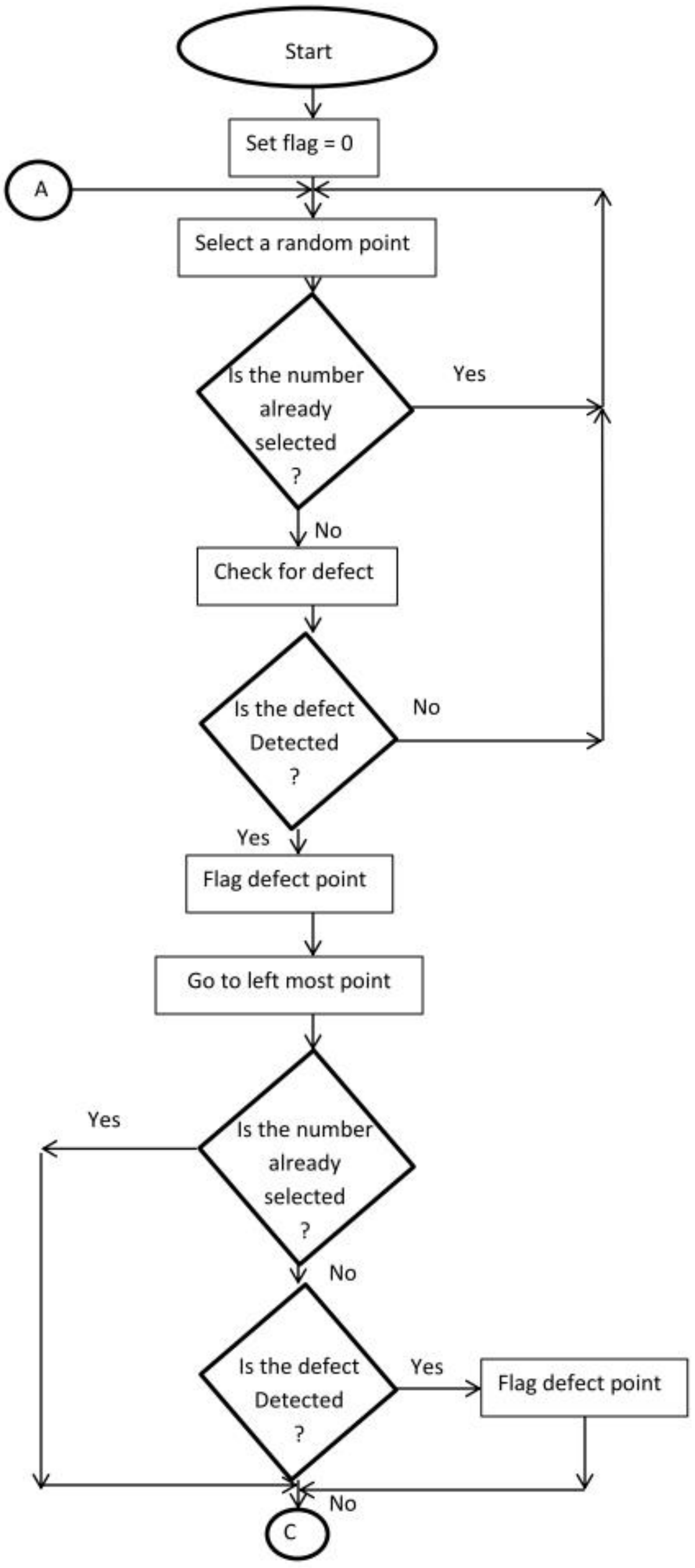
A.4 Flowchart of the Circular Method

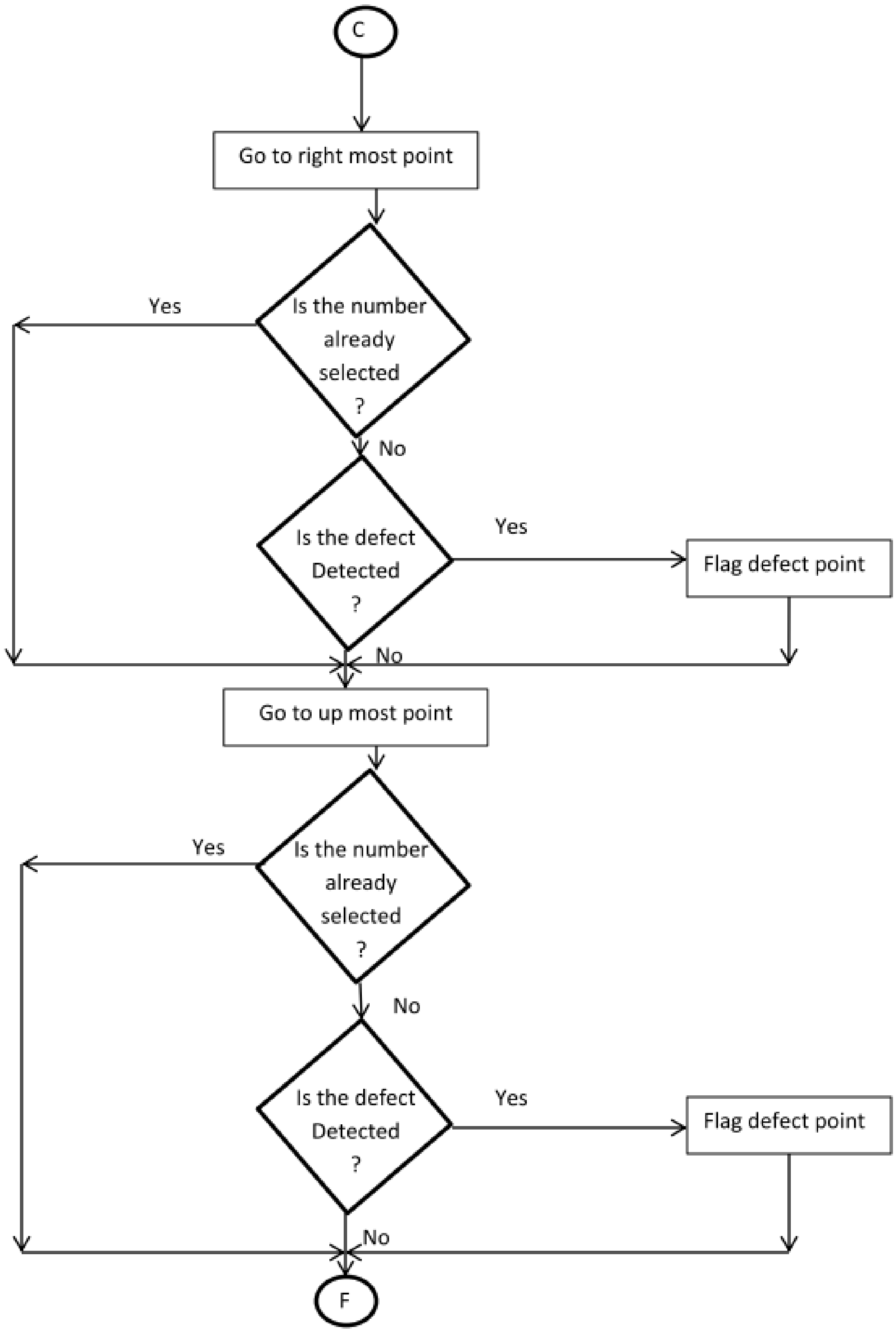


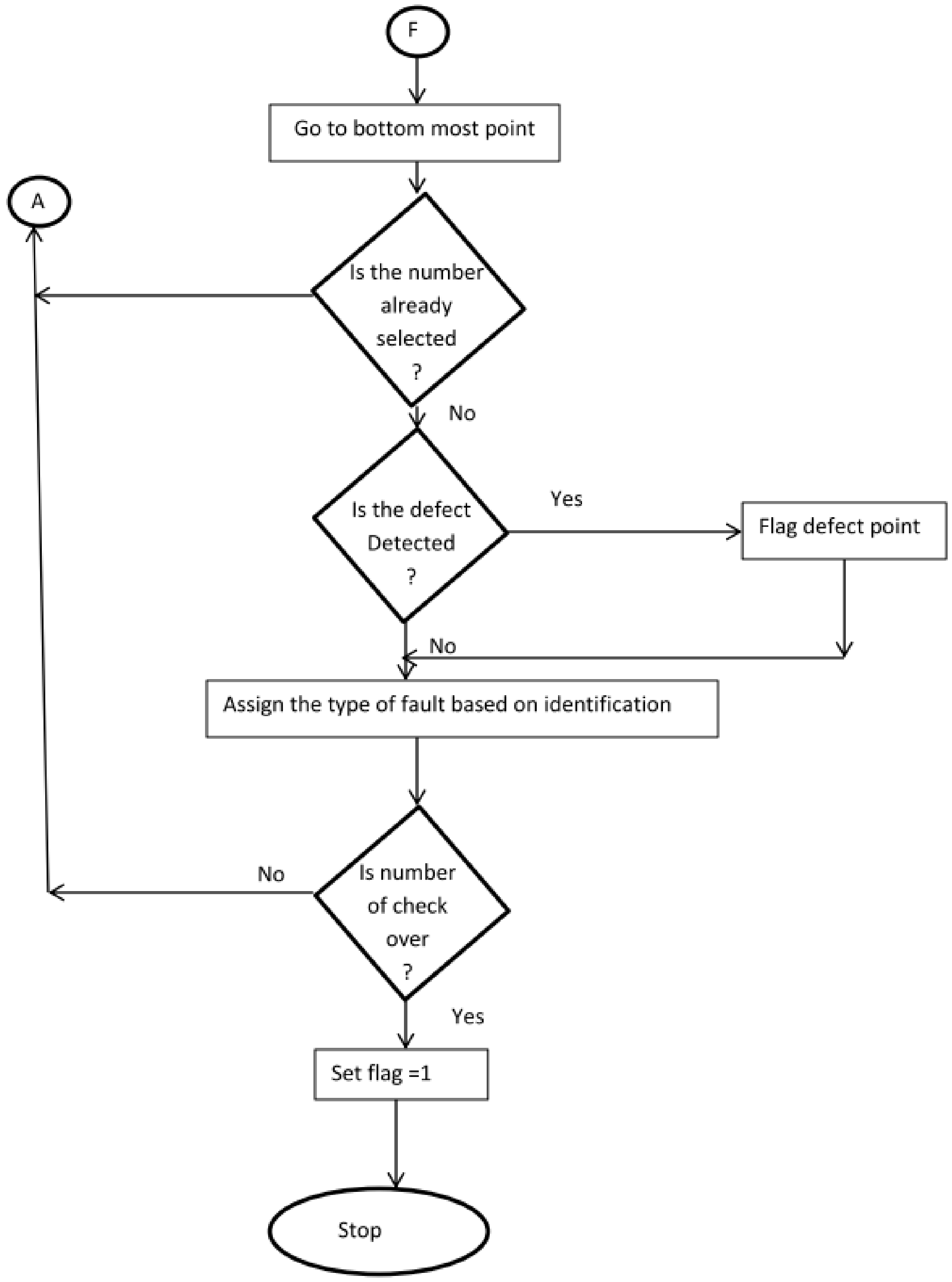

C
Go to right neighbor of 1
Go to right of 3
Yes
Is the number already selected ?
No
Is the defect Detected ?
Yes
Flag defect, assign 3
No
Go to up neighbor of 1
Go to up of 4
Yes
Is the number already selected ?
No
Is the defect Detected ?
Yes
Flag defect, assign 4
No
F


F
Go to down neighbor of 1
Go to down of 5
A
Yes
Is the number already selected ?
No
Is the defect Detected ?
Yes
Flag defect, assign 5
No
No
Is number of check over ?
Yes
Set flag =1
Stop

A.5 Flowchart of the Heuristic Method




C
Go to right most point
Is the number already selected ?
Yes
No
Is the defect Detected ?
Yes
Flag defect point
No
Go to up most point
Is the number already selected ?
Yes
No
Is the defect Detected ?
Yes
Flag defect point
No
F


F
Go to bottom most point
A
Is the number already selected ?
No
Is the defect Detected ?
Yes
Flag defect point
No
Assign the type of fault based on identification
No
Is number of check over ?
Yes
Set flag =1
Stop

11 BIBLIOGRAPHY

[1] F.T. Chan, T.Y. Chen, I.K. Mak, Y.T. Yu, Proportional sampling strategy: guidelines for software testing practitioners, Information and Software Technology 38 (12) (1996) 775–782.

[2] Mirror adaptive random testing , T.Y. Chen, F.-C. Kuo, R.G. Merkel, S.P. Ng, 2004

LNCS 2349.

[3] T.Y. Chen, T.H. Tse, Y.T. Yu, Proportional sampling strategy: a compendium and some insights, The Journal of Systems and Software 58 (2001) 65–81.

[4] R. Cobb, H.D. Mills, Engineering Software under Statistical Quality Control, IEEE Software 7 (1990) 44–56.

[5] I.K. Mak, On the effectiveness of random testing, Master Thesis, Department of Computer Science, University of Melbourne, Australia, 1997.

[6] H.D. Mills, M. Dyer, R.C. Linger, Cleanroom software engineering, IEEE Software 3 (1986) 19–24.

[7] G. Myers, The Art of Software Testing, Wiley, New York, 1979.

[8] R.A. Thayer, M. Lipow, E.C. Nelson, Software Reliability, NorthHolland, Amsterdam, 1978.

[9] T.Y Chen, H.Leung and I.K Mak. Adaptive Random Testing. 2004

[10] Hong Zhu. *Adequate Testing of Computer Software*. An Online Book on

Software Testing, 1995.

[11] http://en.wikipedia.org/wiki/Verification_and_validation_(software)

[12] Efficient and Effective Random Testing Using the Voronoi Diagram, T. Y. Chen and Robert Merkel

[13] P. E. Ammann and J. C. Knight, "Data diversity: an approach to software fault tolerance," IEEE Transactions on Computers, vol. 37, no. 4, pp. 418–425, April 1988.

[14] Chan, F.T., Chen, T.Y., Mak, I.K., Yu, Y.T.: Proportional sampling strategy:

guidelines for software testing practitioners. Information and Software Technology

38 (1996) 775–782

[15] http://www.utdallas.edu/~ewong/SYSM-6310/03-Lecture/02-ART-paper-01.pdf

[16] http://urn.kb.se/resolve?urn=urn:nbn:no:ntnu:diva-17756