Chapter 7 elaborates the Results and Discussion part of work done. This chapter shows the actual implementation of the proposed work and also presents the results that are formed while working on the problem. Further, a brief discussion is done on the various aspects in order to form some good results. In order to reduce software testing time and efforts there should be a mechanism to optimize software testing. This can be considered as an optimization problem. To solve optimization problems there are a number of techniques and one of them is Genetic Algorithms.
CHAPTER 7
SBSE AND AUTOMATIC TEST CASE GENERATION

Software engineering can also be viewed as a searching problem in which a search space consisting of various potential solutions is explored to locate the optimal solutions as discussed in chapter 6. In this chapter, one most important aspect of software engineering that consumes maximum efforts i.e. testing is considered for implementation as a searching problem. Two models have been designed based on metrics and heuristics, implemented and validated.

7.1 Automation of Testing Process

Every program has some particular implementation of requirement specifications. Each program has been made from some requirement specifications of some problem. A program model can be viewed as a function doing mapping of some input to some output as shown in Figure 7.1.

![Figure 7.1: Program Terminology](image)

Software test case design is the primary task in testing, in which some of the inputs from the domain are selected to test the software under test. These inputs are executed on program and outcomes are examined. If the test cases are adequate to express all the cases of input values, there are more chances to find the faults in program, but if test cases are not sufficient then faults will go unnoticed & they propagate with the final product, which will later result in failure. It has been inferred that dynamic testing shows the presence of faults but one cannot certify their absence. Literature shows that there are several methods to assess the reliability of the software which try to accommodate with software engineering life cycle to make the software fully working as well as bind it to the requirements for which it is developed. There are a lot of methods which try to exploit the requirements and then guide the development to the direction which will fulfil them.
7.1.1 Fault Prediction

Software quality is an important activity in software engineering and testing related issues are becoming crucial for software. Although there is diversity in the definition of software quality, it is widely accepted that a project with many defects lacks quality. Methodologies and techniques for predicting the testing effort, monitoring process costs and measuring results can help in increasing efficiency of software testing. Being able to measure the fault-proneness of software can be a key step towards steering the software testing and improving the effectiveness of the whole process. Predictive modelling is the process by which a model is created or chosen to try to predict the best probability of an outcome. The objective of a fault-proneness model is to identify faulty classes and focus testing efforts on them.

Faults in software systems continue to be a major problem. A software bug is an error, flaw, mistake, failure or fault in a computer program that prevents it from behaving as intended. A software fault is a defect that causes software failure in an executable product. In software engineering, the non-conformance of software to its requirements is commonly called a bug. Most bugs arise from mistakes and errors made by people in either a program's source code or its design, and a few are caused by compilers producing incorrect code. Knowing the causes of possible defects as well as identifying general software process areas that may need attention from the initialization of a project could save money, time and work. The possibility of early estimating the potential faultiness of software helps in planning, controlling and executing software development activities.

Many systems are delivered to users with excessive faults despite of a huge amount of development effort going into fault reduction in terms of quality control and testing. It has long been recognized that seeking out fault-prone parts of the system and targeting those parts for increased quality control and testing is an effective approach to fault reduction. Prediction of fault-prone modules provides one way to support software quality engineering through improved scheduling and project control.

It has been observed that during software system development, the majority of faults are found in few of its modules. In most of the cases, 55% of faults exist within 20% of source code. Using software complexity measures, the techniques build models which classify components as likely to contain faults or not. Quality will be improved as more and more faults will be detected. Predicting faults early in the software life cycle can be used to improve software process control and achieve high software
reliability. Timely predictions of faults in software modules can be used to direct cost-effective quality enhancement efforts to modules that are likely to have a high number of faults. Prediction models based on software metrics, can estimate number of faults in software modules.

To predict the fault in software data a variety of techniques have been proposed which includes statistical method, machine learning methods and neural network techniques.

- Statistical methods are used to find an explicit numerical formula, which completely determines the process of classification.
- Machine learning methods are concerned with the design and development of algorithms and techniques to extract rules and patterns out of massive data sets.
- Neural networks, which have already been applied in software engineering applications, to build reliability growth models predict the gross change or reusability metrics. A neural network is trained to reproduce a given set of correct classification examples, instead of producing formulas or rules (Rotenberg, 1999). Neural networks are non-linear sophisticated modelling techniques that are able to model complex functions. Neural network techniques are used when exact nature of input and output is not known. A key feature is that they learn the relationship between input and output through training.

Accurate prediction of fault-prone modules enables the verification and validation activities focused on the critical software components. Fault-proneness models are models that are built from information about the code and its faults and that relate code to faults. It is required that fault-prone prediction models should be efficient and accurate. Such models could be useful during both planning and executing testing activities. Therefore, software developers have a keen interest in software quality models. Planning testing activities could take advantage of information about software fault-proneness for anticipating costs and allocating activities, while test execution one can use this information to evaluate the quality of the results.

7.1.2 Approaches to Automatic Test Case Generations

The main concern in software testing process is test case designing. A test case can be defined as an input to the program, which will be fed to the program and outcomes are inspected. If outcomes are as per specifications, then software passes the testing,
otherwise it is inferred that software faults are present. Software testing can be seen as a process as depicted in Figure 7.2:

![Diagram of software testing process]

**Figure 7.2: Software Testing as a Process**

After generation of test cases one also has to execute them on software under test and observe the output to detect the faults. This is treated as automatic testing and it can also be defined in following three steps:

**STEP I:** Generating the test cases T automatically.
**STEP II:** Executing the test cases T on Program P and producing result R.
**STEP III:** Observing the results R for correctness (i.e. If R is the expected outcome then P is correct otherwise there is a failure and hence, program P contains faults).

In past, there are many techniques adopted for automating first two steps like Random testing, anti-random testing and adaptive random testing etc and back-to-back testing was used for observing the outputs for correctness. Similarly, evolutionary techniques have also been used for generating the test cases automatically. Genetic algorithm is also used to generate test cases automatically for some particular adequacy criteria. In the following sections a brief overview of random testing, ant-random testing and adaptive random testing has been provided and then use of Genetic Algorithm using metric based heuristics have been discussed in detail.

### 7.1.2.1 Random testing

Random testing is a black-box software testing technique where programs are tested by generating random, independent inputs. Results of the output are compared against software specifications to verify that the test output is pass or fail. In nutshell, it is an approach in which software is tested by choosing an arbitrary subset of all possible input values.

In random testing, test cases may be randomly chosen based on a uniform distribution or according to the operational profile. As pointed out by Hamlet, the main merits of random testing include the availability of efficient algorithms to generate its test cases.
and ability to infer reliability and statistical estimates (Weblink 14). In all random testing studies, only the rate of failure-causing inputs is used in the measurement of effectiveness. With random testing, the chances of hitting the failure patterns that is, selecting failure-causing inputs as test cases depend solely on the magnitude of the failure rate. Anti-random testing is a variation of random testing, which is the process of generating random input and sending that input to a system for test. In many situations, random test input does not have an associated expected return value. In such situations, the purpose of random testing is to try to generate a system failure of some sort, such as a hang or denial of service.

Research studies have shown that pure random testing is relatively less effective at discovering bugs than other testing paradigms, such as equivalence partition testing and boundary value analysis testing. However, random testing is appealing because it is typically quick and easy to implement. The idea of Anti-random testing appears to have been introduced by the field of hardware testing. Essentially, Anti-Random testing generates an input set of random values, where the values are as different as possible from each other. The assumption is that similar input to a system will expose similar types of bugs and therefore, an input set that contains values that are very different, can be a better option.

7.1.2.2 Adaptive random testing

Adaptive Random Testing (ART) was proposed as an effective alternative to random testing. It is an attempt to improve the failure-detection effectiveness of random testing. It is based on various empirical observations showing that many program faults result in failures in contiguous areas of the input domain, known as failure patterns. It systematically guides or filters randomly generated candidates, to take advantage of the likely presence of such patterns.

Adaptive random testing makes use of two sets of test cases, namely the executed set and the candidate set, which are disjoint. The executed set is the set of distinct test cases that have been executed but without revealing any failure while the candidate set is a set of test cases that are randomly selected without replacement. The executed set is initially empty and the first test case is randomly chosen from the input domain. The executed set is then incrementally updated with the selected element from the candidate set until a failure is revealed. From the candidate set, an element that is farthest away from all executed test cases is selected as the next test case.
7.1.2.3 Genetic Algorithm and Automatic Test Case Generation

Genetic Algorithm can be used also for automating the test case generation step. GA needs a fitness function to guide the search process. For the purpose of software testing the fitness function can be the criterion using which the test cases may be generated in the most promising areas of search space. These regions can be viewed as areas in state space that comprise of states satisfying some adequacy criteria like boundary value analysis, equivalence class partitioning, path coverage etc. In the proposed model employing Genetic Algorithm to generate the test cases automatically, distance metric based on Boundary Value Analysis and Equivalence Class Partitioning adequacy criteria (discussed in the following section) have been used to identify the most promising regions in the search space and heuristics based on distance metric is used in fitness scaling to improve the convergence of optimization process of test case generation. The various regions of interest in the input space can be identified on the basis of inputs to the program. There can be various input parameter considerations for each program. One can take these inputs to define the peaks or the regions where the test cases can belong to. In the following section some examples are discussed to illustrate the process of identifying the regions in search space.

For example, a simple program of finding greater between two numbers can be viewed as follows:

![Diagram of program for finding greater between two numbers](Image)

**Figure 7.3: View of a Program for Finding Greater Between Two Numbers**
As shown in the Figure 7.3, the inputs to this program can be divided into three regions one (Region no. 1 in the Figure) where \( A < B \), another (Region no. 2) where \( A > B \) and third (Region no. 3) where \( A = B \), where \( A, B \) are the numbers to be used as input.

In another case, requirement specification states that the two given input variables \( A \) and \( B \) are supposed to have values limited by lower bound and upper bound \( lb1, ub1 \) and \( lb2, ub2 \) respectively, then search space can be divided into thirteen different regions of interest using the boundary value analysis and equivalence class partitioning approach as depicted in the Figure 7.4. Four regions (indicated by the no. 1 to 4 in the Figure 7.4) consist of the values lying in the vicinity of boundaries identified on the basis of \( lb1, ub1, lb2 \) and \( ub2 \) and nine regions (indicated by the no. 5 to 13 in the Figure 7.4) of those values which are demarcated by the boundaries. So, test cases to be generated are required to belong to these regions in order to fully test the program.

**Figure 7.4: View of a Program having Two Decision Variables**

If the input space of a program represents the relationship in three-dimensional environment then also the regions can be found on the basis of input values as shown in Figure 7.5. Here, all the dashed and bounded regions will be the regions where test cases must be drawn.
From above discussion, it is concluded that the input space can be divided into different regions and the test case generation process must consider these regions. If the test cases generated by the generation process belongs to all the regions of input space then testing will be effective for all input values. As exhaustive testing is not feasible in all the cases, so this type of test case generation and execution may be beneficial for the testing process. Genetic algorithm is a search process which identifies promising states (i.e. test cases) which are evaluated using some fitness function. So, if the distance from the above regions will be used as fitness function to guide the search process then the test cases generated by genetic algorithm will be effective and efficient.

7.2 Metrics based heuristics in the Design of Test Data

Test data generation is basically a constraint problem which aims to find an input vector $\mathbf{v}$ such that some property $p$ is achieved. The basic notion underpinning search based approaches to test data generation is that some test data are clearly better than others. These good test data either achieve the stated aim or else come close to achieving the aim. The notion of ‘close to’ can generally be defined in a problem specific way i.e. how ‘far away’ the current test input vector is from satisfying the aim and seek to reduce this distance via the search algorithms. The ‘distance’ is of course a metric.

The dynamic test data generation problem aims to find an input vector that causes predicates $p_1, ..., p_n$ to be true at some identified points in program execution. Thus, if
a particular path is to be followed, the appropriate branch predicates must be satisfied (Harman et al., 2004). If data that breaks a specification is found then it is desired that the precondition and the negation of the post-condition to be satisfied. Statements that are associated with them might cause abnormal execution (such as underflow or overflow). But this condition is just another predicate to be broken and hence, some metric (fitness function) is needed in order to satisfy a predicate.

7.3 Proposed Metric and Heuristic

Testing is a process which needs to generate test cases. Test data adequacy criteria are helpful tools for software testers. There are two levels of software testing processes. At the lower level, testing is a process where a program is tested by feeding more and more test cases to it. Here, a test adequacy criterion can be used as a stopping rule to decide when this process can stop. Once the measurement of test adequacy indicates that the test objectives have been achieved, then no further test case is needed. Otherwise, when the measurement of test adequacy shows that a test has not achieved the objectives, more tests must be made. In this case, the adequacy criterion also provides a guideline for the selection of the additional test cases. In this way, an adequacy criterion helps the testers to manage the software testing process so that software quality is ensured by performing sufficient tests. At the same time, the cost of testing is controlled by avoiding redundant and unnecessary tests. This role of adequacy criteria has been considered by some computer scientists (Weyuker, 1986) to be one of the most important. At a higher level, the testing procedure can be considered as repeated cycles of testing, debugging, modifying program code and then testing again. Ideally, this process should stop only when the software has met the required reliability requirements. Although test data adequacy criteria do not play the role of stopping rules at this level, they make an important contribution to the assessment of software dependability.

Therefore, an adequacy criterion is an essential part of any testing method. It plays two fundamental roles. First, an adequacy criterion specifies a particular software testing requirement and hence, determines test cases to satisfy the requirement. The second role that an adequacy criterion plays is to determine the observations that should be made during the testing process.

Although, given an adequacy criterion different methods could be developed to
generate test sets automatically or to select test cases systematically and efficiently, the main features of a testing method are largely determined by the adequacy criterion.

Adequacy criteria is used instead of exhaustive testing as it is not possible to run all test cases because it is not feasible to cover all test suite as the search space is huge (as discussed in the chapter 6, testing can be viewed as a problem of searching an optimal state in a state space). Searching techniques are used in order to solve optimization problems. If there is absence of known optimal solution to a problem, then clearly there is a need to use a search-based approach to seek optimal (or near optimal) solutions.

So, the distance (i.e. a distance metric indicating the distance between actual and desirable) can be used as a guide to generate test cases. Those test cases which are having minimum distance from desired boundaries are more likely to lie in the better test case region as they will satisfy more adequacy criterion. Hence, the heuristic with the distance metric can be stated as “to minimize the distance of test cases with the boundaries of domain values”. This is done with some experiment variables and their domain values. Before that the two main approaches used to design test cases are described as follows:

- **Boundary Value Analysis**
  
  It has been observed that greater number of errors occurs at the boundary of input domain rather than in the centre. It is for this reason that boundary value analysis has been developed as a testing technique that leads to a selection of test cases that exercise boundary values.

  In this case, if an input condition specifies a range bounded by values a and b, then test cases should be designed with values a and b and just above and just below a and b. And if an input condition specifies a number of values then test cases should be developed with values just above and below minimum and maximum.

  In the past, it has been observed that test cases lie in different classes. And programmers commit more mistakes in some classes of test cases. Most important class of these is boundary values as there are more chances for software to fail at boundaries. So, automatic test case generation must focus on these kinds of test case classes which are more crucial for software. Boundary values of a program are described as input boundaries of the variables used in program. For the sake of
drawing the problem a function F is used. F is a function of two variables, x1 and x2. When the function F is implemented as a program, these input variables will have some boundaries: a <= x1 <= b; c <= x2 <= d; The input space of function F is shown in Figure 7.6. Any point within the shaded rectangle is a legitimate point to the function F.

![Figure 7.6: Input Domain of a Function of Two Variables](image)

Boundary value analysis focuses on the boundary of the input space to identify test cases. So, the basic idea is to select five test cases values i.e., minimum, just above the minimum, nominal value, just below the maximum and the maximum as shown in Figure 7.7.

![Figure 7.7: Boundary Value Analysis Test Cases for a Function of Two Variables](image)

Hence, it is clear from the Figure 7.7, where boundary value analysis tests cases are represented that boundary value analysis does not make sense for Boolean variables. Because the extreme values for these variables are TRUE and FALSE. So, the other three slots can't fill for boundary value analysis. Also, boundary
value analysis works well in programs where the program is function of several independent variables that represent bounded physical quantities.

- **Equivalence Class Partitioning**
  Equivalence class partitioning is a black-box testing method that divides the input domain of a program into set of equivalence classes of data for which test cases can be derived. For example, if P is a program to compare two variables x and y, then the valid values of x (from 0 to some INT_MAX) and y (from 0 to some INT_MAX) will divide input of program P into valid and invalid classes as shown in Figure 7.8.

![Figure 7.8: Equivalence Partitioning Test Cases for a Function Of Two Variables (Weblink 13)](image)

Test-case design for equivalence class partitioning is based on an evaluation of equivalence classes for an input condition. An equivalence class represents a set of valid or invalid states for input conditions. Typically, an input condition is a specific numeric value, a range of values, a set of related values or a Boolean condition. These classes are defined according to the following guidelines:

- a) If an input condition specifies the range, one valid and two invalid equivalence classes are defined.
- b) If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
- c) If an input condition specifies a member of a set, one valid and one invalid equivalence classes are defined.
- d) If an input condition is Boolean, one valid and one invalid equivalence classes are defined. By applying the above guidelines test cases for each input domain data item can be developed and executed.
In this research, the researcher carried out the identification of these boundaries automatically through Genetic algorithm, random search and proposed some solutions for genetic algorithm. Genetic algorithm and random testing both starts with some random initial population and then GA uses the fitness of individuals to progress towards the optimums, whereas random testing works randomly throughout the run. For this experiment, the proposed metric (distance from the boundaries) is taken to compute the fitness of the individual chromosome. The algorithms are coded in MATLAB 2012a.

7.4 Genetic Algorithm for Test Case Generation
The proposed genetic algorithm for test case generation for boundary value analysis and equivalence class partitioning is presented in the following section. Firstly, the major components of GAs are discussed and then overall algorithm is presented.

7.4.1 Representation
In proposed GA, value encoding is used in the chromosome i.e. real values are used to represent the input variables $x_1, x_2, ..., x_n$ of the program. The length of the chromosome depends on the number of variables.
Suppose one wish to generate test cases of a program P with input variables $x_1$ and $x_2$. Each of which has their value ranges within some domain say $[c, d]$. Then the values representing the chromosome are $[a_1, a_2]$, where $a_1$ and $a_2$ are the ranges within the respective variables domain $[c, d]$.

7.4.2 Initial population
Initial population is generated randomly as discussed in Representation section. In order to generate initial population, pop_size vectors of c_size length are generated randomly, where pop_size is the size of population, and c_size is the number of variables. For the search space values, initial population is generated with 5 less than lower bound of variable and 5 more than upper bound values, i.e. if lower bound and upper bound are 5 and 15 respectively, then initial values are generated from 0 (5-5) to 20 (15+5). Different values of pop_size are taken under experiments and the best are chosen.
7.4.3 Fitness Function for Boundary Value Analysis

Fitness of each chromosome is determined by its difference from the boundaries of the variable. The more a variable is close to the boundaries the more, it is declared fit. Difference is used as a metric and a common heuristic “minimize this distance” is used.

Fitness(popsize, chromLength, curpop)

- lBound = lower boundary of the variable;
- uBound = upper boundary of the variable;

for I = 1 to popsize
    for j = 1 to chromLength
        diffLower = lBound – curpop(i,j);
        diffUpper = uBound – curpop(i,j);
        moreClose = min(diffLower, diffUpper);
        fitness(i) = moreClose;
    end
end

7.4.4 Fitness Function for Equivalence Class Partitioning

Fitness of each chromosome is determined by its difference from a point p where, the location of point p is somewhere in the middle of two boundaries. The more a variable is close to the point p, the more it is declared fit. So, the heuristic that can be used is “to minimize the distance between the point p and test data”.

Fitness(popsize, chromLength, curpop)

- lBound = lower boundary of the variable;
- uBound = upper boundary of the variable;
- centre = (lbound + ubound) / 2;
- diff = centre - lbound;

for I = 1 to popsize
    for j = 1 to chromLength
        c1 = lbound - diff;
        c2 = lbound + diff;
        c3 = ubound + diff;
diff1 = c1 - curpop(I,j);
diff2 = c2 - curpop(I,j);
diff3 = c3 - curpop(I,j);
moreClose = min(diff1, diff2,diff3);
fitness(i) = moreClose;
end
end
end

7.4.5 Fitness Scaling
The performance of genetic algorithms is essentially improved if fitness scaling is used, i.e., use f(J(xi)) instead of J(xi) as a fitness value, where f(x) is some fixed function known as scaling function. The efficiency of fitness scaling essentially depends on the choice of f.

Fitness scaling converts the raw fitness scores that are returned by the fitness function to values in a range that is suitable for the selection function. Following are the reasons for using fitness scaling:

a) If the value for objective function is negative.

b) To avoid premature convergence i.e. appearances of few super individuals which dominate the selection process and slow the process.

c) In some situations there is no objective function at all. So, it is desired to find parameters for which some system performs in the best possible way. It can be figure out when the performance is better and when it is worse. So, there is a ranking of all possible behaviors.

d) Fitness scaling is performed in order to avoid slow finishing.

All scaling functions can be divided into three categories: Linear, Sigma Truncation and Power Law. Linear scaling methods (such as $f_{\text{linear}}$ below) usually have constants that are not problem dependent, but that may depend on the population characteristics (max, min, mean, etc). Sigma methods include problem depended data. Power scaling takes into account the raw fitness values. Following are the scaling methods used in this research:

a) Linear Scaling: One of the most common scaling techniques is traditional linear scaling. This scaling remaps the fitness values of each individual using the
following equation:

\[ F_{\text{linear}} = a + b f_{\text{raw}} \]  
Eq. (7.1)

Where \( a \) and \( b \) are constants defined by the user. For these trials, the values of \( a \) and \( b \) were tied to specific characteristics of the population.

b) Ranked Scaling: Another scaling option is rank scaling. This is more of a two-step process. First, all individuals are sorted (ranked) by their raw fitness scores. Then, new fitness values are computed solely based on their rank using the equation:

\[ f_{\text{ranked}} = p - 2 \cdot \frac{(r - 1)(p - 1)}{N - 1} \]  
Eq. (7.2)

Where \( r \) is the rank of the individual, \( p \) is the desired selection pressure (best/median ratio) and \( N \) is the size of the population.

c) Exponential scaling: Exponential scaling also begins with ranking all the individuals. But, the new fitness values are instead computed with the following equation:

\[ f_{\text{exponential}} = m^{(r-1)} \]  
Eq. (7.3)

Where each individual’s new fitness is \( m \) times greater than the previous fitness of individual. Low \( m \) can result in high selection pressure and high \( m \) will result in low selection pressure. Low pressure means premature convergence, possibly isolating the entire population in a local maximum and not the true maximum.

d) Top scaling: Top scaling is probably the simplest scaling method. Using this approach, several of the top individuals have their fitness set to the same value (which is proportional to the population size), with all remaining individuals having their fitness values set to zero. This simple concept yields

\[ f_{\text{top}} = s.N \text{ for } r \geq c \text{ and } 0 \text{ for } r < c \]  
Eq. (7.4)

Where \( s \) is some proportionality constant, \( c \) is the number of individuals that will be scaled up and \( N \) is the size of the population. Since this gives several individuals identical fitness levels, regardless of how different their raw scores might be, the diversity of the succeeding generations is increased.

All the four scaling options have arbitrary user inputs. In an attempt at fairness, the values used in the following tests were selected randomly, but only after they proved not to dramatically affect the outcome of the trials. This was done so that no one scaling method would have an advantage due to better tuning by the user. Based on above facts test cases were generated based on the approach shown in Figure 7.9:
Following are the parameters used for generating test cases using above three techniques (as shown in Table 7.1) and some results of experiments are shown in the Figures below.

**Table 7.1: Parameters for Genetic Algorithm**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population Size</td>
<td>50</td>
</tr>
<tr>
<td>No. Of variables</td>
<td>2</td>
</tr>
<tr>
<td>Selection Operator</td>
<td>Roulette Wheel Selection</td>
</tr>
<tr>
<td>Crossover Operator</td>
<td>Arithmetic Crossover</td>
</tr>
<tr>
<td>Generation count</td>
<td>100</td>
</tr>
<tr>
<td>Boundaries of variables</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Case I:</td>
</tr>
<tr>
<td></td>
<td>Lower limit – 5</td>
</tr>
<tr>
<td></td>
<td>Upper limit – 15</td>
</tr>
<tr>
<td></td>
<td>Case II:</td>
</tr>
<tr>
<td></td>
<td>Lower limit – 2</td>
</tr>
<tr>
<td></td>
<td>Upper limit – 20</td>
</tr>
</tbody>
</table>
Below Figure 7.10 shows the output of Test Cases Generated with Random Search with Case-I for boundary values of variables from Table 7.1.

Figure 7.10: Test Cases Generated with Random Search with Case-I for Boundary Values

Below Figure 7.11 shows the output of Test Cases Generated with Random Search with Case-II for boundary values of variables from Table 7.1.

Figure 7.11: Test Cases Generated with Random Search with Case-II for Boundary Values
Below Figure 7.12 shows the output of Test Cases Generated with Genetic Algorithm with Case-I for boundary values of variables from Table 7.1.

![Figure 7.12: Test Cases Generated with Genetic Algorithm with Case-I for Boundary Values](image)

Below Figure 7.13 shows the output of Test Cases Generated with Genetic Algorithm with Case-II for boundary values of variables from Table 7.1.

![Figure 7.13: Test Cases Generated with Genetic Algorithm with Case-II for Boundary Values](image)
Below Figure 7.14 shows the output of Test Cases Generated with Genetic Algorithm with fitness scaling with Case-I for boundary values of variables from Table 7.1.

![Figure 7.14: Test Cases Generated with Genetic Algorithm with Various Fitness Scalings with Case-I for Boundary Values](image)

Below Figure 7.15 shows the output of Test Cases Generated with Genetic Algorithm with fitness scaling with Case-I for boundary values of variables from Table 7.1.

![Figure 7.15: Test Cases Generated with Genetic Algorithm with Various Fitness Scalings with Case-II for Boundary Values](image)

Following Table 7.2 shows some sample results from above experiments with the parameters discussed in Table 7.1.
Table 7.2 Sample Test Data Values Generated from Implementation in Different Algorithms

<table>
<thead>
<tr>
<th>Algorithms generating test cases with adequacy criteria</th>
<th>Sample test cases with Genetic Algorithm</th>
<th>Sample test cases with Random Search</th>
<th>Sample test cases with GA with Fitness Scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable 1 Lower limit 2</td>
<td>1,3,2,4,2</td>
<td>1,4,2,7,9</td>
<td>2,2,1,3,2</td>
</tr>
<tr>
<td>Upper limit 20</td>
<td>16,18,20,18,22</td>
<td>15,20,24,23,28</td>
<td>19,20,20,18,21</td>
</tr>
<tr>
<td>Lower limit 5</td>
<td>3,5,4,7,9</td>
<td>2,5,9,3,8</td>
<td>4,5,6,3,8</td>
</tr>
<tr>
<td>Upper limit 15</td>
<td>13,16,15,18,20</td>
<td>11,24,17,19,22</td>
<td>15,14,16,15,18</td>
</tr>
<tr>
<td>Variable 2 Lower limit 2</td>
<td>5,2,2,1,2</td>
<td>4,10,2,8,9</td>
<td>3,2,2,1,2</td>
</tr>
<tr>
<td>Upper limit 20</td>
<td>15,17,21,20,22</td>
<td>13,22,27,21,20</td>
<td>20,19,20,18,21</td>
</tr>
<tr>
<td>Lower limit 5</td>
<td>7,5,4,6,9</td>
<td>3,5,7,1,8</td>
<td>6,3,5,8</td>
</tr>
<tr>
<td>Upper limit 15</td>
<td>14,18,21,23,15</td>
<td>24,15,18,26,14</td>
<td>14,15,15,17,15</td>
</tr>
</tbody>
</table>

Above results show that genetic algorithm with fitness scaling generate test cases with much better accuracy and therefore, this is a better approach for generating automatic test cases. If a good fitness function is provided to GA it outperforms random searching. So, based on above conclusions further models were introduced for execution of these test cases generated by genetic algorithm with fitness scaling.

7.5 Models for Automatic Test Data Generation

The process of generating the test cases can be automated using various approaches like random, anti-random, adaptive random approach or using GA, but validating the outcome of the software under test is a major issue. If it is carried out manually, then it will be a very cumbersome process and will consume a lot of efforts. So, if it takes a lot of time to execute the test cases on software, then it is better to observe the outputs and then compare observed results with expected ones to find the similarities and dissimilarities. Also, it is better to automate the third step in order to save a lot of time saved. This is a process which needs more consideration than generating the test cases. As different programs behave differently, there must be a program dependant aspect to do the things automatically.

In order to automate the software testing process, some models have been reviewed and some modifications and guidelines have been proposed for those models. The
ideas were taken to implement the models from defensive programming, design by contract, Hoare specification, fault tolerance (N-version programming & Recovery block technique) which have been discussed in the following sections in brief.

7.5.1 Defensive Programming
Defensive Programming suggests that the module should be as much general as possible. In particular, this encourages programmers to include as many checks as possible, even if they are redundant with checks made by callers. Reliability requires systematic approach like defensive programming.

7.5.2 Design by Contract
Software is developed for some predefined requirements, not just with some hypothetical assumptions. Not all of the modules of software are built from scratch. Some of them are developed by hired developers and thus, design by contract (Meyer, 1992) is required. A contract must be signed whenever another party is included in software development. In daily life also, when any task is needed to be performed by someone else then an oral or written contract is signed and an agreement is prepared. For example, when a buyer goes to a furniture shop to purchase a bed then, the worker makes bed as per buyer’s specifications. The buyer pays some money to the worker and in return they provide the required bed with specified requirements. This is the end of contract. Same theory also works in case of software development. When some particular module is to be purchased or developed from other developers, then its reliability is a major concern and there is a need to sign a design contract. So, design by contract plays a vital role in developing modules which are to be hired by someone else for their use.

7.5.3 Hoare’s Specification
Contract between client and implementer can also be understood in terms of some preconditions and post-conditions. Where precondition is defined as a predicate describing the condition at which the function relies on for correct operation. And post condition is defined as a predicate describing the condition that the function establishes after correctly running. With these, the correctness with respect to the specifications is guaranteed. In other words, if the client of a function fulfils the function’s precondition, the function will execute to completion and when it
terminates, the post condition will be true. This is termed as Hoare’s condition or Hoare’s logic or Hoare’s Triples in literature (Hoare, 1995).

The syntax of triple is defined as:

\( \{P\} S \{Q\} \)

Where \( P \) and \( Q \) are predicates of preconditions and post conditions respectively and \( S \) is the program. It simulates as: if one start in a state where \( P \) is true and execute \( S \), \( S \) will terminate in a state where \( Q \) is true. Some simple examples for this are:

\[
\{x = y\} x := x + 5 \{ x = y + 5\}
\]

\[
\{x > 0\} x := x * 3 \{ x > -3\}
\]

\[
\{x=a\} \text{if} (x < 0) \text{then} x := -x \{x=|a|\}
\]

These example shows that if \( \{P\} \) is true and then statement \( S \) is executed then \( \{Q\} \) must be true after full and correct execution of statement \( S \). There are more terms associated with Hoare’s triple for their strength in terms of reliability. As there can be many post conditions so strongest post condition is defined as

If \( \{P\} S \{Q\} \) and for all \( Q' \) such that \( \{P\} S \{Q'\} \), \( Q \Rightarrow Q' \), then \( Q \) is the strongest post-condition of \( S \) with respect to \( P \).

For example,

\[
\{x = 3\} x := x * 2 \{ x > 0 \}
\]

\[
\{x = 3\} x := x * 2 \{ x = 6 \}
\]

Triple second is stronger than first as here \( x=6 \) implies that \( x>0 \).

Weakest precondition is defined as

If \( \{P\} S \{Q\} \) and for all \( P' \) such that \( \{P'\} S \{Q\} \), \( P' \Rightarrow P \), then \( P \) is the weakest precondition wp(S,Q) of \( S \) with respect to \( Q \)

For example,

\[
\{x = 6 && y = 12\} z := x / y \{ z < 1 \}
\]

\[
\{x < y && y > 0\} z := x / y \{ z < 1 \}
\]

\[
\{y \neq 0 && x / y < 1\} z := x / y \{ z < 1 \}
\]

Then Triple third is the weakest precondition as it allows to call the program in most general condition. Relation between Hoare’s triple and these special preconditions is defined as \( \{P\} S \{Q\} \) holds if and only if \( P \Rightarrow \text{wp}(S,Q) \).

In other words, a Hoare Triple is still valid if the precondition is stronger than necessary, but not if it is too weak. Hoare triples are used in proving the correctness of various statements like conditions and looping constructs.
7.5.4 Fault Tolerance

It is the realization that there is always faults (or the potential for faults) in the system and that the design of system is to be in such a way that it will be tolerant of those faults. That is, the system should compensate for the faults and continue to function. The general approach for building fault tolerant systems is redundancy which may be applied at several levels (Weblink 15).

- Information redundancy seeks to provide fault tolerance through replicating or coding the data. For example, a Hamming code can provide extra bits in data to recover a certain ratio of failed bits. Parity memory, ECC (Error Correcting Codes) memory and ECC codes on data blocks are sample uses of information redundancy.

- Time redundancy achieves fault tolerance by performing an operation several times. Timeouts and re-transmissions in reliable point-to-point and group communication are examples of time redundancy. Time redundancy is useful in the presence of transient or intermittent faults. It is of no use with permanent faults. An example of this type of redundancy is TCP/IP’s retransmission of packets.

- Physical redundancy deals with devices and not on data. Addition of extra equipment to enable the system to tolerate the loss of some failed components. Examples of physical redundancy are RAID disks and backup name servers.

When addressing physical redundancy, one can differentiate redundancy from replication. With replication, one has several units operating concurrently and a voting system to select the outcome. With redundancy, only one unit is functioning while the redundant units are standing by to fill in, in case the unit ceases to work.

7.5.5 Levels of Availability

In designing a fault-tolerant system, it must be realized that 100% fault tolerance can never be achieved as it will be very costly. To design a practical system, it is important to consider the degree of replication needed. This will be obtained from a statistical analysis for probable acceptable behavior. Factors that contribute to this analysis are the average worst-case performance in a system without faults and the average worst-case performance in a system with faults. Availability is usually expressed as a percentage of uptime in a given year. The following Table 7.3 shows...
the downtime that will be allowed for a particular percentage of availability, presuming that the system is required to operate continuously. Five nines is the classic standard of availability for telephony. Achieving it entails intensive software testing, redundant processors, backup generators, and earthquake-resilient installation (Weblink 15). If all that ever happens to a system is that lose of power for a day once a year then the reliability is at 99.7% ("two nines"). Availability percentage can be computed by dividing the minutes of uptime by the minutes in a year (or hours of uptime by hours in a year, or anything similar). For example, if a system is expected to be down for three hours, a year on average then the uptime percentage is 1-(180 minutes / 525600 minutes) = 99.97%. The following Table 7.3 shows some availability levels, their common terms, and the corresponding annual downtime.

**Table 7.3: Availability Terminologies**

<table>
<thead>
<tr>
<th>Availability %</th>
<th>Downtime per year</th>
<th>Downtime per month</th>
<th>Downtime per week</th>
</tr>
</thead>
<tbody>
<tr>
<td>90% (&quot;one nine&quot;)</td>
<td>36.5 days</td>
<td>72 hours</td>
<td>16.8 hours</td>
</tr>
<tr>
<td>95%</td>
<td>18.25 days</td>
<td>36 hours</td>
<td>8.4 hours</td>
</tr>
<tr>
<td>97%</td>
<td>10.96 days</td>
<td>21.6 hours</td>
<td>5.04 hours</td>
</tr>
<tr>
<td>98%</td>
<td>7.30 days</td>
<td>14.4 hours</td>
<td>3.36 hours</td>
</tr>
<tr>
<td>99% (&quot;two nines&quot;)</td>
<td>3.65 days</td>
<td>7.20 hours</td>
<td>1.68 hours</td>
</tr>
<tr>
<td>99.5%</td>
<td>1.83 days</td>
<td>3.60 hours</td>
<td>50.4 minutes</td>
</tr>
<tr>
<td>99.8%</td>
<td>17.52 hours</td>
<td>86.23 minutes</td>
<td>20.16 minutes</td>
</tr>
<tr>
<td>99.9% (&quot;three nines&quot;)</td>
<td>8.76 hours</td>
<td>43.8 minutes</td>
<td>10.1 minutes</td>
</tr>
<tr>
<td>99.95%</td>
<td>4.38 hours</td>
<td>21.56 minutes</td>
<td>5.04 minutes</td>
</tr>
<tr>
<td>99.99% (&quot;four nines&quot;)</td>
<td>52.56 minutes</td>
<td>4.32 minutes</td>
<td>1.01 minutes</td>
</tr>
<tr>
<td>99.999% (&quot;five nines&quot;)</td>
<td>5.26 minutes</td>
<td>25.9 seconds</td>
<td>6.05 seconds</td>
</tr>
<tr>
<td>99.9999% (&quot;six nines&quot;)</td>
<td>31.5 seconds</td>
<td>2.59 seconds</td>
<td>0.605 seconds</td>
</tr>
<tr>
<td>99.99999% (&quot;seven nines&quot;)</td>
<td>3.15 seconds</td>
<td>0.259 seconds</td>
<td>0.0605 seconds</td>
</tr>
</tbody>
</table>
7.5.6 Designing Fault Tolerant Systems
The two most popular approaches used to design systems which are fault-tolerant are N-version programming and Recovery block technique.

7.5.6.1 N-Version Programming
Every program has some particular implementation of requirement specifications. One can use multiple versions of the same program and then run them in parallel to check for the quality and reliability of the system (Avizienis, 1975). The usual method to attain reliability of software operation is fault-avoidance (or intolerance). All software defects are eliminated prior to operation. If some defects remain, the operation is reliable only as long as the defects are not involved in program execution. In most large and complex software systems these fault-avoidance conditions have not been successfully attained, regardless of a very large investment of effort and resources and software crashes have occurred during operation. This observation leads to the conjecture that for reliable software operation, redundant software in some form is required to detect, to isolate or to recover from effects of the un-eliminated software defects. N-version Programming is based on triple modular redundancy (TMR), sometimes called triple-mode redundancy. TMR is a fault-tolerant form of N-modular redundancy, in which three systems perform a process and then result is processed by a majority-voting system to produce a single output. If any one of the three systems fails, the other two systems can correct and mask the fault. If the voter fails, then the complete system will fail. This model is used in hardware successfully, but it can't be used in software as it is, because software needs redundancy as well as diversity in the versions. If all versions are same (redundant), and one version fails on some input, then all versions will fail on that input, so just redundancy will not solve the problem. Software requires diversity, so version written with different logics, different algorithms or different programmers etc are used for developing multiple versions of the software. These run in parallel and a voter decides the best among these (Avizienis, 1975).

Figure 7.16 depicts the working of TMR model. N-version programming is defined as the independent generation of \( N \geq 2 \) functionally equivalent programs, called "versions", from the same initial specification, since it bears no implication about the distinctness which is vague and difficult to quantify or even qualify, among the N
versions of a program. "Independent generation of programs" here means that the programming efforts are carried out by N individuals or groups that do not interact with respect to the programming process. Wherever possible, different algorithms and programming languages or translators are used in each effort.

Figure 7.16: Triple Modular Redundancy Model

The general steps of N-version programming are:
1. An initial specification of the requirements of the software is developed. The specification should unambiguously define each and every aspect of requirement (Chen et al., 1995), (Avizienis, 1975).
2. From the specifications, two or more versions of the program are independently developed, each by a group that does not interact with the others. The implementations of these functionally equivalent programs use different algorithms and programming languages as well. The resulting programs are called N-version software (NVS) (Chen et al., 1995). Some N-version execution environment (NVX) is developed which runs the N-version software and makes final decisions of the N-version programs as a whole on the basis of output generated from each individual N-version program. The implementation of the decision algorithms can vary ranging from simple as accepting the most frequently occurring output to some more complex algorithm (Chen et al., 1995), (Avizienis, 1975).

The fundamental principle of NVP is that there are multiple implementations done by different development teams. The same developer cannot make multiple
implementations with multiple understandings of the same specification. The assumption is that when a latent fault activates in one version, then the other versions of the system do not contain the same fault. Research by Knight and Leveson (Knight et al., 1986), (Knight et al., 1990) has shown that the assumption of independence between the faults produced by the different design teams does not hold.

7.5.6.2 Recovery Block

The idea of second model was taken from recovery block technique. Recovery block was first introduced by Horning (Horning et al., 1974). This scheme is analogous to the cold standby scheme for hardware fault tolerance. In this approach, multiple variants of software which are functionally equivalent are deployed in a time redundant fashion. An acceptance test is used to test the validity of the result produced by the primary version. If the result from the primary version passes the acceptance test, this result is reported and execution stops. On the other hand, if the result from the primary version fails the acceptance test, another version from multiple versions is invoked and the result produced is checked by the acceptance test. The execution of the structure does not stop until the acceptance test is passed by one of the multiple versions or until all the versions have been exhausted.

In this model, the alternates correspond to the variants of Program, and the acceptance test to the acceptor. On entry to a recovery block, the state of the system must be saved to permit backward error recovery, i.e., establish a checkpoint. The primary alternate is executed and then the acceptance test is evaluated to provide an acceptance on the outcome of this primary alternate. If the acceptance test is passed, then the outcome is regarded as successful and the recovery block can be exited, discarding the information on the state of the system taken on entry (i.e., checkpoint). However, if the test fails or if any errors are detected by other means during the execution of the alternate, then an exception is raised and backward error recovery is invoked. This restores the state of the system to what it was on entry. After such recovery, the next alternate is executed and then the acceptance test is applied again. This sequence continues until either an acceptance test is passed or all alternates have failed the acceptance test. If all the alternates either fail the test or result in an exception (due to an internal error being detected), a failure exception will be signalled to the environment of the recovery block. Since recovery blocks can be
nested, and then the raising of such an exception from an inner recovery block would invoke recovery in the enclosing block.

In Recovery Block, the design of the acceptance list becomes difficult as it must be independent of the computation used. Moreover, Recovery Block has problems for Real Time Systems because of the sequential operation of the redundant versions.

The significant differences in the recovery block approach from N-version programming are that only one version is executed at a time and the acceptability of results is decided by a test rather than by majority voting. The recovery block technique has been applied to real life systems and has been the basis for the distributed recovery block structure for integrating hardware and software fault tolerance and the extended distributed recovery block structure for command and control applications. Modeling and analysis of recovery blocks are described by Tomek (Tomek et al., 1993).

7.6 Testing Models

Finally on the basis of the concepts discussed in the 6th chapter and 7th chapter of the thesis two different models have been used.

7.6.1 First Model based on N-Version Programming

First is based on N-Version programming. According to this model, the test cases are to be generated automatically using Genetic Algorithm. Problem is viewed as state space and space consists of all possible test cases. Good test cases are identified using GA. To evaluate the test cases, fitness function is used based on the proposed distance metric (derived from boundary value analysis and equivalence class partitioning), to further improve the optimization process, fitness scaling was carried out using the heuristic derived from distance metric. N-versions of the software under test should be written and the test cases generated using GA will be input to those versions and outputs produced by those models will be compared automatically and in case of disagreement the conflict will be resolved on the basis of majority and faulty unit will be identified for debugging.

Figure 7.17 demonstrates the proposed model for automation of software testing using N-version programming.
First model resembles on N-Version programming (NVP) concept of software engineering.

### 7.6.2 Second Model Based on Recovery Block

The idea of the 2nd model was taken from recovery block, defensive programming, design by contract and Hoare’s specification. In all these concepts the output generated by the model is to be validated. The proposed model is depicted in the Figure 7.18.

In this model, to test a function F, a function F’ (Complement of F) should be written such that if function F transforms some input X into Y, then F’ will transform Y into X. Therefore, if the input of F is equal to output of F’, it means that module X is correct. This model can be viewed in Figure 7.19.
The Proposed recovery block mechanism is described as the acceptance test for various programs. Every Function F has been checked by another complement version of that function say F’, which will complement the function F, so that automatic testing can be done. In this model, the test cases are fed to the Function F under test, the output of the F will be fed to Function F’ (complement of F), & output of F’ will be compared to inputs of F, and in case of disagreement it can be inferred that either F or F’ is a faulty unit. In other words, If F is a Function which map input A to B, then one can write a Function F’ which will take B as input and maps it to A. Now, if the input of F and output of F’ matches, they both will test each other by functionality. For example, if one writes a Function to find the factorial (N!) of given number N, then one can also write a Function to find the number N with given factorial (N!). So, if one input is given to first Function say 6 and 720 is the output, then this 720 can be given as an input to second Function and its output can be observed, if it is 6 then both the programs do the testing of each other. If these don’t match, then either of these can be wrong, so necessary actions are taken.

7.7 Guidelines for Proposed Model
Both of the models discussed above are tested with some sample programs and some
observations have been made. N-version model can be applied to all of the programs for testing as all programs can be written in different ways or versions. The problem in this case is that the cost of coding is multiplied by at least three. Because, if two versions of the programs are used and if there is a disagreement between the two after test case execution, then it is hard to decide which of the version is faulty, so one must go with at least three versions. This makes the cost of coding three times more than the normal one. On the other hand, the recovery block based model approach uses the complement of the Function. So only two functions are to be written that is F and F’. These models were tested with many cases. A number of programs are used for experiment and it is found that the programs can be divided into two broad categories, first where the acceptance function F’ can be written easily. In these cases, the function F has one input and one output. For example, a function to compute factorial of a given integer, in such function there is one input i.e. N and one output i.e. N!. In this case one has to write a function F to transform input N to N! and another function F’ to transform N! to N. In the 2nd category, where the inputs of Function F are more than one, then F’ is not just the complement Function. For example, given an unsorted array X of N integer, it is easy to write F to sort that array in ascending order and producing a sorted array Y but to write a function to transform Y array into X array, is not easy. It needs some modification. The things are summarized below:

1. One-input Function: Programs which takes one parameter as their input and generate one output. In these, F’ can be written as complement of Function. The output of Function F is used as input to F’ and F’ converts it back to the main input.

2. More-input Function: Programs which takes more than one parameters as its input. Here the complement Function may not work always as it leads to ambiguous things like matrix multiplication Function where it’s ok to produce resultant matrix C which is equal to product of A and B matrix, but it’s not obvious to get back A and B from C alone.

Some sample programs from each category are taken and observed as follows for their acceptance test:

1. One-Input Function: Following are some programs which needs one parameter as input and their acceptance tests:
   
   a) Factorial of a number: Find out the factorial of given number.
Input: An Integer n whose factorial is needed.

Output: Factorial of input n.

Both the models can be applied effectively and efficiently. As many algorithms are available for finding factorial using loop or recursion using different programming languages. An acceptance test can be written as a Function $F'$ which tells the number whose factorial is given as input to it. Figure 7.20 depicts the stated procedure:

![Figure 7.20: Acceptance Test for Factorial Function](attachment:factorial_function.png)

b) Next Date: Find the next date for the input current date.

Input: Current Date

Output: Next date of the input date

In this example, user enters a date and Function $F$ is required to find the next date of it. In this case also, both the models can be applied as there are many algorithms for finding the next date. For the acceptance test, a Function $F'$ is to be developed which tells the previous date of the date given as input to it. The functionality is depicted in Figure 7.21.

![Figure 7.21: Acceptance Test for Next Date Function](attachment:next_date_function.png)
c) Fibonacci number: Find out the Fibonacci number of the input index.

Input: Index of the Fibonacci sequence

Output: Corresponding number in the Fibonacci sequence

Fibonacci sequence is 0, 1, 2, 3, 5, 8, 13, 21, 34 and so on. It is generated by adding previous two numbers. A number (index) is to be given as input and then output the Fibonacci number of that particular index i.e. input 5 gives value 3 and input 10 gives value 34. There are many algorithms to find it. These can be used and applied for developing many versions. Also one can make a Function which takes the number as input and tells its index in Fibonacci sequence if exists. The process is presented in Figure 7.22.

![Figure 7.22: Acceptance Test for Fibonacci Number Function](image)

d) Reverse a Number: Find the reverse of given n-digit number.

Input: n-digit integer number

Output: n-digit reverse integer number

In this, user enters a number and Function is required to find the reverse of that number. In this case also, both the models can be applied as there are many algorithms available to find the reverse. For the acceptance test, the same Function can be used which takes the reversed number as input and tells back the reverse of it, if it matches the first input then reverse is correct otherwise there is an error. In this case, F and F’ are same. Figure 7.23 shows the above concept.
2. Two-input Function: Following are some programs which needs more than one parameter as input and their acceptance tests:

a) Searching a number: Search the input number in given list and output the location of that (if found) otherwise NULL is output.

   Input: List of Numbers & Number to find out
   Output: index of number in the list if found otherwise NULL

When a number is to be searched in a given list, it will return that index where the number exists in the list. There are a number of algorithms available for searching operation like linear search, binary search etc. So, n-version model can be applied easily to it. For the acceptance test, usual complement Function will not work in this case as from the index it is not possible to get back both the inputs as the output of some Function. Hence, acceptance test can be written that takes the list of numbers and output of Function F as inputs and returns the number at that index in given list, provided the input number and original numbers can be checked for equality. Figure 7.24 states the concept:
b) Matrix Multiplication: Find the multiplication of two matrices A and B of size MxN and NxP respectively.

Input: Two matrices A and B of size MxN and NxP respectively.

Output: One matrix C of size MxP as product of A and B

Matrix multiplication is a traditional problem to be solved by a naive to programming. This can be implemented in many ways by using different programming languages. For acceptance test, a Function F’ is to be developed which takes as input the product of two matrices i.e. C and one of the two matrices A & B and return another matrix. If this matrix equals the other original matrix, then the Function is correct. Figure 7.25 shows the desired functionality.

![Diagram of Matrix Multiplication Function](image)

Figure 7.25: Acceptance Test for Matrix Multiplication Function

c) Sorting of Numbers: It will sort the given array according to given order.

Input: List of numbers as array A

Output: Sorted numbers as array A_sorted

A number of sorting techniques are available for making n-versions. The second model needs some modification in this case. Because if A is the list of unsorted numbers, P is the Function to sort these numbers and G is a Function to unsort some given sorted numbers, then it is not guaranteed that output of G will be equal to A. For example, if A = {4, 5, 3, 8, 1, 12, 43, 11}, then P produce sorted output as {1, 3, 4, 5, 8, 11, 12, 43} and when G tries to unsort these numbers, then there are a number of ways that can be used to unsort it, say {3, 8, 12, 43, 5, 4, 1, 11}. The generated unsorted set is not equal to A. So, Function specification has been met but the purpose
of complement Function is not fulfilled. Therefore, it is implemented using the following approach:

If A is the input list and Function F sort this as \( A_{\text{Sorted}} \), then \( A_{\text{Sorted}} \) is fed to Function G and B is obtained as output, then again this B is fed back to Function F and output is generated as \( B_{\text{Sorted}} \). The generated lists (\( A_{\text{Sorted}} \) and \( B_{\text{Sorted}} \)) must be equal. If the desired result is obtained, then the sorting Function is error-free otherwise, there is some error. Figure 7.26 depicts the stated procedure:

![Figure 7.26: Acceptance Test for Sorting of N Numbers Function](image)

### 7.8 Justification of Search Based Software Engineering

As discussed in previous chapter, there are certain validation criteria for SBSE which must be used for new search based software engineering (Harman et al., 2004). So, above experiments are validated against those validation criteria as below.

1. Base Line validity: To achieve the base line validity, the meta-heuristic search techniques must outperform random search techniques. In above models, test cases generated by genetic algorithm are closer to boundaries than generated by random search. As the distance from the boundary values is much less in case of genetic algorithm, so the results of meta-heuristic approach are better than random
search so the base line validity is validated for above model. Also it tends to be assumed that guided search techniques will outperform the unguided search techniques but in different situations things may vary and depending on the nature of problem this difference can be increased or decreased also. In this study, genetic algorithm is used with various fitness scaling options to guide the global optimisation technique in most effective direction and to optimise the sub-optimal solutions so that each solution can participate in overall search progress.

2. Discovery of known solutions: Meta-heuristic approach applied to software engineering for generating test cases can be compared with the adequacy criteria defined in literature. For example the adequacy criteria chosen in this study are boundary value analysis and equivalence class partitioning. Test cases for various lower and upper boundaries of a variable will decide some example test cases for these adequacy criteria. As shown in table 7.2 some test cases drawn by genetic algorithm with fitness scaling can fit in these example test cases. So the known solutions are also generated by the above models.

3. Discovery of desirable solutions: Meta-heuristic approach must increase the fitness of solutions. The solutions generated by meta-heuristic approaches are fit not only for one adequacy criteria but are also good for more than one criterion. As shown in Table 7.2, the test cases generated tend to be closer to boundaries. Therefore, the test cases according to boundary value analysis are needed to be closer to boundaries, which justify the validation of desired solutions with above approaches too.

4. Efficiency: In many cases a search based approach may be slower than an existing approach, because the search will include repeated execution of fitness function on individuals. So, the meta-heuristic approaches must not be so slow which makes their presence worst. But as shown in the experiment figures results of test case generation were found in the early stages of algorithm processing.

5. Avoiding bias in existing approaches: when meta-heuristic approaches were applied to judge the biasness of existing solution techniques, then the fitness of solutions is not so important. Instead the new and tentative solutions which are previously considered non-standard are more important to be generated by meta-heuristic techniques.
6. Filling problem space gaps left by analytical techniques: Whenever the analytical solutions can generate solutions in a partial problem space and meta-heuristic techniques can generate solutions in all problem space, then the later approaches are considered as filling the gaps left by analytical approaches. As the meta-heuristic techniques have the properties like exploration and exploitation so exploring the whole search space is more likely to happen in this case. It will indeed proceed to generating test cases in whole problem space weather problem space is unimodal or multimodal. So above meta-heuristic techniques generate solutions in all domains as needed so they can fill the necessary gaps in this case.

7. Optimisation of partial or suboptimal analytical solutions: As the existing analytical solutions are more likely to be near to optimal answers, the meta-heuristic techniques start from some random solutions and then they optimise them with the evolutionary approaches. So, the sub-optimal solutions are tending to be optimised till certain level to increase their optimality. It makes the overall quality of solutions good in comparison to solutions generated by analytical techniques.

8. Choice of technique and fitness function: The general application of meta-heuristic techniques to software engineering presents a large number of choices. Just like the various operators of meta-heuristic techniques, the representation, fitness functions, other generational operators like crossover, mutation and the search techniques to be used are also considered. The choice among all to be applied in certain situations must be made carefully. For example, the local search techniques like hill climbing, A* and others can be applied to generate the required solutions, whereas the situations in which these local search techniques will not produce effective solutions, focus should be on global optimisation techniques like genetic algorithm or ant colony optimisation etc. or a mixture of both the techniques can be used, these are called as hybrid evolutionary techniques. It seems simple while reading that the choice will not affect so much on the quality of solutions, but in practice while applying all of these parameters to the problems in software engineering, each of the decision regarding these parameters will significantly affect the performance as well as the quality of the solutions.
7.9 Summary

The metrics based heuristics for search and optimization technique like Genetic algorithm have been used to implement the model and finally the results are compared to give conclusions. The overall results show evolutionary testing to be a promising approach for fully automating test case design for boundary value analysis and equivalence portioning technique of testing. To increase the efficiency and effectiveness and to reduce the overall development cost for software-based systems, a systematic and automatic test case generator is required. Genetic algorithms search for relevant test cases in the input domain of the system under test. The model is proposed for automatic generation of test cases which uses metric as fitness function for genetic algorithm and heuristics are used for fitness scaling. Test cases were generated using random fitness, with normal fitness and with fitness scaling and the outputs of executing test cases were compared. It was found that genetic algorithm with fitness scaling generates test cases with much less difference from boundaries and hence, is a better approach for generating automatic test cases. The models have been reviewed keeping in mind defensive programming, designed by contract, Hoare’s specification and fault tolerance techniques like N-version programming and recovery block technique. The models were tested with many cases and the sample programs were taken and observed for their acceptance test using one input function (factorial of a number, next date, Fibonacci number and reverse of a number) and two input functions (searching, matrix multiplication and sorting). The proposed model is evaluated on the basis of certain validation criteria for search based software engineering proposed by Harman.