UNIT-5

Automated Test Data Generation

WHAT IS AUTOMATED TEST DATA GENERATION?

We require some knowledge of the software for generating test data. This knowledge may be known in terms of functionality and / or internal structure of the software. All techniques given in this book are based on either of the two or any combination of them. We manually write test cases on the basis of selected techniques and execute them to see the correctness of the software. How can we automate the process of generation of test cases / test data? The simplest way is to generate test data randomly, meaning, without considering any internal structure and / or functionality of the software. However, this way may not be an appropriate way to generate test data automatically.

• Test Adequacy Criteria

We may generate a large pool of test data randomly or may use any specified technique. This data is used as input(s) for testing the software. We may keep testing the software if we do not know when to stop testing. How would we come to know that enough testing is performed? This is only possible if we define test adequacy criteria. Once we define this, our goal is to generate a test suite that may help us to achieve defined test adequacy criteria. Some of the ways to define test adequacy criteria are given as:

1. Every statement of the source code should be executed at least once (statement coverage).
2. Every branch of the source code should be executed at least once (branch coverage).
3. Every condition should be tested at least once (condition coverage).
4. Every path of the source code should be executed at least once (path coverage).
5. Every independent path of the source code should be executed at least once (independent path coverage).
6. Every stated requirement should be tested at least once.
7. Every possible output of the program should be verified at least once.
8. Every definition use path and definition clear path should be executed at least once.

There may be many such test adequacy criteria. Effectiveness of testing is dependent on the definition of test adequacy criteria because it sets standards to measure the thoroughness of testing. Our thrust will only be to achieve the defined standard and thus, the definition of test adequacy criteria is very important and significant to ensure the correctness of the software. When our test suite fails to meet the defined criteria, we generate another test suite that does satisfy the criteria. Many times, it may be difficult to generate a large number of test data manually to achieve the criteria and automatic test data generation process may be used to satisfy the defined criteria.

• Static and Dynamic Test Data Generation

Test data can be generated either by statically evaluating the program or by actual execution of the program. The techniques which are based on static evaluation are called static test data generation techniques. Static test data generation techniques do not require the execution of the program. They generally use symbolic execution to identify constraints on input variables for the particular test adequacy criterion. The program is examined thoroughly and its paths are traversed without executing the program. Static test data generation techniques may not be useful for programs containing a large number of paths. The techniques which are based on the actual execution of the program for the generation of test data are called dynamic test data generation techniques. Test data is generated during the execution of the program. If during execution, a desired path is not executed, the program is traced back to find the statement which has diverted the desired flow of the program. A function minimization technique may be used to correct the input variables in order to select and execute the desired path.

APPROACHES TO TEST DATA GENERATION

The approaches to test data generation can be divided into two categories i.e. static and dynamic test data generation. One needs execution of the program (dynamic) and other does not need the execution of the program (static). We may automate any functional testing techniques (boundary value, equivalence partitioning) or structural testing techniques (path testing, data flow testing) for the generation of test data. The program will execute automatically and test data will be generated on the basis of the selected technique. The program execution will continue till the desired test adequacy criterion is achieved.

- ### Random Testing

Random testing generates test data arbitrarily and executes the software using that data as inputs. The output of the software is compared with the expected output based on the inputs generated using random testing. It is the simplest and easiest way to generate test data. For a complex test adequacy criterion, it may not be an appropriate technique because it does not consider the internal structure and functionality of the source code. Random testing is not expensive and needs only a random number generator along with the software to make it functional. The disadvantage of this technique is that it may not even generate test data that executes every statement of the source code. For any reasonably sized software, it may be difficult to attain '100% statement coverage' which is one of the easiest test adequacy criteria. It is a fast technique but does not perform well as it merely generates test data based on probability and has low chances of finding semantically small bugs. A semantically small bug is a bug that is only revealed by a small percentage of the program inputs [EDVA99]. Large software or more complex test adequacy criteria may further increase the problems of random test data generators. However, in the absence of any other technique, it is the only popular technique which is commonly used in practice for the automatic generation of test data.

- ### Symbolic Execution

Many early techniques of test data generation used symbolic execution for the generation of test data in which symbolic values are assigned to variables instead of actual values. The purpose is to generate an expression in terms of input variables. Phil McMinn [MCMI04] has explained the concept effectively as:

> "Symbolic execution is not the execution of a program in its true sense, but rather the process of assigning expressions to program variables as a path is followed through the code structure."

We may define a constraint system with the help of input variables which determines the conditions that are necessary for the traversal of a given path [CLAR76, RAMA76, BOYE75]. We have to find a path and then to identify constraints which will force us to traverse that particular path.

We consider a program for determination of the nature of roots of a quadratic equation. The source code and program graph of the program are given Figure 12.1 and 12.2 respectively. We select the path (1-7, 13, 25, 28-32) for the purpose of symbolic execution.

```
       #include<stdio.h>
       #include<conio.h>

1.     void main()
2.     {
3.     int a,b,c,valid=0,d;
4.     clrscr();
5.     printf("Enter values of a, b and c:\n");
6.     scanf("%d\n %d\n %d",&a,&b,&c);
7.     if((a>=0)&&(a<=100)&&(b>=0)&&(b<=100)&&(c>=0)&&(c<=100)){
8.             valid=1;
9.             if(a==0){
10.            valid=-1;
11.            }
12.    }
13.    if(valid==1){
14.            d=b*b-4*a*c;
15.            if(d==0){
16.            printf("Equal roots");
17.            }
18.            else if(d>0){
19.            printf("Real roots");
20.            }
21.            else{
22.            printf("Imaginary roots");
23.            }
24.    }
25.    else if(valid==-1){
26.            printf("Not quadratic");
27.            }
28.    else {
29.            printf("The inputs are out of range");
30.    }
31.    getch();
32.    }
```

Figure 12.1. Program for determination of nature of roots of a quadratic equation

The input variables a, b and c are assigned the constant variables x, y and z respectively. At statement number 7, we have to select a false branch to transfer control to statement number 13. Hence, the first constraint of the constraint system for this path is:

(i) (x <= 0 or x > 100 or
y <= 0 or y > 100 or
z <= 0 or z > 100)
and (valid = 0)

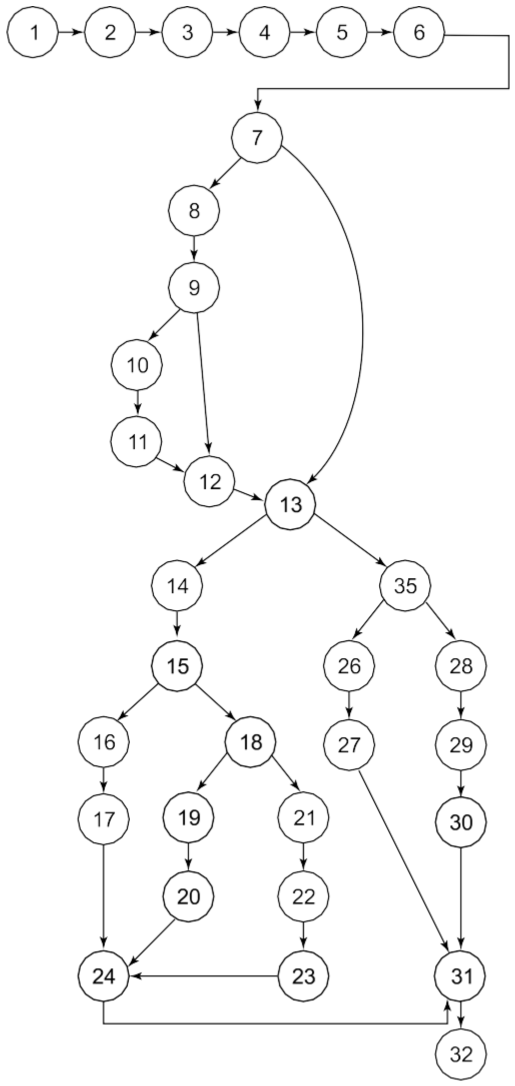

Figure 12.2. Program graph of program given in Figure 12.1

The path needs statement number 13 to become false so that control is transferred to statement number 25. The second constraint of the constraint system for this path is:

(ii) valid != 1

Finally, the path also needs statement number 25 to become false so that control is transferred to statement number 28. Hence, the third constraint of the constraint system for this path is:

(iii) valid != -1

To execute the selected path, all of the above mentioned constraints should be satisfied. One of the test cases that traverse the path (1-7, 13, 25, 28-32) may include the following inputs:

(x = -1, y = 101, z = 101)

Another set of inputs may be (x = -1, y = 90, z = 60). The same path is traversed with both of the test cases. Similarly, if we select x path (1-7, 13, 25-27, 31, 32), constraints of the system are identified as:

(i) (x <= 0 or x > 100
y <= 0 or y > 100
z <= 0 or z > 100)
and (valid = 0)

(ii) valid != 1

(iii) valid = -1

Clearly constraints (i) and (iii) are contradictory, meaning thereby, infeasibility of path. So path (1-7, 13, 25-27, 31, 32) is not feasible. Table 12.1 shows the constraints and identifies feasible/ unfeasible independent paths of the program given in Figure 12.1. Some paths are not feasible, although shown in the program graph for completion of the graph (see row 2 and row 3 of Table 12.1). Infeasible paths will not have any inputs and expected outputs as shown in Table 12.1.

Table 12.1. Constraints and values of paths (feasible/not feasible) of program given in Figure 12.1

S. No.	x	y	z	Expected Output	Path	Constraints	Feasible?
1.	101	50	50	Input values not in range	1-7, 13, 25, 28-32	(x <= 0 or x > 100 or y <= 0 or y > 100 or z <= 0 or z > 100) and (valid = 0) valid != 1 valid != -1	Yes
2.	-	-	-	-	1-7, 13, 25-27, 31, 32	(x <= 0 or x > 100 y <= 0 or y > 100 z <= 0 or z > 100) and (valid = 0) valid != 1 valid = -1	No
3.	-	-	-	-	1-9, 12, 13, 25, 28-32	(x > 0 and x <= 100 and y > 0 and y <= 100 and z > 0 and z <= 100) and x != 0 and (valid = 1) valid != 1	No
4.	0	50	50	Not quadratic	1-13, 25-27, 31, 32	(x > 0 and x <= 100 and y > 0 and y <= 100 and z > 0 and z <= 100) and x = 0 and (valid = -1) valid != 1 valid = -1	Yes

(*Contd.*)

(*Contd.*)

S. No	x	y	z	Expected Output	Path	Constraints	Feasible?
5.	99	0	0	Equal roots	1-9, 12-17, 24, 31, 32	(x > 0 and x <= 100 and y > 0 and y <= 100 and z > 0 and z <= 100) and x != 0 and (valid = 1) valid = 1	Yes
6.	50	50	1	Real roots	1-9, 12-15, 18-20, 24, 31, 32	(x > 0 and x <= 100 and y > 0 and y <= 100 and z > 0 and z <= 100) and x != 0 and (valid = 1) valid = 1	Yes
7.	50	50	50	Imaginary roots	1-9, 12-15, 18, 21-23, 24, 31, 32	(x > 0 and x <= 100 and y > 0 and y <= 100 and z > 0 and z <= 100) and x != 0 and (valid = 1) valid = 1	Yes

Therefore, in symbolic execution, constraints are identified for every predicate node of the selected path. We may generate test data for selected paths automatically using identified constraints.

Christoph. C. Michael [MICH01] and others have discussed some problems of symbolic execution in practice as:

> "One such problem arises in infinite loops, where the number of iterations depends on a non constant expression. To obtain a complete picture of what the program does, it may be necessary to characterize what happens if the loop is never entered, if it iterates once, if it iterates twice, and so on."

We may choose a good number of paths by considering various possibilities in a loop. Thus, it may be a time consuming activity. We may execute the program symbolically for one path at a time. Paths may be selected by a user or by the software using some selection technique.

In addition to loops, there are other constructs which are not easily evaluated symbolically like pointers, linked lists, graphs, trees, etc. There are also problems when the data is referenced indirectly as:

```
x =  (y + k [i]) * 2
```

The value of i should be known in advance to decide which element of the array k is being referred to by k[i]. Hence, the use of pointers and arrays may complicate the process of symbolic execution of a program. Another question that arises is how to handle the function calls to modules where there is no access to the source code? Although any program can be written without using pointers, arrays and function calls, but in practice, their usage is quite popular due to the facilities they offer and may also help to reduce the complexity of the source code. The above mentioned limitations may reduce the applicability of symbolic execution to any reasonable size of the program.

- **Dynamic Test Data Generation**

As against symbolic execution, dynamic test data generation techniques require actual execution of the program with some selected input(s). The values of variables are known during execution of the program. We also determine the program flow with such selected input(s). If the desired program flow / path is not executed, we may carefully examine the source code and identify the node where the flow took the wrong direction. We may use different types of search methods to alter the flow by changing the inputs till the desired path is achieved. This process may be very time consuming and may require many trials before a suitable input is identified. When we change the flow at a particular node, some other flows at different nodes may also change accidentally. Christoph. C. Michael and others [MICH01] have given their views about dynamic test data generation as:

> "This paradigm is based on the idea that if some desired test requirement is not satisfied, the data collected during execution can still be used to determine which tests come closest to satisfying the requirement. With the help of this feedback, test inputs are incrementally modified until one of them satisfies the requirement."

We consider a program given in Figure 12.3 in which statement number 8 contains a condition 'if (a >= 10)'. If we want to select the TRUE branch of this condition, we must choose inputs x and y in such a way that the value of 'a' is greater than or equal to 10, when statement number 8 is executed. How do we come to know the value of 'a'? One way is to execute the program up to statement number 8 and note the value of 'a'. The value of 'a' noted at statement number 8, when inputs given to the program are x and y, is represented as $a_8(x, y)$. We define the following function f(x, y) which is minimal when the TRUE branch is executed at statement number 8.

$$f(x, y) = \begin{cases} 10 - a_8(x, y) & \text{if } a_8(x, y) < 10 \\ 0 & \text{otherwise} \end{cases}$$

```
        #include<stdio.h>
        #include<conio.h>
1.  void main()
2.  {
3.  int x,y,a;
4.  clrscr();
5.  printf("Enter values of x and y:\n");
6.  scanf("%d\n %d", &x, &y);
7.  a=x-y;
8.  if(a >= 10)
9.  {
10.     printf("\nx = %d",x);
11. }
12. else
13. {
14.     printf("\ny = %d",y);
15. }
16. getch();
17. }
```

Figure 12.3. A typical program

This function is also called objective function and the problem of test data generation is now reduced to only function minimization. To get the expected input (say ‘a’ for the program given in Figure 12.3), we have to find values of x and y that minimizes f(x,y). The objective function gives an indication to the test generator about its closeness to reaching the goal. The test generator evaluates function f(x, y) to know how close x and y are to satisfy the present test requirement being targeted. The test generator may further change the values of x and y and evaluate the function f(x, y) again to know what changes in x and y bring the input closer to satisfy the requirement. The test generator may keep on making changes in x and y and evaluates function f(x, y) until the requirement is satisfied. Finally, the test generator may find values of x and y that satisfy the targeted requirement. This is a heuristic technique and the objective function definition is dependent on the goal, which is nothing but the satisfaction of a certain test requirement. The program may, at the first time, execute on randomly generated input(s) and its behaviour is used as the basis of a search for a satisfactory input. Hence, using different types of search methods, the flow can be altered by manipulating the input in a way that the intended branch is taken [EDVA99]. It may require many iterations before a suitable input is found. Dynamic test data generation techniques generate a large amount of data during execution of the program to find expected input(s) for a desired path. Based on test adequacy criteria, a search strategy is adopted and the program is executed automatically till the chosen criteria is satisfied.

TEST DATA GENERATION USING GENETIC ALGORITHM

Evolutionary algorithms provide heuristic search strategy to find a particular solution using operators motivated by genetics and natural selection [MCMI04]. The most popular form of evolutionary algorithm is genetic algorithm in which search is driven by the use of various combinations of input variables in order to satisfy the goal of testing.

Genetic Algorithm (GA) is based on natural genetics and Darwin’s principle of the survival of the fittest. It is used to find solutions for searching and optimization problems. A GA is a search procedure with a goal to find a solution in a multidimensional space. GA is generally many times faster than exhaustive search procedure and is a computer model of biological evolution. When GA is used to solve searching and optimization problems, very good results are obtained. With reference to software testing, GA is used to search the domain of input variables and to find those input variables which satisfy the desired goal of testing. GA is loosely based on the concept of genetics by combining samples to achieve new and fitter individuals [JONE96]. Inputs are combined to generate new inputs which are used for further searching of the desired goal. GA does not make incremental changes to a single structure, but maintains a population of structures from which new structures are created using genetic operators. The evolution is based on two primary operators i.e. mutation and crossover. The power of GA is the technique of applying GA operators (crossover and mutation) to a population of individuals. Despite their randomized nature, GA is not a simple random search. It uses the old knowledge held in a parent population to generate new solutions with improved performance. The population undergoes simulated evolution at each generation. Good solutions are retained and relatively bad ones are discarded and are replaced by fitter new members called offsprings. As given by Ali et al., the significant parameters for a genetic algorithm are [ALI10]:

(i) Initial population
(ii) Operators such as mutation and crossover with their values.

(iii) Fitness function created to guide the search procedure.
(iv) Selection strategy for parents.
(v) Stopping criteria.

- **Initial Population**

The initial population comprises a set of individuals generated randomly or heuristically. The selection of the starting generation has a significant effect on the performance of the next generation. Each individual is represented as a chromosome (binary or gray). A binary string representation is most popular to represent a chromosome. The chromosomes are composed of genes and are subjected to modification by means of mutation and crossover. The process is similar to a natural population of biological creatures where successive generations are conceived, born and raised until they themselves are ready to reproduce. Fitness of each individual is calculated for comparing them and to differentiate their performance. An individual who is near an optimum solution gets a higher fitness value than the one who is far away. During reproduction, two members (chromosomes) are chosen from the generation. The evolutionary process is then based on the GA operators (crossover and mutation), which are applied to them to produce two new members (offspring) for the next generation.

- **Crossover and Mutation**

There are two basic GA operators – crossover and mutation, which are commonly used in practice. However, many variants of both are also designed to find efficient solutions to the problem under investigation. Crossover operates at the individual (chromosome) level. Individuals are represented in the binary form. Crossover selects bits from parents (chromosome) and generates two new offsprings. In crossover operation, two members (parents) are selected from the population. A point along the bit string is selected at random, and the tails of the two bit strings are exchanged. This is known as one point crossover [JONE96]. For example, if two parents are $[V_1, V_2, \ldots V_m]$ and $[W_1, W_2, \ldots\ldots W_m]$, then crossing the chromosomes after the k^{th} gene $(1 \leq k \leq m)$ would produce the offsprings as: $[V_1, V_2, \ldots V_k, W_{k+1}, \ldots W_m]$ and $[W_1, W_2 \ldots W_k, V_{k+1} \ldots V_m]$. If parents are [11111111] and [00000000] and k = 5, the offsprings (children) after the application of crossover operator are [11111000] and [00000111]. A few examples of one point crossover operator are given in Table 12.2.

Table 12.2. Examples of one point crossover operator

Sr. No.	Parents			Offsprings	
	P1	P2	Crossover (k)	C1	C2
1.	11001100	10011111	4	11001111	10011100
2.	11001100	10011111	6	11001111	10011100
3.	11001100	10011111	2	11011111	10001100
4.	10000000	11111111	4	10001111	11110000
5.	10000000	11111111	6	10000011	11111100

Two point crossover operates by selecting two random genes within the parent strings with subsequent swapping of bits between these two genes. If two parents are [V_1, V_2….V_m] and [W_1, W_2….W_m], and the first randomly chosen point is k with ($1 \le k \le m-1$) and the second random point is n with ($k+1 \le n \le m$) this would produce the offsprings as:
[(V_1, V_2….V_k), (W_{k+1}….W_n), (V_{n+1}….V_m)] and [(W_1, W_2,….W_k), (V_{k+1}, …..V_n), (W_{n+1}…W_m)]. Examples of two point crossover operator is given in Table 12.3.

Table 12.3. Examples of two point crossover operator

Sr. No.	Parents		Crossover points		Offsprings	
	P1	P2	k	n	C1	C2
1.	11001100	10011111	2	6	11011100	10001111
2.	11001100	10011111	1	7	10011110	11001101
3.	11111111	00000000	2	5	11000111	00111000
4.	11111111	00000000	7	8	11111110	00000001
5.	11110000	00001111	2	4	11000000	00110000

Mutation changes random bits in the binary string. In the binary code, this simply means changing the state of a gene from 0 to 1 or vice-versa. Mutation is like a random walk through the search space and is used to maintain diversity in the population and to keep the population from prematurely converging on one (local) solution. Mutation avoids local optima and creates genetic material (say input) that may not be present in the current population. Mutation works by randomly changing the chosen bits from 1 to 0 or from 0 to 1. For example, after crossover, the generated offspring is [10001111] and the same may be mutated as [10011111] by changing the 4th bit from 0 to 1. We may also find optimum mutation probability P_m which is the reciprocal of the chromosome size(s) and is given as: $P_m = 1/5$. It would be unlikely for the code to have on average more than one bit of a chromosome mutated. If the mutation probability is too low, there will be insufficient global sampling to prevent convergence to a local optimum. If the rate of mutation is significantly increased, the location of global optima is delayed. After the crossover and mutation operations, we have the original population of parents and the new population of offspring. The survival of parents and offspring depends on the fitness value of every member of the population and is calculated on the basis of a fitness function.

• Fitness Function

GA is used to find the best solution of a problem. This is carried out using a fitness function. The purpose of the fitness function is simply explained by B.F. Jones and others [JONE96] as:

> "Perhaps the most important aspect of using genetic algorithms is the ability to define a suitable fitness function, which is a numeric measure of how close each sample test set is to the goal."

The fitness value is used to compare the individuals and to differentiate their performance. An individual who is near an optimum solution gets a higher fitness value than an individual who is far away. How do we define a fitness function? Every point in the search space is represented by a valid fitness value.

For example, suppose that the function to optimize is f(A) = A^3 where Aε [0,10]. The fitness function here may be the same and is defined as:

$$\text{Fitness} = A^3$$

Table 12.4 gives the chromosomes with their fitness values calculated by the fitness function.

Table 12.4. Chromosomes with fitness values for initial population

Sr. No.	Chromosome	A1	Fitness (f_i)
1.	C1	2	8
2.	C2	4	64
3.	C3	6	216
4.	C4	1	1

The total fitness of the population is calculated as:

$$\text{Fitness} = \sum_{i=1}^{4} f_i = 289$$

Our objective is to achieve a higher fitness value of individuals which is expected to be closer to the global optimum. In this example, a higher fitness value is achieved when the value of A = 10.

- **Selection**

The selection operator selects two individuals from a generation to become parents for the recombination process (crossover and mutation). This selection may be based on fitness value or could be made randomly. If the fitness value is used, then higher value chromosomes will be selected.

- **Algorithm for Generating Test Data**

1. Genetic algorithm begins with an initial population which is randomly generated where each population is represented as a chromosome.
2. Fitness of each individual is calculated for comparing them and to differentiate their performance.
3. An individual who is near an optimum solution is assigned a higher fitness value.
4. A stopping criterion is decided for stopping the test data generation process. It may be based on coverage criteria, number of iterations or the size of the final population. The following steps are repeated until the criterion is satisfied.
   (a) The genetic operators: crossover and mutation are randomly selected. If the crossover operator is selected the following steps are followed:
      (i) Parents are selected from the initial population on the basis of their fitness value.
      (ii) Crossover is performed to generate offsprings with probably better genes / fitness value.

(a) Otherwise the following steps are followed:
   (i) Each offspring is mutated by occasional random alteration of a bit value with changes in some features with unpredictable consequences.
(a) Data is prepared for the next generation.

The flow chart of the above steps is given in Figure 12.4. The process will iterate until the population has evolved to form an optimum solution of the problems or until a maximum number of iterations have taken place. The GA is an evolutionary algorithm where definition of the fitness function for any search is very important. If the fitness function is effective, desired inputs will be selected early and may help us to traverse the desired path of the program under testing.

GA generates first generation of data randomly (initial population) and then follows the steps of the flow chart given in Figure 12.4 to improve the fitness of individuals. On the basis of fitness value, crossover and mutation operators are used to generate offsprings (2nd generation individuals). This process continues until all individuals reach the maximum fitness. The system performs all operations from initial population to the last generation automatically. It does not require user interference. The automated generated test data may give better results with reduced effort and time [PRAV09].

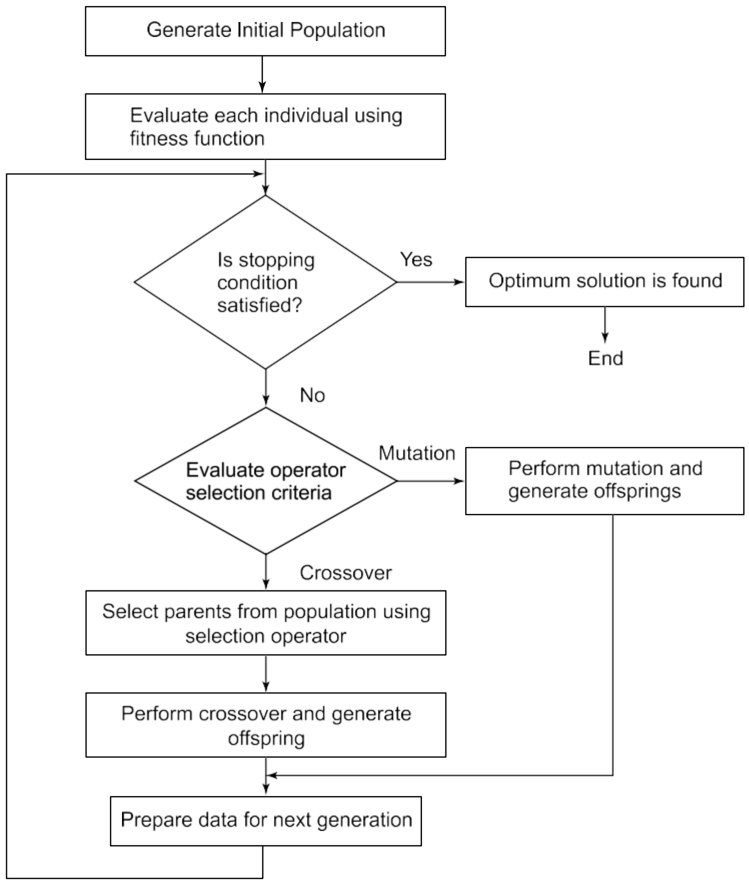
Generate Initial Population
Evaluate each individual using fitness function
Is stopping condition satisfied?
Yes
Optimum solution is found
End
No
Evaluate operator selection criteria
Mutation
Perform mutation and generate offsprings
Crossover
Select parents from population using selection operator
Perform crossover and generate offspring
Prepare data for next generation

Figure 12.4. Flow chart of various steps of genetic algorithm

Example 12.1: Consider the program to divide two numbers given in Figure 12.5. Generate test data using genetic algorithm.

```
#include<stdio.h>
#include<conio.h>

void main()
{
int a, b, c;
printf("Enter value of a and b");
scanf("%d %d",&a, &b);
if(b==0)
{
	printf("Invalid Data");
}
else
{
c=a/b;
}
printf("\n a/b= %d",c);
getch();
}
```

Figure 12.5. Program to divide two numbers

Solution:

The fitness function is given below:
Fitness function

$$F(x) = \frac{x}{y * 100}$$

The criteria for selection of mutation and crossover operators are given below:
If

$F(x) \leq 0.2$, use mutation operator

$0.2 < F(x) < 0.6$, use crossover operator

$F(x) \geq 0.6$, criteria satisfied

The steps for generating test data are given in the following tables. The mutation and crossover (one-point or two-point) bits are randomly selected.

1. First Generation

S. No.	a	b	Operator
1.	5	2	Mutation
2.	100	3	Crossover
3.	80	5	Mutation

a = 5, b=2
00000101 00000010
The 2nd bit is selected randomly for mutation
After mutation
01000101 00000010

a = 100, b=3
01100100 00000011
Two-point crossover is performed. The first randomly chosen point is 4 and the second is 6.
After crossover
01100000 00000111
a = 80, b=5
01010000 00000101
The 7th bit is selected randomly for mutation
After mutation
01010010 00000101

2. Second Generation

S. No.	a	b	Operator
1.	69	2	Crossover
2.	96	7	Mutation
3.	82	5	Mutation

a = 69, b=2
01000101 00000010
One-point crossover is performed with randomly chosen point 5.
After crossover
01000010 00000101

a = 96, b=7
01100000 00000111
The 1st bit is selected randomly for mutation
After mutation
11100000 00000111

a = 82, b=5
01010010 00000101
The 1st bit is selected randomly for mutation
After mutation
11010010 00000101

3. Third Generation

S. No.	a	b	Operator
1.	66	5	Mutation
2.	224	7	Crossover
3.	210	5	Crossover

a = 66, b=5
01000010 00000101
The 1st bit is selected randomly for mutation
After mutation
11000010 00000101

a = 224, b=7
11100000 00000111
Two-point crossover is performed. The first randomly chosen point is 4 and the second is 6.
After crossover
11100100 00000011

a = 210, b=5
11010010 00000101
Two-point crossover is performed. The first randomly chosen point is 5 and the second is 7.
After crossover
11010100 00000011

4. Fourth Generation

S. No.	A	b	Operator
1.	194	5	Crossover
2.	228	3	Criteria satisfied
3.	212	3	Criteria satisfied

a = 194, b=5
11000010 00000101
Two-point crossover is performed. The first randomly chosen point is 5 and the second is 7.
After crossover
11000100 00000011

5. Fifth Generation

S. No.	A	b	Operator
1.	196	3	Criteria satisfied
2.	228	3	Criteria satisfied
3.	212	3	Criteria satisfied

The criteria is satisfied after generating 5th generation population. The testing will be stopped after achieving the defined criteria.

Example 12.2: Consider the program given in Figure 12.1. Generate test data using genetic algorithm.

Solution:

The fitness function is given below:

$$F(x) = (10 - a_8(x, y))/10 \qquad \text{if } a_8(x, y) < 10$$

The criteria for selection of mutation and crossover operators are given below:
If
$F(x, y) \leq 0.2$, use mutation operator
$0.2 < F(x, y) < 0.6$, use crossover operator
$F(x, y) \geq 0.6$, criteria satisfied
The tables below show the steps for generating test data.

1. First Generation

S. No.	x	y	Operator
1.	10	7	Crossover
2.	19	10	Mutation

x = 10, y=7
00001010 00000111
After crossover
00001011 00000110

x = 19, y=10
00010011 00001010
The 14th bit is selected randomly for mutation
After mutation
00010011 00001110

2. Second Generation

S. No.	x	y	Operator
1.	11	6	Crossover
2.	19	14	Crossover

x = 11, y=6
00001011 00000110
Two-point crossover is performed. The first randomly chosen point is 3 and the second is 6.
After crossover
00000111 00001010

x = 19, y=14
00010011 00001110
One-point crossover is performed with randomly chosen point 6.
After crossover
00010010 00001111

3. Third Generation

S. No.	x	y	Operator
1.	7	10	Criteria satisfied
2.	18	15	Criteria satisfied

After the 3rd generation, the criteria are satisfied. Testing will be stopped at this point.

TEST DATA GENERATION TOOLS

The use of tools for the generation of test data is still in its infancy, although some software industries have been using their own tools for the generation of test data. The output of such a tool is a set of test data, which include a sequence of inputs to the system under test. Some tools are also available that accept manually created, automatically generated, predefined test sequences and executes the sequences without human intervention and supervision. A few examples of such tools are Mercury's WinRunner, LoadRunner, Rational Robot and Teleogic Tau Tester. The purpose of these tools is to execute already generated test cases and not to generate test data automatically. We should also not confuse this with modeling tools like Rational Rose, Objecteering, Simulink and Autofocus, which are used for modeling various software engineering activities. The automated test generation tools are different and designed for a specific purpose of test data generation.
Some of the popular test generation tools are given in Table 12.5.

Table 12.5. **Automated test data generation tools**

S. No.	Tool	Language/ platform supported	Description	Remarks
1.	T-VEC	Java/C++	Test cases are generated by applying branch coverage	Easy to use
2.	GS Data Generator	ERP, CRM and data warehouse development	Creates industry specific data for acceptance testing	Focuses on version control and RDBMS repository
3.	MIDOAN	Ada	Used on Ada source code and generates test cases that will be used on branches and decisions.	Product of MIDOAN software engineering solutions for automatic test data generation.
4.	CONFORMIQ	C, C++, Java, Visual Basic	Automates the design of functional tests using black box testing techniques.	It delivers faster test design, higher test quality and better test coverage.
5.	CA DATAMACS test Data Generator (TDG)	Mainframe	Used for mainframe programs	It creates test data from existing files, databases or from scratch.

The software industry is focusing more on the quality of a product instead of increasing functionality. Testing is the most popular and useful way to improve several quality aspects such as reliability, security, correctness, ease of usage, maintainability, etc. If test data is generated automatically, it will reduce the effort and time of testing. Although the process of automated test data generation is still in the early stages, some reasonable success has been achieved in the industry. Further research is needed to develop effective tools and techniques. A special effort is also required to increase the flexibility, user friendliness and ease of use of these tools.