Journal of Universal Computer Science, vol. 11, no. 6 (2005), 898-915
submitted: 16/6/04, accepted: 17/1/05, appeared: 28/6/05 © J.UCS

Automatic Test Data Generation for Data Flow Testing Using a Genetic Algorithm

Moheb R. Girgis
(Department of Computer Science, Faculty of Science
Minia University, El-Minia, Egypt
moheb_r_g@yahoo.com)

Abstract: One of the major difficulties in software testing is the automatic generation of test data that satisfy a given adequacy criterion. This paper presents an automatic test data generation technique that uses a genetic algorithm (GA), which is guided by the data flow dependencies in the program, to search for test data to cover its def-use associations. The GA conducts its search by constructing new test data from previously generated test data that are evaluated as effective test data. The approach can be used in test data generation for programs with/without loops and procedures. The proposed GA accepts as input an instrumented version of the program to be tested, the list of def-use associations to be covered, the number of input variables, and the domain and precision of each input variable. The algorithm produces a set of test cases, the set of def-use associations covered by each test case, and a list of uncovered def-use associations, if any. In the parent selection process, the GA uses one of two methods: the roulette wheel method or a proposed method, called the random selection method, according to the user choice. Finally, the paper presents the results of the experiments that have been carried out to evaluate the effectiveness of the proposed GA compared to the random testing technique, and to compare the proposed random selection method to the roulette wheel method.

Keywords: software testing, automatic test data generation, data flow testing, Genetic algorithms
Categories: D.2.5, K.6.3

1 Introduction

Software testing has two main aspects: test data generation and application of a test data adequacy criterion. A test data generation technique is an algorithm that generates test cases, whereas an adequacy criterion is a predicate that determines whether the testing process is finished, [Frankl, 93]. Several test data adequacy criteria have been proposed, such as control flow-based and data flow-based criteria. One of the major difficulties in software testing is the automatic generation of test data that satisfy a given adequacy criterion.

An automated test data generator is a tool that assists the tester in creating test data. Test data generators can be categorized into three classes: *random test data generators* (e.g., [Mills, 87]; [Voas, 91]), *structural-oriented test data generators* (e.g., [Boyer, 75]; [Clarke, 76]; [Ramamoorthy, 76]; [Howden, 77]; [Korel, 90]; [DeMillo, 91]; [Girgis, 93]), and *data specification generators* (e.g., [Miller, 75]; [Bauer, 79]; [Maurer, 90]). Random test data generators select random test data from

the domain of input variables. Structural-oriented test data generators are based on covering certain structural elements in the program. Most of these generators use symbolic execution to generate test data to meet a testing criterion such as path coverage, branch coverage, def-use coverage, mutation, etc. Data specification generators select test data from program specification, in order to exercise features of the specification.

Recently, the use of genetic algorithms (GAs) in test data generation became the focus of several research studies, (see e.g., [Pei, 94]; [Roper, 95]; [Watkins, 95], [Jones, 96]; [Jones, 98]; [Pargas, 99]; [Bueno, 00]; [Lin, 01]; [Michael, 01]. As far as the author is aware, none of the reported studies have used GAs to generate test data to cover the def-use associations of the program.

This paper presents a structural-oriented technique for automatic test data generation that uses a genetic algorithm, which is guided by the data flow dependencies in the program, to search for test data to fulfil one of the most demanding in the family of data flow path selection criteria, developed by Rapps and Weyuker [Rapps, 85], namely the all-uses criterion. The genetic algorithm conducts its search by constructing new test data from previously generated test data that are evaluated as *effective* test data. In the parent selection process, the GA uses one of two methods: the *roulette wheel method* or a proposed method, called the *random selection method*, according to the user choice. The approach can be used in test data generation for programs with/without loops and procedures.

This paper is organized as follows: Section 2 describes the data flow analysis technique used to implement the all-uses criterion. Section 3 describes the principles of GAs. Section 4 describes the proposed GA for automatic test data generation, and gives the result of applying this algorithm to an example program. Section 5 presents the results of the experiments that are conducted to evaluate the effectiveness of the proposed GA compared to the random testing technique, and to compare the proposed random selection method to the roulette wheel method.

2 The Data Flow Analysis Technique

This section describes the all-uses criterion and the data flow analysis technique used to implement it. Firstly, some definitions used in describing this technique are presented.

The control flow of a program can be represented by a directed graph with a set of nodes and a set of edges. Each node represents a group of consecutive statements, which together constitute a basic block. The edges of the graph are then possible transfers of control flow between the nodes. A path is a finite sequence of nodes connected by edges. A complete path is a path whose first node is the start node and whose last node is an exit node. A path is def-clear with respect to a variable if it contains no new definition of that variable. Figure 2 presents the flow graph of the example program, shown in Figure 1, which determines the middle value of three given integers X, Y, and Z.

Data flow analysis focuses on the interactions between variable definitions (defs) and references (uses) in a program. Variable uses can be split into 'c-uses' and 'p-uses' according to whether the variable use occurs in a computation or a predicate

[Rapps, 85]. Defs and c-uses are associated with nodes, but p-uses are associated with edges. The purpose of the data flow analysis is to determine the defs of every variable in the program and the uses that might be affected by these defs, i.e. the def-use associations. Such data flow relationships can be represented by the following two sets: dcu(i), the set of all variable defs for which there are def-clear paths to their c-uses at node i; and dpu(i,j), the set of all variable defs for which there are def-clear paths to their p-uses at edge (i,j), [Girgis, 85b].

```
 1   1    INTEGER X,Y,Z          12   8    ELSE
 2   1    READ(5,*)X,Y,Z         13   8     IF(X.GE.Y)THEN
 3   1    MID=Z                  14   9      MID=Y
 4   1    IF(Y.LT.Z)THEN         15  10    ELSE
 5   2     IF(X.LT.Y)THEN        16  10      IF(X.GT.Z)THEN
 6   3      MID=Y                17  11       MID=X
 7   4     ELSE                  18  12      END IF
 8   4      IF(X.LT.Z)THEN       19  13     END IF
 9   5       MID=X               20  14    END IF
10   6      END IF                21  14    PRINT*,'MIDDLE VALUE= ', MID
11   7     END IF                 22  14    END
```

Figure 1: Example program (The 1st column represents statement numbers, and the 2nd one represents block numbers)

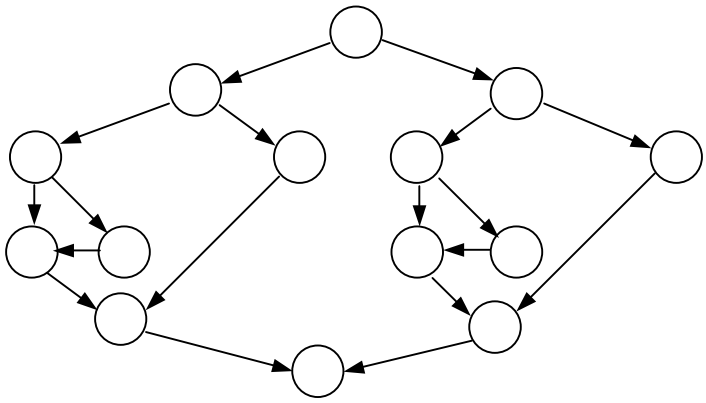

Figure 2: Flow graph for the example program

Using information concerning the location of variable defs and uses, together with the 'basic static reach algorithm' [Allen, 76], the sets dcu(i) and dpu(i,j) can be determined [Girgis, 1985a]. The 'basic static reach algorithm' is used to determine two sets called reach(i) and avail(i). The set reach(i) is the set of all variable defs that "reach" node i. (A def of a variable x in node k is said to reach node i if there is a def-clear path w.r.t. x from node k to node i). The set avail(i) is the set of all "available" variable defs at node i. It is the union of the set of global defs at node i together with

the set of all defs that reach this node and are preserved through it. (Clearly any def of a variable in node i will not preserve any other def of the same variable). Using these two sets, the sets dcu(i) and dpu(i,j) are constructed from the formulae:

$$dcu(i) := reach(i) \cap c\text{-}use(i), \text{ and}$$
$$dpu(i,j) := avail(i) \cap p\text{-}use(i,j),$$

where c-use(i) is the set of variables for which node i contains a global c-use, and p-use(i,j) is the set of variables for which edge (i,j) contains a p-use.

The all-uses criterion requires a def-clear path from each def of a variable to each use (c-use and p-use) of that variable to be traversed. It should be noted that, the all-uses criterion includes all the members of the family of the data flow criteria, developed by Rapps and Weyuker [Rapps, 85], except the all-du-paths criterion. In other words, any complete path satisfying the all-uses criterion also satisfies the others. In order to determine the set of paths that satisfy the all-uses criterion, it is necessary to determine the def-use associations of program variables. As described above, such data flow relationships can be represented by the dcu and dpu sets.

The def-clear paths required to fulfil the all-uses criterion are constructed from the dcu and dpu sets by using the technique described in [Girgis, 93]. These paths are divided into two groups: dcu-paths and dpu-paths. In the dcu-paths list, each dcu-path is represented by: a def-node (a node containing a def of a variable), a c-use-node (a node containing a c-use of that variable), and the set of nodes that must not be included in that path (nodes containing other defs of that variable). These nodes are called killing nodes. In the dpu-paths list, each dpu-path is represented by: a def-node (node containing a def of a variable), p-use-edge (an edge having a p-use of that variable), and the set of killing nodes. Henceforth, the term 'def-use paths' will be used to mean the set of dcu-paths and dpu-paths together. Figures 3 and 4 show the lists of the def-use paths of the example program.

To construct the def-use paths that satisfy the all-uses criterion in the presence of procedure calls, the interprocedural dcu and dpu sets are determined, which represent the def-use associations across procedure boundaries, then the above technique is applied directly to these sets. [Girgis, 00] For programs with loops, only paths in which each loop is iterated zero, one and two times, and satisfy the all-uses criterion are selected, [Girgis, 93].

DCU-Path No.	Variable	Def Node	C-use Node	Killing Nodes
1	Y	1	3	None
2	X	1	5	None
3	Y	1	9	None
4	X	1	11	None
5	MID	1	14	3, 5, 9, 11
6	MID	3	14	1, 5, 9, 11
7	MID	5	14	1, 3, 9, 11
8	MID	9	14	1, 3, 5, 11
9	MID	11	14	1, 3, 5, 9

Figure 3: List of the dcu-paths of the example program.

DPU-Path No.	Variable	Def Node	P-use Edge	Killing Nodes
1	Y	1	1-2	None
2	Z	1	1-2	None
3	Y	1	1-8	None
4	Z	1	1-8	None
5	X	1	2-3	None
6	Y	1	2-3	None
7	X	1	2-4	None
8	Y	1	2-4	None
9	X	1	4-5	None
10	Z	1	4-5	None
11	X	1	4-6	None
12	Z	1	4-6	None
13	X	1	8-9	None
14	Y	1	8-9	None
15	X	1	8-10	None
16	Y	1	8-10	None
17	X	1	10-11	None
18	Z	1	10-11	None
19	X	1	10-12	None
20	Z	1	10-12	None

Figure 4: List of the dpu-paths of the example program.

3 The Principles of Genetic Algorithms

The basic concepts of genetic algorithms (GAs) were developed by Holland [Holland, 75]. GAs are commonly applied to a variety of problems involving search and optimisation. GAs search methods are rooted in the mechanisms of evolution and natural genetics. GAs draw inspiration from the natural search and selection processes leading to the survival of the fittest individuals. GAs generate a sequence of populations by using a selection mechanism, and use crossover and mutation as search mechanisms. [Srinivas, 94]

The principle behind GAs is that they create and maintain a population of individuals represented by chromosomes (essentially a character string analogous to the chromosomes appearing in DNA). These chromosomes are typically encoded solutions to a problem. The chromosomes then undergo a process of evolution according to rules of selection, mutation and reproduction.

Each individual in the environment (represented by a chromosome) receives a measure of its fitness in the environment. Reproduction selects individuals with high fitness values in the population, and through crossover and mutation of such individuals, a new population is derived in which individuals may be even better fitted to their environment. The process of crossover involves two chromosomes swapping chunks of data (genetic information) and is analogous to the process of sexual reproduction. Mutation introduces slight changes into a small proportion of the population and is representative of an evolutionary step. The structure of a simple GA is given below.

```
Simple Genetic Algorithm ()
{
        initialize population;
        evaluate population;
        while termination criterion not reached
          {
                select solutions for next population;
                perform crossover and mutation;
                evaluate population;
          }
}
```

The algorithm will iterate until the population has evolved to form a solution to the problem, or until a maximum number of iterations have taken place (suggesting that a solution is not going to be found given the resources available).

4 A Genetic Algorithm For Test-Data Generation

This section describes the proposed GA for automatic test data generation, which is guided by the data flow dependencies in the program. The algorithm searches for test cases that satisfy the all-uses criterion. Firstly, the major components of this GA are discussed in turn, then the overall algorithm is presented.

4.1 Representation

The proposed GA uses a binary vector as a chromosome to represent values of the program input variables x. The length of the vector depends on the required precision and the domain length for each input variable.

Suppose we wish to generate test cases for a program of k input variables x_1, …, x_k, and each variable x_i can take values from a domain $D_i = [a_i, b_i]$. Suppose further that d_i decimal places are desirable for the values of each variable x_i. To achieve such precision, each domain D_i should be cut into $(b_i - a_i) \cdot 10^{d_i}$ equal size ranges. Let us denote by m_i the smallest integer such that $(b_i - a_i) \cdot 10^{d_i} \le 2^{m_i} - 1$. Then, a representation having each variable x_i coded as a binary string $string_i$ of length m_i clearly satisfies the precision requirement. The mapping from the binary string $string_i$ into a real number x_i from the range $[a_i, b_i]$ is performed by the following formula:

$$x_i = a_i + x_i' \cdot \frac{b_i - a_i}{2^{m_i} - 1}, \qquad (4.1)$$

where x_i' represents the decimal value of the binary string $string_i$, (Michalewicz, 1999).

It should be noted that the above method can be applied for representing values of integer input variables by setting d_i to 0, and using the following formula instead of formula (4.1):

$$x_i = a_i + \text{int}\,(x_i' \cdot \frac{b_i - a_i}{2^{m_i} - 1}), \qquad (4.2)$$

Now, each chromosome (as a test case) is represented by a binary string of length $m = \sum_{i=1}^{k} m_i$; the first m_1 bits map into a value from the range $[a_1, b_1]$ of variable x_1, the next group of m_2 bits map into a value from the range $[a_2, b_2]$ of variable x_2, and so on; the last group of m_k bits map into a value from the range $[a_k, b_k]$ of variable x_k.

For example, let a program have 2 input variables x and y, where $-3.0 \le x \le 12.1$ and $4.1 \le y \le 5.8$, and the required precision is 4 decimal places for each variable. The domain of variable x has length 15.1; the precision requirement implies that the range [-3.0, 12.1] should be divided into at least $15.1 \cdot 10000$ equal size ranges. This means that 18 bits are required as the first part of the chromosome: $2^{17} < 151000 \le 2^{18}$. The domain of variable y has length 1.7; the precision requirement implies that the range [4.1, 5.8] should be divided into at least $1.7 \cdot 10000$ equal size ranges. This means that 15 bits are required as the second part of the chromosome: $2^{14} < 17000 \le 2^{15}$. The total length of a chromosome (test case) is then $m = 18+15=33$ bits; the first 18 bits code x and remaining 15 bits code y. Let us consider an example chromosome:

010001001011010000111110010100010.

By using formula (4.1), the first 18 bits, 010001001011010000, represents x = 1.0524, and the next 15 bits, 111110010100010, represents y = 5.7553. So the given chromosome corresponds to the data values 1.0524 and 5.7553 for the variables x and y, respectively.

4.2 Initial population

As mentioned above, each chromosome (as a test case) is represented by a binary string of length m. We randomly generate *pop_size* m-bit strings to represent the initial population, where *pop_size* is the population size. The appropriate value of *pop_size* is experimentally determined. Each chromosome is converted to k decimal numbers representing values of k input variables $x_1, \ldots, x_k$ (i.e. a test case) by using formula (4.1)/(4.2).

4.3 Evaluation function

The algorithm evaluates each test case by executing the program with it as input, and recording the def-use paths in the program that are covered by this test case. (A test case is said to cover a def-use path, if it causes the program to traverse a path that has a subpath, which starts at the def-node and ends at the c-use node/p-use edge of the def-use path and does not pass through its killing nodes.) The fitness value $eval(v_i)$ for each chromosome v_i ($i = 1, \ldots, pop_size$) is calculated as follows:

$$eval(v_i) = \frac{\text{no. of def - use paths covered by } v_i}{\text{total no. of def - use paths}}$$

The fitness value is the only feedback from the problem for the GA. A test case which is represented by the chromosome v_i is considered *effective* if its fitness value $eval(v_i) > 0$.

4.4 Selection

After computing the fitness of each test case in the current population, the algorithm selects test cases from the effective members of the current population that will be parents of the new population. If none of the members of the current population was effective, all the members of current population are considered the parents of the new population. In the selection process the GA uses one of two methods: the *roulette wheel method* [Goldberg, 89] or a proposed method, called the *random selection method*, according to the user choice. These two methods are described below.

(i) Roulette wheel: For the selection of a new population with respect to the probability distribution based on fitness values, a roulette wheel with slots sized according to fitness is used. Such roulette wheel is constructed as follows:

- Calculate the fitness value $eval(v_i)$ for each chromosome v_i (i = 1, …, *pop_size*).
- Find the total fitness of the population $F = \sum_{i=1}^{pop_size} eval(v_i)$,
- Calculate the probability of a selection p_i for each chromosome v_i (i = 1, …, *pop_size*): $p_i = eval(v_i)/F$.
- Calculate a cumulative probability q_i for each chromosome v_i (i = 1, …, *pop_size*):

$$q_i = \sum_{j=1}^{i} p_j .$$

The selection process is based on spinning the roulette wheel pop_size times; each time we select a single chromosome for a new population in the following way:

- Generate a random (float) number r from the range [0..1].
- If $r < q_1$ then select the first chromosome (v_1); otherwise select the i-th chromosome v_i ($2 \le i \le pop_size$) such that $q_{i-1} < r \le q_i$.

Obviously, some chromosomes would be selected more than once.

(ii) Random selection: In this method, the selection of parents is made randomly, so that every effective member of the current population has an equal chance of being selected for recombination.

Assume that l members of the current population were effective, where $l \le pop_size$. The parents are selected as follows:

```
Isolate the effective members and number them from 1 to l;
For i=1 to pop_size do
Begin
        Generate an random integer number j from the range [0..l];
        Select chromosome v_j from the effective members;
End For;
```

The experiments showed that this selection method produced better results than the roulette wheel method (see Section 5).

4.5 Recombination

In the recombination phase, we use two operators, crossover and mutation, which are the key to the power of GAs. These operators create new individuals from the selected parents to form a new population.

Crossover: It operates at the individual level. During crossover, two parents (chromosomes) exchange sub string information (genetic material) at a random position in the chromosome to produce two new strings (offspring). The objective here is to create better population over time by combining material from pairs of (fitter) members from the parent population. Crossover occurs according to a crossover probability. The probability of crossover p_c gives us the expected number $p_c \cdot$ pop_size of chromosomes, which undergo the crossover operation. We proceed in the following way:
For each chromosome in the (new) population:

- Generate a random (float) number r from the range [0..1];
- If $r < p_c$ then select given chromosome for crossover.

Now we mate selected chromosomes randomly: For each pair of coupled chromosomes we generate a random integer number *pos* from the range [1..m-1] (m is the number of bits in a chromosome). The number *pos* indicates the position of the crossing point. Two chromosomes ($b_1 \ldots b_{pos} b_{pos+1} \ldots b_m$) and ($c_1 \ldots c_{pos} c_{pos+1} \ldots c_m$) are replaced by a pair of their offspring ($b_1 \ldots b_{pos} c_{pos+1} \ldots c_m$) and ($c_1 \ldots c_{pos} b_{pos+1} \ldots b_m$).

Mutation: It is performed on a bit-by-bit basis. Mutation always operates after the crossover operator, and flips each bit with the pre-determined probability. The probability of mutation p_m, gives us the expected number of mutated bits $p_m \cdot m \cdot pop_size$. Every bit (in all chromosomes in the whole population) has an equal chance to undergo mutation, i.e., change from 0 to 1 or vice versa. So we proceed in the following way:
For each chromosome in the current (i.e. after crossover) population and for each bit within the chromosome:

- Generate a random (float) number r from the range [0..1];
- If $r < p_m$ then mutate the bit.

In the traditional GA approach the population would evolve until one individual from the whole set which represents the solution is found. In our case, this would correspond to one group of data items achieving maximum coverage of the program (i.e. traversing all the def-use paths of the program). Whilst this feasible for some programs, the majority of programs cannot be 'covered' by just one group of data items (i.e. one test case) – it might take many groups and several runs of the program to achieve the desired level of testing. So, we let the population evolves until a combined subset of the population achieves the desired level of coverage. This is done by recording which def-use paths of the program each individual has covered and halting the evolution when a set of individuals has traversed the entire def-use paths of program, if possible. The solution is this set.

4.6 Overall Algorithm

The proposed genetic algorithm accepts as input an instrumented version of the program to be tested, the list of def-use paths to be covered, the number of input variables, and the domain and precision of each input variable. Also, it accepts the GA parameters: population size, maximum number of generations, and probabilities of the crossover and mutation. The algorithm produces a set of test cases, the set of def-use paths covered by each test case, and the list of uncovered def-use paths, if any. It should be noted that the instrumentation process and the generation of the program def-use paths are performed by a testing system previously developed by the author [Girgis, 93, 00].

The algorithm uses an integer vector, called the *def-use coverage vector*, to record the traversed def-use paths. In this vector, each element (initially zero) corresponds to a def-use path. Whenever a def-use path is covered, the number of the test case that caused this coverage is stored in the corresponding element of the def-use coverage vector (see Table 1). The algorithm keeps track of all generated test cases that cover new def-use paths. It uses a counter, called nCases, to count them. These test cases are stored for later use. It uses another counter, called nEffective, to count the number of effective members of the current population. This counter indicates whether the current population contains any effective members. The overall GA is presented below.

```
/* A GA algorithm to automatically generate test cases for a given program */
Input:
        Instrumented version P' of the program to be tested P;
        List of def-use paths to be covered;
        Number of program input variables;
        Domain and precision of input data;
        Population size;
        Maximum no. of generations (Max_Gen);
        Probability of crossover;
        Probability of mutation;
Output:
        Set of test cases for P, and the set of def-use paths covered by each test case;
        List of uncovered def-use paths, if any;
Begin
   Step 1: Initialization
            Initialize the def-use coverage vector to zeros;
            Create Initial_Population;
            Current_population ← Initial_Population;
            Set of test cases for P ← ϕ;
            Coverage_Percent ← 0;
            No_Of_Generations ← 0;
            nCases ← 0;
   Step 2: Generate test cases
            nEffective ← 0;
```

```
For each member of current population do
Begin
  Convert the current chromosome to the corresponding set of decimal values;
  Execute P' with this data set as input;
  Evaluate the current test case;
  If (some def-use paths are covered) then
      nCases ← nCases + 1;
      Add effective test case to set of test cases for P;
      Update the def-use coverage vector;
      Update Coverage_Percent;
      nEffective ← nEffective + 1;
  End If
End For;
While (Coverage_Percent ≠ 100 and No_Of_Generations ≤ Max_Gen) do
Begin
  If (nEffective > 0) then
    Select set of parents of new population from effective members of
    current population using roulette wheel method or random selection
    method;
  Else
    Set of parents of new population ← Current_Population;
  End If;
  Create New_Population using crossover and mutation operators;
  Current_Population ← New_Population;
  nEffective ← 0;
  For each member of Current_Population do
  Begin
    Convert current chromosome to the corresponding set of decimal values;
    Execute P' with this data set as input;
    Evaluate the current test case;
    If (some def-use paths are covered) then
       nCases ← nCases + 1;
       Add effective test cases to set of test cases for P;
       Update the def-use coverage vector;
       Update Coverage_Percent;
       nEffective ← nEffective + 1;
    End If
  End For;
  Increment No_Of_Generations;
End While;
```

Step 3: Produce output

```
Return set of test cases for P, and set of def-use paths covered by each test case;
Report on uncovered def-use paths, if any;
End.
```

4.7 Example

To illustrate the operation of the above algorithm, the result of applying the system, which implements it, to the example program (Figure 1), is presented below. Table 1 shows the def-use coverage vector of the example program at the end of the system execution.

```
POP_SIZE:  4
CROSSOVER PROBABILITY: 0.8
MUTATION PROBABILITY: 0.15
NO. OF INPUT VARIABLES: 3
DOMAIN AND PRECESSION OF INPUT VARIABLES:
 1-20, 0; 1-20, 0; 1-20, 0
** GA STARTED **
* INITIAL POPOULATION *
  000011100101110           2,16,10
  111111111010101          20,19,14
  110001110011110          16,18,19
  101101011010100          14,14,13
CASE 1: *** SELECTED ***
 TRAVERSED PATH: 1,8,10,12,13,14
 COVERED DCU-PATHS:  5
 COVERED DPU-PATHS:  3,4,15,16,19,20
 * DEF-USE COVERAGE:              24.1%
 * ACCUMULATED DEF-USE COVERAGE:  24.1%
CASE 2: *** SELECTED ***
 TRAVERSED PATH: 1,8,9,13,14
 COVERED DCU-PATHS: 3,8
 COVERED DPU-PATHS: 13,14
 * DEF-USE COVERAGE:              13.8%
 * ACCUMULATED DEF-USE COVERAGE:  37.9%
CASE 3: *** SELECTED ***
 TRAVERSED PATH: 1,2,3,7,14
 COVERED DCU-PATHS: 1,6
 COVERED DPU-PATHS: 1,2,5,6
 * DEF-USE COVERAGE:              20.7%
 * ACCUMULATED DEF-USE COVERAGE:  58.6%
CASE 4: *** NOT SELECTED ***
* PARENT SELECTION USING ROULETTE WHEEL METHOD *
  000011100101110
  111111111010101
  111111111010101
  000011100101110
* CROSSOVER OPERATION *
SELECTED PARENTS  CROSSOVER     OFFSPRING
                  POSITION
1, 2              3             110011100101110   001111111010101
3, 4              13            000011100101101   111111111010110
* MUTATION OPERATION *
SELECTED CHROMOSOME     MUTATION POSITION     MUTATED CHROMOSOME
1                       5                     110001100101110
2                       7                     001111011010101
3                       14                    000011100101111
4                       2                     101111111010110
4                       15                    101111111010111
* NEW POPOULATION *
  110001100101110          16,16,10
  001111011010101           5,14,14
  000011100101111           2,16,10
```

```
  101111111010111          15,19,15
CASE 5: *** NOT SELECTED ***
CASE 6: *** NOT SELECTED ***
CASE 7: *** NOT SELECTED ***
CASE 8: *** NOT SELECTED ***
* PARENTS = CURRENT POPULATION *
* CROSSOVER OPERATION *
SELECTED        CROSSOVER        OFFSPRING
PARENTS         POSITION
1, 4            11               101111111001110   110001100110111
* MUTATION OPERATION *
SELECTED CHROMOSOME     MUTATION POSITION     MUTATED CHROMOSOME
2                       6                     001110011010101
4                       6                     110000100110111
4                       8                     110000110110111
* NEW POPOULATION *
  101111111001110          15,19,10
  001110011010101           5, 5,14
  000011100101111           2,16,10
  110000110110111          16, 9,15
CASE 9: *** SELECTED ***
 TRAVERSED PATH: 1,8,10,11,12,13,14
 COVERED DCU-PATHS:  4,9
 COVERED DPU-PATHS: 17,18
 * DEF-USE COVERAGE:              13.8%
 * ACCUMULATED DEF-USE COVERAGE:  72.4%
CASE 10: *** SELECTED ***
 TRAVERSED PATH: 1,2,4,5,6,7,14
 COVERED DCU-PATHS:  2,7
 COVERED DPU-PATHS:  7,8,9,10
 * DEF-USE COVERAGE:              20.7%
 * ACCUMULATED DEF-USE COVERAGE:  93.1%
CASE 11: *** NOT SELECTED ***
CASE 12: *** SELECTED ***
 TRAVERSED PATH: 1,2,4,6,7,14
 COVERED DPU-PATHS: 11,12
 * DEF-USE COVERAGE:               6.9%
 * ACCUMULATED DEF-USE COVERAGE: 100.0%
** GA TERMINATED **
** NO. OF GENERATIONS = 3
** GENERATED TEST CASES **
           2,16,10
          20,19,14
          16,18,19
          15,19,10
           5, 5,14
          16, 9,15
```

Dcu-path	1	2	3	4	5	6	7	8	9	
Test Case	3	10	2	9	1	3	10	2	9	
Dpu-path	1	2	3	4	5	6	7	8	9	10
Test Case	3	3	1	1	3	3	10	10	10	10
Dpu-path	11	12	13	14	15	16	17	18	19	20
Test Case	12	12	2	2	1	1	9	9	1	1

Table 1: The def-use coverage vector of the example program

5 Experimental Results

This section presents the results of the experiments that have been carried out to evaluate the effectiveness of the proposed GA compared to the random testing (RT) technique, and to compare the proposed random selection method to the roulette wheel method. A set of 15 small FORTRAN programs is used in the experiments. Two of these programs (Prog#14 and 15) include subroutines. To achieve a fair comparison, the random test data generator was designed to randomly generate sets of pop_size test cases in each iteration. The used GA parameters were as follows: Max_Gen = 100, pc = 0.8, and pm = 0.15.

Prog#	No. of Variables	Pop. Size	Method	No. of Generations	No. of Test Cases	Def-Use Coverage %
1	3	8	GA	4	6	100
			RT	2	6	100
2	3	8	GA	1	6	100
			RT	2	7	100
3	3	8	GA	3	4	100
			RT	35	4	100
4	3	8	GA	16	4	100
			RT	28	4	100
5	3	8	GA	18	6	81.3
			RT	26	6	81.3
6	4	8	GA	3	8	100
			RT	5	8	100
7	3	8	GA	4	5	100
			RT	91	5	100
8	3	10	GA	15	10	63.6
			RT	51	9	58.4
9	2	8	GA	1	3	100
			RT	4	4	100
10	1	8	GA	7	5	100
			RT	3	5	100
11	1	8	GA	2	5	92
			RT	4	5	92
12	3	8	GA	5	5	100
			RT	91	5	100
13	2	4	GA	5	3	96
			RT	2	3	96
14	1	8	GA	2	6	63.6
			RT	15	6	63.6
15	3	8	GA	8	11	87.2
			RT	16	9	81.7

Table 2: A comparison between the GA technique and the random testing (RT) technique

Table 2 shows the results of applying the GA technique and the RT technique to the 15 programs. As can be seen, the GA technique outperformed the RT technique in 12 out of the 15 programs. In 10 of these programs, the GA technique required less number of generations than the RT technique to achieve the same def-use coverage percentage. For Prog#8, the RT technique required 51 generations to cover 58.4% of the def-use paths, while the GA technique required only 15 generations to cover 63.6% of the def-use paths, and for Prog#15, the RT technique required 16 generations to cover 81.7% of the def-use paths, while the GA technique required only 8 generations to cover 87.2% of the def-use paths; i.e. in these two programs, the GA reached higher coverage percentage in fewer generations than the random testing technique.

Prog#	Method	No. of Generations	No. of Test Cases	Def-Use Coverage %
1	Roulette Wheel	4	6	100
	Random Selection	4	6	100
2	Roulette Wheel	1	6	100
	Random Selection	1	6	100
3	Roulette Wheel	3	4	100
	Random Selection	3	4	100
4*	Roulette Wheel	42	4	100
	Random Selection	16	4	100
5*	Roulette Wheel	19	6	81.3
	Random Selection	18	6	81.3
6*	Roulette Wheel	16	8	100
	Random Selection	3	8	100
7	Roulette Wheel	4	5	100
	Random Selection	4	5	100
8*	Roulette Wheel	32	10	63.6
	Random Selection	15	10	63.6
9	Roulette Wheel	1	3	100
	Random Selection	1	3	100
10*	Roulette Wheel	10	5	100
	Random Selection	7	5	100
11*	Roulette Wheel	5	5	92
	Random Selection	2	5	92
12	Roulette Wheel	5	5	100
	Random Selection	5	5	100
13	Roulette Wheel	5	3	96
	Random Selection	9	3	96
14	Roulette Wheel	2	6	63.6
	Random Selection	2	6	63.6
15*	Roulette Wheel	25	13	87.2
	Random Selection	8	11	87.2

Table 3: A comparison between parent selection methods used in the GA technique

It should be noted that, in the cases where less than 100% coverage is achieved, the programs included some def-use paths that cannot be covered by any test data due the existence of infeasible paths.

Table 3 shows a comparison between the two parent selection methods used in the GA technique. As can be seen, the proposed random selection method was better than the roulette wheel method in 7 programs (the stared cases), and the roulette wheel method was better than the random selection method in only one program (the shaded case). In the remaining 7 programs, the performance of both methods was identical.

6 Conclusions

The GA technique presented in this paper is guided by the data flow dependencies in the program to search for test data to fulfil the all-uses criterion. This is the main contribution of this paper. The approach can be used in test data generation for programs with/without loops and procedures. The proposed GA accepts as input an instrumented version of the program to be tested, the list of def-use paths to be covered, the number of input variables, and the domain and precision of each input variable. Also, it accepts the GA parameters: population size, maximum number of generations, and probabilities of the crossover and mutation. The algorithm produces a set of test cases, the set of def-use paths covered by each test case, and a list of uncovered def-use paths, if any.

Experiments have been carried out to evaluate the effectiveness of the proposed GA compared to the random testing technique, and to compare the proposed random selection method to the roulette wheel method. The results of these experiments showed that the GA technique outperformed the random testing technique in 12 out of the 15 programs used in the experiment. In 10 of these programs, the GA technique required less number of generations than the random testing technique to achieve the same def-use coverage percentage. In two programs, the GA reached higher coverage percentage in fewer generations than the random testing technique. The experiments also showed that the proposed selection method produced better results than the roulette wheel method.

References

[Allen, 76] F.E. Allen, J. Cocke, A program data flow analysis procedure, Communication of the ACM, 19 (3), 137-147, 1976.

[Bauer, 79] J. A. Bauer, A. B. Finger, Test plan generation using formal grammars, Proceedings of the 4th International Conference on Software Engineering, IEEE Computer Society, pp. 425-432, 1979.

[Boyer, 75] R. S. Boyer, B. Elspas, K. N. Levitt, SELECT - a formal system for testing and debugging programs by symbolic execution, Proceedings of the International Conference on Reliable software, pp. 234-245, 1975.

[Bueno, 00] P. M. S. Bueno, M. Jino, Identification of potentially infeasible program paths by monitoring the search for test data, The 15th International Conference on Automated Software Engineering (ASE'00), Grenoble, France, 2000.

[Clarke, 76] L. A. Clarke, A system to generate test data and symbolically execute programs, IEEE Transactions on Software Engineering, 2 (3), 215-222, 1976.

[DeMillo, 91] R. A. DeMillo, A. J. Offutt, Constraint-based automatic test data generation, IEEE Transactions on Software Engineering, 17 (9), 900-910, 1991.

[Frankl, 93] P. G. Frankl, S. N. Weiss, An experimental comparison of the effectiveness of branch testing and data flow testing, IEEE Transactions on Software Engineering, 19 (8), 774-787, 1993.

[Girgis, 93] M. R. Girgis, Using symbolic execution and data flow criteria to aid test data selection, The Journal of Software Testing, Verification and Reliability, 3 (2), 101-112, 1993.

[Girgis, 00] M. R. Girgis, A system for interprocedural data flow analysis and testing, International Journal of Applied Mathematics, 3 (2), 133-150, 2000.

[Girgis, 85a] M.R. Girgis, M.R. Woodward, An integrated system for program testing using weak mutation and data flow analysis, Technical Report 85/1, Department of Computer Science, University of Liverpool, U.K, 1985.

[Girgis, 85b] M.R. Girgis, M.R. Woodward, An integrated system for program testing using weak mutation and data flow analysis, Proceedings of Eighth International Conference on Software Engineering, IEEE Computer Society, pp. 313-319, 1985.

[Goldberg, 89] D.E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, Reading, Mass., 1989.

[Holland, 75] J. Holland, Adaptation in Natural and Artificial Systems, ISBN 0 472 08460 7. University of Michigan Press, Ann Arbor, MI, 1975.

[Howden, 77] W. E. Howden, Symbolic testing and the DISSECT symbolic evaluation system, IEEE Transactions on Software Engineering, 3 (4), 266-278, 1977.

[Jones, 96] B. F. Jones, H. -H. Sthamer, D. E. Eyres, Automatic structural testing using genetic algorithms, Software Engineering Journal, 8 (9), 299-306, 1996.

[Jones, 98] B. F. Jones, D. E. Eyres, H. -H. Sthamer, A strategy for using genetic algorithms to automate branch and fault-based testing, The Computer Journal, 41 (2), 98-107, 1998.

[Korel, 90] B. Korel, Automated software test data generation, IEEE Transactions on Software Engineering, 16 (8), 870-879, 1990.

[Lin, 01] J. -C. Lin, P. -L. Yeh, Automatic test data generation for path testing using GAs, Information Sciences, 131 (1-4), 47-64, 2001.

[Maurer, 90] P. M. Maurer, Generating testing data with enhanced context-free grammars, IEEE Software, 7 (4), 1990.

[Michael, 01] C. C. Michael, G. McGraw, M.A. Schatz, Generating Software Test Data by Evolution, IEEE Transactions on Software Engineering, 27 (12), 1085-1110, 2001.

[Michalewicz, 99] Z. Michalewicz, Genetic algorithms + data structures = evolution programs, 3rd Edition, Springer, 1999.

[Miller, 75] E. F. Miller, R. A. Melton, Automated generation of test case data sets, Proceedings of the International Conference on Reliable Software, pp. 51-58, 1975.

[Mills, 87] H. D. Mills, M. D. Dyer, R. C. Linger, Cleanroom software engineering, IEEE Software, 4 (5), 19-25, 1987.

[Pargas, 99] R.P. Pargas, M.J. Harrold, R.R. Peck, Test-Data Generation Using Genetic Algorithms, The Journal of Software Testing, Verification and Reliability, 1999.

[Pei, 94] M. Pei, E. D. Goodman, Z. Gao, K. Zhong, Automated Software Test Data Generation Using A Genetic Algorithm, Technical Report GARAGe of Michigan State University, 1994.

[Ramamoorthy, 76] C. V. Ramamoorthy, S. F. Ho, W. T. Chen, On the automated generation of program test data, IEEE Transactions on Software Engineering, 2 (4), 293-300, 1976.

[Rapps, 85] S. Rapps, E.J. Weyuker, Selecting software test data using data flow information, IEEE Transactions on Software Engineering, 11 (4), 367-375, 1985.

[Roper, 95] M. Roper, I. Maclean, A. Brooks, J. Miller, Wood, M. Genetic Algorithms and the Automatic Generation of Test Data, Technical Report RR/95/195 [EFoCS-19-95], University of Strathclyde, Glasgow G1 1XH, U.K, 1995.

[Srinivas, 94] M. Srinivas, L. M. Patnaik, Genetic algorithms: a survey, IEEE Computer, 27 (6), 17-26, 1994.

[Voas, 91] J. M. Voas, L. Morell, K. W. Miller, Predicting where faults can hide from testing, IEEE Software, 8 (2), 41-48, 1991.

[Watkins, 95] A. E. L. Watkins, A Tool for the Automatic Generation of Test Data Using Genetic Algorithms, In Proceedings of Software Quality Conference, Dundee, Scotland, 1995.