Computing and Informatics, Vol. 26, 2007, 383–401

AUTOMATIC SOFTWARE TEST DATA GENERATION FOR SPANNING SETS COVERAGE USING GENETIC ALGORITHMS

Abdelaziz M. Khamis

Department of Computer Science, ISSR
Cairo University, Egypt
&
Ajman University of Science and Technology, UAE
e-mail: khamis@frcu.eun.eg

Moheb R. Girgis

Department of Computer Science, Faculty of Science
Minia University, Egypt
e-mail: mrgirgis@mailer.eun.eg

Ahmed S. Ghiduk

Department of Mathematics, Faculty of Science
Beni-Suef University, Egypt
&
College of Computing, Georgia Institute of Technology, USA
e-mail: ghiduk@cc.gatech.edu

Manuscript received 25 May 2005; revised 29 January 2007
Communicated by Imre J. Rudas

Abstract. Software testing takes a considerable amount of time and resources spent on producing software. Therefore, it would be useful to have ways to reduce the cost of software testing. The new concepts of spanning sets of entities suggested by Marré and Bertolino are useful for reducing the cost of testing. In fact, to reduce the testing effort, the generation of test data can be targeted to cover the entities

in the spanning set, rather than all the entities in the tested program. Marré and Bertolino presented an algorithm based on the subsumption relation between entities to find spanning sets for a family of control flow and data flow-based test coverage criteria. This paper presents a new general technique for the automatic test data generation for spanning sets coverage. The proposed technique applies to the algorithm proposed recently by Marré and Bertolino to automatically generate the spanning sets of program entities that satisfy a wide range of control flow and data flow-based test coverage criteria. Then, it uses a genetic algorithm to automatically generate sets of test data to cover these spanning sets. The proposed technique employed the concepts of spanning sets to limit the number of test cases, guide the test case selection, overcome the problem of the redundant test cases and automate the test path generation.

Keywords: Genetic algorithms, automatic test-data generation, subsumption, spanning sets

1 INTRODUCTION

Software testing is a main method for improving the quality and increasing the reliability of software now and thereafter the long-term period future. It is a kind of complex, labor-intensive, and time consuming work; it accounts for approximately 50 % of the cost of a software system development. Increasing the degree of automation and the efficiency of software testing certainly can reduce the cost of software design, decrease the time period of software development, and increase the quality of software significantly. The critical point of the problem involved in automation of software testing is of particular relevance of automated software test data generation. Test data generation in software testing is the process of identifying a set of program input data, which satisfies a given testing criterion.

For solving this difficult problem, random, symbolic, and dynamic test data generation techniques have been used in the past. Recently, genetic algorithms have been applied successfully to generate test data.

Random test data generation: It consists of generating inputs at random until a useful input is found [1, 2, 3]. The problem with this approach is clear with complex programs or complex adequacy criteria, where an adequate test input may have to satisfy very specific requirements. In such a case, the number of adequate inputs may be very small compared to the total of inputs, so probability of selecting an adequate input by chance can be low.

Symbolic test data generation: There are many test data generation methods that use symbolic execution to find inputs that satisfy a test requirement [4–8]. Symbolic execution of a program consists of assigning symbolic values to variables in order to come up with an abstract, mathematical characterization of what the program does. Thus, ideally, test data generation can be reduced

to a problem of solving an algebraic expression. A number of problems are encountered in practice when symbolic execution is used. One such problem arises in indefinite loops, where the number of iterations depends on a non-constant expression, and the index of array, where data is referenced indirectly as $a = B[c + d]/10$. Pointer references also present a problem because of the potential for aliasing.

Dynamic test data generation: This paradigm is based on the idea that if some desired test requirement is not satisfied, data collected during execution can still be used to determine which tests come closest to satisfying the requirement [9, 10]. With the help of this feedback, test inputs are incrementally modified until one of them satisfies the requirement. Two limitations are commonly found in dynamic test data generation systems. First many systems make it difficult to generate tests for large programs because they only work on simplified programming languages. Second, many systems use gradient descent techniques to perform function minimization and, therefore, they can stall when they encounter local minima.

Test data generation based on genetic algorithms (GAs): Recently some techniques have been proposed which are based on genetic algorithms to generate test data [11–16]. The new features of GAs make these techniques capable to find the nearly global optimum. All these techniques use genetic algorithms to generate test suites that cover all entities of the tested program which are required by a control flow or data flow coverage criterion.

In [17] Marré and Bertolino introduced the concepts of spanning sets of entities for a coverage criterion which can help reduce the cost of testing without impairing testing efficacy. A spanning set is a minimum subset of entities with the property that any set of test cases covering this subset cover every entity in the program. In addition, they have suggested several applications of spanning sets such as evaluating the ratio between the covered entities and the total number of entities in the tested program, estimating the number of test cases needed to satisfy a selected coverage criterion, preventing redundant test cases, guiding test case selection to cover untested entities, and automating test path generation.

To our knowledge, until now the concepts of spanning sets have not been employed in the process of test data generation.

This paper presents a new general technique for the automatic test data generation for spanning sets coverage. This technique automatically generates spanning sets of program entities that satisfy a family of control flow and data flow-based test coverage criteria by applying the algorithm proposed by Marré and Bertolino. Subsequently, it uses genetic algorithms to automatically generate sets of test data to cover these spanning sets. This technique will be used to evaluate the effectiveness of spanning sets and the subsumption relation between entities in coverage testing.

The paper is organized as follows: Section 2 gives some important definitions. Section 3 describes the general algorithm to find spanning sets of entities. Section 4

describes our proposed technique. Section 5 presents a case study for the proposed technique. Section 6 presents the conclusions and future work.

2 BACKGROUND

2.1 The Ddgraph Model

Typically, in code-based testing strategies a program's structure is analyzed on the program flowgraph, i.e., an annotated digraph which represents graphically the information needed to select the test cases. A program control flow may be mapped onto a flowgraph in different ways. We use a flowgraph representation called ddgraph (decision-to-decision graph) [18].

Definition 1 (Ddgraph)**.** A ddgraph is a digraph $G = (N, A)$, where N is a set of nodes and A is a set of arcs, with two distinguished arcs e_1, e_k (the unique entry arc and the unique exit arc, respectively), such that any other arc in A is reached by e_1 and reaches e_k, and such that for each node $n \in N, n \neq \text{TAIL}(e_1), n \neq \text{HEAD}(e_k)$, $(indegree(n) + outdegree(n)) > 2$, while indegree(TAIL($e_1$)) = 0 and outdegree(TAIL(e_1)) = 1, indegree(HEAD(e_k)) = 1 and outdegree(HEAD(e_k)) = 0.

An arc e in a ddgraph G is an ordered pair of adjacent nodes, called TAIL and HEAD of e, respectively (i.e., $e = (\text{TAIL}(e), \text{HEAD}(e))$. A path p of length q in a ddgraph G is a sequence $p = e_1, e_2, \ldots, e_q$, where $\text{TAIL}(e_{i+1}) = \text{HEAD}(e_i)$ for $i = 1, 2, \ldots, q-1$. A path p is simple if all its nodes, except possibly the first and last ones, are distinct. A complete path in a ddgraph G is a path from the entry node to the exit node of G. Given a path $p = e_1, e_2, \ldots, e_q$, then a path $'p = e_i, \ldots, e_j$ from e_i to e_j, with $1 \leq i \leq j \leq q$, is called a subpath of p.

Code Lines	Ddgraph Component
1–9	e_1
10	n_2
11–12	e_2
13	n_3
14	n_4
15–17	e_4
18–19	e_6
20–24	e_7
25	e_8

Table 1. Correspondence between instructions in SORT and arcs and nodes in ddgraph G_{SORT}

Figure 1 shows an example program SORT (adapted from [19]) and the corresponding ddgraph G_{SORT}. The distinguished arcs of G_{SORT} are e_1 (the entry arc) and e_8 (the exit arc). The correspondence between the instructions in the program

```
void sort (a, n)  /*SORT program*/
1   int a[];
2   int n;
3   {
4     int sortupto;
5     int maxpos;
6     int mymax;
7     int index;
8     sortupto = 1;
9     maxpos = 1;
10    while(sortupto < n){
11      mymax = a[sortupto];
12      index = sortupto +1;
13      while (index <= n) {
14        if (a[index]> mymax){
15          mymax = a[index];
16          maxpos = index;
17        }
18        index = index +1;
19    }
20    index = a[sortupto];
21    a[sortupto] = mymax;
22    a[maxpos] = index;
23    sortupto = sortupto +1;
24  }
25  }
```

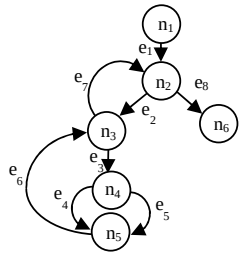

Fig. 1. Program SORT and ddgraph G_{SORT}

(as numbered in Figure 1) and arcs and nodes in the ddgraph is shown in Table 1. Note that arcs e_3 and e_5 and node n_5 do not correspond to any program instruction.

2.2 Def-Use Ddgraph

The occurrences of a variable in a program can be associated with the following events:

Def. A statement storing a value in a memory location of a variable creates a definition (def) of the variable.

Use. A statement drawing a value from the memory location of a variable is a use of the currently active definition of the variable. In particular, when the variable appears on the right-hand side of an assignment statement, it is called a computational use (c-use); when the variable appears in the predicate of the conditional control transfer statement, it is called a predicate use (p-use).

Killing. A statement kills the currently active definition of a variable when the value associated with it becomes unbound.

Data flow testing considers the possible interactions between definitions and uses of variables. To analyze these interactions, programs are represented as annotated flowgraph called def-use ddgraph [17].

Given a ddgraph corresponding to a program, for every variable in the program, a def-use ddgraph is derived, in which each arc is annotated with a sequence (which may be empty) of the symbols d or u, to represent that the variable of interest is defined or referenced in the program block represented by such arc. Note that for a predicate use (which occurs in this ddgraph model at a decision node) each arc leaving the node at which the predicate occurs is labeled with a use symbol u. Figure 2 shows two def-use ddgraphs corresponding to the occurrences of variable index and variable a in program SORT.

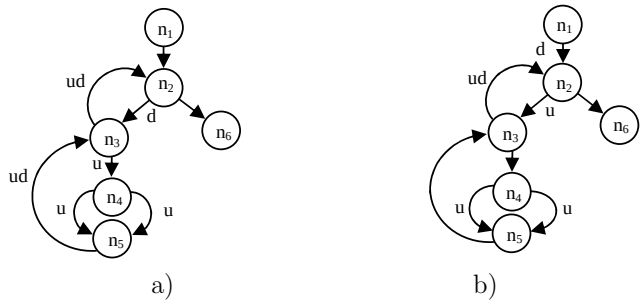

Fig. 2. a) The def-use ddgraph for variable index, b) The def-use ddgraph for variable a, in Program SORT

Given a def-use ddgraph G for variable X, a def-clear path with respect to X is a path $p = e, e_1, e_2, \ldots, e_q, é$ on G, with $q \geq 0$, such that X may be defined in e, and is not redefined or killed in any of the arcs $e_1, e_2, \ldots, e_q$. For example, $p_1 = e_2, e_3, e_5, e_6$ is a def-clear path for variable index in G_{SORT} (because index is defined in e_2 and is not redefined or killed on e_3 or e_5). On the other hand, $p_2 = e_2, e_3, e_4, e_6, e_7$ is not a def-clear path for index in G_{SORT} (since index is redefined in e_6).

Definition 2 (Dua)**.** Let d and u be two arcs in G, and X be a variable. The triple $[d, u, X]$ is said to be a definition-use association or a dua if X has a global definition in d, a global use in u, and there is a def-clear path wrt X from d to u. For example, $T_1 = [e_1, e_7, index]$ and $T_2 = [e_6, e_4, index]$ are duas in G_{SORT}. On the other hand, $T_3 = [e_7, e_4, index]$ is not a dua, since index is defined in e_7, index is used in e_4, but there is no a def-clear path wrt index from e_7 to e_4.

2.3 Coverage Testing Criteria

Coverage criteria require a set of entities of the program flowgraph to be covered when the tests are executed. A set of complete paths satisfy a criterion if it covers the set of entities associated with that criterion. Depending on the criterion

selected, the entities to be covered may be derived from the program control flow or form the program data flow. In [17], Marré and Bertolino re-defined a family of popular control flow and data flow test coverage criteria ([20–22]) and denoted the corresponding set $E_c(G)$ of entities of G for each test coverage criterion c. Table 2 shows some of these coverage criteria and identifies the entities of the program flow-graph that are associated with them. In this table, $G = (N, A)$ denotes a ddgraph, and $\wp$ denotes the set of all complete paths in G.

Coverage Criterion c	Set $E_c(G)$ of entities for ddgraph G and coverage criterion c
All-Paths	$\{p \in \wp : p$ is a complete path in $G\}$
All-k-Paths	$\{p \in \wp : p$ is a complete path in G and no loop in p is iterated more than k times$\}$
All-Branches	A
All-Statements	$\{e \in A : e$ is associated with at least one instruction$\}$
All-Du-Paths	$\{p \in \wp : \exists T = [d, u, X] \in D(G)$ such that p is a simple def-clear path wrt X from d to $u\}$

Table 2. Coverage criteria and entities to be covered to satisfy these criteria

For example:

- The set of entities for the ddgraph $G_{SORT} = (N_{SORT}, A_{SORT})$ and All-Branches criterion is $E_{All-Branches}(G_{SORT}) = A_{SORT} = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$.
- The set of entities for the ddgraph GSORT and All-Statements criterion is $E_{All-Statements}(G_{SORT}) = A_{SORT} = \{e_1, e_2, e_3, e_4, e_6, e_7, e_8\}$.
- The set of entities for the ddgraph G_{SORT} and All-Uses criterion is $E_{All-Uses}(G_{SORT}) = D(G_{SORT}) = \{[e_1, e_8, n], [e_1, e_2, n], [e_1, e_7, n], [e_1, e_3, n], [e_1, e_2, a], [e_1, e_7, a], [e_7, e_2, a], [e_7, e_7, a], [e_7, e_4, a], [e_7, e_5, a], [e_1, e_4, a], [e_1, e_5, a], [e_6, e_6, index], [e_1, e_2, sortupto], [e_1, e_7, sortupto], [e_7, e_8, sortupto], [e_1, e_8, sortupto], [e_7, e_2, sortupto], [e_7, e_7, sortupto], [e_1, e_7, maxpos], [e_4, e_7, maxpos], [e_2, e_7, mymax], [e_2, e_4, mymax], [e_4, e_7, mymax], [e_4, e_4, mymax], [e_2, e_5, mymax], [e_4, e_5, mymax], [e_2, e_3, index], [e_6, e_7, index], [e_2, e_7, index], [e_2, e_4, index], [e_6, e_4, index], [e_6, e_3, index], [e_2, e_5, index], [e_6, e_5, index], [e_2, e_6, index]\}$.

2.4 Spanning Sets

A selected coverage criterion c determines for a given ddgraph G a set $E_c(G)$ of entities to be covered. In general, it is possible to derive a subset of $E_c(G)$ with the property that a set of complete paths covering all the entities in it will cover every c-entity in $E_c(G)$ [17]. For instance, for the ddgraph G_{SORT}, any set of complete paths covering the set of arcs: $\{e_3, e_4, e_5, e_6\} \subseteq E_{All-Branches}(G_{SORT})$ will cover every arc in $E_{All-Branches}(G_{SORT})$. This happens because any complete path that exercises these arcs must evidently exercises arcs e_1, e_2, e_7, and e_8 as well.

In other terms, for a selected criterion c the coverage of some c-entities automatically guarantees the coverage of other c-entities. This property implies a natural ordering between c-entities, according to how easily they can be covered. This ordering relationship between entities is called a subsumption [17]. The intuitive underlying idea is that if c-entity E_1 subsumes c-entity E_2, then one can forget about E_2 provided that one cares about E_1.

Definition 3 (Subsumption)**.** Let G be a ddgraph, c be a coverage criterion, and E_1 and E_2 be two entities in $E_c(G)$. Then, E_1 subsumes E_2 if every complete path that covers E_1 covers E_2 as well.

The entities that are not subsumed by other entities are said to be "unconstrained" (i.e., "not guaranteed"): the coverage of an unconstrained entity is not guaranteed by the coverage of any other entity. More precisely, an entity E is said to be unconstrained entity if there exists no other entity $\acute{E}$ that subsumes E without being itself subsumed by E. A smallest subset of c-entities with such a property is called a spanning set.

Definition 4 (Spanning set of entities)**.** Let G be a ddgraph and c be a coverage criterion. A subset U of $E_c(G)$ is said to be a spanning set of entities for G and c if a set of paths $\wp$ that covers every entity in U covers all the entities in $E_c(G)$; and if there is any other set $\acute{U} \subseteq E_c(G)$ such that any set of paths covers every entity in $\acute{U}$ covers all the entities in $E_c(G)$, then $|U| \leq |\acute{U}|$.

2.5 Genetic Algorithms (GAs)

Genetic algorithms (GAs) are commonly applied to a variety of problems involving search, optimization and machine learning within the AI domain. The principle behind GAs is that they create and maintain a population of individuals represented by chromosomes. In GAs each of a problem's parameters is represented as a binary string. Borrowing from biology, an encoded parameter can be thought of as a gene, where the parameter's values are the gene's alleles. The string produced by the concatenation of all the encoded parameters forms a genotype (chromosome). Each genotype specifies an individual which is in turn a member of a population.

The GAs starts by creating an initial population of individuals, each represented by randomly generated genotype. The fitness of individuals is evaluated in some problem-dependent way, and the GAs tries to evolve highly fit individuals from the initial population.

The genetic search process is iterative: evaluating, selecting, and recombining strings in the population during each iteration (generation) until reaching some termination condition.

The basic algorithm of GAs, where $P(t)$ is the population strings at generation number t, is as follows:

1. initialize $P(t)$
2. evaluate $P(t)$
3. while (termination condition not satisfied) do
   {
4. select $P(t+1)$ from $P(t)$;
5. recombine $P(t+1)$;
6. evaluate $P(t+1)$;
7. $t = t+1$;
   }

In the evaluation step, the fitness of each individual is determined. Evaluation of each string (individual) is based on a fitness function that is problem-dependent.

The selection step is used to find pairs of individuals that will be mated to contribute to the next generation. Selection of a string depends on its fitness relative to that of other strings in the population. Most often, the two individuals are selected at random, but each individual's probability of being chosen is proportional to its fitness. Thus, selection is done on the basis of relative fitness.

The crossover (or recombination), fills the role played by sexual reproduction in nature. The process of crossover involves two chromosomes swapping chunks of data (genetic information). Mutation introduces slight changes into a small proportion of population and is representative of an evolutionary step.

The above algorithm will iterate until the population has evolved to form a solution to the problem, or until a termination condition is satisfied. In our case the chromosome represents test data to satisfy the given criterion.

3 FINDING SPANNING SETS

In [17], Marré and Bertolino presented a technique to find a spanning set of entities for a given ddgraph G and a given coverage criterion c. For a given ddgraph G and a coverage criterion c, one can construct a digraph that represents the subsumption relationship. In this ddgraph, the nodes are the entities for G and c, and there is an arc from node E_2 to node E_1 if and only if E_1 subsumes E_2. The digraph thus obtained is called the c-subsumption digraph of G and is denoted by $S_c(G)$.

Clearly, if a c-subsumption digraph includes a strongly connected component M, whenever an entity in M is covered by a test path, then all entities in M are covered by this same test path (i.e., each entity in M subsumes every other entity in M). Hence, any entity in a strongly connected component M of $S_c(G)$ may be chosen to represent the whole set of entities in M. The entity represents a strongly connected component M called representative of that component and is denoted by $\mathrm{rep}(M)$.

By reducing the strongly connected components of a subsumption digraph, we could obtain a directed acyclic graph by merging all nodes in each strongly connected component M to a single node n_M. The digraph obtained is called a reduced c-subsumption digraph and is denoted by $R_c(G)$.

Let U be the set of all the leaves of the reduced c-subsumption digraph (i.e., the nodes with no exit arc) and take one representative for each. It can be proved that U is a spanning set of (unconstrained) entities.

Procedure FIND-A-SPANNING-SET-OF-ENTITES(G: ddgraph; $E_c(G)$: set of entities): set of entities;
1. for each $E_1, E_2 \in E_c(G)$, $E_1 \neq E_2$, do SUBSUMPTION(E_1, E_2, G);
2. construct $S_c(G) = (V_{S_c(G)}, E_{S_c(G)})$;
3. construct $R_c(G) = (V_{R_c(G)}, E_{R_c(G)})$;
4. $U = \{\text{rep}(M)$: M is a leaf of $R_c(G)\}$;
5. return(U).

Fig. 3. The procedure FIND-A-SPANNING-SET-OF-ENTITIES

In the procedure FIND-A-SPANNING-SET-OF-ENTITIES (Figure 3), steps 2, 3, and 4 do not depend on the type of entities manipulated, and rep(M) denotes the entity chosen to represent the strongly connected component M. On the other hand, evaluating whether entity E_1 subsumes entity E_2 (step 1) depends on the notion of "coverage" associated with the particular type of entities considered. Thus, there is a specific SUBSUMPTION (E_1, E_2, G) procedure for each possible type of entity (arc, dua, class of duas, and path).

4 THE PROPOSED TECHNIQUE

This section describes our proposed technique to test C++ programs. This technique incorporates Marré and Bertolino's spanning sets-based techniques into testing processes and using these techniques with genetic algorithms techniques to solve the problem of deriving test data that covers the spanning sets. Figure 4 shows the overall diagram of our proposed technique.

It performs the following tasks:

- Analysis and reformatting of source code.
- Generating spanning sets of program entities to be covered.
- Generating a test data using genetic algorithms.

The technique performs these tasks in three stages. We give a detailed description of these three stages of the technique in the following subsections.

4.1 Analysis Phase

The analysis and reformatting module, shown in Figure 5, has been built to perform the following tasks:

1. Classify program statements and reformat some of them to facilitate the construction of the program flow graph.

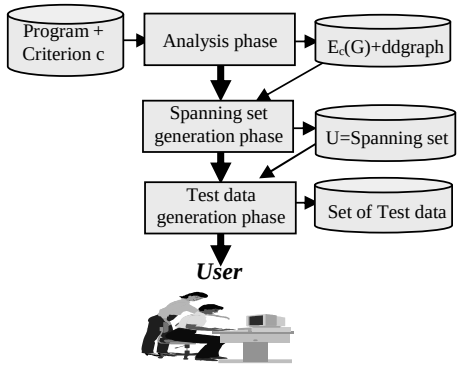

Fig. 4. The block diagram of the proposed technique

2. Construct the control flow graph CFG of the reformatted version of the program.
3. Reduce the control flow graph to obtain the program ddgraph G using the algorithm proposed by Bertolino and Marré in [18].
4. Produce the set $E_c(G)$ of entities to be covered that satisfy a coverage criterion c (e.g. the set of duas).

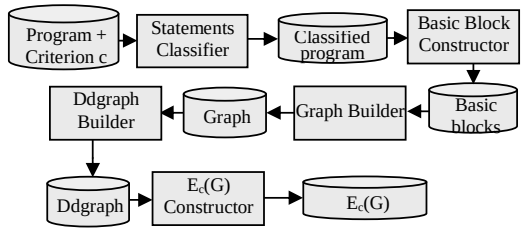

Fig. 5. Analysis phase

4.2 Spanning Set Generation Phase

Now, we describe how our proposed technique finds a spanning set of entities for a given program and a given coverage criterion c.

We use the algorithm of Marré and Bertolino [17] to find the spanning set of entities that satisfy the given coverage criterion c. Figure 6 shows the overall diagram of the spanning set generation phase.

It performs the following tasks to find a spanning set:

1. Construct the subsumption digraph $S_c(G)$ that represents the subsumption relationship between the entities.

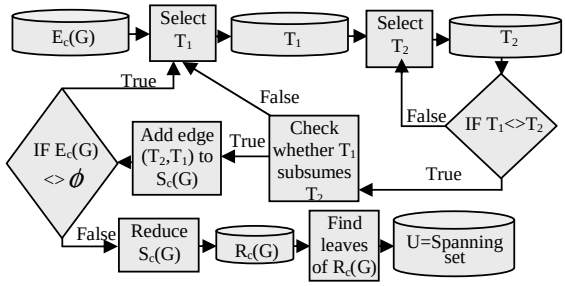

Fig. 6. Spanning set generation phase

2. Reduce the subsumption digraph $S_c(G)$ to obtain the reduced c-subsumption digraph $R_c(G)$.
3. Produce the spanning set of entities U which consists of the leaves of the reduced EcE-subsumption digraph $R_c(G)$.

4.3 Test Data Generation Phase

This phase uses a genetic algorithm to generate test data that will execute a set of paths that covers the spanning set. It performs the following tasks to find a set of test data:

1. Find the set of test data that form the initial population.
2. Determine the fitness of each individual which is based on a fitness function that is problem-dependent.
3. Select two individuals that will be mated to contribute to the next generation.
4. Apply the crossover and mutation processes.

The above algorithm will iterate until the population has evolved to form a solution to the problem, or until a termination condition is satisfied. Figure 7 shows the flowchart of the above algorithm.

Now, we present the whole algorithm of our proposed technique as follows:

Algorithm: Generate test data using a GA to cover the spanning set of entities that satisfy a given coverage criterion c.

Input: Prog := tested program, c := a coverage criterion.

Output: Test_cover := set of test data that cover the entities of the spanning set of the given coverage criterion c.

Declare:

CFG: the control flow graph of Prog.
G: ddgraph (the reduced graph of CFG).
$E_c(G)$: the set of all entities that satisfy criterion c.

$S_c(G)$: the subsumption digraph.
$R_c(G)$: the reduced c-subsumption digraph.
U := list of test requirements(spanning set).
T: test requirement for which a test case is to be generated
InitialPopulation, NewPopulation: set of test cases
MaxAttempts(): function that returns true when maximum number of attempts for target T has been exceeded and false otherwise.
TimeOver(): function that returns true when the time limit is exceeded and false otherwise.
Begin
/* Step 1: Analysis the program under test */
[1] Classify the program's statements
[2] Construct the program's basic blocks
[3] Build the program's control flow graph CFG
[4] Reduce CFG to obtain the ddgraph G
[5] Construct the set of entities Ec(G)
/* Step 2: Find the spanning set of entities that satisfy
the coverage criterion c */
[6] for each $E_1, E_2 \in E_c(G)$, $E_1 \neq E_2$, do SUBSUMPTION(E_1, E_2, G);
[7] construct $S_c(G) = (V_{S_c(G)}, E_{S_c(G)})$;
[8] construct $R_c(G) = (V_{R_c(G)}, E_{R_c(G)})$;
[9] $U = \{\text{rep}(M)$: M is a leaf of $R_c(G)\}$;
/* Step 3: Generate test */
/*step 3.1 Initialize and set up */
[10] Create and Initialize Test_cover.
[11] Generate InitialPopulation of test cases.
/* Step 3.2: Generate test data */
[12] While (U not empty and not TimeOver()) do
[13] Select uncovered entity T from U.
[14] while (T not covered and not MaxAttempts()) do
[15] Evaluate fitness values of current population.
[16] Sort current population according to fitness.
[17] Select parents of NewPopulation.
[18] Apply genetic operators (crossover and mutation) on the selected parents to generate NewPopulation.
[19] Execute tested program on each member of NewPopulation.
[20] Update Test_cover and access U to delete targets that are satisfied or have max attempts.
[21] Endwhile
[22] Endwhile
/* Step 3.3: Clean up and return */
[23] Test_cover := test cases that satisfy U.
[24] Return(Test_cover, U).
[25] End.

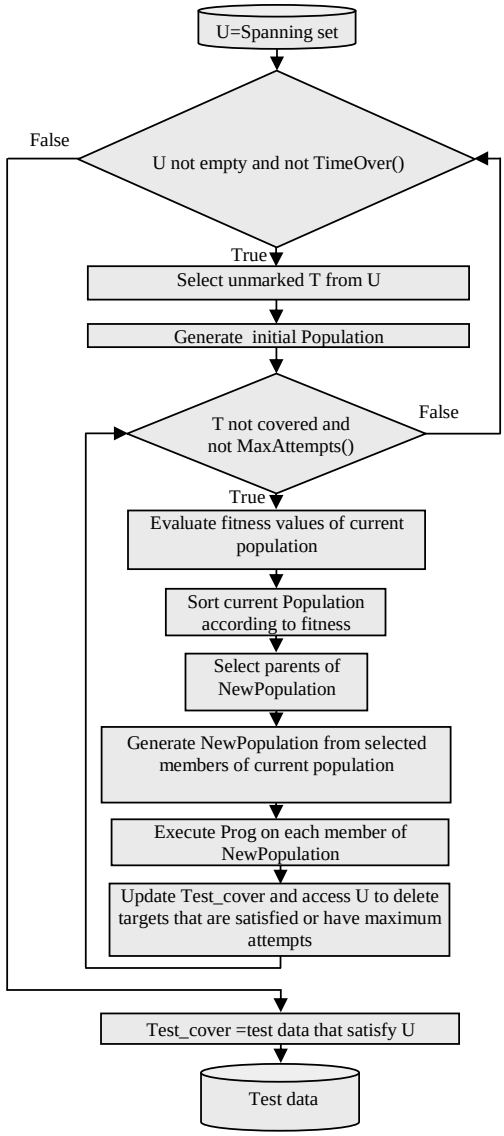

Fig. 7. Test data generation phase

5 A CASE STUDY

In the following paragraphs, we give a detailed description of the technique as applied to the example program given in Figure 1 to achieve all-use coverage criterion.

In the analysis phase, our proposed technique uses a function, BUILDGRAPH(), to construct the control flow graph CFG of the program. Then, it constructs the reduced flow graph ddgraph of G according to Definition 1 in Section 2 by using the procedure REDUCE given in [18]. Figure 1 shows the example program SORT and the corresponding ddgraph G_{SORT}.

Then, the technique produces the set $E_{All-Uses}(G_{SORT})$ of entities for the ddgraph G_{SORT} and All-Uses criterion. The elements of the set $E_{All-Uses}(G_{SORT})$ are given in Section 2.3.

In the spanning set generation phase, the technique constructs the subsumption digraph $S_{All-Uses}(G_{SORT})$ that represents the subsumption relationship between the entities. Then, it reduces the subsumption digraph $S_{All-Uses}(G_{SORT})$ to obtain the reduced c-subsumption digraph $R_{All-Uses}(G_{SORT})$, shown in Figure 8.

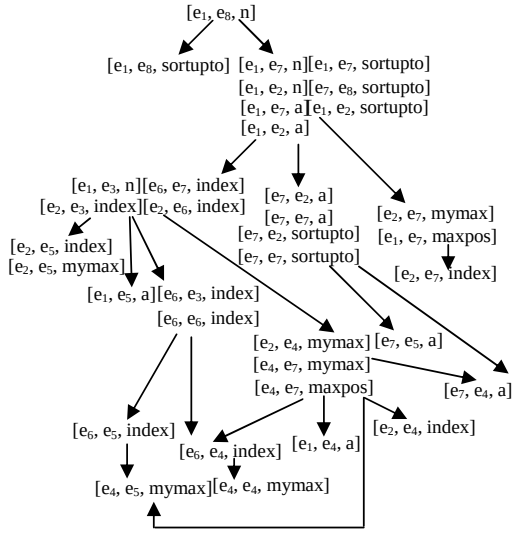

Fig. 8. Reduced subsumption digraph for GSORT and all-uses criterion

The technique produces the spanning set of entities $U_{All-Uses}(G_{SORT}) = \{[e_7, e_4, a], [e_7, e_5, a], [e1, e4, a], [e_1, e_5, a], [e_1, e_8, sortupto], [e_4, e_4, mymax], [e_4, e_5, mymax], [e_2, e_7, index], [e_2, e_4, index], [e2, e5, index]\}$ which consists of the leaves of the reduced c-subsumption digraph $R_{All-Uses}(G_{SORT})$.

In the test data generation phase, the proposed technique selects an element of the spanning set of entities $U_{All-Uses}(G_{SORT})$. Then, the genetic algorithm is used to generate a test data that will execute a path that covers the selected element of the spanning set. This phase iterates for each element in the spanning set. The parameters of the genetic algorithm can be adapted according to the parameters suggested by Girgis in [16] (i.e., population size = 10, maximum number of generation = 100, the probability of crossover operator = 0.8, the probability of mutation operator = 0.15). The selection of the initial population for a subsequent

run can be enhanced by using one of the previously generated data as part of the new initial population, such that the program entity covered by this data and the one to be covered have some common element. For example, the dua $[e_7, e_5, a]$ and $[e_7, e_4, a]$ have the same definition point e_7. So, the data generated to cover the first dua can be used as part of the initial population for the second dua.

Theoretically, to cover n program entities we need n test cases. But our technique reduces this number of test cases to the number of elements in the spanning set of the required entities which is less than the number of the required entities.

In the case study, the number of test cases is 10 (cardinality of the spanning set) instead of 36 (the cardinality of the set of the required entities). Consequently, the ratio of reducing the test cases is 72 %.

6 CONCLUSIONS AND FUTURE WORK

Marré and Bertolino [17] have introduced the concepts of spanning sets of entities for coverage testing. They have suggested several potential applications of spanning sets as reducing and estimating the number of tests and evaluating test adequacy more effectively. They have provided a method to derive a spanning set that is parameterized in the subsumption relation between entities. This method does not handle the problem of deriving test data.

In this paper, we presented a new general technique that combines the concept of spanning set with a genetic algorithm to automatically generate test data for spanning set coverage. Our proposed technique applies the algorithm introduced by Marré and Bertolino to automatically generate the spanning sets of program entities that satisfy a vast array of control flow and data flow-based test coverage criteria. Then, it uses a genetic algorithm to automatically generate sets of test data to cover these spanning sets. The proposed technique employed the concepts of spanning sets to set a bound to the number of test cases; for example the bound of test cases is 10 in the case study. In addition, our technique overcomes the problem of the redundant test cases and guides the test case selection by concentrating only on the elements of the spanning set. The results of the case study are very encouraging because they have shown that the cost of testing reduced by 72 %.

Our future work will concentrate on developing a fitness function definition for each one the family of coverage criteria that have been suggested by Marré and Bertolino in [17]. In addition, the parameters of the genetic algorithms such as chromosomes representation, genetic operators, and initial population will be improved to be suitable for this family of criteria. A further work is also required to determine the effect of the population size and control the genetic operators so that they don't destroy the data types of the program's input. Also, the problems of infeasible paths identification and stopping the system from trying to find solutions in the presence of infeasible paths will be investigated in our future work.

Now, a software tool is being implemented to evaluate the efficiency of this technique. This tool will be used to evaluate the effectiveness of the spanning sets

and the subsumption relation between entities in coverage testing. Also, this tool will be used to generate a set of test data to cover the selected spanning sets using genetic algorithms. Experiments will be conducted using this tool to compare the proposed technique with conventional techniques for automatic test data generation.

REFERENCES

[1] Mills, H. D.—Dyer, M. D.—Linger, R. C.: Cleanroom Software Engineering. IEEE Software, Vol. 4, 1987, No. 5, pp. 19–25.

[2] Voas, J. M.—Morell, L.—Miller, K. W.: Predicting Where Faults Can Hide from Testing. IEEE, Vol. 8, 1991, No. 2, pp. 41–48.

[3] Thévenod-Fosse, P.—Waeselynck, H.: STATEMATE: Applied to Statistical Software Testing. ACM SIGSOFT Proceedings of the 1993 International Symposium on Software Testing and Analysis, Software Engineering Notes, June 1993 pp. 78–81.

[4] Clarke, L. A.: A System to Generate Test Data and Symbolically Execute Programs. IEEE Transactions on Software Engineering, Vol. 2, 1976, No. 3, pp. 215–222.

[5] King, J. C.: Symbolic Execution and Program Testing. Communications of the ACM, Vol. 19, 1976, No. 7, pp. 385–394.

[6] Howden, W. E.: Symbolic Testing and the DISSECT Symbolic Evaluation System. IEEE Transactions on Software Engineering, Vol. 3, 1977, No. 4, pp. 266–278.

[7] Lindquist, T. E.—Jenkins, J. R.: Test-Case Generation with IOGen. IEEE Software, Vol. 5, 1988, No. 1, pp. 72–79.

[8] Girgis, M. R.: Using Symbolic Execution and Data Flow Criteria to Aid Test Data Selection. The Journal of Software Testing, Verification and Reliability, Vol. 3, 1993, No. 2, pp. 101–112.

[9] Korel, B.: Automated Software Test Data Generation. IEEE Transactions on Software Engineering, Vol. 16, 1990, No. 8, pp. 870–879.

[10] Ferguson, R.—Korel, B.: The Chaining Approach for Software Test Data Generation. ACM TOSEM, Vol. 5, 1996, No. 1, pp. 63–86.

[11] Roper, M.—Maclean, I.—Brooks, A.—Miller, J.—Wood, M.: Genetic Algorithms and the Automatic Generation of Test Data. Technical Report RR/95/195[EFoCS-19-95].

[12] Pargas, R. P.—Harrold, M. J.—Peck, R. R.: Test Data Generation Using Genetic Algorithms. Journal of Software Testing, Verifications and Reliability, Vol. 9, 1999, pp. 263–282.

[13] Lin, J.—Yeh, P.: Automatic Test Data Generation for Path Testing Using GAs. Journal of Information Science, Vol. 131, 2001, pp. 47–64.

[14] Michael, C. C.—McGraw, G. E. Schatz, M. A.: Generating Software Test Data by Evolution. IEEE Transactions on Software Engineering, Vol. 27, 2001, No. 12, pp. 1085–1110.

[15] Bueno, P. M. S.— Jino, M.: Automatic Test Data Generation for Program Paths Using Genetic Algorithms. International Journal of Software Engineering and Knowledge Engineering, Vol. 12, 2002, No. 6, pp. 691–709.

[16] Girgis, M. R.: Automatic Test Data Generation for Data Flow Testing Using Genetic Algorithm. Journal of Universal Computer Science, Vol. 11, 2005, No. 5, pp. 898–915.

[17] Marré, M.—Bertolino, A.: Using Spanning Sets for Coverage Testing. IEEE Transactions on Software Engineering, Vol. 29, 2003, No. 11, pp. 974–984.

[18] Bertolino, A.—Marré, M.: Automatic Generation of Path Covers Based on the Control Flow Analysis of Computer Programs. IEEE Transactions on Software Engineering, Vol. 20, 1994, No. 12, pp. 885–899.

[19] Wong, W. E.: On Mutation and Data Flow. Ph. D. thesis, SERC-TR-149-P, Software Engineering Research Center, Purdue University, December 1993.

[20] Beizer, B.: Software Testing Techniques. Second Edition, van Nostrand Reinhold, New York, 1990.

[21] Rapps, S.—Weyuker, E. J.: Selecting Software Test Data Using Data Flow Information. IEEE Transactions on Software Engineering, Vol. 11, 1985, No. 4, pp. 367–375.

[22] Frankl, P. G.— Weyuker, E. J.: An Applicable Family of Data Flow Testing Criteria. IEEE Transactions on Software Engineering, Vol. 14, 1988, No. 10, pp. 1483–1498.

Abdelaziz M. Khamis is a professor of computer science; currently, the works at Ajman University of Science and Technology, UAE. He received a Ph. D. in computer science from Essex University, UK. His research interests include software engineering, formal methods, object-oriented design and programming, object-oriented application frameworks, aspect-oriented programming, and software testing. His qualifications to conduct research in these areas are founded on over 25 years of research experience in academia (1982–2007).

Moheb R. Girgis is a member of the IEEE Computer Society. He received the B. Sc. degree from Mansoura University, Egypt, in 1974, the M. Sc. degree from Assuit University, Egypt, in 1980, and the Ph. D. from the University of Liverpool, England, in 1986. He is an associate professor at Minia University, Egypt. His research interests include software engineering, information retrieval, genetic algorithms, and networks.

Ahmed S. Ghiduk is an assistant lecturer at Beni-Suef University, Egypt. He received the B. Sc. degree from Cairo University, Egypt, in 1994, and the M. Sc. degree from Minia University, Egypt, in 2001. Currently, he is a Ph. D. student at College of Computing, Georgia Institute of Technology, USA. His research interests include software engineering and genetic algorithms. His thesis research concerns software test data generation using genetic algorithm.