SOFTWARE TESTING, VERIFICATION AND RELIABILITY, VOL. 2, 203–213 (1992)

Dynamic Method for Software Test Data Generation

BOGDAN KOREL
Department of Computer Science, Wayne State University, Detroit, MI 48202, U.S.A.

SUMMARY

Test data generation in program testing is the process of identifying a set of test data which satisfies a given testing criterion. Existing pathwise test data generators proceed by selecting program paths that satisfy the selected criterion and then generating program inputs for these paths. One of the problems with this approach is that unfeasible paths are often selected; as a result, significant computational effort can be wasted in analysing those paths. In this paper, an approach to test data generation, referred to as a dynamic approach for test data generation, is presented. In this approach, the path selection stage is eliminated. Test data are derived based on the actual execution of the program under test and function minimization methods. The approach starts by executing a program for an arbitrary program input. During program execution for each executed branch, a search procedure decides whether the execution should continue through the current branch or an alternative branch should be taken. If an undesirable execution flow is observed at the current branch, then a real-valued function is associated with this branch, and function minimization search algorithms are used to locate values of input variables automatically, which will change the flow of execution at this branch.

KEY WORDS Software testing Test coverage tool Automated test data generation Path selection Function minimization Path unfeasibility

1. INTRODUCTION

Test data generation in software testing is the process of identifying program test data which satisfy selected testing criteria. This paper focuses on structural testing coverage criteria that require certain program elements, or combinations thereof, to be exercised, e.g. statement coverage, branch coverage (Huang, 1975), or data flow coverage (Laski and Korel, 1983; Rapps and Weyuker, 1985). Those testing criteria are usually supported by testing coverage tools that automatically identify program elements that have been covered during testing. After initial testing, programmers face the problem of finding test data to exercise program elements that are reported by a testing coverage tool as not covered, e.g. branches not yet covered. Derivation of test data to exercise those remaining elements requires a good understanding of the program under test from programmers and can be very labour-intensive and expensive. The process of test data generation can become even more aggravating in software maintenance, especially in regression testing. Here programmers usually modify ‘someone else’s’ programs, which are often poorly or

0960–0833/92/040203–11$10.50
© 1992 by John Wiley & Sons, Ltd.

Received 18th January 1993
Revised 27th April 1993

only partially understood. Several structural techniques for testing modified programs have been proposed in the literature (e.g. Harrold and Soffa, 1988; Leung and White, 1989). They usually require testing of the modified parts of programs or those parts which are affected by change. Finding test data which exercise modified elements of partially understood programs can be exceptionally difficult.

If the process of test data generation could be automated, the cost of software development and maintenance could be reduced significantly. A test data generator is a tool that assists a programmer in the generation of test data for a program. The test data generation problem is defined as follows: for a given program element, e.g. a statement, find a program input on which this element is executed. There are two types of test data generators that could be applied for this problem: random test data generators (Bird and Munoz, 1983) or pathwise test data generators (Bicevskis *et al.*, 1979; Boyer *et al.*, 1975; Clarke, 1976; Howden, 1977; Ramamoorthy *et al.*, 1976; Korel, 1990a). The existing pathwise test data generators reduce the problem of test data generation to a 'path' problem, i.e. they proceed by selecting a program path that will exercise the selected program element. Then, if possible, program input for that path is generated. If the program input is not found for the selected path, then the next path is selected. This process is repeated until: (1) a path is selected for which a program input on which this path is traversed is found; or (2) the designated resources have been exhausted, e.g. a time limit. Most of the pathwise test data generators (Bicevskis *et al.*, 1979; Boyer *et al.*, 1975; Clarke, 1976; Howden, 1977; Ramamoorthy *et al.*, 1976) use symbolic evaluation to derive input data. Symbolic evaluation involves executing a program using symbolic values of variables instead of actual values. Once a path is selected, symbolic evaluation is used to generate a path constraint, which consists of a set of equalities and inequalities on the program's input variables; this path constraint must be satisfied for the path to be traversed. Input data for the selected path are found by solving these equalities and inequalities. There are several difficulties which prevent this method from being used for 'real' programs (Ince, 1987; Muchnick and Jones, 1981; DeMillo *et al.*, 1987). The major difficulty is the inefficient handling of arrays, dynamic data structures (pointers), and procedures. An alternative approach of finding program input to traverse the selected path has been proposed by Korel (1990a). This approach is based on actual execution of a program under test, dynamic data flow analysis, and function minimization methods. Test data are developed using actual values of input variables. Since the approach is based on the actual program execution, values of array indexes and pointers are known at each step of program execution, and this approach exploits this information in order to overcome several limitations of the methods based on symbolic evaluation.

In the existing pathwise test data generators, program paths are usually selected automatically by the test data generator or are provided by the user. The major weakness (Ince, 1987; Muchnick and Jones, 1981; DeMillo *et al.*, 1987) of automatic path selection is path unfeasibility. Since the path selection is based purely on a control flow graph of the program, this method very often leads to the selection of unfeasible paths; in this case significant computational effort can be wasted in analysing those unfeasible paths. However, if users are to provide the paths to be traversed, this requires significant effort to select the 'feasible' program paths; as a result, the degree of automation of the test data generation process can be significantly degraded.

In this paper, an approach to automated test data generation (Korel, 1990b), referred to as a dynamic approach for test data generation, is presented. This approach attempts to break away from the conventional 'pathwise' approach to test data generation by eliminating the path selection stage. The approach starts by initially executing a program with arbitrary program input. When the program is executed, the program execution flow is monitored. During the program execution for each executed branch, a search procedure decides whether the execution should continue through the current branch or an alternative branch should be taken because, for instance, the current branch does not lead to the execution of the selected program element, e.g. a statement. If an undesirable execution flow at the current branch is observed, then a real-valued function is associated with this branch. Function minimization search algorithms are used to find automatically new input data that will change the flow of execution at this branch.

The organization of this paper is as follows: in the next section, basic concepts and notation are introduced. Section 3 presents the dynamic approach for test data generation. A method for solving subgoals is presented in Section 4. In Section 5, the concluding section, future research is discussed.

2. BASIC CONCEPTS

A *flow graph* for program Q is a directed graph $C = (N, A, s, e)$ where (1) N is a set of nodes, (2) A is a binary relation on N (a subset of $N \times N$), referred to as a set of edges, and (3) s and e are, respectively, unique entry and unique exit nodes; $s, e \in N$. For the sake of simplicity, the analysis is restricted to a subset of structured Pascal-like programming language constructs, namely: **sequencing**, **if-then-else** and **while** statements. A node in N corresponds to the smallest single-entry, single-exit executable part of a statement in Q that cannot be further decomposed; such a part is referred to as a *node*. A single node corresponds to an assignment statement, an input or output statement, or the ⟨expression⟩ part of an **if-then-else** or **while** statement, in which case it is called a *test node*. An *edge* $(n_i, n_j) \in A$ corresponds to a possible transfer of control from node n_i to node n_j. For instance, (2, 3), (7, 8), and (7, 9) are edges in the program of Figure 1. An

```
var
A: array [1..100] of integer;
n,min,max: integer;
c,i: integer;

          begin
1           input (n,A) ;
2           min := A[1] ;
3           max := A[1] ;
4           c := 0;
5           i := 2 ;
6           while i <= n do
              begin
7,8             if max < A[i] then max := A[i] ;
9               if min >= A[i] then
                  begin
10,11               if min = A[i] then c :=c + 1
12                  else min := A[i];
                  end;
13              i := i + 1 ;
              end ;
14          output (max,min,c) ;
          end;
```

Figure 1. A sample program

edge (n_i, n_j) is called a *branch* if n_i is a test node. Each branch in the control flow graph can be labelled by a predicate, referred to as a *branch predicate*, describing the conditions under which the branch will be traversed. For example, in the program of Figure 1, branch (6, 7) is labelled '$i \leq n$', branch (7, 8) is labelled 'max $< A[i]$', and branch (7, 9) is labelled 'max $\geq A[i]$'.

An input variable of a program Q is a variable that appears in an input statement, e.g. read(x), or it is an input parameter of a procedure. Input variables may be of different types, e.g. integer, real, boolean, etc. Let $I = (x_1, x_2, \ldots, x_n)$ be a vector of input variables of program Q. The domain D_{x_i} of input variable x_i is a set of all values which x_i can hold. The *domain* D of the program Q is a cross product, $D = D_{x_1} \times D_{x_2} \times \ldots \times D_{x_n}$, where each D_{x_i} is the domain for input variable x_i. A single pont $\mathbf{x}$ in the n-dimensional input space D, $\mathbf{x} \in D$, is referred to as a *program input*.

A *path* P in a control flow graph is a sequence $P = <n_{k_1}, n_{k_2}, \ldots, n_{k_q}>$ of nodes, such that $n_{k_1} = s$, and for all i, $1 \leq i < q$, $(n_{k_i}, n_{k_{i+1}}) \in A$. A path is *feasible* if there exists a program input $\mathbf{x}$ for which the path is traversed during program execution, otherwise the path is *unfeasible*.

3. DESCRIPTION OF THE DYNAMIC APPROACH

The dynamic approach of test data generation is presented for the node problem; however, it can be extended to other types of problems, i.e. branch problem, definition-use chain problem, etc. The node problem is stated as follows:

> Given node Y in a program. The goal is to find a program input $\mathbf{x}$ on which node Y will be executed.

Observe that in the branch problem one would search for a program input $\mathbf{x}$, on which a selected branch would be executed. The pathwise methods of test data generation reduce the node problem to a 'path' problem, i.e. they proceed by selecting a program path from the entry node to node Y, and then finding, if possible, a program input for that path. As already stated, the major weakness of automatic path selection is the likely unfeasibility of a chosen path. The pathwise approach is probably best suited for programs with a relatively small number of paths to reach the selected node. However, for programs with complex control structures and usually an infinite number of paths to reach the selected node, the pathwise approach becomes inefficient.

The dynamic approach of test data generation (Korel, 1990b) differs from existing techniques in that the path selection stage is eliminated. The dynamic approach starts by initially executing a program for an arbitrary program input. During the program execution for each executed branch, a search procedure decides whether the execution should continue through this branch or an alternative branch should be taken (because, for instance, the current branch does not lead to the selected node). In the latter case, a new program input must be found to change the flow of execution at the current branch. For example, suppose that the goal is to find a program input to execute node 20 in a program that is represented by the control flow graph of Figure 2, and that this program is initially executed on an arbitrary program input on which the following path is traversed: $\langle s, 1, 2, 3, 4, 6, 7, 2, 8, 9, 25 \rangle$. It is obvious that when branch (9, 25) is executed, the program execution should be terminated at test node 9, and a new program input $\mathbf{x}$ must be found to change the flow of execution at this node to execute branch (9, 10).

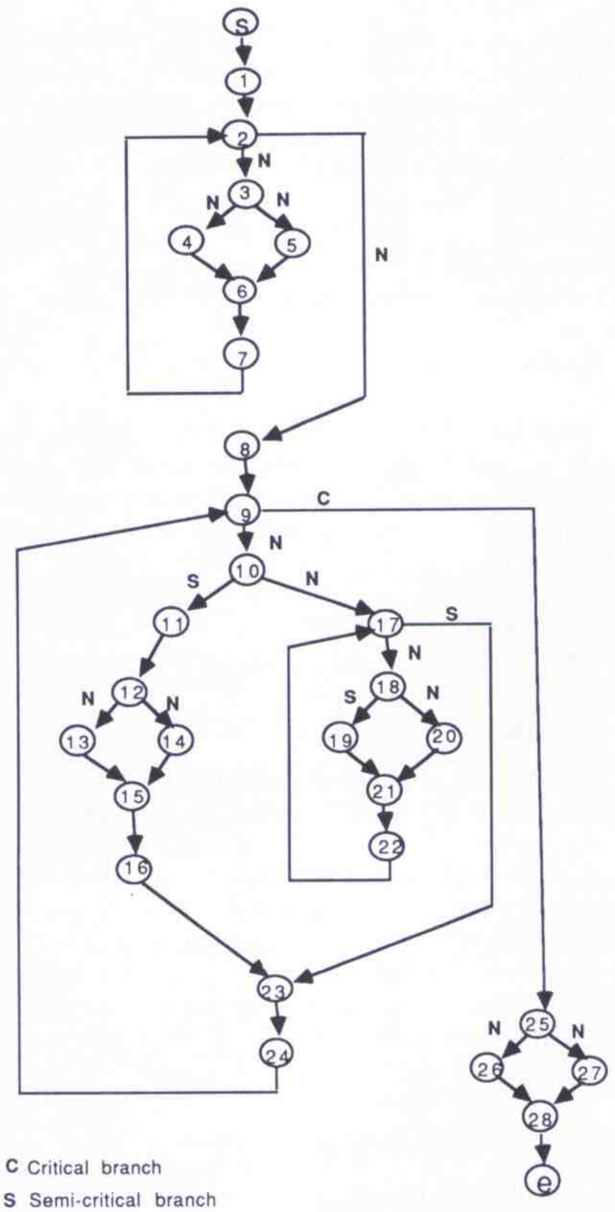

Figure 2. A control flow graph with branch classification with respect to node 20

The general idea of the dynamic approach is to concentrate on 'essential' branches, i.e. branches which 'influence' the execution of the selected node Y, and to ignore 'non-essential' branches, i.e. branches which in no way influence the execution of this node. For instance, branch (2, 3) in Figure 2 does not influence the execution of node 20 because, assuming termination of loop 2–7, branch (2, 8) will always be executed. For this purpose, every branch in a program is classified into three categories. The branch classification proposed in this paper is based solely on flow graph information, and it is determined prior to the program execution. The search procedure uses this classification during program execution to continue or to suspend the execution at the current branch. In the latter case, the search procedure determines a new program input.

Before classifying branches, the scope of control influence for **if** and **while**-statements is defined:

(a) **if Z then** B_1 **else** B_2
    X is in the scope of control influence of Z iff X appears in B_1 or B_2.
(b) **while** Z **do** B**:**
    X is in the scope of control influence of Z iff X appears in B.

The scope of control influence captures the influence between test nodes and nodes that can be chosen to be executed or not to be executed by these test nodes. For instance, test node 10 in Figure 2 has influence on the execution of node 20, but it has no influence on the execution of node 24. As a result, node 20 is in the scope of control influence of node 10, but node 24 is not.

3.1. Branch classification

3.1.1. A critical branch

A branch (n_i, n_j) is called a *critical branch* with respect to node Y iff (1) Y is in the scope of control influence of n_i, and (2) there is no path from n_i to Y through branch (n_i, n_j).

If a critical branch is executed, the program execution is terminated because its execution cannot lead to the selected node Y, and the search procedure has to find a program input $\mathbf{x}$ that will change the flow of execution at this point; as a result, an alternative branch will be executed. For instance, branch (9, 25) in Figure 2 is a critical branch with respect to node 20 because once this branch is executed there is no way to reach node 20. In this case, a new program input must be found on which alternative branch (9, 10) will be taken.

3.1.2. A semi-critical branch

A branch (n_i, n_j) is called a *semi-critical branch* with respect to node Y iff (1) Y is in the scope of control influence of n_i, and (2) there is no acyclic path from n_i to Y through branch (n_i, n_j).

If a semi-critical branch is executed, the program execution is terminated and the search procedure tries to find a new program input to change the flow at this branch. For instance, branch (10, 11) is a semi-critical branch with respect to node 20 because to reach this node the program has to iterate loop 9–24 and reach again node 10. It is obvious that the alternative branch (10, 17) is more 'promising' at this point. If the search for a new program input to execute branch (10, 17) is unsuccessful the search procedure allows the program execution to continue through branch (10, 11), hoping that on the next execution of test node 10 branch (10, 17) will be taken.

3.1.3. A non-essential branch

A branch (n_i, n_j) is called a *non-essential branch* with respect to node Y iff it is not a critical or a semi-critical branch.

If a non-essential branch is executed, the program execution is allowed to continue through this branch. For example, branches (3, 4) and (3, 5) are non-essential branches with respect to node 20 because they do not influence the execution of this node. Consequently, the execution can continue through either of them.

The following is the classification of branches with respect to node 20 for the flow graph of Figure 2:

critical branches: (9, 25);

semi-critical branches: (10, 11), (17, 23), (18, 19);

all the remaining branches are classified as non-essential.

Summarizing: if, during program execution, the current branch is a non-essential branch, the program execution is continued through this branch. If the current branch is a critical or semi-critical branch, the program execution is terminated and the search for a new program input, which will change the flow of execution at the current branch, is performed. If this search process of finding a new program input fails, then (a) the program execution is continued through this branch in the case of the semi-critical branch, or (b) the whole search process is terminated with a failure in the case of the critical branch.

Example 1

Consider the control flow graph of Figure 2. The goal is to find a program input $\boldsymbol{x}$ on which node 20 will be executed. For this purpose, the program is executed on an initial program input $\boldsymbol{x}^0$; suppose that the following path is traversed on this input: $\langle s, 1, 2, 3, 4, 6, 7, 2, 8, 9, 25\rangle$. All branches on this path are non-essential except for branch (9, 25), which is a critical branch. The search procedure terminates the execution at this point and tries to solve the first subgoal, i.e. to find a program input on which branch (9, 10) will be traversed. Let $\boldsymbol{x}^1$ be the solution to the first subgoal, and suppose that the following path is traversed: $\langle s, 1, 2, 3, 4, 6, 7, 2, 8, 9, 10, 11\rangle$. The last branch (10, 11) in this path is semi-critical; the search procedure has to solve the second subgoal, i.e. to find a program input on which branch (10, 17) will be taken rather than (10, 11). Suppose that the search fails to solve the second subgoal; in this case, the search procedure continues the program execution through branch (10, 11) and the following path is traversed: $\langle s, 1, 2, 3, 4, 6, 7, 2, 8, 9, 10, 11, 12, 14, 15, 16, 23, 24, 9, 10, 11\rangle$. The search procedure has to solve the third subgoal, i.e. to find a program input $\boldsymbol{x}$ on which branch (10, 17) will be executed on the second iteration of loop 9–24. Let $\boldsymbol{x}^3$ be the solution to the third subgoal, and suppose that the following path is traversed: $\langle s, 1, 2, 3, 4, 6, 7, 2, 8, 9, 10, 11, 12, 14, 15, 16, 23, 24, 9, 10, 17, 23\rangle$. Since branch (17, 23) is semi-critical, the next subgoal is to find a program input on which branch (17, 18) will be taken. This process of solving subgoals is repeated until the solution to the main goal is found, i.e. until node 20 is executed, or no progress can be made (as a result, the search procedure fails to find the solution).

4. SOLVING SUBGOALS

In the dynamic approach, the problem of finding inupt data is reduced to a sequence of subgoals where each subgoal is solved using function minimization search techniques that use branch predicates to guide the search process. Without loss of generality, it is assumed that the branch predicates are simple relational expressions (inequalities and equalities). That is, all branch predicates are of the following form: E_1 *op* E_2, where E_1 and E_2 are arithmetic expressions, and '*op*' is one of $\{<,\leq,>,\geq,=,\neq\}$. Each branch predicate E_1 *op* E_2 can be transformed to the equivalent predicate of the form: F *rel* 0, where F and '*rel*' are given in Table I.

F is a real-valued function, referred to as a *branch function*, which is (1) positive (or zero if *rel* is '$<$') when a branch predicate is false or (2) negative (or zero if *rel* is '$=$' or '$\leq$') when the branch predicate is true. It is obvious that F is actually a function of program input $\mathbf{x}$. The branch function can be evaluated for any input data by executing the program. For instance, the true branch of a test 'if $y>z$ then...' has a branch function F, whose value can be determined for a given input by executing the program and evaluating the z–y expression.

Let $\mathbf{x}^0$ be an initial program input on which the program is executed. If node Y is executed, $\mathbf{x}^0$ is the solution to the test data generation problem; if not, the first subgoal must be solved. Let $\langle n_{p_1}, n_{p_i}, \ldots, n_{p_{i+1}}\rangle$ be a program path traversed on input $\mathbf{x}^0$, such that for all j, $1 \leq j < \mathrm{i}$, if n_{p_j} is a test node $(n_{p_j}, n_{p_{j+1}})$ is a non-essential branch, and $(n_{p_i}, n_{p_{i+1}})$ is a critical or semi-critical branch. Let (n_{p_i}, n_{p_k}) be an alternative branch of test node n_{p_i} and $F_i(\mathbf{x})$ be a branch function of branch (n_{p_i}, n_{p_k}).

The goal, now, is to find a value of $\mathbf{x}$ that will preserve the traversal of path $\langle n_{p_1}, n_{p_2}, \ldots, n_{p_i}\rangle$, referred to as a *constraint path*, and cause $F_i(\mathbf{x})$ to be negative (or zero) at n_{p_i}; as a result, alternative branch (n_{p_i}, n_{p_k}) will be executed. More formally, the goal is to find a program input $\mathbf{x} \in D$ satisfying: $F_i(\mathbf{x})$ rel_i 0, subject to the constraint: $\langle n_{p_1}, n_{p_2}, \ldots, n_{p_i}\rangle$ is traversed on $\mathbf{x}$, where rel_i is one of $\{<, \leq, =\}$.

This problem is similar to the minimization problem with constraints because the function $F_i(\mathbf{x})$ can be minimized using numerical techniques for constrained minimization, stopping when $F_i(\mathbf{x})$ becomes negative (or zero, depending on rel_i). Because of a lack of assumptions about the branch function and constraints, direct-search methods (Gill and Murray, 1974; Glass and Cooper, 1965) are used. The direct-search method progresses towards the minimum using a strategy based on the comparison of branch function values only. The simplest strategy of this form is that known as the *alternating variable method*,

Table I.

Branch predicate	Branch function F	*rel*
$E_1 > E_2$	$E_2 - E_1$	$<$
$E_1 \geq E_2$	$E_2 - E_1$	$\leq$
$E_1 < E_2$	$E_1 - E_2$	$<$
$E_1 \leq E_2$	$E_1 - E_2$	$\leq$
$E_1 = E_2$	abs $(E_1 - E_2)$	$=$
$E_1 \neq E_2$	$-$abs $(E_1 - E_2)$	$<$

which consists of minimizing the current branch function with respect to each input variable in turn. The search proceeds in this manner until all input variables are explored. After completing such a cycle, the procedure continuously cycles around the input variables until the solution is found or no progress (decrement of the branch function) can be made for any input variable. In the latter case, the search process fails to find the solution.

During experimentation with the dynamic approach, it has been observed that the requirement of the exact traversal of the constraint path may often be too rigid. Notice that in order to verify whether or not the constraint is violated for some input $\boldsymbol{x}$, it is enough to check whether the constraint path is traversed during program execution. If the constraint path is not traversed, then the constraint violation is reported to the search procedure. In some cases, this requirement may prevent efficient solving of subgoals because many unnecessary constraints (conditions) are imposed on the search. This situation is described in Example 2.

Example 2

Consider the program of Figure 1. The goal is to find a program input $\boldsymbol{x}$ (i.e. values for input variables: n, $A[1]$, $A[2]$,..., $A[100]$) on which node 12 will be executed. For this purpose, the program is executed on the following initial program input: $n=14$, $A[1]=1$, $A[2]=2$,..., $A[100]=100$, and the following path is traversed: $\langle s, 1, 2, 3, 4, 5, 6, 7, 8, 9, 13\rangle$. Branch (9, 13) is a semi-critical branch with respect to node 12. Let $F_1(\boldsymbol{x})$ be a branch function of branch (9, 10) where values of $F_1(\boldsymbol{x})$ can be evaluated by computing the $A[i]$-*min* expression at node 9. The first subgoal is to find a program input $\boldsymbol{x}$ such that $F(\boldsymbol{x}) \leq 0$, subject to the constraint: path $\langle s, 1, 2, 3, 4, 5, 6, 7, 8, 9\rangle$ is traversed on $\boldsymbol{x}$. It should be clear that this subgoal cannot be solved under the current constraint, and the search procedure continues the program execution through branch (9, 13). In this case, loop 6–13 iterates and the following path is traversed: $\langle s, 1, 2, 3, 4, 5, 6, 7, 8, 9, 13, 6, 7, 8, 9, 13\rangle$. An attempt to solve the next subgoal to execute branch (9, 10) fails again because there is no solution for this subgoal under the current constraint. The major reason for the failure of the search is the constraint path; in particular, execution of branch (7, 8) on this path. This prevents the execution of branch (9, 10). Branches (7, 8) and (7, 9) are non-essential branches with respect to node 12. It should be clear that branch (7, 9) instead of branch (7, 8) should be allowed to be executed.

For this reason, the search procedure has been modified by relaxing the constraint of traversal of a constraint path. Let $\langle n_{p_1}, n_{p2},\ldots, n_{p_i}\rangle$ be a constraint path. If, during solving a subgoal, constraint violations do not allow a solution to a subgoal to be found, the 'constraint path' requirement is relaxed: any path from the entry node to node n_{p_i} is allowed (i.e. constraint violations occur only on critical branches). Clearly, if during program execution a critical branch is executed before reaching node n_{p_i}, the execution is suspended, and the constraint violation is reported to the search procedure. This modification often leads to a more efficient search. For instance, in Example 2 when the constraint is relaxed, the solution to the test data generation problem is found on the first iteration of the loop 6–13.

For many programs, the evaluation of the branch function is the most time consuming portion of the search. By considering only those input variables which have influence on the current branch function, the possibility of a fruitless search can be significantly reduced. Dynamic data flow analysis (Korel, 1990a) can be applied to determine those input variables that are responsible for the current value of the branch function on a given program input. This is essential when programs with a large number of input variables are considered, e.g. programs with large input arrays (more detailed description of the dynamic data flow oriented search can be found in the work of Korel, 1990a). Experiments have shown that the dynamic data flow oriented search can significantly reduce the search time.

5. CONCLUSIONS

In this paper, a dynamic approach to test data generation has been presented. This approach differs from existing techniques in that the path selection stage is eliminated. Test data are derived based on the actual execution of the program under test and function minimization methods. It has been shown that the test data generation problem can be reduced to a sequence of subgoals that are solved using function minimization methods. The approach described in this paper has been implemented as an extension of the existing experimental test data generation tool TESTGEN (Korel, 1989). Initial experiments have shown that the dynamic approach efficiently generates test data for many programs. However, more research has to be done in order to understand better the advantages and limitations of this approach. The main extensions should go into the generality and robustness of this approach for use in the real world. To be of practical value, the method has to be extended to arbitrary programs, e.g. programs with procedure calls. In addition, the branch classification presented in this paper is based solely on control flow graph information. Data flow analysis may significantly enhance this classification, which should lead to more efficient test data generation.

References

Bicevskis, J., Borzovs, J., Straujumus, U., Zarins, A. and Miller, E. (1979) 'SMOTL—a system to construct samples for data processing program debugging', *IEEE Transactions on Software Engineering*, **5** (1), 60–66.

Bird, D. and Munoz, C. (1983) 'Automatic generation of random self-checking test cases', *IBM Systems Journal*, **22** (3), 229–245.

Boyer, R., Elspas, B. and Levitt, K. (1975) 'SELECT—a formal system for testing and debugging programs by symbolic execution', *SIGPLAN Notices*, **10** (6), 234–245.

Clarke, L. (1976) 'A system to generate test data and symbolically execute programs', *IEEE Transactions on Software Engineering*, **2** (3), 215–222.

DeMillo, R., McCracken, W., Martin, R. and Pasafiume, J. (1987) *Software Testing and Evaluation*, Benjamin Cummings.

Gill, P. and Murray, W. (1974) *Numerical Methods for Constrained Optimization*, Academic Press, New York.

Glass, H. and Cooper, J. (1965) 'Sequential search: a method for solving constrained optimization problems', *Journal of the ACM*, **12** (1), 71–82.

Harrold, M. J. and Soffa, M. L. (1988) 'An incremental approach to unit testing during maintenance', *Proceedings of the Conference on Software Maintenance*, pp.362–367.

Howden, W. (1977) 'Symbolic testing and the DISSECT symbolic evaluation system', *IEEE Transactions on Software Engineering*, **4** (4), 266–278.

Huang, J. (1975) 'An approach to program testing', *ACM Computing Surveys*, **7** (3), 113–128.

Ince, D. (1987) 'The automatic generation of test data', *The Computer Journal*, **30** (1), 63–69.

Korel, B. (1989) 'TESTGEN—A structural test data generation system', Technical Report CSC-89-001, Department of Computer Science, Wayne State University.

Korel, B. (1990a) 'Automated software test data generation', *IEEE Transactions on Software Engineering*, **16** (8), 870–879.

Korel, B. (1990b) 'A dynamic approach of automated test data generation', *Proceedings of the Conference on Software Maintenance*, San Diego, California, pp.311–317.

Laski, J. and Korel, B. (1983) 'A data flow oriented program testing strategy', *IEEE Transactions on Software Engineering*, **9** (3), 347–354.

Leung, H. and White, L. (1989) 'Insights into regression testing', *Proceedings of the Conference on Software Maintenance*, pp.60–69.

Muchnick, S. and Jones, N. (1981) *Program Flow Analysis: Theory and Applications*, Prentice Hall International.

Ramamoorthy, C., Ho, S. and Chen, W. (1976) 'On the automated generation of program test data', *IEEE Transactions on Software Engineering*, **2** (4), 293–300.

Rapps, S. and Weyuker, E. (1985) 'Selecting software test data using data flow information', *IEEE Transactions on Software Engineering*, **11** (4), 367–375.