Automated Software Test Data Generation

BOGDAN KOREL, MEMBER, IEEE

Abstract—**Test data generation in program testing is the process of identifying a set of test data which satisfies given testing criterion. Most of the existing test data generators [6], [8], [10], [16], [30] use symbolic evaluation to derive test data. However, in practical programs this technique frequently requires complex algebraic manipulations, especially in the presence of arrays. In this paper we present an alternative approach of test data generation which is based on actual execution of the program under test, function minimization methods, and dynamic data flow analysis. Test data are developed for the program using actual values of input variables. When the program is executed, the program execution flow is monitored. If during program execution an undesirable execution flow is observed (e.g., the "actual" path does not correspond to the selected control path) then function minimization search algorithms are used to automatically locate the values of input variables for which the selected path is traversed. In addition, dynamic data flow analysis is used to determine those input variables responsible for the undesirable program behavior, leading to significant speedup of the search process. The approach of generating test data is then extended to programs with dynamic data structures, and a search method based on dynamic data flow analysis and backtracking is presented. In the approach described in this paper, values of array indexes and pointers are known at each step of program execution, and this approach exploits this information to overcome difficulties of array and pointer handling; as a result, the effectiveness of test data generation can be significantly improved.**

Index Terms—**Automated test generation, dynamic data flow analysis, function minimization, software testing, symbolic evaluation.**

I. Introduction

SOFTWARE testing is very labor-intensive and expensive; it accounts for approximately 50% of the cost of a software system development [1], [28]. If the testing process could be automated, the cost of developing software should be reduced significantly. Of the problems involved in testing software, one is of particular relevance here: the problem of developing test data. Test data generation in software testing is the process of identifying program input data which satisfy selected testing criterion. A test data generator is a tool which assists a programmer in the generation of test data for a program. There are three types of test data generators: pathwise test data generators [6], [8], [10], [16], [30], data specification generators [3], [19], [24], [25], and random test data generators [7]. This paper focuses on pathwise test data generators which are tools that accept as input a computer program and a testing criterion (e.g., total path coverage, statement coverage, branch coverage, etc.) and then automatically generate test data that meet the selected criterion. The basic operation of the pathwise generator consists of the following steps: program control flow graph construction, path selection, and test data generation. The path selector automatically identifies a set of paths (e.g., near-minimal set of paths) to satisfy selected testing criterion. Once a set of test paths is determined, then for every path in this set the test generator derives input data that results in the execution of the selected path.

Most of the pathwise test data generators [6], [8], [10], [16], [30] 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. A number of algorithms have been used for the inequality solution. As pointed out in [18], [27], symbolic evaluation is a promising approach; however, there are still several problems which require additional research, e.g., the problem of array element determination. This problem occurs when the index of an array depends on input values; in this case, the array element that is being referenced or defined is unknown. This problem occurs frequently during symbolic evaluation. Inefficient solutions exist, for in the worst case all possible index values can be enumerated. Though there has been some work on this problem [30] and a related problem for record structures [27], the results are still unsatisfactory.

In this paper we present an alternative approach of test data generation, referred to as a dynamic approach of test data generation, which 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. When the program is executed on some input data, the program execution flow is monitored. If, during program execution, an undesirable execution flow at some branch is observed then a real-valued function is associated with this branch. This function is positive when a branch predicate is false and negative when the branch predicate is true. Function minimization search algorithms are used to automatically locate values of input variables for which the function becomes negative. In addition, dynamic data flow analysis is used to determine input variables which are responsible for the undesirable program behavior, leading to significant speed-up of the search process. In this approach, arrays and dynamic data structures can be handled precisely

Manuscript received December 19, 1988; revised March 24, 1990. Recommended by W. Howden.

The author is with the Department of Computer Science, Wayne State University, Detroit, MI 48202.

IEEE Log Number 9036267.

0098-5589/90/0800-0870$01.00 © 1990 IEEE

because during program execution all variables values, including array indexes and pointers, are known; as a result, the effectiveness of the process of test data generation can be significantly improved.

The organization of this paper is as follows. In the next section basic concepts and notations are introduced. Section III shows that the test data generation problem can be reduced to the sequence of subgoals where each subgoal can be solved using function minimization search techniques. The basic search procedure for solving subgoals is presented in Section IV. Dynamic data flow concepts and their application in the test data generation process are discussed in Section V. In Section VI, the test data generation is then extended onto programs with dynamic data structures. Finally, in the Conclusions further research is outlined.

II. Basic Concepts

A *flow graph* of 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, we restrict our analysis 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-exist executable part of a statement in Q that cannot be further decomposed; such a part is referred to as an instruction. A single instruction 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 instruction.

An *edge* $(n_i, n_j) \in A$ corresponds to a possible transfer of control from instruction n_i to instruction n_j. For instance, (2, 3), (6, 7), and (6, 8) are edges in the program of Fig. 1. An edge (n_i, n_j) is called a *branch* if n_i is a test instruction. Each branch in the control flow graph can be labeled 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 Fig. 1 branch (5, 6) is labeled "$i <$ high," branch (6, 7) is labeled "max $< A[i]$," and branch (6, 8) is labeled "max $\geq A[i]$."

An input variable of a program Q is a variable which 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 $I = (x_1, x_2, \cdots, x_n)$ 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. By the *domain D* of the program Q we mean a cross product, $D = D_{x_1} \times D_{x_2} \times \cdots \times D_{x_n}$, where each D_{x_i} is the domain for input variable x_i. A single point $\boldsymbol{x}$ in the n-dimensional input space D, $\boldsymbol{x} \in D$, is referred to as a *program input*.

A *path P* in a control flow graph is a sequence $P = <n_{k_1}, n_{k_2}, \cdots, n_{k_q}>$ of instructions, 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

```
      var
      A: array[1..100] of integer;
      low,high,step: integer;
      min,max: integer;
      i: integer;

      begin
1     input (low,high,step,A);
2     min := A[low];
3     max := A[low];
4     i := low + step;
5     while i < high do
          begin
6,7           if max < A[i] then max := A[i];
8,9           if min > A[i] then min := A[i];
10            i := i + step;
          end;
11    output (min,max);
      end;
```

Fig. 1. A sample program.

if there exists a program input $\boldsymbol{x}$ for which the path is traversed during program execution, otherwise the path is infeasible.

III. A Test Data Generation Problem

Let $P = <n_{k_1}, n_{k_2}, \cdots, n_{k_q}>$ be a path in the program. The goal of the test data generation problem is to find a program input $\boldsymbol{x} \in D$ on which P will be traversed. We shall show that this problem can be reduced to a sequence of subgoals where each subgoal is solved using function minimization search techniques.

Without loss of generality, we assume that the branch predicates are simple relational expressions (inequalities and equalities). That is, all branch predicates are of the following form:

$$E_1 \text{ op } E_2$$

where E_1 and E_2 are arithmetic expressions, and op is one of $\{<, \leq, >, \geq, =, \neq\}$. In addition, it is assumed that predicates do not contain AND's, OR's or other boolean operators.

Each branch predicate E_1 op E_2 can be transformed to the equivalent predicate of the form

$$F \text{ rel } 0$$

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 $\boldsymbol{x}$. Symbolic evaluation can be used to find explicit representation of the $F(\boldsymbol{x})$ in terms of the input variables. However, in practical programs this technique frequently requires complex algebraic manipulations, especially in the presence of arrays. For this reason, we shall consider the alternative approach in which the branch function is 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 computed for a given input by executing the program and evaluating the z-y expression.

TABLE I

Branch Predicate	Branch Function F	rel
$E_1 > E_2$	$E_2 - E_1$	$<$
$E_1 \geq E_2$	$E_2 - E_1$	$\leq$
$E_1 < E_2$	$E_1 - E_2$	$<$
$E_1 \leq E_2$	$E_1 - E_2$	$\leq$
$E_1 = E_2$	$\mathrm{abs}(E_1 - E_2)$	$=$
$E_1 \neq E_2$	$\mathrm{abs}(E_1 - E_2)$	$\leq$

Let $\boldsymbol{x}^0$ be the initial program input (selected randomly) on which the program is executed. If P is traversed, $\boldsymbol{x}^0$ is the solution to the test data generation problem; if not, we have to solve the first subgoal. Let $T = < t_{p_1}, t_{p_2}, \cdots, t_{p_z} >$ be a program path traversed on $\boldsymbol{x}^0$, and let $P_1 = < n_{k_1}, n_{k_2}, \cdots, n_{k_i} >$ be the longest subpath of P, referred to as a successful subpath of P on $\boldsymbol{x}^0$, such that for all j, $1 \leq j \leq i$, $n_{k_j} = t_{p_j}$. P_1 represents a successfully traversed part of P on input $\boldsymbol{x}^0$; the branch violation occurs on execution of branch $(n_{k_i}, n_{k_{i+1}})$. Let $F_i(\boldsymbol{x})$ be a branch function of branch $(n_{k_i}, n_{k_{i+1}})$. The first subgoal, now, is to find a value of $\boldsymbol{x}$ which will preserve the traversal of P_1 and cause $F_i(\boldsymbol{x})$ to be negative (or zero) at n_{k_i}; as a result, $(n_{k_i}, n_{k_{i+1}})$ will be successfully executed. More formally, we want to find a program input $\boldsymbol{x} \in D$ satisfying

$$F_i(\boldsymbol{x}) \ \mathrm{rel}_i \ 0$$

subject to the constraint:

$$P_1 \text{ is traversed on } \boldsymbol{x},$$

where rel_i is one of $\{<, \leq, =\}$.

This problem is similar to the minimization problem with constraints because the function $F_i(\boldsymbol{x})$ can be minimized using numerical techniques for constrained minimization [12], [13] until $F_i(\boldsymbol{x})$ becomes negative (or zero, depending on rel_i). The search procedure for solving subgoals is presented in Section IV.

Let $\boldsymbol{x}^1$ be the solution to the first subgoal. Now, either the selected path P is traversed (as a consequence, $\boldsymbol{x}^1$ is the solution to the main goal), or the second subgoal must be solved. In the latter case, let $P_2 = < n_{k_1}, n_{k_2}, \cdots, n_{k_i}, n_{k_{i+1}}, \cdots, n_{k_m} >$ be the successful subpath of P traversed on $\boldsymbol{x}^1$. Let $F_m(\boldsymbol{x})$ be the branch function of branch $(n_{k_m}, n_{k_{m+1}})$. The second subgoal is to find a program input $\boldsymbol{x}$ which satisfies $F_m(\boldsymbol{x})$ rel_m 0, subject to the constraint: P_2 is traversed on $\boldsymbol{x}$. This process of solving subgoals is repeated until the solution to the main goal is found, or one of the subgoals cannot be solved. In the latter case, the search procedure fails to solve the test data generation problem.

IV. Basic Search Procedure

We now turn our attention to the question of how to conduct a search to find the solution to a subgoal. Because of a lack of assumptions about the branch function and constraints, we have selected direct-search methods [12], [13], which progress towards the minimum using a strategy based on the comparison of branch function values only. The main advantage of these search methods is that they do not require regularity and continuity of the branch function and the existence of derivatives. The most simple strategy of this form is that known as the alternating variable method which consists of minimizing with respect to each input variable in turn. We start searching for a minimum with the first input variable x_1 (using a one-dimensional search procedure) while keeping all the other input variables constant until the solution is found (the branch function becomes negative) or the positive minimum of the branch function is located. In the latter case, the search continues from this minimum with the next input variable x_2. The search proceeds in this manner until all input variables $x_1, \cdots, x_n$ are explored in turn. 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 even if the positive minimum of the branch function is located.

Now we shall briefly describe the one-dimensional search procedure for solving, for instance, the first subgoal. The one-dimensional search procedure [13] consists of two major phases, an "exploratory search" and a "pattern search." In the exploratory search, the selected input variable x_j is increased and decreased by a small amount, while the remaining input variables are held constant. These are called the exploratory moves. For each variable change, the program is executed and the constraint is checked for possible violation by comparing successful subpath P_1 with the path which is actually being traversed. If P_1 has been traversed, branch function $F_i(\boldsymbol{x})$ is evaluated for the new input. On the other hand, if P_1 has not been traversed, the constraint violation is reported. In these exploratory moves, the value of the branch function is compared to the value of the branch function for the previous input. In this way, it is possible to indicate a direction in which to proceed, that is, to make a larger move. If the branch function is improved (decreased) when x_j is decreased, the search should proceed in the direction of decreasing x_j. If, on the other hand, the branch function is improved when x_j is increased, the search should proceed in the direction of increasing x_j. If both the decrement and the increment of x_j do not cause the improvement of the branch function, the exploratory search fails to determine the direction for the search; in this case, the next input variable is selected for consideration.

Assuming that the exploratory moves are able to indicate a direction in which to proceed, a larger move called a pattern move (pattern search) is made. After a pattern move, the program is executed and the constraint is checked for possible violation. If the constraint violation has not occurred and branch $(n_{k_i}, n_{k_{i+1}})$ has not been taken, the branch function is evaluated and its value is compared to the value of the branch function for the previous input. If branch function $F_i(x)$ is improved for the given step, then the new value of the branch function replaces the old one, and another larger move is made in the same direction. A series of pattern moves is made along this direction as long as the branch function is improved by each pattern move. The magnitude of the step for the pattern move is roughly proportional to the number of successful steps previously encountered. However, if the branch function is not improved, then the old value of the branch function is retained. If, after a succession of successful moves, a pattern move fails because the value of the branch function is not improved, then exploratory moves would be made to indicate a new direction. If a pattern move fails because of a constraint violation, the search would continue in this direction, with a reduction of the step size as necessary, until a successful move is made. At this point, we would again make exploratory moves to indicate a new direction.

This process continues until the branch function becomes negative (or zero) or no progress (decrement of the branch function) can be made for any input variable during the exploratory search. In the latter case, the search procedure fails to find the solution to the test data generation problem.

Example 1

We work through a simple example to illustrate the basic approach of test data generation. Consider the program of Fig. 1 which is supposed to determine minimum and maximum values for selected elements of array A (determined by input variables low, step, and high).

Given is the following path $P = < s, 1, 2, 3, 4, 5, 6, 8, 10, 5, 6, 8, 9, 10, 5, 11>$, the goal of the test data generation is to find a program input x (i.e., values for the input variables: low, high, step, $A[1]$, $A[2]$, $\cdots$, $A[100]$) which will cause P to be traversed. In the first step, all input variables receive initial values; suppose the following values have been assigned: low = 39, high = 93, step = 12, $A[1] = 1$, $A[2] = 2$, $\cdots$, $A[100] = 100$. The program is executed on this input, and the following successful subpath $P_1 = < s, 1, 2, 3, 4, 5, 6 >$ of P is traversed; the violation occurs at branch (6, 8). Let $F_1(x)$ be a branch function of branch (6, 8), where values of $F_1(x)$ can be evaluated by computing the "$A[i]$-max" expression at 6. Note that the branch predicate of branch (6, 8) is max $\geq A[i]$. Our first subgoal is to find a program input x such that $F_1(x) \leq 0$, subject to the constraint: P_1 is traversed on x.

Suppose that input variables are ordered in the following way: $A[1]$, $A[2]$, $\cdots$, $A[100]$, high, step, low. The direct search starts with input variable $A[1]$. In the exploratory move, $A[1]$ is increased by one, while the remaining variables are kept constant. It should be noted that because $A[1]$ is declared as integer, the minimal increment of $A[1]$ is one. The program is executed on this input and the value of $F_1(x)$ is evaluated at 6; the value of branch function $F_1(x)$ is unchanged. $A[1]$ is decreased by one, but the value of $F_1(x)$ is unchanged. The next input variable $A[2]$ is selected. The exploratory moves for $A[2]$, $A[3]$, $\cdots$, $A[38]$ do not cause any change in the value of $F_1(x)$. The increment of $A[39]$ by one causes the decrement in the value of $F_1(x)$. This step defines a promising direction for search. Suppose that the value of $A[39]$ is increased by 100 in the pattern move, i.e., $A[39] = 139$. On this input, subpath P_1 is traversed and branch (6, 8) is taken. As a result, this input is the solution to the first subgoal.

When the program is executed on this input, the following successful subpath $P_2 = < s, 1, 2, 3, 4, 5, 6, 8 >$ of P is traversed; the violation occurs at branch (8, 10). Let $F_2(x)$ be a branch function of branch (8, 10), where values of $F_2(x)$ can be evaluated by computing the "min-$A[i]$" expression at 8. The second subgoal is to find a program input x such that $F_2(x) \leq 0$, subject to the constraint: P_2 is traversed on x.

Suppose that input variables are ordered in the same manner as for the first subgoal. Exploratory moves for $A[1]$, $\cdots$, $A[38]$ do not cause any improvement in the value of branch function $F_2(x)$. The decrement of $A[39]$ by one causes the decrement in the value of $F_2(x)$. This step defines a direction for search. By performing the series of exploratory and pattern moves on $A[39]$ and reducing the size of the pattern move, the value of $A[39] = 51$ can be found. This input is the solution to the second subgoal, i.e., subpath P_2 is traversed and branch (8, 10) is taken.

When the program is executed on this input, the following successful subpath $P_3 = < s, 1, 2, 3, 4, 5, 6, 8, 10, 5, 6 >$ of P is traversed. The violation occurs on the second execution of branch (6, 8). Let $F_3(x)$ be a branch function of branch (6, 8), where values of $F_3(x)$ can be evaluated by the "$A[i]$-max" expression. The third subgoal is to find a program input x such that $F_3(x) \leq 0$, subject to the constraint: P_3 is traversed on x.

Exploratory moves for $A[1]$, $\cdots$, $A[62]$ do not cause any improvement in the value of branch function $F_3(x)$. Observe that the exploratory moves for $A[39]$ cause the constraint violation, i.e., P_3 is not traversed. Only the decrement of $A[63]$ causes the improvement of branch function $F_3(x)$. Suppose that the value of $A[63]$ is decreased by 100 in the pattern move, i.e., $A[63] = -37$. This move finds the solution to the third subgoal.

When the program is executed on this input, the following successful subpath $P_4 = < s, 1, 2, 3, 4, 5, 6, 8, 10, 5, 6, 8, 9, 10, 5 >$ of P is traversed. The violation occurs on the branch (5, 11). Let $F_4(x)$ be a branch function of branch (5, 11), where values of $F_4(x)$ can be evaluated by the "high-i" expression. The fourth subgoal is to find a program input x such that $F_4(x) \leq 0$, subject to the constraint: P_4 is traversed on x.

In the first step all elements of array A are tried out. However, none of the exploratory moves for array elements $A[1]$, $\cdots$, $A[100]$ cause an improvement in the value of branch function $F_4(x)$. Only the decrement of input variable high causes the improvement of branch function $F_4(x)$. This exploratory move defines a direction for the change of the variable high. The sequence of the pattern moves can determine the final value of high = 67. The solution, high = 67, $A[39] = 51$, $A[63] = -37$ (where the rest of the input variables have the initial values), to the fourth subgoal is also the solution to the test generation problem.

V. Dynamic Data Flow Based Search

In this section, we present a heuristic approach, based on dynamic data flow analysis, which can significantly speedup the search process presented in the previous section. It originated during experiments with a recently implemented prototype of the automated test data generation system TESTGEN [23].

For many programs the evaluation of the branch function is the most time-consuming portion of the search. It is felt that it would be more efficient to keep the number of evaluations to a minimum. One of the most essential factors deciding about the reduction of branch function evaluations is the arrangement of input variables for consideration. For instance, the arrangement of input variables in Example 1 is rather rigid and in a sense "blind." It does not give any preference to a variable that has a very good chance of moving toward the solution quickly. For example, when subgoal 4 in Example 1 is considered, selection of variable high, as the first variable, will quickly lead to the solution. However, the "blind" rule requires that all array elements be evaluated before variable high is considered. It should be obvious that a different arrangement of input variables for consideration can lead to the different performance of the search process. This is essential when we deal with programs with a large number of input variables, e.g., programs with large input arrays (hundreds or even thousands of array elements).

Thus, the important question arises as to which are the most promising input variables to explore. The approach which we propose in this paper is based on dynamic data flow analysis which allows determining those input variables which are responsible for the current value of the branch function on the given program input.

We now introduce the basic concepts of dynamic data flow analysis [21], [22], [2], that is, those concerned with data flow along the path which has been traversed during program execution. We shall later show how to use this information to guide the search process.

Let $T = <n_{k_1}, n_{k_2}, \cdots, n_{k_q}>$ be a path that is traversed on a program input $\boldsymbol{x}$. A use of variable v in T is an instruction n_{k_i} in which this variable is referenced. A definition of variable v in T is an instruction n_{k_i} which assigns a value to that variable. Let $U(n_{k_i})$ be a set of variables whose values are used in n_{k_i} and $D(n_{k_i})$ be a set of variables whose values are defined in n_{k_i}. To illustrate these concepts consider the following assignment instruction

$$Y\colon\ a[i + j] := b[j + 3] + v - a[k - 3];$$

at some point of program execution. Since it is possible to determine, by instrumentation, values of array indexes during program execution, we can determine which array elements are used or defined. For example, assume that $i = 2, j = 3$, and $k = 4$ just before Y is executed. In this case, the particular array elements $b[6]$ and $a[1]$ are used; also, the scalar variables v, i, j, and k are all used in Y. By the same token, the only variable defined in Y is array element $a[5]$.

In what folows we introduce the concept of data flow influence (dependence) between instructions in T.

Definition 1: Let n_{k_p} and n_{k_t} be two instructions in T. We say that n_{k_p} directly influences n_{k_t} by variable v, $p < t$, iff

1) $v \in U(n_{k_t})$,
2) $v \in D(n_{k_p})$, and
3) for all j, $p < j < t$, $v \notin D(n_{k_j})$.

This influence describes a situation where one instruction assigns a value to an item of data and the other instruction uses that value. The influences between instructions in T can be represented graphically as an influence network, where each link between instructions represents direct influence between them. The example of an influence subnetwork is presented in Fig. 2. In this subnetwork, for instance, instruction 3 directly influences instruction 6 by variable max.

Definition 2: We say that an input variable x_i influences instruction n_{k_t} in T iff there is a sequence $<n_{r_1}, n_{r_2}, \cdots, n_{r_w}>$ of instructions from T such that:

1) n_{r_1} is an input instruction which defines x_i,
2) $n_{r_w} = n_{k_t}$.
3) n_{r_1} directly influences n_{r_2} by x_i, and
4) for all j, $1 < j < w$, there exists a variable v such that n_{r_j} directly influences $n_{r_{j+1}}$ by v.

From the influence subnetwork of Fig. 2 it is easy to determine that input variable $A[39]$ influences instruction 6 because input instruction 1 directly influences instruction 3 by $A[39]$ and instruction 3 directly influences instruction 6 by variable max. By the same token, we can determine that input variables $A[51]$, low, and step influence test instruction 6. Consequently, these input variables influence branch function $F_1(\boldsymbol{x})$ in subgoal 1 of Example 1. In the same manner, we can determine that input variables $A[39]$, $A[51]$, low, and step influence $F_2(\boldsymbol{x})$ in subgoal 2; input variables $A[39]$, $A[63]$, low, and step influence $F_3(\boldsymbol{x})$ in subgoal 3; input variables high, low, and step influence branch function $F_4(\boldsymbol{x})$ in subgoal 4.

This information can be used to speed up the search during the solution of subgoals by considering only those input variables which have influence on a given branch function. As a result, the possibility of a fruitless search can be significantly reduced. For instance, while solving the fourth subgoal, we are effectively removing two hundred (2*100) evaluations of branch function $F_4(\boldsymbol{x})$; if the number of elements in A had been, for example, one

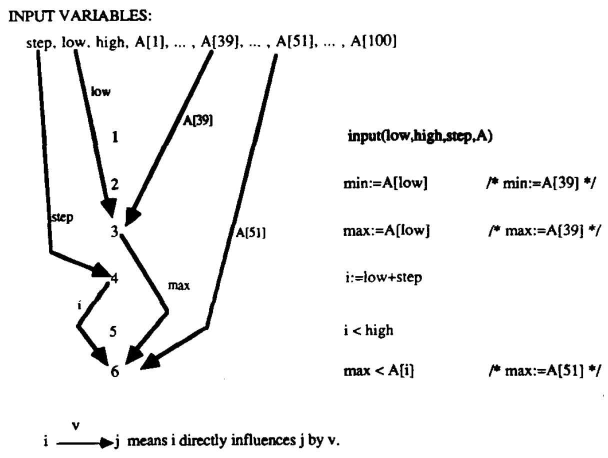


Fig. 2. The influence subnetwork for subpath P_1 from Example 1.

thousand then two thousand evaluations could have been saved.

Further improvement of the search process can be gained by reducing the number of constraint violations. The difficulty with the search method, presented in the previous section, is that when repeatedly encountering a constraint violation it is necessary to reduce the size of a pattern step until the constraint is satisfied. After many such withdrawals the search will be quite slow. We can reduce the number of constraint violations during the search by finding those input variables which have minimal effect, or no effect at all, on the constraint, i.e., those input variables which have minimal or no influence on branch predicates of a successful subpath. For instance, while solving the third subgoal of Example 1, input variable $A[39]$ influences, in subpath P_3, predicates of branches (6, 8) and (8, 10); input variables low and step influence predicates of branches (5, 6), (6, 8), (8, 10), and (5, 6); input variable $A[63]$ does not influence any of the branch predicates in P_3. It is obvious that $A[63]$ should be selected as the first variable for consideration because changing $A[63]$ will not cause any constraint violation. Similarly, solving the fourth subgoal, variables low and step influence six branch predicates in P_4, whereas variable high influences two branch predicates. Input variable high should be selected as a first variable to explore because by changing the value of high the risk of constraint violation is minimized.

Let $P' = < n_{k_1}, n_{k_2}, \cdots, n_{k_i} >$ be a successful subpath of P on a given program input x. With every input variable x_j which influences branch function $F(x)$ of branch $(n_{k_i}, n_{k_{i+1}})$ associated is the integer $r(x_j)$, referred to as a risk factor, equal to the number of branch predicates in P' which are influenced by x_j. On this basis, input variables influencing $F(x)$ are sorted with respect to $r(x_j)$ in the ascending order. The major presumption of this heuristic is that the lower $r(x_j)$, the lower the risk of constraint violation while changing x_j.

Thus the input variables in Example 1 are arranged in the following way:

SUBGOAL 1	Input variables	A[39]	A[51]	low	step
	Risk Factor	0	0	1	1
SUBGOAL 2	Input variables	A[39]	A[51]	low	step
	Risk Factor	1	1	2	2
SUBGOAL 3	Input variables	A[63]	A[39]	low	step
	Risk Factor	0	2	4	4
SUBGOAL 4	Input variables	high	low	step	-
	Risk Factor	2	6	6	-

This arrangement of input variables leads to the solution in 21 trials (program executions). On the other hand, the "blind" arrangement of variables from Example 1 requires 497 trials to find the solution.

During the search, special attention should be paid to input variables that influence array indexes, that is, those input variables that influence the selection of the array elements during program execution. For example, suppose that while solving the first subgoal in Example 1 input variables are ordered in the following way: low, $A[39]$, $A[51]$, step, and suppose that during the exploration of variable low its value has been modified, e.g., low = 20. If the search continues from this point with input variable $A[39]$, then the search will fail because $A[39]$ and $A[51]$ do not influence $F_1(x)$ any more. It is easy to see that array elements $A[20]$ and $A[32]$ now influence branch function $F_1(x)$. Therefore each time an input variable receives a new value and this variable influences the index variable, a new set of input variables influencing a branch function should be derived.

VI. Dynamic Data Structures

The next extension of the automated test data generation involves records and pointers. Records provide a grouping facility for data items. A record is a collection of data items, each of which is said to occupy a "field" of the record. A distinct name is associated with each field, and access to individual items within a record is via these field identifiers. Consequently, every field in a record can be treated as a separate variable.

Pointers, however, create unique problems since the pointer variable actually represents two variables: the pointer itself and the record pointed at. A nameless record of a given type is created by calling the standard procedure **new**(p). Storage is reserved for the record but no value is assigned to it. Once the dynamic record has been created, it can be referenced by pointers.

In our approach, we treat every record, created dynamically by the Pascal procedure **new**, as a separate variable [9]. For this purpose, a list of dynamic records is created and manipulated during program execution. In this way, it is possible to determine not only which dynamic records are pointed to by pointer variables at every point of program execution, but also which fields in every dynamic record are referenced or modified. To distinguish between dynamic records, a unique name must be assigned to each of them. For the purpose of the presentation, the following notation will be used: rec_i. The first execution of the **new** procedure assigns the name rec_1 to the first dynamic record created. The next execution of **new** assigns the name rec_2 to the next record. Every subsequent execution of **new** increases the value of i by one. In this manner, a unique name for every dynamic record is guaranteed. Consequently, every field in a dynamic record is uniquely identified by rec_i.field_name.

We now turn our attention to the question of how to generate test data in the presence of pointers. The search method based on dynamic data flow analysis and backtracking will be illustrated in the following example.

Example 2

Consider the Pascal procedure FIND of Fig. 3. This procedure accepts as input an integer element y and a dynamic data structure pointed to by pointer variable L. The goal of test data generation is to find a program input which will cause the traversal of the following path $P = <s, 1, 2, 3, 4, 7, 8, 3, 4, 7, 9, 3, 4, 5, 6, 3>$.

One of the main problems of the test data generation, in this case, is to find a shape of the input data structure which will cause traversal of the selected path. In this example we assume that the search procedure doesn't have any knowledge about the desired shape or a class of shapes of the input data structure (e.g., a binary tree, a double linked list, a directed graph etc). Since the only information available is the declaration part of the dynamic record, it is assumed that the input dynamic data structure is a directed graph.

The brute force approach to find an input data structure would be to search systematiclaly through the space of all possible input data structure "shapes" until the solution is found. The search method presented in this paper yields the same answer with far fewer trials. Its basic idea is to use dynamic data flow analysis and backtracking. In this method, the goal of finding the input data structure to traverse a selected path is achieved by solving a sequence of subgoals, where subgoals are divided into two categories: arithmetic and pointer subgoals. If the given subgoal is of arithmetic type then the branch function is constructed and the direct search method described in the previous section is applied. On the other hand, if the given subgoal is of pointer type, then the search procedure based

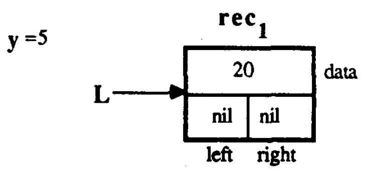
```
type
NodePointer = ^Node;
   Node = record
      data : integer ;
      left : NodePointer ;
      right: NodePointer;
   end ;

procedure FIND (L: NodePointer; y: integer; var q: NodePointer);
var
p: NodePointer ;

begin
1      p := L;
2      q := nil;
3      while p <> nil do
          begin
4            if y = p^.data then
               begin
5                 q := p ;
6                 p := nil;
               end
            else
7,8         if y < p^.data then p := p^.left
9                     else p := p^.right;
          end;
end {FIND};
```

Fig. 3. A sample Pascal procedure.

on dynamic data flow analysis and backtracking is applied.

A subgoal of pointer type, for instance the first subgoal, is solved by using, initially, dynamic data flow analysis to determine those input pointer variables which influence this subgoal (pointer branch predicate). Then the search procedure determines subgoal solution by systematically assigning possible values for those variables (note that a branch function cannot be constructed from the branch predicate of pointer type). When the first subgoal has been solved, the search procedure attempts to solve the next subgoal (if necessary). If the next subgoal cannot be solved within the constaints of the current solution of the first subgoal, then a new solution for the first subgoal is sought. Clearly, when it is realized at some point (subgoal) that a certain value of an input pointer variable can in no way lead to a solution, then the search procedure must backtrack to the previous subgoal to assign a new value to this input variable. If a new solution is found, again the search procedure attempts to solve the next subgoal, constrained of course by the new solution. The search procedure will continue attempting to solve the subgoals, backtracking again and again when necessary, until the solution to the main goal is found or no way to consistently solve the subgoals can be found.

We now show, in more detail, how dynamic data flow analysis and backtracking are used to guide the process of finding the shape of the input data structure. Suppose that the following initial input has been generated:



Procedure FIND from Fig. 3 is executed on this input, and the successful subpath $P_1 = <s, 1, 2, 3, 4, 7, 8, 3>$ of P shown in Fig. 4 is traversed.

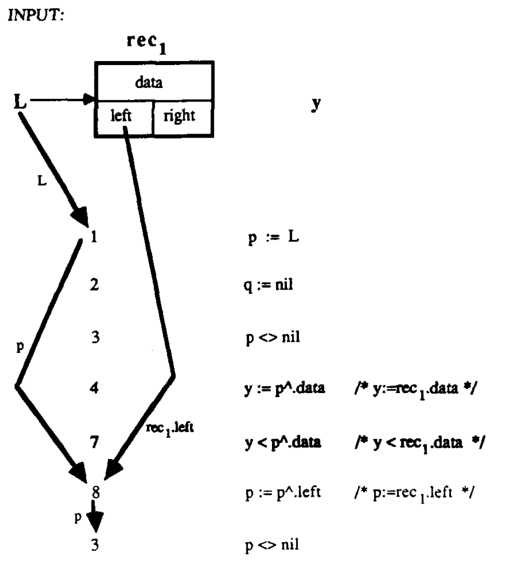
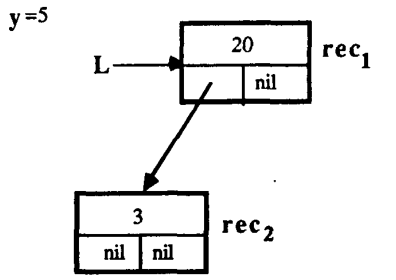

INPUT:
rec1
data
left
right
L
y
L
p
1
2
3
4
7
8
3
p
rec1.left
p := L
q := nil
p <> nil
y := p^.data /* y:=rec1.data */
y < p^.data /* y < rec1.data */
p := p^.left /* p:=rec1.left */
p <> nil

Fig. 4. The influence subnetwork for subpath P_1 from Example 2.

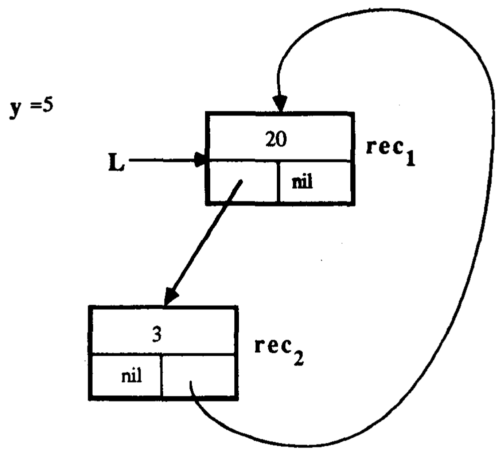
The violation occurs on the second execution of branch (3, 4). The first subgoal is to find values for input variables y, L, rec_1.data, rec_1.left, and rec_1.right such that P_1 is traversed and branch (3, 4) is taken.

Dynamic data flow analysis (see Fig. 4) determines that rec_1.left influences the branch predicate of (3, 4). It should be clear that the only meaningful values which can be assigned to rec_1.left at this moment are nil and adr(rec_1), where adr(rec_1) is an address of rec_1. In addition, a new record rec_2 can be created, and adr(rec_2) can be assigned to rec_2.left. For the sake of simplicity, we assume that the input data structure (directed graph) cannot contain a record pointing to itself. As a result, a new record rec_2 is created and adr(rec_2) is assigned to rec_1.left. It should be noted that each time a new record is created by the search procedure, the nil-value is assigned to all pointer fields of the record and all integer (or real) fields receive random value. Thus rec_2.data receives a random value 3; rec_2.right and rec_2.left receive nil-values. Consequently, the following input data structure is created:

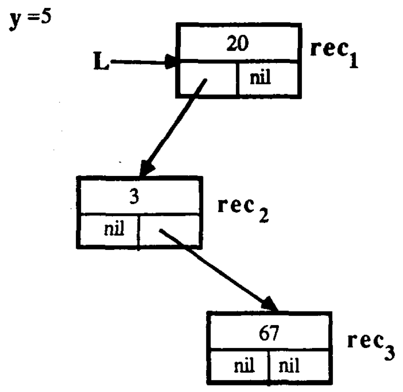


Procedure FIND is executed on this input, and the following successful subpath $P_2 = < s, 1, 2, 3, 4, 7, 8, 3, 4, 7, 9, 3 >$ of P is traversed. The violation occurs on the third execution of branch (3, 4). The second subgoal is to find values for input variables y, L, rec_1.data, rec_1.left, rec_1.right, rec_2.data, rec_2.left, and rec_2.right such that P_2 is traversed and branch (3, 4) is taken.

Dynamic data flow analysis determines that rec_2.right influences the branch predicate of (3, 4). The only values which can be assigned to rec_2.right at this point are adr(rec_1) or adr(rec_3), where rec_3 is a new dynamic record. In the first attempt the search procedure assigns adr(rec_1) to rec_2.right, and the following input data structure is created:

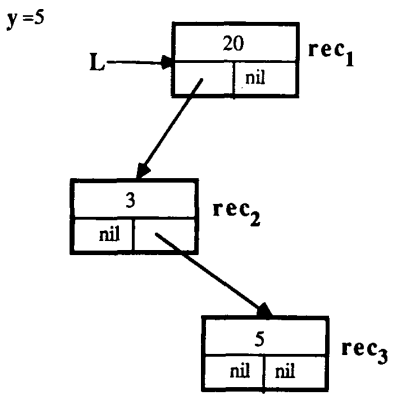


Procedure FIND is executed on this input, and the following successful subpath $P_3' = < s, 1, 2, 3, 4, 7, 8, 3, 4, 7, 9, 3, 4 >$ of P is traversed. The violation occurs on the execution of branch (4, 5). Since the branch predicate of branch (4, 5) contains an arithmetic expression, we can apply the search procedure described in the previous section to solve the third subgoal. However, this procedure fails to find the solution to this subgoal. It should be obvious that the selected path P cannot be traversed for this shape of the input data structure. For this reason, we have to backtrack to the second subgoal and assign the second possible value to rec_2.right. Thus a new record rec_3 is created and adr(rec_3) is assigned to rec_2.right; rec_3.data receives a random value 67. The following input data structure is created:



Procedure FIND is executed, and the following successful subpath $P_3 = < s, 1, 2, 3, 4, 7, 8, 3, 4, 7, 9, 3, 4 >$ of P is traversed. The violation occurs on the execution of branch (4, 5). Let $F_3(x)$ be a branch function of branch (4, 5), where values of $F_3(x)$ can be evaluated by the abs(y-p^.data) expression. Thus the third subgoal is to find a program input x such that $F_3(x) = 0$ subject to the constraint: P_3' is traversed on x.

Dynamic data flow analysis determines that input variables rec_3.data and y influence branch function $F_3(x)$. Using the search procedure described in the previous section, the value of rec_3.data can be found equal to 5. The following solution to the third subgoal is also the solution to the test data generation problem:



VII. Conclusions

The test data generation approach described in this paper is based on program execution, dynamic data flow analysis, and the function minimization methods. It has been shown that the test data generation problem can be reduced to a sequence of subgoals. Function minimization methods are used to solve these subgoals. Moreover, dynamic data flow analysis is applied to speed up the search process by identifying those input variables that influence undesirable program behavior; as a result, the number of fruitless tries can be significantly reduced. The potential value of this approach is exhibited, for instance, by the fact that the efficiency of the search does not depend upon the size of input arrays. The approach of test data generation has then been extended to programs with dynamic data structures, and the search method, which uses dynamic data flow analysis and backtracking to determine the shape of the input dynamic data structure, has been presented. In the approach described in this paper, values of array indexes and pointers are known at each step of program execution, and this approach exploits this information to overcome difficulties of array and pointer handling in the process of test data generation.

Several attempts to use "actual" program execution to derive test data has been reported in the literature [4], [26], [29]. One technique of test data generation for a selected path was described by Miller and Spooner [26]; this technique requires, in the first step, to find by hand a partial solution to the test data generation problem, involving all nonfloating-point input variables. On the basis of this partial solution, a straight-line program, which corresponds to the selected path, is derived from the original program. A real-valued function for the whole path is chosen which is negative when at least one of the branch predicates is false and positive when all the branch predicates are true. Test data are derived by executing the straight-line program and applying a numerical optimization algorithm to maximize this function. No test data generator which uses this techniques has been reported. Another technique described by Benson [4] uses executable assertions in conjunction with optimum search algorithms in order to test program automatically. In this technique the error function, which relates the number of assertions violated during program execution to the values of the input variables, is introduced. The optimum search techniques are used to find the values of the input variables for which the maximum number of assertions are violated.

The approach presented in this paper makes no claim of optimality. It opens, rather, the way for a wide spectrum of test data generation methods based on the program execution and dynamic data flow analysis. The main extensions should go into the generality and robustness of this approach for use in the real world. Moreover, to fully understand the power and limitations of the dynamic approach of test data generation additional research is required. We now highlight some directions for further research.

A. Static Analysis

One direction of the further research is to incorporate static analysis (e.g., dependence analysis [5], [15], [20]) in the process of test data generation. For example, for the program of Fig. 1 static analysis can determine that input variables low, high, and step always influence predicate "$i <$ high." Consequently, there is no need to apply dynamic data flow analysis (to determine influencing input variables) while solving subgoals related to this predicate. This can speed up the search, especially when there is a significant number of subgoals associated with this predicate along the selected path. In addition, static analysis can be used to reduce the amount of information recorded during program execution, which is required to perform dynamic data flow analysis, e.g., used/defined variables. For example, for subgoals associated with the predicate " max $< A[i]$" in the program of Fig. 1, static analysis can determine that, in order to find the influencing input variables, no recording of used/defined variables in the "if min $> A[i]$ then min := $A[i]$" statement is required. We should stress, however, that static analysis can be expensive, especially in the presence of procedures; therefore, more research is required to determine a tradeoff between dynamic data flow analysis and static analysis in the test data generation process.

B. Symbolic Evaluation

One of the problems of the dynamic approach of test data generation is its very limited ability to detect path infeasibility. If the selected path is infeasible and the infeasibility is not detected, a large number of attempts can be performed before the search procedure terminates and a lot of effort can be wasted. Symbolic evaluation, on the other hand, is capable of detecting, to a certain extent, path infeasibility. We, therefore, believe that combination of both techniques for test data generation can be advantageous. For example, symbolic evaluation can be used to check path consistency before dynamic approach of test data generation is used. In addition, application of symbolic evaluation over mixtures of actual and symbolic data [17] in the test data generation process should be investigated.

C. Procedures

To be of practical value, the dynamic approach of test data generation has to be extended to programs with procedures. This does not seem to pose difficulties because it is possible by instrumentation to identify variables that are used or defined in a procedure call for the actual execution; as a result, dynamic data flow analysis can precisely determine influencing input variables in the presence of procedure calls. On the other hand, static analysis, in general case, fails to identify the used/defined variables in the procedure call.

D. Global Optimization

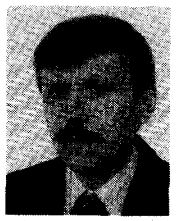
The function minimization algorithm applied in our approach of test data generation is based on the direct search method published in [12], [13]. One of the problems of this method is that it allows only to find a local minimum. In many cases this can prevent solving subgoals, especially for branch functions with several local minimums. There exists an extensive research in the area of global optimization, e.g., [31], and several techniques have been developed to find a global optimum. The research is needed to investigate the application of those techniques in the dynamic approach of test data generation.

Acknowledgment

I would like to thank the referees and the editor, W. Howden, for their helpful comments.

References

[1] D. Alberts, "The economics of software quality assurance," in *AFIPS Conf. Proc. 1976 Nat. Computer Conf.*, vol. 45. Montvale, NJ: AFIPS Press, pp. 433-442.
[2] R. Balzer, "EXDAMS—Extendable debugging and monitoring system," in *1969 Spring Joint Computer Conf., AFIPS Conf. Proc.*, vol. 34. Montvale, NJ: AFIPS Press, pp. 576-580.
[3] J. Bauer and A. Finger, "Test plan generation using formal grammars," in *Proc. 4th Int. Conf. Software Engineering*, 1979, pp. 425-432.
[4] J. Benson, "Adaptive search techniques applied to software testing," *ACM Perform. Eval. Rev.*, vol. 10, no. 1, pp. 109-116, Spring 1981.
[5] J. Bergeretti and B. Carre, "Information-flow and data-flow analysis of while-programs," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 37-61, Jan. 1985.
[6] J. Bicevskis, J. Borzovs, U. Straujums, A. Zarins, and E. Miller, "SMOTL—A system to construct samples for data processing program debugging," *IEEE Trans. Software Eng.*, vol. SE-5, no. 1, pp. 60-66, Jan. 1979.
[7] D. Bird and C. Munoz, "Automatic generation of random self-checking test cases," *IBM Syst. J.*, vol. 22, no. 3, pp. 229-245, 1983.
[8] R. Boyer, B. Elspas, and K. Levitt, "SELECT—A formal system for testing and debugging programs by symbolic execution," *SIGPLAN Notices*, vol. 10, no. 6, pp. 234-245, June 1975.
[9] F. Chan and T. Chen, "AIDA—A dynamic data flow anomaly detection system for Pascal programs," *Software—Practice and Experience*, vol. 17, no. 3, pp. 227-239, Mar. 1987.
[10] L. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Trans. Software Eng.*, vol. SE-2, no. 3, pp. 215-222, Sept. 1976.
[11] R. DeMillo, W. McCracken, R. Martin, and J. Passafiume, *Software Testing and Evaluation*. Benjamin/Cummings, 1987.
[12] P. Gill and W. Murray, Eds., *Numerical Methods for Constrained Optimization*. New York: Academic, 1974.
[13] H. Glass and L. Cooper, "Sequential search: A method for solving constrained optimization problems," *J. ACM*, vol. 12, no. 1, pp. 71-82, Jan. 1965.
[14] R. Fairly, "An experimental program-testing facility," *IEEE Trans. Software Eng.*, vol. SE-1, no. 4, pp. 350-357, Dec. 1975.
[15] J. Ferrante, K. Ottenstein, and J. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319-349, July 1987.
[16] W. Howden, "Symbolic testing and the DISSECT symbolic evaluation system," *IEEE Trans. Software Eng.*, vol. SE-4, no. 4, pp. 266-278, 1977.
[17] —, *Functional Program Testing and Analysis*. New York: McGraw-Hill, 1987.
[18] D. Ince, "The automatic generation of test data," *Comput. J.*, vol. 30, no. 1, pp. 63-69, 1987.
[19] W. Jessop, J. Kanem, S. Roy, and J. Scanlon, "ATLAS—An automated software testing system," in *Proc. 2nd Int. Conf. Software Engirieeering*, 1976.
[20] B. Korel, "The program dependence graph in static program testing," *Inform. Processing Lett.*, vol. 24, pp. 103-108, Jan. 1987.
[21] —, "PELAS—Program error locating assistant system," *IEEE Trans. Software Eng.*, vol. 14, no. 9, pp. 1253-1260, Sept. 1988.
[22] B. Korel and J. Laski, *Dynamic program slicing," Inform. Processing Lett.*, vol. 29, no. 3, pp. 155-163, Oct. 1988.
[23] B. Korel, "TESTGEN—A structural test data generation system," Dep. Comput. Sci., Wayne State Univ., Detroit, MI, Tech. Rep. CSC-89-001, 1989.
[24] N. Lyons, "An automatic data generation system for data base simulation and testing," *Data Base*, vol. 8, no. 4, pp. 10-13, 1977.
[25] E. Miller, Jr. and R. Melton, "Automated generation of testcase datasets," *SIGPLAN Notices*, vol. 10, no. 6, pp. 51-58, June 1975.
[26] W. Miller and D. Spooner, "Automatic generation of floating-point test data," *IEEE Trans. Software Eng.*, vol. SE-2, no. 3, pp. 223-226, Sept. 1976.
[27] S. Muchnick and N. Jones, Eds., *Program Flow Analysis: Theory and Applications*. Englewood Cliffs, NJ: Prentice-Hall International, 1981.
[28] G. Myers, *The Art of Software Testing*. New York: Wiley, 1979.
[29] M Paige, "Data space testing," *ACM Perform. Eval. Rev.*, vol. 10, no. 1, pp. 117-127, Spring 1981.
[30] C. Ramamoorthy, S. Ho, and W. Chen, "On the automated generation of program test data," *IEEE Trans. Software Eng.*, vol. SE-2, no. 4, pp. 293-300, Dec. 1976.
[31] H. Ratschek, *New Computer Methods for Global Optimization*. New York: Halsted, 1988.



Bogdan Korel (M'87) was born in Poland. He received the M.S. degree in electrical engineering from the Technical University of Kiev, USSR, and the Ph.D. degree in systems engineering from Oakland University, Rochester, MI, in 1986.

He is an Assistant Professor in the Department of Computer Science at Wayne State University, Detroit, MI. His research interests include automatic software testing and debugging, software development environments, and distributed systems.

Dr. Korel is a member of the IEEE Computer Society and the Association for Computing Machinery.