A DYNAMIC APPROACH OF TEST DATA GENERATION

Bogdan Korel

Department of Computer Science
Wayne State University
Detroit, MI 48202 USA

ABSTRACT

Test data generation in program testing is the process of identifying a set of test data which satisfies a selected testing criterion. Existing pathwise test data generators [Bic79,Boy75,Cla76,How77,Ram76,Kor90] proceed by selecting a program path(s) and then generating a program input for that path. One of the problems in this approach is that infeasible paths are often selected; as a result, significant computational effort can be wasted in analyzing those paths. In this paper, we present a novel approach of test data generation, referred to as a dynamic approach of test data generation, in which the path selection stage is eliminated. In our approach, test data are derived based on the actual execution of the program under test, dynamic data flow analysis, 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 automatically locate values of input variables which will change the flow of execution at this branch. In addition, dynamic data flow analysis is used to determine input variables which are responsible for the undesirable program behavior, leading to the speed-up of the search process.

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, which require that certain program elements, or combinations thereof, be exercised (e.g., statement coverage, branch coverage, data flow coverage, etc). After initial testing, programmers face the problem of finding test data to exercise not covered program elements, e.g., branches. Derivation of test data to exercise those remaining elements requires from programmers a good understanding of the program under test and can be very labor-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 only partially understood. Several structural techniques for testing modified programs have been proposed in the literature, e.g., [Har88,Leu89]. 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 test data generation could be automated, the cost of software development and maintenance should be reduced significantly. A test data generator is a tool which assists a programmer in the generation of test data for a program. In this paper the test data generation problem is defined as follows: for a given program element, e.g., a modified statement, find a program input on which this element is executed. There are two types of test data generators which could be applied for this problem: random test data generators [Bir83] and pathwise test data generators [Bic79,Boy75,Cla76,How77,Ram76,Kor90]. The existing pathwise test data generators reduce the problem of test data generation to a "path" problem, that is, they proceed by selecting a program path which will exercise the selected program element. Then, if possible, a program input for that path is generated via symbolic evaluation. 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. As pointed out in [Inc87,Muc81,DeM87], there are some weaknesses with the existing pathwise test data generators; those weaknesses are mainly associated with symbolic evaluation and the path selection process. Symbolic evaluation is a promising approach; however, there are several difficulties which prevent this method from being currently used for "real" programs. The major difficulty is inefficient handling of arrays, dynamic data structures (pointers), and procedures. An alternative approach of finding a program input to traverse the selected path has been recently proposed in [Kor90]. This approach is based on program execution and dynamic data flow analysis. The initial experience has shown that this approach can in many cases efficiently handle programs with arrays and pointers.

In the existing pathwise test data generators, program paths can be selected automatically by the test data generator or provided by the user. The major weakness of automatic path selection is path infeasibility. Since the path selection is based purely on a control flow graph of the program, this method leads very often to selection of infeasible paths; in this case, the significant computational effort can be wasted in analyzing those infeasible paths. In the second method, users provide paths to be traversed. This, however, requires significant effort on the user side to select the "feasible" program paths; as a result, the degree of automation of the test data generation process can be significantly degraded. One attempt to alleviate the infeasibility problem, in the automatic method of path selection, was reported in [Cla78,Woo79]. In this technique, symbolic evaluation is used to detect, if possible, path infeasibility in early stages of path selection. Although this technique can partially alleviate the infeasibility problem, the test data generator can often fail to find input data to traverse the selected path because of undetected infeasibility or the failure of the generator to find input data due to the weaknesses of symbolic evaluation.

The test data generation approach presented in this paper attempts to break away from the conventional "pathwise" approach to test data generation. It differs from the existing techniques in that the path selection stage is eliminated, i.e., program paths which should be executed are not selected. This approach is based on actual execution of the program under test, dynamic data flow analysis, and function minimization methods. Test data are developed using actual values of input variables. The approach starts by executing initially a program for an arbitrary program input. When the program is executed, the program execution flow is monitored. 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 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 automatically locate values of input variables which will change the flow of execution at this branch. In addition, dynamic data flow analysis [Kor90] is used to determine input variables which are responsible for the undesirable program behavior, leading to the speed-up of the search process. In our approach, arrays and dynamic data structures can be handled precisely because during program execution all variable values, including index variables and pointer variables, are known. As a result, the effectiveness the test data generation can be improved. The approach described in this paper has been implemented as an extension of the existing test data generation tool TESTGEN [Kor89].

The organization of this paper is as follows: In the next section basic concepts and notations are introduced. Section 3 describes the approach of test data generation for a node problem. The basic search method is presented in Section 4. Dynamic data flow concepts and a search method using these concepts are presented in Section 5. Finally, in the Conclusions further research is outlined.

2. 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-exit executable part of a statement in Q that cannot be further decomposed. 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. An edge (n_i,n_j) is called a **branch** if n_i is a test node, e.g., (7,8), and (7,9) are branches in the program of Figure 1. 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 Figure 1 branch (6,7) is labeled "i < high," branch (9,10) is labeled "min ≥ A[i]," and branch (9,14) is labeled "min < 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 $I=(x_1,x_2,...,x_n)$ 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, $D = D_{x_1} \times D_{x_2} \times ... \times D_{x_n}$, where each D_{x_i} is the domain for input variable x_i. A single point **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 $P = <n_{k_1},n_{k_2},...,n_{k_q}>$ of nodes, such that $n_{k_1}=s$, and for all i, $1 \le i < q$, $(n_{k_i},n_{k_{i+1}}) \in A$. A path is feasible if there exists a program input **x** for which the path is traversed during program execution; otherwise, the path is infeasible.

3. DESCRIPTION OF THE APPROACH.

We present a novel approach of test data generation which differs from the existing techniques that the path selection stage is eliminated. We present our approach for the node problem; however, this approach can be extended to other types of problems, i.e., branch problem, definition-use chain problem [LaKo83,Wey85], etc. The node problem, in this paper, is stated as follows:

Given node Y in a program. The goal is to find a program input **x** on which node Y will be executed.

Observe that in the branch problem we would search for a program input **x**, on which a selected branch were executed. The present techniques of test data generation reduce the node problem to a "path" problem, that is, 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. If the program input is not found, the next path is selected. This process is repeated until a path is selected for which a program input is found, or the designated resources are exhausted, e.g., computer-time limit for the search. The major weakness [Inc87,Muc81,DeM87] associated with this technique is the significant computational effort can be wasted in analyzing infeasible paths which are selected in the path selection stage.

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

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

Figure 1. A sample program.

In our approach the path selection stage is eliminated, i.e., paths, which should be traversed, are not generated. The approach starts by executing initially a program for an arbitrary program input. During 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 doesn't lead to the selected node). In the latter case, a new program input must be determined 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 which 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: <s,1,2,3,4,6,7,2,8,9,25>. 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 **x** must be found to change the flow of execution at this node, that is, to execute branch (9,10).

The general idea of our 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 doesn't 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 four 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 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 which can be chosen to execute or not execute 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.

Branch Classification:

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_j 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 **x** which 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 when this branch is executed there is no way to reach node 20. In this case, a new program input must be found on which branch (9,10) will be taken.

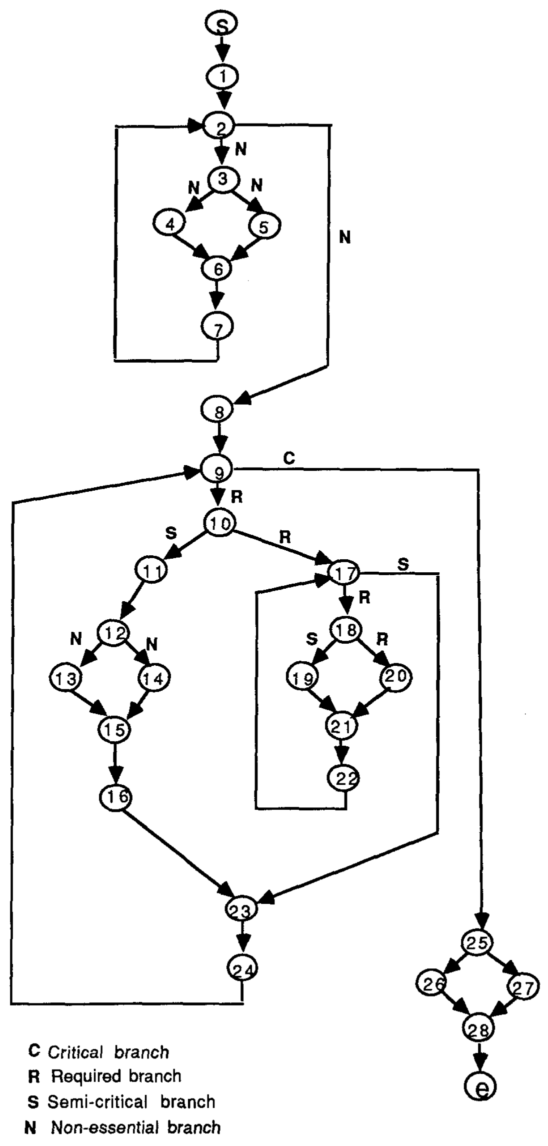


Figure 2. A control flow graph with branch classification.

2. **A required branch**. A branch (n_i,n_j) is called a required branch with respect to node Y iff (1) Y is in the scope of control influence of n_i, and (2) there is a acyclic path from n_i to Y through branch (n_i,n_j).

If a required branch is executed, the program execution is allowed to be continued through this branch. For example, branch (10,17) is a required branch because its execution leads "closer" to the selected node 20, and it is obvious that the program execution should continue through this branch.

3. **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.

4. **A non-essential branch**. A branch (n_i,n_j) is called a non-essential branch with respect to node Y iff Y is not in the scope of control influence of n_i.

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 don't influence the execution of this node. Consequently, the execution can continue through either of them.

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

Critical branches: (9,25).
Required branches: (9,10), (10,17), (17,18), (18,20).
Semi-critical branches: (10,11), (17,23), (18,19).
Non-essential branches: (2,3), (2,8), (3,4), (3,5), (12,13), (12,14).

Observe, that branches (25,26) and (25,27) are not classified because the search procedure prevents the execution of those branches by terminating the program execution at critical branch (9,25).

Summing up, if during program execution the current branch is a non-essential or required 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 the 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:
We now work through a simple example to present our approach. Consider the control flow graph of Figure 2. The goal is to find a program input **x** on which node 20 will be executed. For this purpose, the program is executed on an initial program input $\mathbf{x}^0$; suppose that the following path is traversed on this input: <s,1,2,3,4,6,7,2,8,9,25>. All branches on this path are non-essential ones except branch (9,25) which is a critical branch. The search procedure terminates the execution at this point and tries to solve the first subgoal, that is, to find a program input on which branch (9,10) will be traversed. Let $\mathbf{x}^1$ be the solution to the first subgoal, and suppose that the following path is traversed: <s,1,2,3,4,6,7,2,8,9,10,11>. 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: <s,1,2,3,4,6,7,2,8,9,10,11,12,14,15,16,23, 24,9,10,11>. The search procedure has to solve the third subgoal, i.e., to find a program input **x** on which branch (10,17) will be executed on the second iteration of loop 9-24. Let $\mathbf{x}^3$ be the solution to the third subgoal, and suppose that the following path is traversed: <s,1,2,3,4,6,7,2,8,9,10,11,12,14, 15,16,23,24,9,10,17,23>. 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 will be executed, or no progress can be made (as a result, the search procedure fails to find the solution).

4. SOLVING SUBGOALS.

Now we turn our attention to the question of how to find a new program input which will change the flow of execution at the current branch. 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 **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 Table 1.

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 ≤) when the branch predicate is true. It is obvious that F is actually a function of program input **x**. Symbolic evaluation can be used to find explicit representation of the F(**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 consider the alternative approach in which the branch function is evaluated for any program input by executing the program. For instance, the true branch of a test "if y > z then.." has a branch function F, whose values can be computed for a given input by executing the program and evaluating the z-y expression.

TABLE 1:

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

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, we have to solve the first subgoal. Let $<n_{p_1},n_{p_2},...,n_{p_i},n_{p_{i+1}}>$ be a program path traversed on input $\mathbf{x}^0$, such that for all j, $1 \leq j < i$, $(n_{p_j},n_{p_{j+1}})$ is a non-essential or required 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 **x** which will preserve the traversal of path $<n_{p_1},n_{p_2},...,n_{p_i}>$, 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, we want to find a program input $\mathbf{x} \in D$ satisfying:

$$F_i(\mathbf{x}) \ \mathbf{rel}_i \ 0$$

subject to the constraint:

$$<n_{p_1},n_{p_2},...,n_{p_i}> \text{ is traversed on } \mathbf{x},$$

where $\mathbf{rel}_i$ 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 [Gill74,Gla65], stopping when $F_i(\mathbf{x})$ becomes negative (or zero, depending on rel_i). To verify whether or not the constraint has been violated for some input **x**, it is enough to execute the program and check whether the constraint path has been traversed. If the constraint path has not been traversed, we say that the constraint is violated on a given input **x**.

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 [Gill74,Gla65], 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 $x_1,...,x_n$ 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 [Gla65] consists of two major phases, an "exploratory search" and "pattern search." In the exploratory search, the selected input variable x_j is incremented and decreased by a small amount, while the remaining input variables are held constant. For each variable change, the program is executed and the values of the branch function are compared. In this way, it is possible to indicate a direction in which to proceed, that is, to make a larger move. Assuming that the exploratory moves are able to indicate a direction in which to proceed, a larger move called a pattern move 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_{p_i},n_{p_k}) has not been taken, the branch function is evaluated and its value is compared to the value of the branch function for the previous program input. If branch function $F_i(\mathbf{x})$ 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 search process continues until the branch function becomes negative (or zero) or a minimum is reached. In the latter case, the last value of x_j is retained and the search continues with the next input variable. 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.

One of the difficulties which we faced during experimentations with this approach was caused by the requirement of the exact traversal of the constraint path. In many cases, this requirement was very rigid because many unnecessary constraints (conditions) were imposed on the search, seriously limiting its efficiency. This situation is described in Example 2.

Example 2:
Consider the program of Figure 1. The goal is to find a program input **x** (i.e., values for input variables: low, high, 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: low=39, high=53, A[1]=1, A[2]=2,..., A[100]=100, and the following path is traversed: <s,1,2,3,4,5,6,7,8,9,14>. Branch (9,14) is a semi-critical branch with respect to node 12. Let $F_1(\mathbf{x})$ be a branch function of branch (9,10), where values of $F_1(\mathbf{x})$ can be evaluated by computing the A[i]-min expression at node 9. The first subgoal is to find a program input **x** such that $F(\mathbf{x}) \leq 0$, subject to the constraint: path <s,1,2,3,4, 5,6,7,8,9> is traversed on **x**. It should be clear that this subgoal cannot be solved under current constraint, and the search procedure continues the program execution through branch (9,14). In this case, loop 6-14 iterates and the following path is traversed: <s,1,2,3,4,5,6,7,8,9,14,6,7,8,9,14>. An attempt to solve the next subgoal to execute branch (9,10) fails again because there is no solution for this subgoal under current constraint. The major reason of the failure of the search is the constraint requiring the execution of branch (7,8); 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 the branch (7,9) instead of branch (7,8) should be allowed to be executed.

For this reasons, we have modified the search procedure by relaxing the constraint: If, during solving a subgoal, constraint violations do not allow one to find a solution, the constraint, requiring the full traversal of a constraint path, is "relaxed" by requiring only "partial" preservation of the constraint path traversal. In particular, if a constraint violation occurs on a non-essential branch, then this violation is ignored and the execution continues through the current branch. However, if the constraint violation occurs on a required branch, the execution is not allowed to continue and the constraint violation is reported to the search procedure. Note that violation on a required branch means execution of a critical or semi-critical branch; it should be clear why execution is not allowed to continue through these branches. This modification leads to the very efficient search in

Example 2 because, during solving the first subgoal, the unnecessary constraint is eliminated; as a result, the execution of branch (7,9) is allowed.

5. DYNAMIC DATA FLOW ORIENTED SEARCH.

In this section, we present a heuristic method, based on dynamic data flow analysis, which can significantly speed-up the process of finding solutions to subgoals. 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 in deciding about the reduction of branch function evaluations is the arrangement of input variables for consideration. It should be obvious that a different arrangement of input variables for consideration leads to the different performance of the search process. The arrangement of input variables in the search procedure presented in the previous section is rather rigid and in a sense "blind." It doesn't give any preference to a variable that has a very good chance of getting the search to the solution quickly. 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, during the solution of subgoals, the important question arises as to which are the most promising input variables to explore. The heuristic method presented in this paper is based on dynamic data flow analysis which allows determining these input variables which are responsible for the current value of the branch function on a given program input.

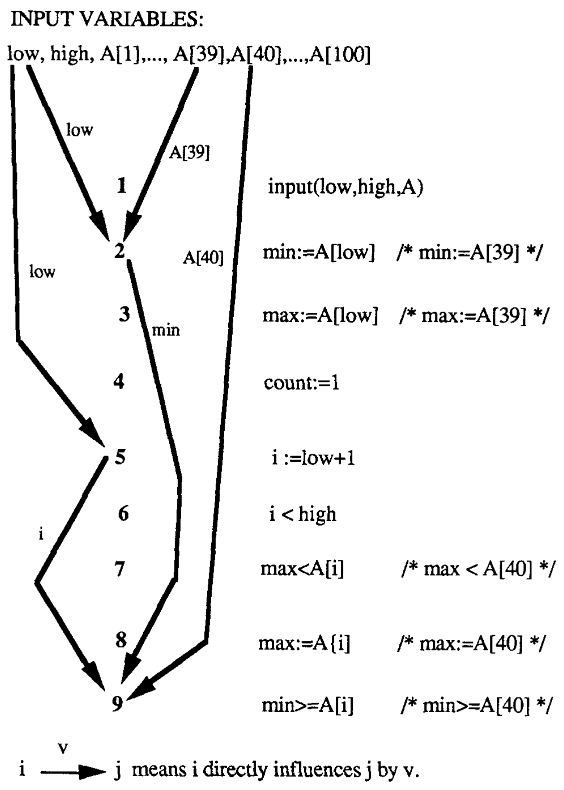
We now introduce the basic concepts of dynamic data flow analysis [Kor88b,Kor90], that is, those concerned with data flow along the path which has been traversed during program execution. Let $T=<n_{k_1},n_{k_2},...,n_{k_q}>$ be a path that is traversed on a program input **x**. A use of variable v in T is a node n_{k_i} in which this variable is referenced. A definition of variable v in T is a node 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}. Let n_{k_p} and n_{k_t} be two nodes 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 node assigns a value to an item of data and the other node uses that value. For instance, in the traversed path in Figure 3, node 2 directly influences node 9 by variable min.

We say that an input variable x_i **influences** node n_{k_t} in T iff there is a sequence $<n_{r_1},n_{r_2},...,n_{r_w}>$ of nodes from T such that: (1) n_{r_1} is an input node 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.

For example, in Figure 3 input variable A[39] influences node 9 because input node 1 directly influences node 2 by A[39] and node 2 directly influences node 9 by variable min.



```
1  input(low,high,A)
2  min:=A[low]    /* min:=A[39] */
3  max:=A[low]    /* max:=A[39] */
4  count:=1
5  i :=low+1
6  i < high
7  max<A[i]       /* max < A[40] */
8  max:=A{i]      /* max:=A[40] */
9  min>=A[i]      /* min>=A[40] */
```

$i \xrightarrow{v} j$ means i directly influences j by v.

Figure 3. A traversed path of the program of Figure 1 and its influence network on input: low=39, high=53, A[1]=1, A[2]=2, ... , A[100]=100.

The influences between nodes in the traversed path can be represented graphically as an influence network, where each link between nodes represents direct influence between them. The example of an influence network is presented in Figure 3. From this network it easy to determine that input variables A[39], A[40], and low influence node 9. Consequently, these variables influence branch function F(**x**) of branch (9,10) in the first subgoal of Example 2.

This information is used by the search procedure to speed-up the search during the solution of subgoals by considering only those input variables which have influence on the current branch function. As a result, the possibility of a fruitless search can be significantly reduced. For instance, while solving the first subgoal (from Example 2), we are effectively removing 2*38 evaluations of branch function F(**x**). The preliminary experiments with TESTGEN [Kor89] have shown that the dynamic data flow oriented search can significantly reduce the search time (depending on the size of input arrays).

6. CONCLUSIONS.

In this paper, we have presented a novel approach of test data generation which differs from the existing techniques that the path selection stage is eliminated. In our approach, test data are derived based on the actual execution of the program under test, dynamic data flow analysis, and function minimization methods. It has been shown that the test data generation problem can be reduced to a sequence of subgoals which are solved using function minimization methods. Moreover, dynamic data flow analysis is applied to speed up the search process by identifying those input variables that influence undesirable program behavior. The initial experiments with this approach has shown that it can alleviate the infeasibility problem and efficiently handle programs with arrays and dynamic data structures.

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. To be of practical value, the method has to be extended to arbitrary programs, e.g., programs with procedure calls. This does not seem to pose difficulties because it is possible to identify the variables that are used or defined in a procedure call for the actual execution. The branch classification presented in this paper is based solely on the control flow graph information. We believe that data flow information can significantly enhance this classification leading to more efficient test data generation.

8. REFERENCES.

[Bic79] J. Bicevskis, J. Borzovs, U. Straujumus, A. Zarins, E. Miller, "SMOTL - A system to construct samples for data processing program debugging," IEEE Trans. on Software Eng., vol. SE-5, No. 1, Jan. 1979, pp. 60-66.

[Bir83] D. Bird, C. Munoz, "Automatic generation of random self-checking test cases," IBM Systems Journal, vol. 22, No. 3, 1983, pp. 229-245.

[Boy75] R. Boyer, B. Elspas, K. Levitt, "SELECT - A formal system for testing and debugging programs by symbolic execution," SIGPLAN Notices, vol. 10, No. 6, June 1975, pp. 234-245.

[Cla76] L. Clarke, "A system to generate test data and symbolically execute programs," IEEE Trans. on Software Eng., vol. SE-2, No. 3, Sept. 1976, pp. 215-222.

[Cla78] L. Clarke, "Automatic Test Data Selection Techniques," Infotech State Of the Art Report on Software Testing, September 1978.

[DeM87] R. DeMillo, W. McCracken, R. Martin, J. Pasafiume, Software Testing and Evaluation, The Benjamin/Cummings Publishing Company, 1987.

[Gil74] P. Gill, W. Murray, Ed., Numerical Methods for Constrained Optimization, New York: Academic, 1974.

[Gla65] H. Glass, J. Cooper, "Sequential search: a method for solving constrained optimization problems," Journal of ACM, vol. 12, No. 1, Jan. 1965, pp. 71-82.

[Har88] M. Harrold, M. Soffa, "An incremental approach to unit testing during maintenance," Proceedingd of the Conference on Software Maintenance, Phoenix, AZ, October 1988, pp. 362-367.

[How77] W. Howden, "Symbolic testing and the DISSECT symbolic evaluation system," IEEE Trans. on Software Eng., vol. SE-4, No. 4, 1977, pp. 266-278.

[Inc87] D. Ince, "The automatic generation of test data," The Computer Journal, vol. 30, No. 1, 1987, pp. 63-69.

[Kor85] B. Korel, J. Laski, "A tool for data flow oriented program testing," SoftFair II, Proc. of 2-d Conf. on Software Development, Tools, Techniques, and Alternatives, San Francisco, Dec. 4-5, 1985, pp. 34-38.

[Kor88a] B. Korel, J. Laski, "Dynamic program slicing," Information Processing Letters, vol. 29, No. 3, October 1988, pp. 155-163.

[Kor88b] B. Korel, "PELAS - Program Error Locating Assistant System," IEEE Trans. on Software Eng., vol. SE-14, No. 9, Sept. 1988, pp. 1253-1260.

[Kor89] B. Korel, "TESTGEN - A Structural Test Data Generation System," Sixth International Conference on Testing Computer Software, Washington, D.C., May 22-25, 1989.

[Kor90] B. Korel, "Automated Software Test Data Generation," IEEE Trans. on Software Eng., vol. SE-16, No. 8, August 1990, pp. 870-879.

[LaKo83] J. Laski, B. Korel, "A data flow oriented program testing strategy," IEEE Trans. on Software Eng., vol. SE-9, No. 3, May 1983, pp. 347-354.

[Leu89] H. Leung, L. White, "Insights into regression testing," Proceedings of the Conference on Software Maintenance, October 1989, pp. 60-69.

[Muc81] S. Muchnick, N. Jones, Ed., Program flow analysis: Theory and Applications, Prentice-Hall International, 1981.

[Ram76] C. Ramamoorthy, S. Ho, W. Chen, "On the automated generation of program test data," IEEE Trans. on Software Eng., vol. SE-2, No. 4, Dec. 1976, pp. 293-300.

[Wey85] E. Weyuker, S. Rapps, "Selecting software test data using data flow information," IEEE Trans. on Software Eng., vol. SE-11, No. 4, April 1985. pp. 367-375.

[Woo79] J. Woods, "Path Selection for Symbolic Execution Systems," Ph. D. Thesis, University of Massachusetts, August 1979.