The Chaining Approach for Software Test Data Generation

ROGER FERGUSON
Lawrence Technological University
and
BOGDAN KOREL
Illinois Institute of Technology

Software testing is very labor intensive and expensive and accounts for a significant portion of software system development cost. If the testing process could be automated, the cost of developing software could be significantly reduced. Test data generation in program testing is the process of identifying a set of test data that satisfies a selected testing criterion, such as statement coverage and branch coverage. In this article we present a *chaining approach* for automated software test data generation which builds on the current theory of execution-oriented test data generation. In the chaining approach, test data are derived based on the actual execution of the program under test. For many programs, the execution of the selected statement may require prior execution of some other statements. The existing methods of test data generation may not efficiently generate test data for these types of programs because they only use control flow information of a program during the search process. The chaining approach uses data dependence analysis to guide the search process, i.e., data dependence analysis automatically identifies statements that affect the execution of the selected statement. The chaining approach uses these statements to form a sequence of statements that is to be executed prior to the execution of the selected statement. The experiments have shown that the chaining approach may significantly improve the chances of finding test data as compared to the existing methods of automated test data generation.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*test data generation*

General Terms: Experimentation, Measurement, Performance

Additional Key Words and Phrases: Data dependency, dynamic analysis, heuristics, program execution

This research was partially supported by the NSF grant CCR-9308895.
Authors' addresses: R. Ferguson, Department of Computer Science, Lawrence Technological University, 21000 West Ten Mile Road, Southfield, MI 48075-1058; B. Korel, Department of Computer Science, Illinois Institute of Technology, 10 West 31st Street, Chicago, IL 60616; email: korel@charlie.iit.edu.

1. INTRODUCTION

Software testing is a very labor intensive and hence very expensive process. It can account for 50% of the total cost of software development [Alberts 1976; DeMillo et al. 1987; Myers 1979]. If the process of testing could be automated [Ince 1987; Jessop et al. 1976], significant reductions in the cost of software development could be achieved. Data generation for software testing is the process of identifying program inputs which satisfy selected testing criteria. Structural test coverage criteria require that certain program elements be evaluated (e.g., statement coverage, branch coverage, data flow coverage, etc.). After initial testing, programmers face the problem of finding additional test data to evaluate program elements not covered, e.g., statements not yet covered. Finding input test data to evaluate those remaining elements requires a good understanding of the program under test from programmers and can be very labor intensive and expensive, increasing the overall cost of the software. This is because programmers do not always work on their own code; for example, during maintenance programmers modify someone else's programs, which are often poorly, or only partially, understood.

The test data generation problem in this article is defined as follows: for a given program element, find a program input on which this element is executed. In this article, we will concentrate on finding test data for program statements. However, the approach is also applicable for branch testing and data flow testing. A test data generator is a tool which assists a programmer in the generation of test data for the program. There are three types of test data generators that could be applied for the test data generation problem: *random* test data generators [Bird and Munoz 1982], *path-oriented* test data generators [Boyer et al. 1975; Clarke 1976; Korel 1990a; Ramamoorthy et al. 1976], and *goal-oriented* test data generators [Korel 1990b; 1992]. The path-oriented approach is the process of selecting a program path(s) to the selected statement and then generating input data for that path. The path to be tested can be generated automatically, or it can be provided by the user. The goal-oriented approach is the process of generating input test data to execute the selected statement irrespective of the path taken, i.e., the path selection stage is eliminated.

For some programs, the execution of the selected statement may require prior execution of some other statements in the program. Our experience with the path-oriented approach and goal-oriented approach has shown that for these types of programs the control flow graph is not sufficient to guide the searching process to generate input data to reach the selected statement. In this article, we extend the goal-oriented approach of test data generation [Korel 1990b; 1992]. Our new approach, referred to as the *chaining approach*, uses data dependencies to guide the search process—i.e., data dependence analysis is used to identify statements that affect execution of the selected statement. By requiring that these statements be executed prior to reaching the selected statement, the chances of executing the selected statement may be increased. Our experiments have shown that

```
program sample;
var
target,i : integer;
a, b: array [1..10] of integer;
fa, fb: boolean;
begin
1     input (a,b,target);
2     i := 1;
3     fa := false;
4     fb := false;
5     while (i ≤ 10) do begin
6         if (a[i] = target) then
7             fa := true;
8         i := i + 1;
      end;
9     if (fa = true) then begin
10        i := 1;
11        fb := true;
12        while (i ≤ 10) do begin
13            if (b[i] ≠ target) then
14                fb := false;
15            i := i + 1;
          end;
      end;
16    if (fb = true) then
17        writeln ('message 1')
18    else writeln ('message 2');
end.
```

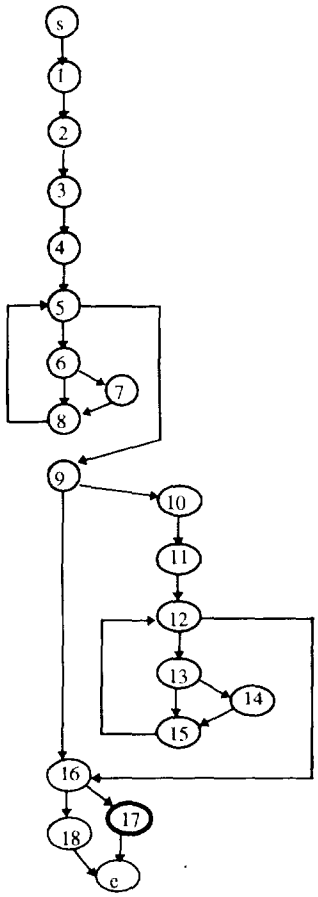


Fig. 1. A sample program and the corresponding flow graph.

the chaining approach may significantly increase the chances of finding test data as compared to the existing methods of test data generation.

The organization of this article is as follows. Section 2 introduces basic concepts and notations. Section 3 overviews existing test data generation techniques. Section 4 presents the chaining approach. Section 5 describes the search process. Section 6 presents the results of an experimental study, and Section 7 discusses future research.

2. BASIC CONCEPTS

A program structure is represented by a graph model. A *control flow graph* of a program 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, $s, e \in N$. A node corresponds to an assignment statement, an input or output statement, or the predicate of a conditional or loop statement, in which case it is called a *test node*. An *edge* $(n, m) \in A$ corresponds to a

possible transfer of control from node n to node m. An edge (n, m) is called a *branch* if n is a test node. Figure 1 shows a control flow graph of a sample program.

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 "$a[i]$ = target."

An input variable of a program is a variable which appears in an input statement, e.g., input(x). Let $I = (x_1, x_2, \ldots, x_n)$ be a vector of input variables of the program. 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 we mean a cross product, $D = D_{x_1} \times D_{x_2} \times \ldots \times D_{x_n}$, where each D_{x_i} is the domain for input variable x_i. A single point $\mathbf{x}$ in the n-dimensional input space D, $\mathbf{x} \in D$, is referred to as a program input $\mathbf{x}$.

A *path* P in a control flow graph from node n_{k_1} to node n_{k_q} is a sequence $P = \langle n_{k_1}, n_{k_2}, \ldots, n_{k_q} \rangle$ of nodes, such that for all i, $1 \leq i < q$, $(n_{k_i}, n_{k_{i+1}}) \in A$. For example, $P = \langle 3, 4, 5, 6, 8, 5, 9 \rangle$ is a path in the control flow graph of Figure 1 from node 3 to node 9. A path is *feasible* if there exists program input on which the path is traversed during program execution; otherwise the path is *infeasible*.

A *use* of variable $\mathbf{v}$ is a node in which a variable $\mathbf{v}$ is referenced. In particular, a use can be: (1) an assignment statement, (2) an output statement, (3) the predicate of a conditional or loop statement. A *definition* of variable $\mathbf{v}$ is a node which assigns a value to variable $\mathbf{v}$. In particular, a definition of variable $\mathbf{v}$ can be: (1) an assignment statement, (2) an input statement. For example, node 5 is a use of variable i, and node 7 is a definition of variable fa in the program of Figure 1. Let $U(n)$ be a set of variables whose values are used at node n, and let $D(n)$ be a set of variables whose values are defined at n.

A *definition-clear path from* n_{k_1} *to* n_{k_q} *with respect to variable* v is a path $\langle n_{k_1}, n_{k_2}, \ldots, n_{k_q} \rangle$ in the control flow graph that starts at node n_{k_1} and ends at node n_{k_q} and for all $1 < i < q$, $v \notin D(n_{k_i})$, i.e., variable v is not modified along the path. Let S be a set of variables. A *definition-clear path from* n_{k_1} *to* n_{k_q} *with respect to* S is a path $\langle n_{k_1}, n_{k_2}, \ldots, n_{k_q} \rangle$ in the control flow graph which starts at node n_{k_1} and ends at node n_{k_q} and for all i, $1 < i < q$, $(D(n_{k_i}) \cap S) = \emptyset$—i.e., none of the variables from S are modified along the path. For example, path $\langle 11, 12, 13, 15, 12, 16 \rangle$ is a definition-clear path with respect to a set of variables $\{fa, fb\}$.

Control dependence captures the dependence between test nodes and nodes that have been chosen to be executed by these test nodes. The control dependency is defined as [Ferrante et al. 1987]: let Y and Z be two nodes and (Y, X) be a branch of Y. Node Z *postdominates* node Y iff Z is on every path from Y to the exit node e. Node Z postdominates branch (Y, X) iff Z is on every path from Y to the program exit node e through branch (Y, X). Z is *control dependent* on Y iff Z postdominates one of the branches of Y and Z does not postdominate Y. For example, in the program of Figure 1, node 7 is

control dependent on node 6 because node 7 postdominates branch (6, 7), and node 7 does not postdominate node 6.

3. EXISTING METHODS OF TEST DATA GENERATION

This section overviews the existing methods of automated test data generation. The test data generation problem in this article is defined as follows.

Given node g (referred to as the goal node) in a program, the objective is to find program input $\mathbf{x}$ on which node g will be executed.

There are three types of test data generation methods that could be applied for this problem: *random* test data generation [Bird and Munoz 1982], *path-oriented* test data generation [Boyer et al. 1975; Clarke 1978; Korel 1990a; Ramamoorthy et al. 1976], and *goal-oriented* test data generation [Korel 1990b; 1992].

The path-oriented approach reduces the problem of test data generation to a "path" problem—i.e., a program path is selected (automatically or by the user) which leads to node g. Then program input $\mathbf{x}$ that results in the execution of the selected path is derived. If the program input is not found for the selected path, a different path is selected which also leads to node g. This process is repeated until a program path is selected for which a program input $\mathbf{x}$ is found or when the designated resources have been exhausted, e.g., search time limit.

Two methods have been proposed to find program input $\mathbf{x}$ to execute the selected path P—namely, *symbolic execution* [Boyer et al. 1975; Clarke 1975; Howden 1977] and *execution-oriented test data generation* [Korel 1990a]. The symbolic execution is a program analysis method that generates an algebraic expression over the symbolic input values—i.e., symbolic execution generates path constraints—which consist of a set of equalities and inequalities on the program's input variables. These constraints must be satisfied for path P to be traversed. A number of algorithms have been used for solving these constraints [Clarke 1976; Ramamoorthy et al. 1976]. Symbolic execution is a promising approach; however, there are several problems that still need to be overcome. These weaknesses include difficulty with dynamic data structures, arrays, and the handling of procedures.

The alternative approach, referred to as an execution-oriented approach of test data generation, for finding program input for the selected path was proposed in Korel [1990a]. This approach is based on actual execution of a program under test and function minimization methods. In this approach, the goal of finding a program input is achieved by solving a sequence of subgoals, where function minimization methods [Gill and Murray 1974; Glass and Cooper 1965] are used to solve these subgoals. Since the approach is based on the actual execution, values of array indexes and pointers are known at each step of program execution, and this approach exploits this information in order to overcome some limitations of the methods based on symbolic execution.

As pointed out by DeMillo and Ince [DeMillo et al. 1987; Ince 1987], there are some weaknesses with the existing path-oriented test data generators;

those weaknesses are mainly associated with the path selection process. For the path selection process, the disadvantage of not knowing if the selected path is feasible makes the path-oriented approach of limited use for certain classes of programs. Significant computational effort can be wasted in analyzing these infeasible paths that are generated. One attempt to alleviate the infeasibility problem was reported by Clarke [1979]. In this technique, symbolic execution [Clarke 1976; Howden 1977] is used to detect, if possible, path infeasibility in early stages of path selection. Although this technique can 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 execution.

The path-oriented approach is probably best suited for programs with a relatively small number of paths to reach the selected node. However, for programs with complex control structures and usually an infinite number of paths to reach the selected node, the path-oriented approach may be inefficient because only control information is used in the process of path selection.

Other research in the path-oriented approach was offered by DeMillo and Offutt [1991; 1993] as part of their constraint-based testing. This is a fault-based method that uses algebraic constraints to generate test cases designed to find particular types of faults. The approach creates a path expression constraint for each statement in the program which describes all paths up to, but not including loops, that statement. Each path to the statement is represented as a disjunctive clause. Symbolic evaluation is used to represent constraints, and heuristics methods are used to solve these constraints. The constraint-based approach falls into the category of path-oriented methods of test data generation. As with other path-oriented approaches, this approach may have problems with selection of feasible paths for programs with complex control structures because this approach is not able to identify statements that affect the execution of the selected statement. Additionally, since only the first two array elements are treated as separate variables [DeMillo and Offutt 1993] during symbolic evaluation, the constraint-based approach may not efficiently generate test cases for certain classes of programs with arrays.

The goal-oriented approach of test data generation [Korel 1990b; 1992] differs from the path-oriented test data generation in that the path selection stage is eliminated. The approach starts by initially executing a program with arbitrary program input. When the program is executed, the program execution flow is monitored. During the program execution, the search procedure decides whether the execution should continue through the current branch or whether an alternative branch should be taken because, for example, the currently executed branch does not lead to the execution of the selected node. If an undesirable execution flow at the current branch is observed, then a real-valued function is associated with branch. Function minimization search algorithms are used to find automatically new input that will change the flow execution at this branch. The general idea of the goal-oriented approach is to concentrate only on branches which "influence" the execution of the goal

node g and to ignore branches which in no way influence the execution of node g. For instance, branch (5, 6) in Figure 1 does not influence the execution of node 11 because, assuming termination of loop 5–8, branch (5, 9) will always be executed. Therefore any execution within the loop is allowed. However, branch (9, 16) is an essential branch with respect to node 11 because if this branch is executed then node 11 cannot be reached. If branch (9, 16) is executed then the search procedure suspends the program execution and tries to identify a new input for which the alternative branch (9, 10) is taken. All branches in a program are classified into different categories. The search procedure [Korel 1990b; 1992] uses this classification during program execution to continue or to suspend the execution at the current branch. This branch classification [Korel 1990b; 1992] is based solely on flow graph information, and it is determined prior to the program execution.

4. GENERAL DESCRIPTION OF THE CHAINING APPROACH

In this section, we describe a new test data generation approach referred to as the *chaining approach*. The approach has been presented for the node problem; however, this approach can be used for other types of problems, e.g., the branch problem and the definition-use chain problem [Laski and Korel 1983; Rapps and Weyuker 1985]. The chaining approach is an extension of the goal-oriented approach [Korel 1990b; 1992]. The main limitation of the path- and goal-oriented test data generation methods is that only the control flow graph is used to guide the search process. This limited amount of information may make some nodes very difficult to reach. For example, prior to reaching node 17 in the program of Figure 1, node 7 must be executed, and node 14 cannot be executed. The path-oriented method is not able to identify that node 7 affects execution of node 17 and that node 14 should be omitted during execution, because only a control flow graph is used in the process of path selection. As a result, the path-oriented approach may generate a large number of paths before the "right" path is selected, because the path selection process "blindly" generates paths to node 17. Similarly, the goal-oriented approach identifies that all nodes from 1 to 15 do not affect the execution of node 17. As a result, the goal-oriented approach, in most cases, will fail to reach node 17 (notice that only when one of the elements of array a equals the value of *target* and all elements of array b equal to the value of *target* will node 17 be executed).

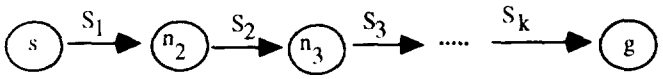
The chaining approach uses data dependency analysis to guide the test data generation process. The basic idea of the chaining approach is to identify a sequence of nodes to be executed prior to execution of node g. The chaining approach uses the following data dependency concepts to identify such a sequence:

Let p be a node and v a variable used in p. By a *last definition* n of variable v at node p we mean a node which satisfies the following conditions: (1) $v \in D(n)$, (2) $v \in U(p)$, and (3) there exists a definition-clear path of v from n to p. The last definition is a node that assigns a value to

variable v, and this value may potentially be used by node p. For example, node 4 is a last definition of variable *fb* at node 16 in Figure 1. By a *set of all last definitions* of node p, represented by $LD(p)$, we mean a set of all last definitions of all variables used in p. For example, the set of all last definitions of node 16 in the program of Figure 1 is $LD(16) = \{4, 11, 14\}$.

The chaining approach starts by executing a program for an arbitrary program input **x**. During program execution for each executed branch (p, q), the search process, which is described in more detail in Section 5, decides whether the execution should continue through this branch or whether an alternative branch should be taken (because, for instance, the current branch does not lead to goal node g). If an undesirable execution flow at the current branch (p, q) is observed, then a real-valued function (defined in Section 5) is associated with this branch. Function minimization search algorithms are used to find automatically new input that will change the flow execution at this branch. If, at this point, the search process cannot find program input **x** to change the flow of execution at branch (p, q), then the chaining approach attempts to alter the flow at node p by identifying nodes that have to be executed prior to reaching node p. As a result, the alternative branch at p may be executed. It is important to note that the goal-oriented approach would have declared failure of the search at this point [Korel 1990b; 1992]. We will refer to node p of the branch (p, q) as a *problem* node. The chaining approach finds a set $LD(p)$ of last definitions of all variables used at problem node p. By requiring that these nodes be executed prior to the execution of problem node p, the chances of altering the flow execution at problem node p may be increased. Such a sequence of nodes to be executed is referred to as an *event sequence* and is used to control the execution of the program by the chaining approach.

Example 4.1. Suppose that node 11 is a goal node in the program of Figure 1. The chaining approach starts by executing the program for an arbitrary input **x**, e.g., $a = (27, 58, 78, 4, 89, 21, 54, 85, 35, 96)$, $b = (45, 99, 6, 45, 2, 63, 28, 15, 94, 22)$, target = 56. At node 9 the flow of execution proceeds down the false branch (9, 16). At this point execution is suspended, since branch (9, 16) does not lead to node 11. The search process tries to identify a new program input to change the flow of control at node 9. Assuming the search process cannot find a new program input **x** to alter the flow at node 9, the chaining approach identifies node 9 as a problem node and finds a set of last definitions of node 9, $LD(9) = \{3, 7\}$, to determine nodes that are to be executed prior to reaching problem node 9. Since node 3 already was executed before node 9, node 7 is selected. The chaining approach requires now that node 7 is reached first, then problem node 9, and finally node 11. In order to reach node 7, any path from the start node s to node 7 can be traversed; after node 7 is reached, only those paths can be traversed that do not modify variable *fa* before node 9 is reached. This sequence of nodes will cause the change of execution at node 9 and the execution of node 11. The following program input will cause that the specified sequence of nodes is traversed: $a = (56, 58, 78, 4, 89, 21, 54, 85, 35, 96)$, $b = (45, 99, 6, 45, 2, 63, 28, 15, 94, 22)$, target = 56. Using the last



$E = \langle (s, S_1), (n_2, S_2), \ldots, (n_k, S_k), (g, \emptyset) \rangle$.

Fig. 2. A graphical representation of event sequence E.

definitions of problem nodes to identify a sequence of node executions may significantly increase the chances of finding program inputs to execute the selected nodes. For this particular example the chaining approach will always find input data to execute node 11 regardless of the initial values of input variables.

4.1 AN EVENT SEQUENCE

In this section we present the concept of an event sequence. Event sequences are generated by the chaining approach and then used to "guide" the execution of the program during the search process. Informally, by an *event* we mean the fact that a node is executed. Similarly, an *event sequence* refers to a sequence of executed nodes. More formally:

—An *event sequence* E is a sequence $\langle e_1, e_2, \ldots, e_k \rangle$ of events where each *event* is a tuple $e_i = (n_i, S_i)$ where $n_i \in N$ is a node and S_i a set of variables referred to as a *constraint set*. For every two adjacent events, $e_i = (n_i, S_i)$ and $e_{i+1} = (n_{i+1}, S_{i+1})$ there exists a definition-clear path with respect to S_i from n_i to n_{i+1}.

The event sequence (also referred to as a *chain*) identifies a sequence of nodes that are to be executed during program execution. A constraint set associated with each event $e_i = (n_i, S_i)$ identifies the constraints imposed on the execution from a given node n_i to node n_{i+1} of the next event e_{i+1} in the event sequence. It is required that all variables in the constraint set are not modified during program execution between node n_i and node n_{i+1}. For example, the following is an event sequence $E = \langle (s, \emptyset), (7, \{fa\}), (9, \emptyset), (11, \emptyset) \rangle$ in the program of Figure 1. There are four events: $e_1 = (s, \emptyset)$, $e_2 = (7, \{fa\})$, $e_3 = (9, \emptyset)$, and $e_4 = (11, \emptyset)$. This event sequence requires that the start node s is first executed, followed by the execution of node 7, followed by the execution of node 9, and finally execution of node 11. A constraint is imposed on the execution between nodes 7 and 9; it is required that the value of fa is not modified during program execution between nodes 7 and 9. However, there are no constraints imposed on the execution between nodes s and 7 and between nodes 9 and 11—i.e., the execution can follow any path between nodes s and 7 and between nodes 9 and 11. Graphical representation of a "general" event sequence is shown in Figure 2. A graphical representation of the event sequence $E = \langle (s, \emptyset), (7, \{fa\}), (9, \emptyset), (11, \emptyset) \rangle$ for the program of Figure 1 is shown in Figure 3.

An event sequence is *feasible* if there exists a program input on which the event sequence is traversed. For example, the event sequence $E = \langle (s, \emptyset), (7, \{fa\}), (9, \emptyset), (11, \emptyset) \rangle$ is feasible because this sequence will be traversed during program execution on the following input: $a = (56, 58, 78, 4, 89, 21,$

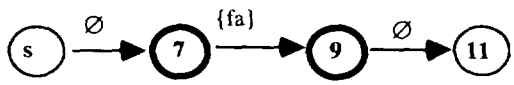


Fig. 3. A graphical representation of event sequence $E = \langle (s, \emptyset), (7, \{fa\}), (9, \emptyset), (11, \emptyset) \rangle$.

54, 85, 35, 96), b = (45, 99, 6, 45, 2, 63, 28, 15, 94, 22), target = 56. On the other hand, event sequence $\langle (s, \emptyset), (3, \{fa\}), (9, \emptyset), (11, \emptyset) \rangle$ is not feasible because there is no input to traverse this sequence.

4.2 GENERATING EVENT SEQUENCES

In this section we describe the process of generating event sequences during the search process. Initially, for the given goal node g, the following initial event sequence is generated: $E_0 = \langle (s, \emptyset), (g, \emptyset) \rangle$. Suppose that during the search process a problem node p is encountered—i.e., a test node at which the execution should be changed, but the searching procedure is not able to find a program input that will alter execution at p (the search process is described in Section 5). In this case, the chaining approach finds a set of last definitions $LD(p)$ of problem node p, $LD(p) = (d_1, d_2, \ldots, d_N)$. This set is used to generate N event sequences. Each newly generated event sequence contains an event associated with problem node p and an event associated with last definition d_i. The following event sequences are generated:

$$E_1 = \langle (s, \emptyset), (d_1, D(d_1)), (p, \emptyset), (g, \emptyset) \rangle$$

$$E_2 = \langle (s, \emptyset), (d_2, D(d_2)), (p, \emptyset), (g, \emptyset) \rangle$$

$$\ldots$$

$$E_N = \langle (s, \emptyset), (d_N, D(d_N)), (p, \emptyset), (g, \emptyset) \rangle$$

For the sake of presentation we assume that each definition modifies only one variable. As a result, a constraint set associated with each last definition d_i in E_i is a one-element set $D(d_i)$ that requires that a variable defined in d_i is not modified between d_i and p.

The chaining approach selects one of the event sequences from the list—for example, E_1—and tries to find program input on which event sequence E_1 is traversed. If during this process a new problem node p_1 is encountered, e.g., between the start node s and d_1, the chaining approach determines a set of last definitions for p_1, $LD(p_1) = (f_1, f_2, \ldots, f_M)$, and generates M new event sequences: $E_{1_1}, E_{1_2}, \ldots, E_{1_M}$. These event sequences are constructed from E_1 by inserting two events to E_1: one related to p_1 and the second related to f_i. The following event sequences are generated:

$$E_{1_1} = \langle (s, \emptyset), (f_1, D(f_1)), (p_1, \emptyset), (d_1, D(d_1)), (p, \emptyset), (g, \emptyset) \rangle$$

$$\ldots$$

$$E_{1_M} = \langle (s, \emptyset), (f_M, D(f_M)), (p_1, \emptyset), (d_1, D(d_1)), (p, \emptyset), (g, \emptyset) \rangle$$

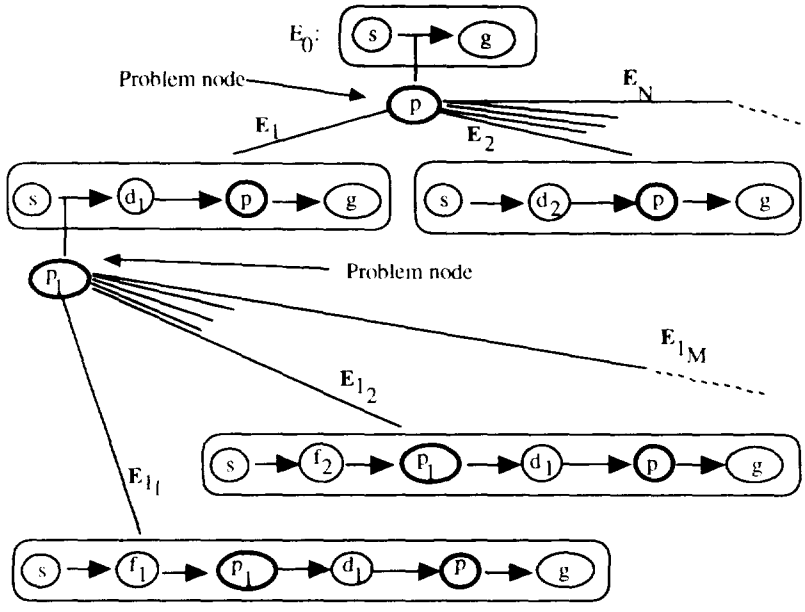


Fig. 4. A search tree generated by the chaining approach.

The chaining approach selects the next event sequence from the set of already generated sequences and tries to find program input on which this event sequence is traversed. If such an input is found then the input to execute the goal is also found. If the input is not found then new event sequences may be generated, and the chaining approach selects the next event sequence to explore.

The process of generating event sequences may be organized in a form of a tree referred to as a *search tree*. The initial event sequence E_0 represents the root of the tree. Each event sequence E_i derived from E_0 (by using a set of last definitions $LD(p) = (d_1, d_2, \ldots, d_N)$ and problem node p) is a child of E_0. Similarly, for each event sequence E_i new event sequences $E_{i_1}, E_{i_2}, \ldots, E_{i_M}$ can be generated that include a new problem node p_1 and last definitions of p_1, $LD(p_1) = (f_1, f_2, \ldots, f_M)$. These new event sequences $E_{i_1}, E_{i_2}, \ldots, E_{i_M}$ are children of E_i. This process of generating event sequences creates the tree-like structure shown in Figure 4.

The chaining approach traverses the search tree in a depth-first search traversal, attempting to find an event sequence E for which a program input is found that "executes" the selected event sequence. For example, after the search process fails to find program input for E_1 (shown in Figure 4) it generates new event sequences $E_{1_1}, E_{1_2}, \ldots, E_{1_M}$. The chaining approach explores then E_{1_1} to E_{1_M} (under the assumption that no new event sequences are generated). Only after all children of E_1 are explored, the search process explores E_2. This traversal process continues until the search tree has been completely traversed or until the designated search resources are exhausted, e.g., time limit.

In order to control the "depth" of the search, a limit may be imposed on the level of the search. For example, event sequences $E_1, E_2, \ldots, E_N$ shown in Figure 4 are on the first level, whereas event sequences $E_{1_1}, E_{1_2}, \ldots, E_{1_M}$ are in the second level. If the search limit is imposed on the first level, then event sequences $E_1, E_2, \ldots, E_N$ are only explored. The event sequences on the second level are not even generated. Let k be a search level that determines the level of generation of event sequences in the search tree. Let us assume that each test node (potential problem node) has N last definitions. Further, let us assume that a particular problem node can occur only once in the event sequence. In this case the search tree would contain N^K event sequences which could be used to search for an input for the goal node. Notice that the goal-oriented approach of test generation [Korel 1990b; 1992] is equivalent to the chaining approach with the level-0 search limit.

We now show more formally as to how a new event sequence is generated from a given sequence E. Let $E = \langle e_1, e_2, \ldots, e_{i-1}, e_i, e_{i+1}, \ldots, e_m \rangle$ be an event sequence. Suppose (1) that the execution partially traversed the event sequence up to event e_i and (2) that a problem node p is encountered between events e_i and e_{i+1}. Let d be a last definition of problem node p. A new event sequence is generated from E by inserting two events into sequence: $e_d = (d, D(d))$ and $e_p = (p, \emptyset)$. Event e_p is always inserted between events e_i and e_{i+1}; however, event e_d, in general, may be inserted in any position between e_1 and e_p. Suppose that e_d is inserted between events e_k and e_{k+1}. The following event sequence is then generated:

$$E' = \langle e_1, e_2, \ldots, e_{k-1}, e_k, e_d, e_{k+1}, \ldots, e_{i-1}, e_i, e_p, e_{i+1}, \ldots, e_m \rangle.$$

Insertion of new events into the sequence requires modifications of the corresponding constraint sets associated with each event. This update is done in three steps:

(1) $S_d = S_k \cup D(d)$
(2) $S_p = S_i$
(3) for all j, $k + 1 \leq j \leq i$, $S_j = S_j \cup D(d)$

The constraint set S_d of e_d equals the constraint set of the preceding event e_k and a variable defined by d. The constraint set S_p of e_p equals the constraint set of e_i. Finally in step (3), all constraint sets between e_{k+1} and e_i are modified by including a variable defined at d.

Example 4.2.1. Suppose that node 17 is a goal node in the program of Figure 1. The chaining approach generates the following initial event sequence $E_0 = \langle (s, \emptyset), (17, \emptyset) \rangle$ and starts the search by executing the program for an arbitrary input **x**, e.g., a = (27, 58, 78, 4, 89, 21, 54, 85, 35, 96), b = (45, 99, 6, 45, 2, 63, 28, 15, 94, 22), target = 56. The execution starts at the entry node s and proceeds to node 16. At node 16 the flow of execution proceeds down the false branch (16, 18). At this point execution is suspended, since branch (16, 18) does not lead to node 17. The search

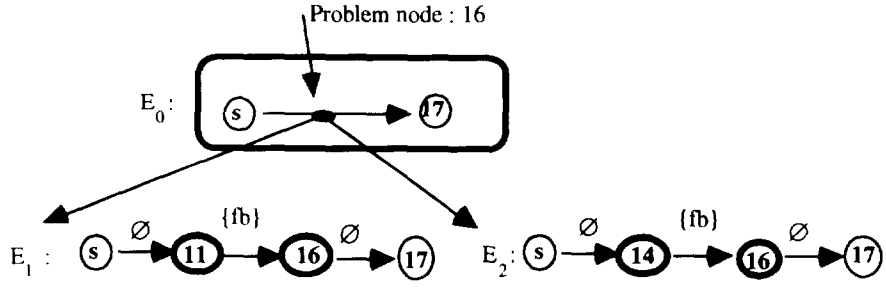


Fig. 5. A search tree generated in Example 4.2.1.

process tries to identify a new program input; however, in most cases it will fail. Node 16 is the problem node; therefore, the set of last definitions of node 16 is identified, $LD(16) = \{4, 11, 14\}$. The chaining approach uses this set to generate new event sequences. The following sequences are generated and are graphically shown in Figure 5:

$$E_1 = \langle (s, \varnothing), (11, \{fb\}), (16, \varnothing), (17, \varnothing) \rangle$$

$$E_2 = \langle (s, \varnothing), (14, \{fb\}), (16, \varnothing), (17, \varnothing) \rangle$$

Notice that event sequence $\langle (s, \varnothing), (4, \{fb\}), (16, \varnothing), (17, \varnothing) \rangle$ is not generated because this was the event sequence for which the search was not successful. The chaining approach selects the first event sequence E_1. During the search a new problem node is found at node 9. The set of last definitions of node 9 is identified $LD(9) = \{3, 7\}$, and two new event sequences are generated:

$$E_{1_1} = \langle (s, \varnothing), (3, \{fa\}), (9, \varnothing), (11, \{fb\}), (16, \varnothing), (17, \varnothing) \rangle$$

$$E_{1_2} = \langle (s, \varnothing), (7, \{fa\}), (9, \varnothing), (11, \{fb\}), (16, \varnothing), (17, \varnothing) \rangle$$

The chaining approach selects E_{1_1}; however, the search fails, and no new event sequences are generated. When event sequence E_{1_2} is selected, the chaining approach finds an input on which E_{1_2} is traversed; on this input goal node 17 is also executed. The following input may be generated: $a = (56, 58, 78, 4, 89, 21, 54, 85, 35, 96)$, $b = (56, 56, 56, 56, 56, 56, 56, 56, 56, 56)$, target = 56. The search tree generated for this example is shown in Figure 6.

5. THE SEARCH PROCESS

In this section we describe the process of finding input data to "traverse" a given event sequence. The problem is stated as follows:

—Given an event sequence $E = \langle e_1, e_2, \ldots, e_k \rangle$, the goal is to find program input **x** on which event sequence E is traversed.

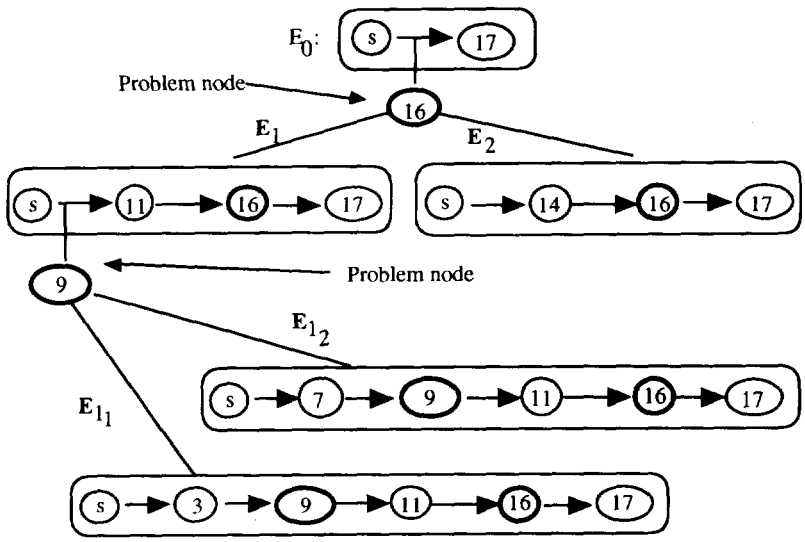


Fig. 6. The entire search tree for Example 4.2.1.

The search starts by initially executing a program for an arbitrary program input. During the program execution for each executed branch, the search procedure decides whether the execution should continue through this branch or whether an alternative branch should be taken. In the latter case a new input must be found to change the flow of execution at the current branch. For this purpose every branch in the program is classified into different categories. The branch classification is determined prior to the program execution. The search process uses this classification during program execution to continue or suspend the execution at the current branch. In the latter case, the search procedure determines a new program input.

Branch Classification

For every two adjacent events $e_i = (n_i, S_i)$ and $e_{i+1} = (n_{i+1}, S_{i+1})$ in the event sequence E a branch classification is computed. The branch classification relates to the situation that event e_i has already occurred (node n_i was executed), and now the goal is to reach node n_{i+1} without modifying any variable in S_i.

Critical Branch. A branch (p, q) is called a critical branch with respect to events e_i and e_{i+1} iff (1) there does not exist a definition-clear path from p to n_{i+1} or n_i with respect to S_i through branch (p, q) and (2) there exists a definition-clear path from p to n_{i+1} with respect to S_i through the alternative branch of (p, q).

If a critical branch (p, q) is executed, then program execution is suspended because the execution cannot lead to n_{i+1} or because the path is not a definition-clear path from q to n_{i+1} or n_i with respect to S_i. Notice that node n_i can be executed several times before reaching node n_{i+1}. The search

algorithm attempts to find a program input **x** which will change the flow of execution at critical branch (p, q); as a result, an alternative branch of (p, q) is executed. If a program input is found, the execution continues through the alternative branch. However, if program input **x** cannot be found, then the search is suspended, and node p is "declared" as a problem node (for which new event sequences may be generated). For instance, for two adjacent events $(11, \{fb\})$ and $(16, \emptyset)$ in some event sequence for the program of Figure 1, branch (13, 14) is critical because once this branch is executed the value of fb is modified in node 14. In this case, a new program input must be found on which alternative branch (13, 15) is taken.

Semicritical Branch. A branch (p, q) is called a semicritical branch with respect to events e_i and e_{i+1} iff (1) (p, q) is not a critical branch, (2) n_{i+1} is control dependent on test node p, and (3) there does not exist an acyclic definition-clear path from p to n_{i+1} with respect to S_i through branch (p, q).

Recall that the control dependence has been described in Section 2. If a semicritical branch (p, q) is executed, the program execution is terminated, and the search algorithm tries to find a new program input **x** to change the flow at this branch (p, q). If the search for a new program input is unsuccessful, then the program execution is allowed to continue through branch (p, q), hoping that on the next execution of test node p the alternative branch will be taken. For instance, branch (12, 13) is a semicritical branch with respect to events $(11, \{fb\})$ and $(16, \emptyset)$ because to reach node 16 the program has to iterate loop 12–15 and reach again node 12 without modifying fb. Clearly, if a semicritical branch is followed, the execution has to iterate, and there is a "danger" of executing a critical branch; it is obvious that the alternative branch of the semicritical branch is more "promising" at this point.

The remaining branches are classified as nonessential. If a nonessential branch is executed, the program execution is allowed to continue through this branch.

Example 5.1. Given the following event sequence:

$$E_{2_2} = \langle (s, \emptyset), (7, \{fa\}), (9, \emptyset), (11, \{fb\}), (16, \emptyset), (17, \emptyset) \rangle$$

for the program of Figure 1, the following branch classification is computed:

	Pair of events	Critical branches	Semicritical branches
1:	$(s, \emptyset), (7, \{fa\})$	(5, 9)	(6, 8)
2:	$(7, \{fa\}), (9, \emptyset)$	—	(5, 6)
3:	$(9, \emptyset), (11, \{fb\})$	(9, 16)	—
4:	$(11, \{fb\}), (16, \emptyset)$	(13, 14)	(12, 13)
5:	$(16, \emptyset), (17, \emptyset)$	(16, 18)	—

This branch classification is used to control the program execution. At the beginning of program execution classification 1 is used until node 7 is executed. Classification 2 is then used until node 9 is reached. The search procedure then uses classification 3 until node 11 is reached, and so on. Each time when an "event" occurs, then the next classification is used to control program execution.

Branch Predicate	Function F	*rel*
$A_1 > A_2$	$A_2 - A_1$	$<$
$A_1 \geq A_2$	$A_2 - A_1$	$\leq$
$A_1 < A_2$	$A_1 - A_2$	$<$
$A_1 \leq A_2$	$A_1 - A_2$	$\leq$
$A_1 = A_2$	$\text{abs}(A_1 - A_2)$	$=$
$A_1 \neq A_2$	$-\text{abs}(A_1 - A_2)$	$<$

Fig. 7. Branch functions of arithmetic predicates.

Finding Input Data

The problem of finding input data is reduced to a sequence of subgoals where each subgoal is solved using function minimization search techniques that use branch predicates to guide the search process. Each branch (p, q) in the flowgraph is labeled by a predicate, referred to as a *branch predicate*, describing the conditions under which the branch will be traversed. There are two types of branch predicates: Boolean and arithmetic predicates. Boolean predicates involve Boolean variables. On the other hand, arithmetic predicates are of the following form: A_1 *op* A_2, where A_1 and A_2 are arithmetic expressions, and *op* is one of $\{<, \leq, >, \geq, =, \neq\}$. For the sake of presentation, we assume that the arithmetic branch predicates are simple relational expressions (inequalities and equalities).

Each branch predicate A_1 *op* A_2 can be transformed to the equivalent predicate of the form F *rel* 0, where F and *rel* are given in Figure 7. F is a real-valued function, referred to as a *branch function*, which is (1) positive (or zero if *rel* is $<$) when a branch predicate is false or (2) negative (or zero if *rel* is $=$ or $\leq$) when the branch predicate is true. It is obvious that F is actually a function of program input $\mathbf{x}$. The branch function is evaluated for any program input by executing the program. For instance, the true branch of a test statement "if $a[i]$ = target then . . ." (node 6 in Figure 1) has a branch function F, whose value can be computed for a given input by executing the program and evaluating the "*abs* ($a[i]$ - target)" expression.

Let $E = \langle e_1, e_2, \ldots, e_m \rangle$ be an event sequence. The goal is to find program input $\mathbf{x}$ on which E will be traversed. The goal of finding a program input $\mathbf{x}$ is achieved by solving a sequence of subgoals. Let $\mathbf{x}^0$ be the initial program input (selected randomly) on which the program is executed. If E is traversed, $\mathbf{x}^0$ is the solution to the test data generation problem. Suppose, however, that a critical or semicritical branch (p,q) was executed between events e_i and e_{i+1}—i.e., the event sequence was partially traversed up to event e_i, and the critical or semicritical branch was encountered when the execution was supposed to reach node n_{i+1}. Let $E' = \langle e_1, e_2, \ldots, e_i \rangle$ be an event subsequence of E that was successfully traversed. In this case we have to solve the first subgoal. Let (p, t) be an alternative branch of branch (p, q). There are two possible cases depending on the type of branch predicate of (p, t).

If a branch predicate of (p, t) is of a Boolean type then the search is terminated, and node p is declared as a problem node for the critical

branch; in the case of a semicritical branch the execution continues through branch (p, q).

On the other hand, if the branch predicate of (p, t) is of an arithmetic type then the following procedure is used: let $F_1(\mathbf{x})$ be a branch function of branch (p, t). The first subgoal is to find a value of $\mathbf{x}$ which causes $F_1(\mathbf{x})$ to be negative (or zero) at node p; as a result, (p, t) will be successfully executed. More formally, we want to find a program input $\mathbf{x}$ satisfying $F_1(\mathbf{x})$ rel_i 0 subject to the constraint: $E' = \langle e_1, e_2, \ldots, e_i \rangle$ is traversed on $\mathbf{x}$, where rel_i is one of $\{=, <, \leq\}$.

This problem is similar to the minimization problem with constraints because the function $F_i(\mathbf{x})$ can be minimized using numerical techniques for constrained minimization, stopping when $F_i(\mathbf{x})$ becomes negative (or zero, depending on rel_i). Because of a lack of assumptions about the branch function and constraints, direct-search methods [Gill and Murray 1974; Glass and Cooper 1965] are used. The direct-search method progresses toward the minimum using a strategy based on the comparison of branch function values only. The simplest strategy of this form is that known as the *alternating-variable method*, which consists of minimizing the current branch function with respect to each input variable in turn. The search proceeds in this manner until all input variables are explored in turn. After completing such a cycle, the procedure continuously cycles around the input variables until either 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.

Once input $\mathbf{x}^1$ for the first subgoal is found, the program continues execution until either the whole event sequence is traversed or a new branch violation occurs at some other node. In the latter case, the second subgoal has to be solved. The process of solving subgoals is repeated until the solution $\mathbf{x}$ to the main goal node is found, or until no progress can be made at some node, in which case, the search process fails to find the solution for a given event sequence; in the latter case, this node is declared as a problem node, and new event sequences may be generated.

6. EXPERIMENTAL STUDY

The goal of the experiment was to compare the following methods of test data generation: random test data generation, path-oriented test data generation, goal-oriented test data generation, and the chaining approach of test data generation on two levels of search: level 1 and level 3. Notice that the goal-oriented approach is equivalent to the chaining approach with the level-0 search limit. In the path-oriented approach the execution-oriented method [Korel 1990a] was used for finding program input for selected paths.

The chaining approach has been developed as a part of the test data generation system TESTGEN [Korel 1989; 1995]. TESTGEN supports test data generation for programs written in a subset of Pascal. The following

execution-oriented test data generation methods are implemented: path-oriented, goal-oriented, and chaining methods.

Experiment

The experiment was performed on a personal computer with a 60MHz Pentium processor. In the experiment, 11 programs were used. For every program the following procedure was applied: for every node in the program, the goal was to find program input on which the selected node was executed.

Since initial data for the search were generated randomly, for some nodes the success of the search may depend on the initial input data. In order to reduce the element of "luck" during the search, for each node the search was repeated 10 times. Consequently, for each program the total number of tries was 10*number of nodes. For each try the limit of five minutes was imposed on the search—i.e., if the search was unsuccessful after five minutes the search was terminated and was considered unsuccessful. This procedure was used for every test data generation method except the random method.

In the random method, input data were generated randomly, and during program execution all executed nodes were marked as covered. The random approach was limited to five minutes; this corresponded to 200,000 to 700,000 executions (random tries) depending on the program. If at least one input was generated on which a particular node was executed, the search was considered successful for the node.

In the path-oriented test data generation the paths were generated according to the following procedure. Initially all acyclic paths to reach a selected node were generated. If the search was not successful then all paths with maximum two occurrences of any node were generated (this corresponds to one loop iteration). If the search was not successful then all paths with maximum three occurrences of any node were generated (two loop iterations). This process was repeated until either the solution was found or the search time limit was exhausted.

Measurements

During the experiment the following information was measured for every program:

(a) Success rate
(b) Coverage
(c) Average time of successful search
(d) Maximum time of successful search
(e) Average time of unsuccessful search
(f) Maximum time of unsuccessful search

Success rate SR is defined as follows:

$$SR = \frac{100\% * (\text{Number of successful tries})}{(m * \text{Number of program nodes})}$$

where $m = 10$ is the repetition factor for the search, i.e., in order to reduce the factor of "luck" during the search the experiment was repeated m times for each node.

Coverage represents the percentage of nodes for which at least one try was successful in finding input data during the experiment.

Programs

The experiments were performed for programs of different complexity. The following Pascal programs were used in the experiment: FORMAT, STACK, BANK, BIGBANK, BSEARCH, SAMPLE, BUBBLE, DAYS, FIND, GCD, and TRITYP. The source code of these programs can be found in Korel and Ferguson [1995]. The last five programs represent the Pascal version of the five Fortran programs used in the test data generation experiment for mutation testing described in DeMillo and Offutt [1991].

Program BANK maintains a bank account and supports simple transactions: open an account, deposit, withdraw, balance. Program STACK maintains stack operations like push, pop, empty, full. The input to program Stack are requests for stack operations. FORMAT performs text formatting within the selected column width (text is represented as an array of characters). BIGBANK is an extended version of the program BANK and supports transactions for different types of accounts. BSEARCH is a binary search program. SAMPLE is the program of Figure 1. BUBBLE performs array sorting using bubble sort. TRITYP accepts as input the relative lengths of the sides of a triangle and classifies the triangle as equilateral, isosceles, scalene, or illegal. DAYS calculates the number of days between the two given days. GCD computes the greatest common divisor of a sequence of numbers. FIND accepts an input array A of integers and an index F; it returns the array with every element to the left of $A[F]$ less than or equal to $A[F]$ and every element to the right of $A[F]$ greater or equal to $A[F]$.

Program	No. of nodes	No. of branches	No. of lines
BANK	85	50	155
STACK	40	22	78
FORMAT	77	42	137
BIGBANK	277	166	483
BSEARCH	18	10	34
SAMPLE	18	12	28
BUBBLE	13	6	26
TRITYP	49	46	76
DAYS	28	10	47
GCD	83	44	122
FIND	25	18	38

Results

SR:	Success Rate
C:	Coverage
ASuc:	Average Successful Search Time
Msuc:	Maximum Successful Search Time
AUnSuc:	Average Unsuccessful Search Time
MUnSuc:	Maximum Unsuccessful Search Time

Time was measured in seconds with the precision of 0.1 second.

BANK

	SR	C	ASuc	MSuc	AUnSuc	MUnSuc
Random		67%				
Path-oriented	63%	63%	0.1	0.5	300	300
Goal-oriented	72%	95%	0.3	2	0.3	1
Chaining (level 1)	92%	100%	0.7	6	4	9
Chaining (level 3)	98%	100%	1.5	16	12.5	13

FORMAT

	SR	C	ASuc	MSuc	AUnSuc	MUnSuc
Random		74%				
Path-oriented	61%	61%	0.1	0.1	300	300
Goal-oriented	84%	94%	0.2	1	0.4	5
Chaining (level 1)	91%	94%	0.6	19	13	22
Chaining (level 3)	94%	94%	0.7	47	56	93

BIGBANK

	SR	C	ASuc	MSuc	AUnSuc	MUnSuc
Random		68%				
Path-oriented	60%	60%	0.1	1	300	300
Goal-oriented	73%	85%	7	17	8	15
Chaining (level 1)	89%	97%	20	230	149	286
Chaining (level 3)	95%	99%	34	290	300	300

STACK

	SR	C	ASuc	MSuc	AUnSuc	MUnSuc
Random		79%				
Path-oriented	90%	90%	0.1	0.1	300	300
Goal-oriented	92%	97%	0.3	9	0.3	1
Chaining (level 1)	95%	100%	0.4	10	9.5	16
Chaining (level 3)	100%	100%	1	25	—	—

BSEARCH

	SR	C	ASuc	MSuc	AUnSuc	MUnSuc
Random		100%				
Path-oriented	100%	100%	0.1	0.1	—	—
Goal-oriented	94%	100%	0.1	0.1	0.1	0.1
Chaining (level 1)	100%	100%	0.1	0.1	—	—

SAMPLE

	SR	C	ASuc	MSuc	AUnSuc	MUnSuc
Random		95%				
Path-oriented	95%	95%	6	10	300	300
Goal-oriented	61%	61%	0.1	0.4	0.1	1
Chaining (level 1)	91%	95%	0.1	1	0.1	1
Chaining (level 3)	100%	100%	0.1	2	—	—

BUBBLE

	SR	C	ASuc	MSuc	AUnSuc	MUnSuc
Random		100%				
Path-oriented	100%	100%	0.1	0.1	—	—
Goal-oriented	100%	100%	0.1	0.1	—	—

TRITYP

	SR	C	ASuc	MSuc	AUnSuc	MUnSuc
Random		77%				
Path-oriented	98%	100%	15	280	300	300
Goal-oriented	26%	84%	0.1	1	0.1	1
Chaining (level 1)	61%	100%	0.6	2	1.1	3
Chaining (level 3)	99%	100%	1.3	17	5	11

DAYS

	SR	C	ASuc	MSuc	AUnSuc	MUnSuc
Random		100%				
Path-oriented	94%	100%	0.1	1	0.1	0.1
Goal-oriented	95%	100%	0.1	1	0.1	0.1
Chaining (level 1)	98%	100%	0.1	1	0.1	0.1
Chaining (level 3)	98%	100%	0.1	1	0.1	1

GCD

	SR	C	ASuc	MSuc	AUnSuc	MUnSuc
Random		100%				
Path-oriented	98%	100%	1.8	61	300	300
Goal-oriented	90%	95%	0.2	1	0.2	1
Chaining (level 1)	99%	100%	0.4	3	2.3	3
Chaining (level 3)	100%	100%	0.4	8	—	—

FIND

	SR	C	ASuc	MSuc	AUnSuc	MUnSuc
Random		100%				
Path-oriented	99.6%	100%	6	48	136	136
Goal-oriented	96%	100%	0.1	1	0.1	0.1
Chaining (level 1)	100%	100%	0.1	1	—	—

Conclusions of the Experiment

For small programs with relatively uncomplicated structure the random approach and path-oriented approach performed well. However, for programs with more-complex control structures and when execution of some statements depends on execution of other statements, these methods did not perform well in the experiment. The results of the experiment indicate that the chaining approach may increase the chances of generating test data. This is demonstrated by the increased value of the success rate for the chaining approach. Notice that the larger the value of the success rate, the larger the chances that a particular method may generate input for the selected node.

DeMillo and Offutt [1991] reported the results of the test data generation experiment for mutation testing using the constraint-based approach. The experiment was performed for five programs. We have created a Pascal version of these programs and used them in our experiment: BUBBLE, DAYS, FIND, GCD, and TRITYP. For these programs the path-oriented and chaining approaches did not have any major problems in generating program inputs to execute statements (even random testing was very successful for these programs, except program TRITYP). Since the constraint-based approach [DeMillo and Offutt 1991] is a path-oriented approach, it also did not have problems in generating program inputs for these programs. However, the programs used by DeMillo and Offutt in their experiment are quite simple with respect to generating program inputs. We believe that the constraint-based approach may not be efficient in generating program inputs for more-complex programs—for example, BANK, FORMAT, and BIGBANK, for which path-oriented methods did not perform very well in our experiment, as opposed to the chaining approach. For these programs the major challenge is to identify an executable path(s) among a large number of infeasible paths. Since the constraint-based approach uses only a control flow graph to identify paths, it may waste a lot of effort "blindly" exploring infeasible paths.

7. CONCLUSIONS

In this article, we have presented the chaining approach of test data generation that is an extension of the existing execution-oriented methods of test data generation. The existing techniques use only the control flow graph in the search process. The chaining approach uses data dependency analysis to guide the search process. As a result, the effectiveness of the process of test data generation may be improved.

The chaining approach was presented for the node problem. However, it may be also used for branch testing and data flow testing (definition-use chain coverage) [Laski and Korel 1983; Rapps and Weyuker 1985]. In order to use the chaining approach for branch and data flow coverage different initial event sequences are generated. Recall that for the node problem the following initial event sequence is generated: $E = \langle(s, \emptyset),(g, \emptyset)\rangle$, where g is a goal node. In branch testing the goal is to execute a selected branch (p, q),

and the following initial event sequence is generated: $E = \langle (s, \emptyset), (p, \emptyset), (q, \emptyset) \rangle$. In data flow testing the goal is to "evaluate" the definition-use pair (d,u), and the following initial event sequence is generated: $E = \langle (s, \emptyset), (d, D(d)), (u, \emptyset) \rangle$. When the initial event sequence is generated, the search process is identical to the search process used for the node problem.

We do not claim that the chaining approach is optimal. Instead, we open the way for new directions of research into test data generation methods based on dependency analysis. One possible direction of future research is to use program slicing [Korel and Laski 1988; Weiser 1982] in the process of test data generation. During test data generation, multiple executions of the program are required. Program slicing may improve the efficiency of the test data generation because in many cases a program slice may be executed rather than the whole program. This may improve the efficiency of test data generation for programs with expensive loops, i.e., loops with a large number of loop iterations.

REFERENCES

ALBERTS, D. 1976. The economics of software quality assurance. In *AFIPS Conference Proceedings: 1976 National Computer Conference*. Vol. 45. AFIPS Press, 433–442.

BIRD, D. AND MUNOZ, C. 1982. Automatic generation of random self-checking test cases. *IBM Syst. J. 22*, 3, 229–245.

BOYER, R., ELSPAS, B., AND LEVITT, K. 1975. SELECT—A formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not. 10*, 6 (June), 234–245.

CLARKE, L. 1976. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng. 2*, 3, 215–222.

CLARKE, L. 1979. Automatic test data selection techniques. Infotech State of the Art Report on Software Testing, Infotech International. Sept.

DE MILLO, R. AND OFFUTT, A. 1991. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng. 17*, 9, 900–910.

DEMILLO, R. AND OFFUTT, A. 1993. Experimental results from an automatic test data generator. *ACM Trans. Softw. Eng. Methodol. 2*, 9, 109–127.

DE MILLO, R., MCCRACKEN, W., MARTIN, R., AND PASSAFIUME, J. 1987. *Software Testing and Evaluation*. Benjamin/Cummings, Menlo Park, Calif.

FERRANTE, J., OTTENSTEIN, K., AND WARREN, J. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*, 5, 319–349.

GILL, P. AND MURRAY, W., Eds. 1974. *Numerical Methods for Constrained Optimization*. Academic, New York.

GLASS, H. AND COOPER, L. 1965. Sequential search: A method for solving constrained optimization problems. *J. ACM 12*, 1, 71–82.

HOWDEN, W. E. 1977. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Trans. Softw. Eng. 4*, 4, 266–278.

INCE, D. 1987. The automatic generation of test data. *Comput. J. 30*, 1, 63–69.

JESSOP, W., KANEM, J., ROY, S., AND SCANLON, J. 1976. ATLAS—An automated software testing system. *In Proceedings of the 2nd International Conference on Software Engineering*.

KOREL, B. 1989. TESTGEN—A structural test data generation system. In the *6th International Conference on Software Testing* (Washington, D.C.). USPDI, Washington, D.C.

KOREL, B. 1990a. Automated test data generation. *IEEE Trans. Softw. Eng. 16*, 8, 870–879.

KOREL, B. 1990b. A dynamic approach of automated test data generation. In the *Conference on Software Maintenance* (San Diego, Calif.). 311–317.

KOREL, B. 1992. Dynamic method for software test data generation. *J. Softw. Testing Verif. Reliab. 2*, 4, 203–213.

KOREL, B. 1995. TESTGEN—An execution-oriented test data generation system. Tech. Rep. TR-SE-95-01, Dept. of Computer Science, Illinois Inst. of Technology, Chicago.

KOREL, B. AND LASKI, J. 1988. Dynamic program slicing. *Inf. Process. Lett. 29,* 3, 155–163.

KOREL, B. AND FERGUSON, R. 1995. Chaining approach of test data generation—Experimental results. Tech. Rep. TR-SE-95-02, Dept. of Computer Science, Illinois Inst. of Technology, Chicago.

LASKI, J. AND KOREL, B. 1983. Data flow oriented program testing strategy. *IEEE Trans. Softw. Eng. 9,* 3, 347–354.

MYERS, G. 1979. *The Art of Software Testing.* John Wiley and Sons, New York.

RAMAMOORTHY, C., HO, S., AND CHEN, W. 1976. On the automated generation of program test data. *IEEE Trans. Softw. Eng. 2,* 4, 293–300.

RAPPS, S. AND WEYUKER, E. 1985. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng. SE-11,* 4, 367–375.

WEISER, M. 1982. Program slicing. *IEEE Trans. Softw. Eng. SE-10,* 4, 352–357.

Received December 1993; revised October 1994; accepted November 1995