Generating Test Data For Branch Coverage

Neelam Gupta
Dept. of Computer Science
The University of Arizona
Tucson, AZ 85721
ngupta@cs.arizona.edu

Aditya P. Mathur
Dept. of Computer Science
Purdue University
West Lafayette, IN 47907
apm@cs.purdue.edu

Mary Lou Soffa
Dept. of Computer Science
Univ. of Pittsburgh
Pittsburgh, PA 15260
soffa@cs.pitt.edu

Abstract

Branch coverage is an important criteria used during the structural testing of programs. In this paper, we present a new program execution based approach to generate input data that exercises a selected branch in a program. The test data generation is initiated with an arbitrarily chosen input from the input domain of the program. A new input is derived from the initial input in an attempt to force execution through any of the paths through the selected branch. The method dynamically switches among the paths that reach the branch by refining the input. Using a numerical iterative technique that attempts to generate an input to exercise the branch, it dynamically selects a path that offers less resistance. We have implemented the technique and present experimental results of its performance for some programs. Our results show that our method is feasible and practical.

Keywords - *Path testing, branch testing, iterative relaxation technique, testing tools.*

1 Introduction

An important technique used in testing of software is structural testing, in which a particular type of program element is selected for coverage. Path coverage is an example of structural testing, and requires exercising a fixed sequence of statements which constitute a path from the start of the program to a termination point. Other examples of structural testing include statement, branch, and data flow (e.g., definition to use) coverage, which require exercising a given program element using any path through that element. A critical problem during structural testing, and the focus of this paper, is to generate input data that exercises a selected element in the program.

In generating input data for path coverage, little flexibility is afforded the test case generator since the complete path is specified. Thus, given an input test case, if the path is not executed using this input, the only option is to try to change the input such that the selected path is executed.

However, for the coverage criteria that utilize only a single program element (e.g., statement or branch), there is more flexibility in that not only can the input data be modified but the different paths that reach the element can also be used when generating test input. Thus, there is more flexibility when generating test data for this type of coverage. Path selection is thus an important component of test case generation when a number of potential paths can be used to exercise a program element. In this paper, we focus on branch coverage, although the technique presented can also be used for statement or data flow coverage by dynamically selecting a path among the paths that exercise a given statement or a given definition-use pair.

There are a number of approaches that can be used to generate data that executes a branch. One approach is essentially to use the same technique as path coverage in that a particular path is chosen that goes through the selected branch and an attempt is made to generate test data for that path [11, 8, 1, 6]. If the effort is unsuccessful, another path is chosen, and the process continues until the branch is exercised or no more paths can be tried. The major drawback of this approach is the impact of infeasible paths. If an infeasible path is selected, then a significant computational effort could be spent before the path is abandoned and another one selected.

Another approach is not to select the path initially, but to dynamically generate the path by changing the branch outcome of one branch at a time, using some input. A previously developed technique that utilizes this approach uses function minimization to change the input, in an attempt to force execution toward the selected branch [10]. In [3], the authors improve upon the performance of their approach in [10] by using the data dependence analysis, in addition to the control flow information, to guide the test data generation process. But in both these approaches, the predicate where it is determined that control cannot reach the selected branch, a function associated with the branch predicate is minimized. The function minimization is used to change the input so that the predicate evaluates to a value that makes it possible for execution to reach the selected branch. The minimization in both these approaches is carried out with respect to one input variable at a time and continuously cycles through the relevant input variables, until a solution is reached or no progress can be made. If unsuccessful, the technique backtracks to the preceding branch predicate, and repeats the process by minimizing the function associated with that predicate. If the set of paths from the start node to the test branch contains a large number of branch predicates, this method may take an unacceptable large number of program executions.

A domain testing algorithm that combines the path selection process with test data generation is described in [5]. It uses the function minimization approach described in [11] more efficiently and requires selection of fewer inputs from the input domain. But still, the number of points selected from the input domain for each influencing variable, is a function of the maximum number of different values that can be assigned to the input variable, which can be very large in a general program.

In this paper, we present a new approach that generates input data for branch coverage by dynamically selecting a path in an attempt to exercise a test branch in a given program. Since a branch can be exercised by several paths in the program, path selection is an important problem in generating test data to exercise the test branch. Different paths can offer different resistance to a test data generation technique that attempts to force execution through them. We presented an iterative relaxation technique in [8] to generate test data for a given path. The new idea here is how we use the approach presented in [8] to guide the path selection process by dynamically switching execution to a path that offers less resistance to force execution to reach the given branch.

Our technique starts with a given test branch in a program and an arbitrarily chosen input in the input domain of the program. If the program is executed with the input and the test branch is not executed, a path that exercises the test branch is selected based on the outcomes of predicates called the search predicates. The arbitrarily chosen input is refined using a set of linear constraints that expresses the conditions for execution of this path. The program execution, with the modified input, may either exercise the test branch or switch the outcome of a search predicate on the current path, resulting in a new candidate path for the next iteration. Our algorithm compares the path resistance of the current path with that of the new path. If the new candidate path offers lesser resistance to test data generation, the new path is used for refining the input in the next iteration. Otherwise, the input is further refined using the current path. Therefore, the path selection in our approach is dynamic, guided by the execution behavior of the branch predicates on the selected path.

During this search, the paths that are detected as infeasible or identified as likely infeasible are removed from the set of paths being explored. The algorithm terminates when either an input is found that exercises the selected branch, or all the paths that reach the branch are identified as infeasible or likely to be infeasible.

The technique is general in that it can handle programs with branch predicates that compute general nonlinear functions of input. Since it is based on extension of the technique described in [8], it inherits all its properties. Therefore, it can handle programs with pointers, arrays and function calls. Using the Unified Numerical Approach (UNA) described in [9] to compute a feasible solution of a set of linear constraints, it can generate test data for a mix of real and integer inputs.

The organization of the paper is as follows. An overview of the approach is given in the next section. The algorithm for generating test data for a given branch is described and illustrated with an example in section 3. The implementation of the technique and experimental results are described in section 4. The related work is described in section 5 and the method is summarized in section 6.

2 Overview

In this paper, we address the problem of generating a program input I, in the input domain of a program, such that a given test branch B in the program is exercised when the program is executed using I. Our method uses the control flowgraph representation of the given program. Using the control flowgraph G of the program, we derive another flowgraph G' that precisely contains all the paths from the start node to the test branch B. This is essentially the **reachability graph** for the branch B and can be computed by a backward pass over the control flowgraph of the pro-

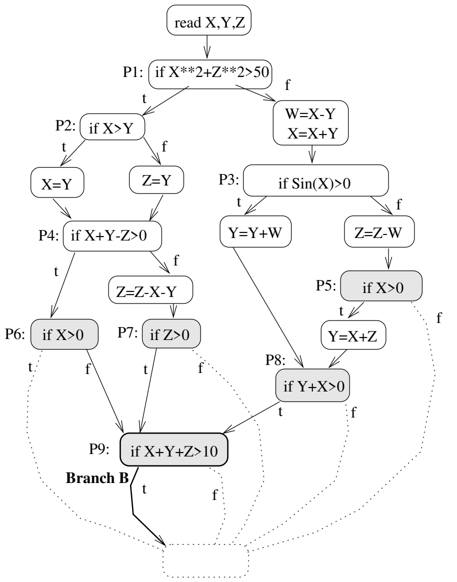

Figure 1. An example program flowgraph

gram. For example, in the flowgraph G of the program shown in Figure 1, the reachability graph G' derived for the test branch B is shown with solid edges. As shown in the figure, every branch predicate in G' can be classified as either a **search predicate** or a **critical predicate**.

Search Predicates

The branch predicates in G' that have all their outcomes to nodes that reside within G' are called the search predicates. The search predicates guide the search for a suitable path by switching between paths in G' during different iterations. A search predicate has the property that irrespective of its branch outcome, there exists a path in G' from the search node to B. For example, in the reachability graph G' for the test branch B in Figure 1, $P1, P2, P3$, and $P4$ are the search predicates.

Critical Predicates

The branch predicates in G' that have at least one branch outcome to a node not in G' are called the critical predicates. It is necessary that a critical predicate evaluates to a branch outcome to a node within G' for the program execution through the critical node to reach B. For example, $P5, P6, P7, P8$, and $P9$ are the critical predicates in the reachability graph G' for the test branch B in Figure 1. We now explain how these search and critical predicates are used in the path selection process for the test branch B.

Path Selection

Our test data generation algorithm is initiated with an arbitrarily chosen input. If this input does not exercise the test branch B, we derive a flowgraph G'' by ignoring the critical predicates in G'. The flowgraph G'' shown in Figure 2 is derived from the flowgraph G' in Figure 1 by ignoring the critical predicates in G'. The nodes in G'' correspond to either the search predicates or the unconditional statements in G'. Note that the critical predicate $P9$ is converted to an unconditional node in G''. If the flowgraph G'' is executed using the arbitrarily chosen input, the outcomes of search predicates will define a path P'' in G'' exercising B. This is always true since there are no critical predicates in G''. We select the corresponding path P in G' as our initial choice to force execution through B. We derive the set of linear constraints for the branch predicates on P using the method described in [8] and solve them by the Unified Numerical Approach described in [9] to obtain a refined input.

If the refined input does not exercise test branch B, then we execute G'' using the refined input. If a search predicate on P'' switches its outcome when G'' is executed using the refined input, then a new candidate path for next iteration is defined by this refined input. If the **path resistance** of this path, as defined below, is less than path resistance of P then the path is switched in the next iteration. Otherwise, the input is further refined in the next iteration with respect to the current path P. We define the path resistance of a path P as follows:

$$\mathbf{PathResistance(P)} = \frac{\mathbf{(m+1).n.e^r}}{\mathbf{Threshold - icp}}$$

- **m** is the number of times algorithm has switched from the current path P to a different path. Note that $m + 1$ is used to define a meaningful path resistance for a path that has not been switched so far.
- **n** is the total number of branch predicates on the path P.
- **r** is the total number of branch predicates on P that compute nonlinear functions of the input.
- **Threshold** is the maximum number of times the constraint set of a path is allowed to be inconsistent before it is classified as Likely Infeasible Path.
- **icp** is the number of times the constraint set for P has been found to be inconsistent.

It is clear that the path resistance increases as the number of predicates on a path increases. Also, if the algorithm has switched from P to another path a large

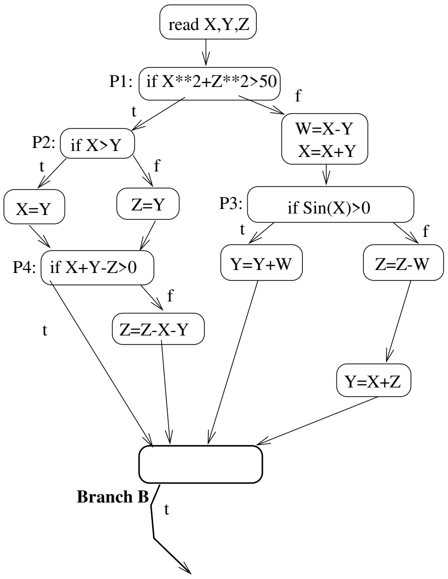

Figure 2. Derivation of G'' for the example

number of times then the path P is offering large resistance to generation of an input that exercises it. As shown in [8] if all the branch predicates on a path compute linear functions of the input, then either an input to exercise the path can be generated in one iteration or the path is guaranteed to be infeasible. Therefore, a path with all the predicates computing linear functions of input offers less resistance to iterative test data generation technique. For a path with branch predicates computing general nonlinear functions of input, a linear approximation to the nonlinear predicate function is derived in the neighborhood of the current input. To refine the current input, the constraints corresponding to the nonlinear predicates are derived from their linear approximations at the current input. Therefore, it may take several iterations to refine the current input to generate test data for path with branch predicates computing nonlinear functions of the input. Thus, the path resistance increases much faster with the increase in the number of branch predicates computing nonlinear functions of input as compared with the increase in total number of predicates on the path. A repeated occurrence of inconsistent set of constraints for a given path indicates that the path is offering more resistance to forcing execution through it. As a result, the path switching in our approach is guided by the complexity of predicates on the path as well as the execution behavior of the predicates on the path. We describe our test data generation algorithm in detail in the next section.

3 The Test Data Generation Algorithm

In this section, we present our algorithm to generate test data that exercises a given branch in a program with numeric input, arrays, assignments, conditionals and loops. In order to handle a mix of integer and real inputs, we use the Unified Numerical Approach described in [9] to compute a feasible solution of a set of linear constraints. The technique can be extended to nonnumeric input such as characters and strings by providing mappings between numeric and nonnumeric values. The main steps of the algorithm are outlined in Figure 3. We now describe the steps of our algorithm in detail.

The method begins with a test branch B in the flowgraph G of a given program and an arbitrarily chosen input I_0 in the input domain of the program. The program is executed on I_0. If B is exercised using I_0, then I_0 is the desired program input and the algorithm terminates; otherwise the algorithm dynamically searches for a path to B for which it can generate an input to exercise B.

Step 1: Construction of the flowgraph. Using the flowgraph G of the given program, we derive a flowgraph G' that precisely contains all the paths from the starting node of the program to the test branch B. This flowgraph is essentially the reachability subgraph from the start node to B. We derive G' by taking a backward pass over G starting at B. Let us consider the example in Figure 1. Let $I_0 = (1, 2, 3)$ be an arbitrarily chosen input from the input domain of the program. Executing the flowgraph G using I_0, we find that B is not executed. Therefore, we derive the flowgraph G' that precisely contains all the paths from the start node to B, shown by solid edges in Figure 1. The branch predicates $P1, P2, P3, P4$ are the search predicates and $P5, P6, P7, P8, P9$ are the critical predicates in G'.

We maintain a set called IP, which contains all the paths that have been identified as Infeasible Paths by the algorithm. We also maintain a set called LIP that contains all the paths that have been identified as Likely Infeasible Paths by the algorithm. The iteration counts $nIter$ and k are initialized to zero and the sets IP and LIP of infeasible and likely infeasible paths respectively are initialized to empty.

Step 2: Initial Path Selection. Construct the flowgraph G'' by ignoring the critical predicates in G'. Now, the only branch predicates in G'' are the search predicates. Hence, executing G'' using I_0 will identify a path in G'. We select this path as our initial path P. This path has the property that all the search predi-

GENERATE_BRANCH_TEST_DATA(G, B, I_0)
Input: The flowgraph G of a Program, a test Branch B, an initial program Input I_0
Output: A Program Input I_f on which B is executed

```
        begin
            If B executed on I_0 then return(I_0);
Step 1:     Construct a flowgraph G' that precisely contains all the paths to B.
            k=0; IP, LIP = {};
Step 2:     Let G'' be a flowgraph obtained from G' by removing the critical predicates.
            Let P be the path in G' corresponding to the path exercised by G'' using input I_0.
            icp(P) = 0; nIter = 0; return(INPUTGEN(I_0, P));
        end

        procedure INPUTGEN(I_k, P)
Step 3:     If nIter++ ≥ Limit then print("Stop Limit Reached"); return(); endif
            Derive S, the set of linear constraints on increments to I_k,
                from the desired branch outcomes of the branch predicates on P.
Step 4:     If S is inconsistent then
                If the inconsistent branch predicates are linear then
                    IP = IP ∪ {P}
                else
                    If ++icp(P) ≥ Threshold then
                        LIP = LIP ∪ {P}
                    endif
                endif
Step 5:         Derive a consistent subset of S and solve it to compute I_{k+1}.
            else
                Solve S to compute I_{k+1}.
            endif
            If B executed using I_{k+1} then return(I_{k+1}) endif
Step 6:     Let P'' be the path exercised by G'', using input I_{k+1},
                such that the corresponding path P'(in G') ∉ IP, LIP.
            If (no such P'' exists) then
                print("B unreachable"); return();
            endif
            If P ≠ P' then
                If PathResistance(P) ≤ PathResistance(P') then
                    return(INPUTGEN(I_{k+1}, P));
                else
                    return(INPUTGEN(I_{k+1}, P'));
                endif
            else
                return(INPUTGEN(I_{k+1}, P));
            endif
        endprocedure
```

Figure 3. Algorithm to generate test data to execute a given branch.

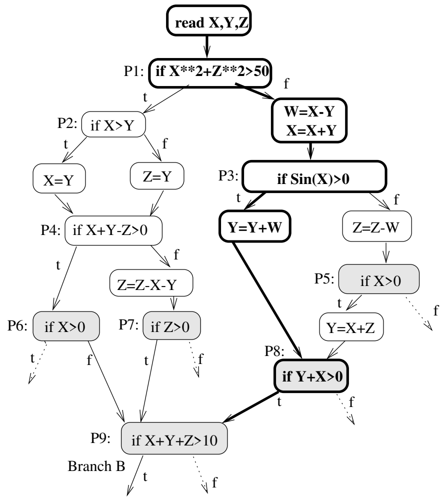

Figure 4. Input $I_0 = (1, 2, 3)$**.**

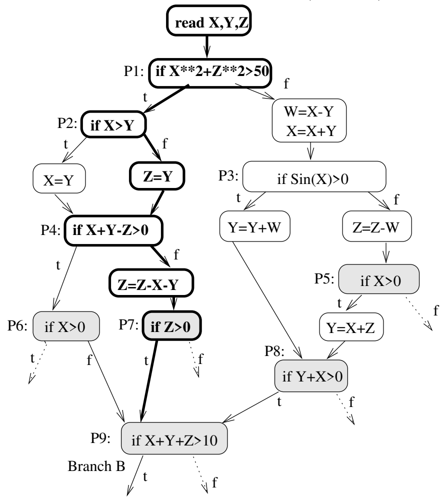

Figure 5. Input $I_1 = (-3, 6.1, 10)$**.**

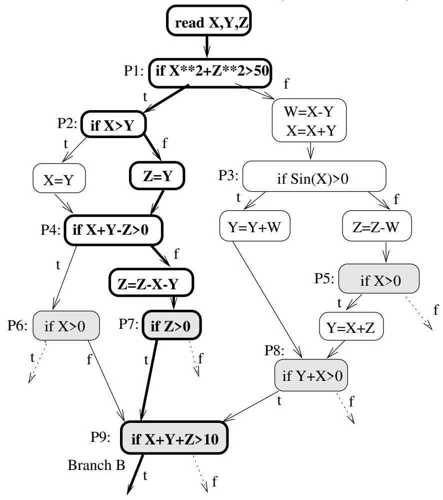

Figure 6. Input $I_2 = (-3, 10.4, 20)$**.**

cates in G' evaluate to their desired outcomes for this path using I_0. The infeasibility count icp of path P is initialized to zero. Then the procedure INPUTGEN to compute an input that exercises B is invoked with the arbitrarily chosen input I_0 and P . For our example in Figure 1, the flowgraph G'' is shown in Figure 2. Let $I_0 = (1, 2, 3)$ be the arbitrarily chosen input for this example. The path P selected by executing G'' using I_0 is:

$$P = \{read(X, Y, Z), P1, W = X - Y, X = X + Y,$$

$$P3, Y = Y + W, P8, P9\}$$

Step 3 : Derivation of linear constraints. In this step, if the iteration count has not reached its limit, we derive a linear constraint imposed on the increments to the current input for each branch predicate on P. The value of the function computed by each branch predicate on the current input determines by how much the current values of the variables in the input must be incremented so that the branch predicate evaluates, or continues to evaluate, to the desired branch outcome. A linear arithmetic representation is derived for the function of the input computed by each branch predicate on the path. If the function of the input computed by the branch predicate is linear, then this representation is exact; otherwise, it approximates the function by its tangent plane in the neighborhood of current input. For the details of computing linear representation for a branch predicate on path, please refer to [8]. Using the value of the function computed by a branch predicate at the current input and the linear arithmetic representation of the function computed by the branch predicate at the current input, a linear constraint on the increments to the current input is derived for each of the branch predicates on P. The linear constraints for the branch predicates on P in our example are:

$$P1 : 3\Delta X + 7\Delta Z \leq 40; \;\; P3 : -\Delta X - \Delta Y > -0.14$$
$$P8 : 2\Delta X + \Delta Y > -4; \;\; P9 : 2\Delta X + \Delta Y + \Delta Z > 3$$

Step 4: Detection of Infeasible paths. If the set of linear constraints derived in step 3 is inconsistent and the branch predicates corresponding to the inconsistent constraints compute linear functions of input, then the selected path is infeasible. Therefore, we add it to the set IP. If the branch predicates corresponding to the inconsistent constraints are nonlinear, then we increment the infeasibility count icp of this path. If the infeasibility count of the path reaches the $Threshold$, we conclude that the path is likely to be infeasible and add it to set LIP. The set of constraints in our example are consistent hence this step is not executed.

Step 5: Consistent subset of linear constraints. If the set of constraints computed in Step 3 are inconsistent, then remove the constraints, one at a time, for the branch outcomes closest to B until we obtain a consistent set of constraints. If the last constraint removed from the set corresponds to a search predicate, then a constraint for its alternate branch is added to the constraint set, and the search predicate is converted into an unconditional node. If the last constraint removed from the set corresponds to a critical predicate, then the constraint corresponding to the closest search predicate on the path is replaced by the constraint for its alternate branch outcome. The search predicate is converted into unconditional node and the constraints for other critical predicates between the search predicate and the above critical predicate are removed from the set. We check the resulting set of constraints for consistency and use the above method recursively to obtain a consistent subset of constraints. We then solve this subset of consistent constraints to obtain a feasible solution of the constraint set. This feasible solution generates a new candidate path for resistance comparison in Step 6. If the set of constraints obtained in Step 3 are consistent, then we solve them to obtain a feasible solution of the constraint set. The feasible solution gives increments for the current input to obtain the refined input. Considering our example, solving the consistent set of linear constraints obtained in Step 3, we obtain $\Delta X = -4$, $\Delta Y = 4.1$ and $\Delta Z = 7$. Thus, $I_1 = (-3, 6.1, 10)$. By executing the flowgraph G with I_1, we see that B is not exercised.

Step 6: Path Switching. In this step, we execute the flowgraph G'' using the refined input obtained in the previous step. Let P'' be the path defined by the outcomes of the search predicates in G''. If $P'' \in IP \ or \ LIP$, then we remove the branch on P'' of the farthest search predicate and convert the search predicate into an unconditional node. All the descendants of this search predicate, that were earlier converted to an unconditional node, are converted back to search nodes. We again execute the modified flowgraph G'' with the refined input obtained in the step 5. We repeat this process until a path $P'' \notin IP,\ LIP$ is obtained. If no such path exists, then we conclude that the test branch B is unreachable. Let P' be the path in G' corresponding to the path P'' in G''. If P' is not the same path as P then we compute the path resistance of P' as explained in the previous section and compare it with the path resistance of P. If the path resistance of P' is less than the path resistance of P, then the path to be explored in the next iteration is switched to P'. Otherwise, the input is further refined in the next iteration using the current path P. In our example, by executing G' using I_1, we see that the search predicate $P1$ changes its outcome and gives the following path P' as the candidate path to be used in the next iteration.

$$P' = \{read(X,Y,Z), P1, P2, Z = Y, P4,\\ Z = Z - X - Y, P7, P9\}$$

We compute the path resistance of P as well as P' in order to decide if we should switch the path to P' in the next iteration. Note that m is 0 for both the paths since none of the paths have been switched so far.

$$PathResistance(P) = \frac{1*4*e^2}{Threshold} = \frac{29.5}{Threshold}$$

$$PathResistance(P') = \frac{1*5*e^1}{Threshold} = \frac{13.5}{Threshold}$$

Thus, the path resistance of the path P' is less than the path resistance of the path P. Therefore, we switch the path and use P' to refine I_1 in the next iteration as shown in Figure 5 We derive the set of linear constraints for the branch outcomes of the branch predicates on the path P' given below:

$P1 : -5\Delta X + 21\Delta Z > -59$; $P2 : \Delta X - \Delta Y \leq 9.1$
$P4 : \Delta X \leq 3$; $P7 : -\Delta X > -3$
$P9 : \Delta Y > 3.9$

Solving the above set of constraints, we get $\Delta X = 0$, $\Delta Y = 4$ and $\Delta Z = 10$. Therefore, $I_2 = (-3, 10.4, 20)$. Executing G' on I_2, we see that B is exercised. Using I_2, the execution follows the path shown in bold in Figure 6 and the branch predicate $P9$ also also evaluates to "true". Therefore, I_2 is the desired input to exercise the test branch B in the program in Figure 1.

4 Experimental Results

We implemented our technique and generated numeric input data to demonstrate the feasibility of our approach. We considered scientific programs from [12] with integer and floating point inputs. We used Unified Numerical Approach (UNA) described in [9] to compute a feasible solution of the set of linear constraints in each iteration. We conducted experiments for exponential integral program called *expint* and random generator program called *gamdev*. The exponential integral program was selected because it has branch predicates computing linear and nonlinear functions of input. It also has a mix of integer and floating point inputs. The random generator program was selected because it makes function calls to another function called **ran1** and also has branch predicates computing nonlinear functions of input which make calls to *ran1*. We thought this was really a challenging program because the subsequent calls to *ran1* by predicate functions receive values that are not expected to have any pattern. We used the number of iterations and the time taken in seconds as a measure of the performance of the technique. We conducted experiments on Windows NT platform using a 400MHz Pentium II machine with 128MB RAM. The performance of our technique is summarized below:

Exponential Integral Program: This program called *expint* has 30 branches in total, out of which only 4 compute nonlinear functions of the input. All other branches compute linear functions of input along each of the path that exercises the branch. Two of these branches that compute linear function of input took 2 iterations each because the path that was selected in the first iteration was detected infeasible. Therefore, new candidate path was generated for the next iteration. The input generated in second iteration exercised the test branch. Other two branches compute a function of the index of a loop and a constant predefined in the program. When the iteration count was exceeded for these branches, the corresponding constraints were analyzed and it was found that these branches corresponded to exiting the program if the exponential integral did not converge in a predefined number of iterations. Therefore, to test these branches, the predefined number of executions of the respective loops were necessary. The test data for two of the linear branches that took one iteration each also exercised one nonlinear branch each. Test data to cover the remaining two nonlinear branches took 8 iterations and 5 iterations respectively. Their time performance is shown in Table 1. For both of these branches, the constraint set was found to be consistent in each of the iterations. Besides, the alternate candidate paths in each iteration offered more resistance to test data generation for this branch due to the presence of more nonlinear predicates. Hence, the path never switched from the initial path and the initial input got refined with respect to this path in subsequent iterations.

Random Number Generator Program: This program called *gamdev* has 12 branches in all. Three branches computing a linear function of input took only one iteration each to generate data. Test data for one of these linear branches also exercised 3 other linear branches not exercised so far. Test data for another linear branch exercised 4 branches computing nonlinear functions of the input. The time performance and the number of iteration taken by the remaining 2 branches that compute nonlinear functions of the input are shown in Table 1. In case of branch 1, the search predicate switched its outcome after first iteration. This generated a new candidate path for next iteration. But the resistance of the new path was more than the current path due to presence of two additional nonlinear predicates. Hence the path was not switched in second iteration. The refined input generated by the second iteration exercised the test branch. Our experiments illustrate that our technique is able to compute input data for test branches. In addition, the decision to switch the path is dynamic and meaningful. Its time performance indicates that the technique is practical in the experiments conducted so far. The experimental results also show that the technique can effectively handle infeasible paths in programs.

Benchmark	Branch	# of Iterations	Time(sec)
Expint	1	8	216.7
	2	5	56.9
Gamdev	1	2	12.9
	2	1	7.4

Table 1. Time performance for nonlinear branches of *expint* and *gamdev*.

5 Related Work

The earliest approaches to automatically generate test data for a given branch proceeded with selecting a path containing the test branch. They used symbolic evaluation to detect path infeasibility to some extent and then attempted to generate an input to exercise the selected path [1, 2]. If an input could not be generated for a path, then the next path was selected. These approaches are limited in handling dynamic data structures such as arrays and pointers since they use symbolic evaluation. Derivation of symbolic representations in general involves complex algebraic manipulations. A general nonlinear system of inequalities thus obtained is then solved to obtain a feasible solution. The approach presented in this paper derives the linear constraints, required to iteratively refine an arbitrarily chosen input, directly from the program representations of the functions computed by branch predicates. This approach obviates the need to derive exact symbolic representations for the branch predicates.

A program execution based approach for generating test data to exercise a given node is described in [10]. It attempts to change the flow of execution for one branch predicate at a time on the set of paths being explored, by attempting to minimize a function associated with the branch predicate. In [3], the authors improve upon the performance of their approach in [10] by using the data dependence analysis, in addition to control flow information, to guide the test data generation process. But, both these approaches are inefficient in handling infeasible paths. Another program execution based approach to generate test data for a given branch is described in [7]. It attempts to detect infeasible paths by constructing a set of global constraints for the program and using constraint solving techniques. In this method, the global constraints are generated for the whole program even if a single node is to be exercised.

An approach to construct program slices that contain smaller number of influencing predicates for a given test element is described in [4]. The rationale of this approach is that a path with smaller number of influencing branch predicates is more likely to be feasible. As illustrated in this paper, the number of branch predicates on a path is only one of the factors contributing to the infeasiblity of a path. The complexity of the branch predicates is also an important factor to be considered, along with other factors, during the path selection for a given test element.

Our program execution based approach for generating an input to exercise a given branch is general in its applicability as it can handle branch predicates that compute general nonlinear functions of the input. Since it based on program execution, it can handle dynamic data structures such as arrays and pointers. The time performance of our technique is governed by the semantic complexity of the predicates on the paths exercising the test branch. It may take hours to generate test data by hand for the random number generator program that we considered for experimentation, whereas it took a few seconds to generate the test data automatically. Our algorithm identifies infeasible linear paths and likely infeasible paths and removes them from the set of paths being explored. Therefore, it can effectively handle infeasible paths through the given branch.

6 Conclusions

We have presented a new program execution based approach to generate test data for a given branch in a program. The method dynamically switches to a path that offers relatively less resistance to generation of an input to force execution through it to reach the given branch. The method is general and applicable to programs with general nonlinear expressions. Being execution based, it can handle different programming language features. It is suitable for automation and has potential to provide a practical solution to the test data generation problem.

References

[1] L.A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 3, pages 215-222, September 1976.

[2] R.A. DeMillo and A.J. Offutt, "Constraint-based Automatic Test Data Generation," *IEEE Transactions on Software*, Vol. 17, No. 9, pages 900-910, September 1991.

[3] R. Ferguson and B. Korel, "The Chaining Approach for Software Test Data Generation," *ACM Transactions on Software Engineering Methodology,* Vol 5, No.1, pages 63-86, January 1996.

[4] I. Forgacs and A Bertolino, "Feasible Test Path Selection by Principle Slicing," *Proceedings of the ACM SIGSOFT International Conference ESEC/FSE'97,* LNCS 1301, Zurigo, 22-25 September, 1997.

[5] I. Forgacs and A Hajnal, "An Applicable Test Data Generation Algorithm for Domain Errors," In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, Clearwater Beach, Florida, March, 1998.

[6] M.J. Gallagher and V.L. Narsimhan, "ADTEST: A Test Data Generation Suite for Ada Software Systems," *IEEE Transactions on Software Engineering*, Vol. 23, No. 8, pages 473-484, August 1997.

[7] A. Gotlieb, B. Botella, and M. Rueher, "Automatic Test Data Generation using Constraint Solving Techniques," *International Symposium on Software Testing and Analysis*, 1998.

[8] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa, "Automated Test Data Generation using An Iterative Relaxation Method," *ACM SIGSOFT Sixth International Symposium on Foundations of Software Engineering(FSE-6),* pages 231-244, Orlando, Florida, November 1998.

[9] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa, "UNA Based Iterative Test Data Generation and its Evaluation," *14th IEEE International Conference on Automated Software Engineering(ASE'99),* pages 224-232, Cocoa Beach, Florida, October 1999.

[10] B. Korel, "A Dynamic Approach of Test Data Generation," *Conference on Software Maintenance,* pages 311-317, San Diego, CA, November 1990.

[11] B. Korel, "Automated Software Test Data Generation," *IEEE Transactions on Software Engineering,* Vol 16, No.8, pages 870-879, August 1990.

[12] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, "Numerical Reci pes in C" Cambridge University Press, 1992.