Automated Test Data Generation Using An Iterative Relaxation Method

Neelam Gupta
Dept. of Computer Science
Purdue University
West Lafayette, IN 47907
ngupta@cs.purdue.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
University of Pittsburgh
Pittsburgh, PA 15260
soffa@cs.pitt.edu

Abstract

An important problem that arises in path oriented testing is the generation of test data that causes a program to follow a given path. In this paper, we present a novel program execution based approach using an iterative relaxation method to address the above problem. In this method, test data generation is initiated with an arbitrarily chosen input from a given domain. This input is then iteratively refined to obtain an input on which all the branch predicates on the given path evaluate to the desired outcome. In each iteration the program statements relevant to the evaluation of each branch predicate on the path are executed, and a set of linear constraints is derived. The constraints are then solved to obtain the increments for the input. These increments are added to the current input to obtain the input for the next iteration. The relaxation technique used in deriving the constraints provides feedback on the amount by which each input variable should be adjusted for the branches on the path to evaluate to the desired outcome.

When the branch conditions on a path are linear functions of input variables, our technique either finds a solution for such paths in one iteration or it guarantees that the path is infeasible. In contrast, existing execution based approaches may require an unacceptably large number of iterations for relatively long paths because they consider only one input variable and one branch predicate at a time and use backtracking. When the branch conditions on a path are nonlinear functions of input variables, though it may take more then one iteration to derive a desired input, the set of constraints to be solved in each iteration is linear and is solved using Gaussian elimination. This makes our technique practical and suitable for automation.

Keywords - path testing, dynamic test data generation, predicate slices, input dependency set, predicate residuals, relaxation methods.

1 Introduction

Software testing is an important stage of software development. It provides a method to establish confidence in the reliability of software. It is a time consuming process and accounts for 50% of the cost of software development [10]. Given a program and a testing criteria, the generation of test data that satisfies the selected testing criteria is a very difficult problem. If test data for a given testing criteria for a program can be generated automatically, it can relieve the software testing team of a tedious and difficult task, reducing the cost of the software testing significantly. Several approaches for automated test data generation have been proposed in the literature, including random [2], syntax based [5], program specification based [1, 9, 12, 13], symbolic evaluation [4, 6] and program execution based [7, 8, 10, 11, 14] test data generation.

A particular type of testing criteria is **path coverage**, which requires generating test data that causes the program execution to follow a given path. Generating test data for a given program path is a difficult task posing many complex problems [4]. Symbolic evaluation [4, 6] and program execution based approaches [7, 10, 14] have been proposed for generating test data for a given path. In general, symbolic evaluation of statements along a path requires complex algebraic manipulations and has difficulty in handling arrays and pointer references. Program execution based approaches can handle arrays and pointer references efficiently because array indexes and pointer addresses are known at each step of program execution. But, one of the major challenges to these methods is the impact of infeasible paths. Since there is no concept of inconsistent constraints in these methods, a large number of iterations can be performed before the search for input is abandoned for an infeasible path. Existing program execution based methods [7, 10] use function minimization search algorithms to locate the values of input variables for which the selected path is traversed. They consider one branch predicate and one input variable at a time and use backtracking. Therefore, even when the branch conditions on the path are linear functions of input, they may require a large number of iterations for long paths.

In this paper, we present a new program execution based approach to generate test data for a given path. It is a novel approach based on a relaxation technique for iteratively refining an arbitrarily chosen input. The relaxation technique is used in numerical analysis to improve upon an approximate solution of an equation representing the roots of a function [15]. In this technique, the function is evaluated at the approximate solution and the resulting value is used to provide feedback on the amount by which the values in the approximate solution should be adjusted so that it becomes an exact solution of the equation. If the function is linear, this technique derives an exact solution of the equation from an approximate solution in one iteration. For nonlinear functions it may take more than one iteration to derive an exact solution from an approximate solution.

In our method, test data generation for a given path in a program is initiated with an arbitrarily chosen input from a given domain. If the path is not traversed when the program is executed on this input, then the input is iteratively refined using the relaxation technique to obtain a new input that results in the traversal of the path. To apply the relaxation technique to the test data generation problem, we view each branch condition on the given path as a function of input variables and derive two representations for this function. One representation is in the form of a subset of input and assignment statements along the given path that must be executed in order to evaluate the function. This representation is computed as a slicing operation on the data dependency graph of the program statements on the path, starting at the predicate under consideration. Therefore, we refer to it as a **predicate slice**. Note that a predicate slice always provides an exact representation of the function computed by a branch condition. Using this exact representation in the form of program statements, we derive a **linear arithmetic representation** of the function computed by the branch condition in terms of input variables. An arithmetic representation of the function in terms of input variables is necessary to enable the application of numerical analysis techniques since a program representation of the function is not suitable for this purpose. If the function computed by a branch condition is a linear function of the input, then its linear arithmetic representation is exact. When the function computed by a branch condition is a nonlinear function of the input, its linear arithmetic representation approximates the function in the neighborhood of the current input.

These two representations are used to refine an arbitrarily chosen input to obtain the desired input as follows. Let us assume that by executing a predicate slice using the arbitrarily chosen input, we determine that a branch condition does not evaluate to the desired outcome. In this case, the evaluation of the branch condition also provides us with a value called the **predicate residual** which is the amount by which the function value must change in order to achieve the desired branch outcome. Now using the linear arithmetic representation and the predicate residual, we derive a **linear constraint on the increments** for the current input. One such constraint is derived for each branch condition on the path. These linear constraints are then solved simultaneously using Gaussian elimination to compute the increments for the current input. A new input is obtained by adding these increments to the current input. Since the constraints corresponding to all the branch conditions on the path are solved simultaneously, our method attempts to change the current input so that all the branch predicates on the path evaluate to their desired outcomes when their predicate slices are executed on the new input.

If all the branch conditions on the path are linear functions of the input (i.e., the linear arithmetic representations of the predicate functions are exact), then our method either derives a desired input in one iteration or guarantees that the path is infeasible. This result has immense practical importance in accordance with the studies reported in [6]. A case study of 3600 test case constraints generated for a group of Fortran programs has shown that the constraints are almost always linear. For this large class of paths our method is able to detect infeasibility, even though the problem of detecting infeasible paths is unsolvable in general. If such a path is feasible, our method is extremely efficient as it finds a solution in exactly one iteration.

If at least one branch condition on the path is a nonlinear function of the input, then the increments for the current input that are computed by solving the linear constraints on the increments may not immediately yield a desired input. This is because the set of linear constraints on the increments are derived from the linear arithmetic representations (which in this case are approximate) of the corresponding branch conditions. Therefore it may take more than one iteration to obtain a desired input. Even when the branch predicates on the path are nonlinear functions of the input, the set of equations to be solved to obtain a new input from the current input are linear and are solved by Gaussian elimination. Gauss elimination algorithm is widely implemented and is an established method for solving a system of linear equations. This makes our technique practical and suitable for automation.

The important contributions of the novel method presented in this paper are:

- It is an innovative use of the traditional relaxation technique for test data generation.
- If all the conditionals on the path are linear functions of the input, it either generates the test data in one iteration or guarantees that the path is infeasible. Therefore, it is efficient in finding a solution as well as powerful in detecting infeasibility for a large class of paths.
- It is a general technique and can generate test data even if conditionals on the given path are nonlinear functions of the input. In this case also, the number of iterations with inconsistent constraints can be used as an indication of a potential infeasible path.
- The set of constraints to be solved in this method is always linear even though the path may involve conditionals that are non-linear functions of the input. A set of linear constraints can be automatically solved using Gaussian elimination whereas no direct method exits to solve a set of arbitrary nonlinear constraints. Gaussian elimination has been widely implemented and experimented algorithm. This makes the method practical and suitable for automation.
- It is scalable to large programs. The number of program executions required in each iteration are independent of the path length and are bounded by number of input variables. The size of the system of linear equations to be solved using Gaussian elimination increases with the number of branch predicates on the path, but the increase in cost is significantly less than that of the existing techniques.

The organization of this paper is as follows. An overview of the method is presented in the next section. The algorithm for test data generation is described in section 3. It is illustrated with examples involving linear and nonlinear paths, loops and arrays. Related work is discussed in section 4. The important features of the method are summarized and our future work is outlined in section 5.

2 Overview

We define a program module M as a directed graph $G = (N, E, s, e)$, where N is a set of nodes, E is a set of edges, s is a unique entry node and e is a unique exit node of M. A node n represents a single statement or a conditional expression, and a possible transfer of control from node n_i to node n_j is mapped to an *edge* $(n_i, n_j) \in E$. A *Path* $P = \{n_1 = s, n_2, ..., n_{k+1}\} \in G$ is a sequence of nodes such that $(n_i, n_{i+1}) \in E$, for $i = 1$ to k.

A variable i_k is an *input variable* of the module M if it either appears in an input statement of M or is an input parameter of M. The domain D_k of input variable i_k is the set of all possible values it can hold. An input vector $I = (i_1, i_2 ..., i_m) \in (D_1 \times D_2 \times ... D_m)$, where m is the number of inputs, is called a *Program Input*. In this paper, we may refer to the *program input* by *input* and use these terms interchangeably.

A conditional expression in a multi-way decision statement is called a **Branch Predicate**. Without loss of generality, we assume that the branch predicates are simple relational expressions (inequalities and equalities) of the form $E_1 \;\; op \;\; E_2$, where E_1 and E_2 are arithmetic expressions, and op is one of $\{<, \leq, >, \geq, =, \neq\}$.

If a branch predicate contains boolean variables, we represent the "true" value of the boolean variable by a numeric value zero or greater and the "false" value by a negative numeric value. If a branch predicate on a path is a conjunction of two or more boolean variables such as in $(A \;\; AND \;\; B)$, then such a predicate is considered as multiple branch predicates $A \geq 0$ and $B \geq 0$ that must simultaneously be satisfied for the traversal of the path. If a branch predicate on a path is a disjunction of two or more boolean variables such as in $(A \;\; OR \;\; B)$, then at a time only one of the branch predicates $A \geq 0$ or $B \geq 0$ is considered along with other branch predicates on the path. If a solution is not found with one branch predicate then the other one is tried.

Each branch predicate $E_1 \;\; op \;\; E_2$ can be transformed to the equivalent branch predicate of the form $F \;\; op \;\; 0$, where F is an arithmetic expression $E_1 - E_2$. Along a given path, F represents a real valued function called a **Predicate Function**. F may be a direct or indirect function of the input variables. To illustrate this, let us consider the branch predicate

$$BP2 : (W + Z) > 100$$

for the conditional statement $P2$ in the example program in Figure 1. The predicate function $F2$ corresponding to the branch predicate $BP2$ is

$$F2 : W + Z - 100.$$

Along path $P = \{\ 0,\ 1,\ P1,\ 2,\ P2,\ 4,\ 5,\ 6,\ P4,\ 9\ \}$, the predicate function $F2 : W + Z - 100$ indirectly represents the function $2X - 2Y + Z - 100$ of the input variables X, Y, Z. We now state the problem being addressed in this paper:

Problem Statement: *Given a program path P which is traversed for certain evaluations (true or false) of branch predicates $BP_1, BP_2 \cdots BP_n$ along P, generate a program input $I = (i_1, i_2 ..., i_m) \in (D_1 \times D_2 \times ... D_m)$ that causes the branch predicates to evaluate such that P is traversed.*

We present a new method for generating a program input such that a given path in a program is traversed when the program is executed using this input. In this method, test data generation is initiated with an arbitrarily chosen input from a given domain. If the given path is not traversed on this input, a set of linear constraints on increments to the input are derived using a relaxation method. The increments obtained by solving these constraints are added to the input to obtain a new input. If the path is traversed on the new input then the method terminates. Otherwise, the steps of refining the input are carried out iteratively to obtain the desired input. We now briefly review the relaxation technique as used in numerical analysis for refining an approximation to the solution of a linear equation.

The Relaxation Technique

Let (x_0, y_0) be an approximation to a solution of the linear equation

$$ax + by + c = 0. \tag{1}$$

In general, substituting (x_0, y_0) in the lhs of the above equation would result in a non zero value r_0 called the residual, i.e.,

$$ax_0 + by_0 + c = r_0.$$

If increments Δx and Δy for x_0 and y_0 are computed such that they satisfy the linear constraint given by

$$a\Delta x + b\Delta y = -r_0,$$

then,

$$a(x_0 + \Delta x) + b(y_0 + \Delta y) + c = 0.$$

Therefore,

$$(x_1, y_1) = (x_0 + \Delta x, y_0 + \Delta y)$$

is a solution of equation (1).

In order to formulate the test data generation problem as a relaxation technique problem, we view the predicate function corresponding to each branch predicate on the path as a function of program input. To apply the above relaxation technique, a **Linear Arithmetic Representation** in terms of the relevant input variables is required for each predicate function. We first derive an exact program representation called a **Predicate Slice** for the function computed by each predicate function and then use it to derive a linear arithmetic representation. The two representations are used in an innovative way to refine the program input.

The Predicate Slice

The exact program representation of a predicate function, the Predicate Slice, is defined as follows:

Definition: The **Predicate Slice** $S(BP, P)$ of a branch predicate BP on a path P is the set of statements that compute values upon which the value of BP may be directly or indirectly data dependent when execution follows the path P.

In other words, $S(BP, P)$ is a slice over data dependencies of the branch predicate BP using a program consisting of

```
0:  read(X,Y,Z)
1:  U = ( X - Y ) * 2
P1: if( X > Y ) then
2:       W = U
3:  else W = Y endif
P2: if ( W + Z ) > 100 then
4:       X = X - 2
5:       Y = Y + W
6:       write("Linear")
P3: elseif ( X^2 + Z^2 >= 100 ) then
7:       Y = X * Z + 1
8:       write("Nonlinear: Quadratic")
    endif
P4: if ( U > 0 ) then
9:       write(U)
P5: elseif ( Y - Sin(Z) ) > 0 then
10:      write("Nonlinear: Sine")
    endif
```

```
0:  read(X,Y,Z)
```
Statements in Predicate Slice S(BP1, P)

```
0:  read(X,Y,Z)
1:  U = ( X - Y ) * 2
2:  W = U
```
Statements in Predicate Slice S(BP2, P)

```
0:  read(X,Y,Z)
1:  U = ( X - Y ) * 2
```
Statements in Predicate Slice S(BP4, P)

Figure 1: An Example Program and Predicate Slices on a path P={0,1,P1,2,P2,4,5,6,P4,9}

only input and assignment statements preceding BP on the path P. We illustrate the above definition using the example program in Figure 1. Consider the path P

$$P = \{0, 1, P1, 2, P2, 4, 5, 6, P4, 9\}.$$

Let BPi denote the i^{th} branch predicate along the path P. The predicate slices corresponding to the branch predicates $BP1$, $BP2$ and $BP4$ along path P are:

$$S(BP1, P) = \{0\};$$
$$S(BP2, P) = \{0, 1, 2\};$$
$$S(BP4, P) = \{0, 1\}.$$

As illustrated by the above examples, predicate slices include only input and assignment statements. The value of a predicate function for an input can be computed by executing the corresponding predicate slice on the input.

Note that a predicate slice is not a conventional static slice since it is computed over the statements along a path. It is also not a dynamic slice because it is computed statically using the input and assignment statements along a path and is not as precise as the dynamic slice. To illustrate the latter we consider the code segment given in Figure 2:

```
input(I, J, Y);
A[I] = Y;
If (A[J] > 0) then...;
```

Figure 2: A code segment on a path using an array.

When I $\neq$ J, the evaluation of BP: (A[J] > 0) is not data dependent on the assignment statement. Whereas, if I = J, the evaluation of BP is data dependent on the assignment statement. Therefore, the predicate slice for the branch predicate BP will consist of the input statement as well the assignment statement. In other words, the predicate slice is a path oriented static slice.

The concept of predicate slice enables us to evaluate the outcome of each branch predicate on the path irrespective of the outcome of other branch predicates. The predicate slices for the branch predicates on the path can be executed using an arbitrary input even though the path may not be traversed on that input. This is possible because there are no conditionals in a predicate slice. After execution of a predicate slice on an input, the value of the corresponding predicate function can be computed and the branch outcome evaluated.

There is a correspondence between the outcomes of the execution of the predicate slices on an input and the traversal of the the path on that input. If all the branch predicates on the path evaluate to their desired outcomes, by executing their respective predicate slices on an input and computing the respective predicate functions, the path will be traversed when the program is executed using this input. If any of the branch predicates on the path does not evaluate to its desired outcome when its predicate slice is executed on an input, the path will not be traversed when the program is executed using this input.

Conceptually, a predicate slice enables us to view a predicate function on the path as an independent function of input variables. Therefore, our method can simultaneously force all branch predicates along the path to evaluate to their desired outcomes. In contrast, the existing program execution based methods [7, 10] for test data generation attempt to satisfy one branch predicate at a time and use backtracking to fix a predicate satisfied earlier while trying to satisfy a predicate that appears later on the path. They cannot consider all the branch predicates on the path simultaneously because the path may not be traversed on an intermediate input.

The predicate slice is also useful in identifying the relevant subset of input variables, on which the value of the predicate function depends. This subset of input variables is required so that a linear arithmetic representation of the predicate function in terms of these input variables can be derived. The subset of the input variables on which the value computed by a predicate function depends can only be determined dynamically as illustrated by the example in Figure 2. Therefore, given an input and a branch predicate on the path, the corresponding predicate slice is executed using this input and a dynamic data dependence graph based upon the execution is constructed. The relevant input variables for the corresponding predicate function are determined by taking a dynamic slice over this dependence graph.

Note that if only scalars are referenced in a predicate slice and the corresponding predicate function, then the subset of input variables on which the predicate function depends can be determined statically from the predicate slice. Execution of the predicate slice on the input data followed by a dynamic slice to determine relevant input variables is necessary to handle arrays. We define this subset of input variables as the Input Dependency Set.

Definition: The **Input Dependency Set** $ID(BP, I, P)$ of a branch predicate BP on an input I along a path P is the subset of input variables on which BP is, directly or indirectly, data dependent. These input variables can be identified by executing the statements in the predicate slice $S(BP, P)$ on input I and taking a dynamic slice over the dynamic data dependence graph.

For example, executing $S(BP2, P)$ on an input $I_0 = (1, 2, 3)$, we note that the evaluation of $BP2$ depends on the input variables X,Y and Z. Therefore, $ID(BP2, I_0, P) = \{X, Y, Z\}$.

Now we explain how we use the input dependency set to derive the linear arithmetic representation in terms of input variables for a predicate function for a given input.

Deriving the Linear Arithmetic Representation of a Predicate Function

Given a predicate function and its input dependency set ID for an input I, we write a general linear function of the input variables in ID. Then, we compute the values of the coefficients in the general linear function so that it represents the tangent plane to the predicate function at I. This gives us a **Linear Arithmetic Representation** for the predicate function at I.

For example, the predicate function $F2 : W + Z - 100$ has $ID = \{X, Y, Z\}$ for the input $I_0 = (1, 2, 3)$. A general linear function for the inputs in ID is

$$f(X, Y, Z) = aX + bY + cZ + d.$$

Here, a, b and c are the slopes of f with respect to input variables X, Y and Z respectively and d is the constant term.

If the slopes a, b and c above are computed by evaluating the corresponding derivatives of the predicate function at the input I_0 and the constant term is computed such that the linear function f evaluates to the same value at I_0 as that computed by executing the corresponding predicate slice on I_0 and evaluating the predicate function, then $f(X, Y, Z) = 0$ represents the tangent plane to the predicate function at input I_0. This gives us the linear arithmetic representation for the predicate function at I_0.

If the predicate function computes a linear function of the input, then the above tangent plane $f(X, Y, Z) = 0$ is the exact representation for the predicate function. Whereas if a predicate function computes a nonlinear function of the input, then the above tangent plane $f(X, Y, Z) = 0$ will approximate the predicate function in the neighborhood of the input I_0.

We illustrate this by deriving the linear arithmetic representation for the predicate function $F2$ at the input $I_0 = (1, 2, 3)$. We approximate the derivatives of a predicate function by its divided differences. To compute a at I_0, we execute $S(BP2, P)$ at I_0 and at

$$(X_0, Y_0, Z_0) + (\Delta X, 0, 0) = (1, 2, 3) + (1, 0, 0) = (2, 2, 3),$$

where we have chosen $\Delta X = 1$, for a unit increment in the input variable X. Then, we compute the divided difference:

$$\frac{F2(X_0 + \Delta X, Y_0, Z_0) - F2(X_0, Y_0, Z_0)}{\Delta X} = \frac{-97 + 99}{1} = 2.$$

This gives the value of $a = 2$. We compute the value of b by executing the predicate slice $S(BP2, P)$ at I_0 and at

$$(X_0, Y_0, Z_0) + (0, \Delta Y, 0) = (1, 2, 3) + (0, 1, 0) = (1, 3, 3),$$

and computing the divided difference of $F2$ at these two points with respect to y. This gives b equal to -2. Similarly, we get c equal to 1. We compute the value of d by solving for d from the equation

$$a + 2b + 3c + d = F2(I_0).$$

Substituting the values of a, b, c and $F2(I_0)$ in this equation and solving for d, we get d equal to -100. Therefore, we obtain the linear arithmetic representation for $F2$ at I_0 as

$$2X - 2Y + Z - 100.$$

In this example, $F2$ computes a linear function of the input. Therefore, its linear arithmetic representation at I_0 computed as above is the exact representation of the function of inputs computed by $F2$. Also, only those input variables that influence the predicate function $F2$ appear in this representation.

In this paper, we have approximated the derivatives of a predicate function by its divided differences. Tools have been developed to compute derivative of a program with respect to an input variable [3]. With these tools, we can get exact derivative values rather than using divided differences. Therefore, our technique for deriving a linear arithmetic representation for a predicate function can be very accurately implemented for automated testing.

Using the method explained above, we derive a linear arithmetic representation at the current input for each predicate function on the given path. In order to derive a set of linear constraints on the increments to the current input from these linear arithmetic representations, we execute the predicate slices of all the branch predicates on the current input and compute the values of corresponding predicate functions. We use these values of the predicate functions to provide feedback for computing the desired increments to the current input.

The Predicate Residuals

The values of the predicate functions at an input, defined as Predicate Residuals, essentially place constraints on the changes in the values of the input variables that, if satisfied, will provide us with a new input on which the desired path is followed.

Definition: The **Predicate Residual** of a branch predicate for an input is the value of the corresponding predicate function computed by executing its predicate slice at the input.

If a branch predicate has the relational operator " = ", then a non zero predicate residual gives the exact amount by which the value of the predicate function should change by modifying the input so that the branch evaluates to its desired outcome. Otherwise, a predicate residual gives the least (maximum) value by which the predicate function's value must be changed (can be allowed to change), by modifying the program input, such that the branch predicate evaluates (continues to evaluate) to the desired outcome. We explain this with examples given below.

If a branch predicate evaluates to the desired outcome for a given input, then it should continue to evaluate to the desired outcome. In this case, the predicate residual gives the maximum value by which the predicate function's value can be allowed to change, by modifying the program input, such that the branch predicate continues to evaluate to the desired outcome. To illustrate this, let us consider the path P in the example program in Figure 1. Using an input $I = (1, 2, 110)$, the branch predicate $BP2$ evaluates to the desired branch for the path P to be traversed. The value of the predicate function $F2$ at $I = (1, 2, 110)$ and hence the predicate residual at this input is 8. Therefore, the value of the predicate function can be allowed to decrease at most by 8 due to a change in the program input, so that the predicate function continues to evaluate to evaluate to a positive value.

On the other hand, if a predicate does not evaluate to the desired outcome, the predicate residual gives the least value by which the predicate function's value must be changed, by modifying the program input, such that the branch predicate evaluates to the desired outcome. For example, using the input $I_0 = (1, 2, 3)$ the branch predicate $BP2$ does not evaluate to the desired branch for the path P to be traversed. The value of the predicate function and hence the predicate residual at $I_0 = (1, 2, 3)$ is -99. Therefore, the input should be modified such that the value of the predicate function increases at least by 99 for the branch predicate $BP2$ to evaluate to its desired outcome.

The predicate residuals essentially guide the search for a program input that will cause each branch predicate on the given path P to evaluate to its desired outcome. We compute a predicate residual at the current input for each branch predicate on the given path. Once we have a predicate residual and a linear arithmetic representation at the current input for each predicate function, we can apply the relaxation technique to refine the input.

Refining the input

The linear arithmetic representation and the predicate residual of a predicate function at an input essentially allow us to map the change in the value of the predicate function to changes in the program input. For each predicate function on the path P, we derive a linear constraint on the increments to the program input using the linear representation of the predicate function and the value of the corresponding predicate residual. This set of linear constraints is then solved simultaneously using Gaussian elimination to compute increments to the input. These increments are added to the input to obtain a new input.

We illustrate the derivation of linear constraint corresponding to the predicate function $F2$. The branch predicate $BP2$ evaluates to "false", when $S(BP2, P)$ is executed on the arbitrarily chosen input $I_0 = (1, 2, 3)$, whereas it should evaluate to "true" for the path P to be traversed. The residual value -99 and the linear function

$$2X - 2Y + Z - 100$$

are used to derive a **linear constraint**

$$2\Delta X - 2\Delta Y + \Delta Z > 99. \tag{2}$$

Note that the constant term d does not appear in this constraint. Intuitively, this means that the increments to the input I_0 should be such that the value of predicate function $F2$ changes more than 99 so as to force $F2$ to evaluate to a positive value and therefore force the corresponding branch predicate $BP2$ to evaluate to its desired outcome, i.e., "true" on the new input. For instance, $\Delta X = 1, \Delta Y = 1, \Delta Z = 100$ is one of the solutions to the above constraint. We see that $BP2$ evaluates to "true" when $S(BP2, P)$ is executed on $I_1 = (2, 3, 103)$.

The linear constraint derived above from the predicate residual to compute the increments for the current input, is an important step of this method. It is through this constraint that the value of a predicate function at the current input provides feedback to the increments to be computed to derive a new input. Since this method computes a new program input from the previous input and the residuals, it is a relaxation method which iteratively refines the program input to obtain the desired solution.

We would like to point out here that when the relational operator in each branch predicate on the path is "=", this method reduces to Newton's Method for iterative refinement of an approximation to a root of a system of nonlinear functions in several variables. To illustrate this, let us consider the linear constraint in equation (2). Let us assume that the relational operator in the corresponding branch predicate $BP2$ is "=" and for simplicity let $F2$ be a function of a single variable X. Then the linear constraint in equation 2 reduces to

$$\Delta X = \frac{99}{2}$$

which is of the form

$$X_{n+1} = X_n - \frac{F2(X_n)}{F2'(X_n)}.$$

In general, the branch predicates on a path will have equalities as well as inequalities. In such a case, our method is different from Newton's Method for computing a root of a system of nonlinear functions in several variables. But since the increments for input are computed by stepping along the tangent plane to the function at the current input, we expect our method to have convergence properties similar to Newton's Method.

In our discussion so far, we have assumed that the conditionals are the only source of predicate functions. However, in practice some additional predicates should also be considered during test data generation. First, constraints on inputs may exist that may require the introduction of additional predicates (e.g., if an input variable I is required to have a positive value, then the predicate $I > 0$ should be introduced). Second, we must introduce predicates that constrain input variables to have values that avoid execution errors (e.g., array bound checks and division by zero). By considering the above predicates together with the predicates from the conditionals on the path a desired input can be found. For simplicity, in the examples considered in this paper we only consider the predicates from the conditionals.

3 Description of the Algorithm

In this section, we present an algorithm to generate test data for programs with numeric input, arrays, assignments, conditionals and loops. 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 our algorithm are outlined in Figure 3. We now describe the steps of our algorithm in detail and at the same time illustrate each step of the algorithm by generating test data for a path along which the predicate functions are linear functions of the input. Examples with nonlinear predicate functions are given in the next section.

The method begins with the given path P and an arbitrarily chosen input I_0 in the input domain of the program. The program is executed on I_0. If P is traversed using I_0, then I_0 is the desired program input and the algorithm terminates; otherwise the steps of iterative refinement using the relaxation technique are executed.

We illustrate the algorithm using the example from section 2, where the path

$$P = \{0, 1, P1, 2, P2, 4, 5, 6, P4, 9\}$$

in the program of Figure 1, with initial program input $I_0 = (1, 2, 3)$ is considered. The path P is not traversed when the program is executed using I_0. Thus, the iterative relaxation method as discussed below is employed to refine the input.

Step 1. Computation of Predicate Slices.

For each node n_i in P that represents a branch predicate, we compute its Predicate Slice $S(n_i, P)$ by a backward pass over the static data dependency graph of input and assignment statements along the path P before n_i. The predicate slices for the branch predicates on the path P are:

$$S(BP1, P) = \{0\},$$
$$S(BP2, P) = \{0, 1, 2\},$$
$$S(BP4, P) = \{0, 1\}.$$

Step 2. Identifying the Input Dependency Sets.

For every node n_i in P that represents a branch predicate, we identify the input dependency set $ID(n_i, I_k, P)$ of input variables on which n_i is data dependent by executing the predicate slice $S(n_i, P)$ on the current input I_k and taking a dynamic slice over the dynamic data dependence graph. The input dependency sets for the branch predicates on the path P computed by executing the respective predicate slices on P using the input $I_0 = (1, 2, 3)$ are:

$$ID(BP1, I_0, P) = \{X, Y\},$$
$$ID(BP2, I_0, P) = \{X, Y, Z\},$$
$$ID(BP4, I_0, P) = \{X, Y\}.$$

Note that all input and assignment statements along the path P need be executed at most once to compute the input dependency sets for all the branch predicates along the path P.

Step 3. Derivation of Linear Arithmetic Representations of the Predicate Functions.

In this step, we construct a linear arithmetic representation for the predicate function corresponding to each branch predicate on P. For each branch predicate n_i in P, we first formulate a general linear function of the input variables in the set $ID(n_i, I_k, P)$. For example, the linear formulations for the predicate functions corresponding to the branch predicates on path P are:

$$F1 : a_1 X + b_1 Y + d_1,$$
$$F2 : a_2 X + b_2 Y + c_2 Z + d_2,$$
$$F4 : a_4 X + b_4 Y + d_4.$$

The coefficients a_i, b_i and c_i of the input variables in the above linear functions represent the slopes of the i^{th} predicate function with respect to input variables X, Y and Z respectively. We approximate these slopes with respective divided differences.

To compute the slope coefficient with respect to a variable, we execute the predicate slice $S(n_i, P)$ and evaluate the predicate function F at the current input $I_k = (i_1, .., i_j, .., i_m)$ and at $I_k + (0, .., \Delta i_j, ..0)$ and compute the divided difference

$$\frac{F(I_k + (0, .., \Delta i_j, .., 0)) - F(I_k)}{\Delta i_j}.$$

This gives the value of the coefficient of i_j in the linear function for the predicate function F corresponding to node n_i in P. Similarly, we compute the other slope coefficients in the linear function.

In the example being considered, evaluating $F1$ by executing $S(BP1, P)$ at $(1, 2, 3)$ and $(2, 2, 3)$ and computing the divided difference with respect to X, we get $a_1 = 1$. Similarly, for $F2$ and $F4$, we get $a_2 = 2$, and $a_4 = 2$. Computing the divided differences with respect to Y using $(1, 2, 3)$ and $(1, 3, 3)$, we get $b_1 = -1$, $b_2 = -2$, and $b_4 = -2$; and computing the divided difference with respect to Z using $(1, 2, 3)$ and $(1, 2, 4)$, we get $c_2 = 1$.

To compute the constant term d_i, we execute the corresponding predicate slice at I_k and evaluate the value of the predicate function. The values of input variables in I_k and the slope coefficients found above are substituted in the linear function, and it is equated to the value of the predicate function at I_k computed above. This gives a linear equation in one unknown and it can be solved for the value of the constant term.

For the example being considered, we substitute the slope coefficients a_i, b_i and c_i computed above and $X = 1$, $Y = 2$ and $Z = 3$, in the general linear formulations for the predicate functions $F1$, $F2$ and $F4$. Then, we equate the general linear formulations to their respective values at $I_0 = (1, 2, 3)$ and obtain the following equations in d_i:

$$F1 : 1 - 2 + d_1 = -1,$$
$$F2 : 2 - 4 + 3 + d_2 = -99,$$
$$F4 : 2 - 4 + d_4 = -2.$$

Solving for the constant terms d_i, we get $d_1 = 0$, $d_2 = -100$, and $d_4 = 0$. Therefore, the linear arithmetic representations for the three predicate functions of P are given by:

$$F1 : X - Y,$$
$$F2 : 2X - 2Y + Z - 100,$$
$$F4 : 2X - 2Y.$$

If a predicate function is a linear function of input variables then the slopes computed above are exact and this method results in the exact representation of the predicate function.

Input: A path $P = \{n_1, n_2, .., n_k\}$ and an Initial Program Input I_0
Output: A Program Input I_f on which P is traversed
procedure TESTGEN(P, I_0)
    If P traversed on I_0 **then** $I_f = I_0$ **return**
step 1: **for** each Branch Predicate n_i on P, **do** Compute Predicate Slice $S(n_i, P)$ **endfor**
    k=0
    while not Done **do**
        for each Branch Predicate n_i on P, **do**
step 2: Execute $S(n_i, P)$ on input I_k to compute input dependency set $ID(n_i, I_k, P)$.
step 3: Compute the linear representation $L(ID(n_i, I_k, P))$ for the predicate function for n_i
step 4: Compute Predicate Residual $R(n_i, I_k, P)$
step 5: Construct a linear constraint using $R(n_i, I_k, P)$ and $L(ID(n_i, I_k, P))$ for the computation of increment to I_k
        endfor
step 6: Convert inequalities in the constraint set to equalities
step 7: Solve this system of equations to compute increments for the current program input.
        Compute the new program input I_{k+1} by adding the computed increments to I_k
        if the execution of the program on input I_{k+1} follows path P **then** $I_f = I_{k+1}$; Done = True
        else k++ **endif**
    endwhile
endprocedure

Figure 3: Algorithm to generate test data using an iterative relaxation method.

If a predicate function computes a nonlinear function, the linear function computed above represents the tangent plane to the predicate function at I_k. In the neighborhood of I_k, the inequality derived from the tangent plane closely approximates the branch predicate. Therefore, if the predicate function evaluates to a positive value at a program input in the neighborhood of I_k, then so does the linear function and vice versa. These linear representations and the predicate residuals computed in subsequent step are used to derive a set of linear constraints which are used to refine I_k and obtain I_{k+1}.

Step 4: Computation of Predicate Residuals.

We execute the predicate slice corresponding to each branch predicate on P at the current program input I_k and evaluate the value of the predicate function. This value of the predicate function is the predicate residual value $R(n_i, I_k, P)$ at the current program input I_k for a branch predicate n_i on P. The predicate residuals at I_0 for the branch predicates on P are:

$$R(BP1, I_0, P) = -1,$$
$$R(BP2, I_0, P) = -99,$$
$$R(BP4, I_0, P) = -2.$$

Step 5: Construction of a System of Linear Constraints to be solved to obtain increments for the current input.

In this step, we construct linear constraints for computing the increments ΔI_k for the current input I_k, using the linear representations computed in step 3 and predicate residual values computed in step 4.

We first convert the linear arithmetic representations of the predicate functions into a set of inequalities and equalities. If a branch predicate should evaluate to "true" for the given path to be traversed, the corresponding predicate function is converted into an inequality/equality with the same relational operator as in the branch predicate. On the other hand, if a branch predicate should evaluate to "false" for the given path, the corresponding predicate function is converted into an inequality with a reversal of the relational operator used in the branch predicate. If a branch predicate has = relational operator and should evaluate to "false" for the given path to be traversed, then we convert it into two inequalities, one with the relational operator > and the other with the relational operator <. If the corresponding predicate function evaluates to a positive value at I_k, then we consider the inequality with > operator else we consider the one with < operator. This discussion also holds when a branch predicate has $\neq$ relational operator and should evaluate to "true" for the given path to be traversed. This set of inequalities/equalities gives linear representations of the branch predicates on P as they should evaluate for the given path to be traversed.

Converting the linear arithmetic representations for the predicate functions on the path P into inequalities, we get:

$$F1 : X - Y > 0,$$
$$F2 : 2X - 2Y + Z - 100 > 0,$$
$$F4 : 2X - 2Y > 0.$$

Now using the linear arithmetic representations at I_k of the branch predicates as they should evaluate for the traversal of path P and using the predicate residuals computed at I_k, we apply the relaxation technique as described in the previous section to derive a set of linear constraints on the increments to the input.

By applying the relaxation technique to the linear arithmetic representations computed above and the predicate residuals computed in the previous step, the set of linear constraints on increments to I_0 are derived as given below:

$$F1 : \Delta X - \Delta Y > 1$$
$$F2 : 2\Delta X - 2\Delta Y + \Delta Z > 99$$
$$F4 : 2\Delta X - 2\Delta Y > 2$$

Note that the constant terms d_i from the linear arithmetic representations do not appear in these constraints.

Step 6: Conversion of inequalities to equalities.

In general, the set of linear constraints on increments derived in the previous step may be a mix of inequalities and equalities. For automating the method of computing the solution of this set of inequalities, we convert it into a system of equalities and solve it using Gaussian elimination. We convert inequalities into equalities by the addition of new variables. A simultaneous solution of this system of equations gives us the increments for I_k to obtain the next program input I_{k+1}.

Converting the inequalities to equalities in the constraint set, for the example being considered, by introducing three new variables u, v and w, we get:

$$\begin{aligned}&F1 : \Delta X - \Delta Y - u = 1,\\ &F2 : 2\Delta X - 2\Delta Y + \Delta Z - v = 99,\\ &F4 : 2\Delta X - 2\Delta Y - w = 2,\end{aligned}$$

where we require that $u, v, w > 0$.

Step 7: Solution of the System of Linear Equations.

We simultaneously solve the system of linear equations obtained in the previous step using Gaussian elimination. If the number of branch predicates on the path is equal to the number of unknowns (input variables and new variables) and it is a consistent nonsingular system of equations, then a straightforward application of Gaussian elimination gives the solution of this system of equations. If the number of branch predicates on the path is more than the number of unknowns, then the system of equations is *overdetermined* and there may or may not exist a solution depending on whether the system of equations is consistent or not. If the system of equations is consistent then again a solution can be found by applying Gaussian elimination to a subsystem with the number of constraints equal to the number of variables, and verifying that the solution satisfies the remaining constraints. If the system of equations is not consistent, it is possible that the path is infeasible. In such a case, a consistent subsystem of the set of linear constraints is solved using Gaussian elimination and used as program input for the next iteration. A repeated occurrence of inconsistent systems of equations in subsequent iterations strengthens the possibility of the path being infeasible. A testing tool may choose to terminate the algorithm after a certain number of occurrences of inconsistent systems with the conclusion that the path is likely to be infeasible.

If the number of branch predicates on the path is less than the number of unknowns, then the system of equations is *underdetermined* and there will be infinite number of solutions if the system is consistent. In this case, if there are n branch predicates, we select n unknowns and formulate the system of n equations expressed in these n unknowns. The other unknowns are the free variables. The n unknowns are selected such that the resulting system of equations is a set of n linearly independent equations. Then, this system of n equations in n unknowns is solved in terms of free variables, using Gaussian elimination. The values of free variables are chosen and the values of n dependent variables are computed. The solution obtained in this step gives the values by which the current program input I_k has to be incremented to obtain a next approximation I_{k+1} for the program input. We execute the predicate slices and evaluate the predicate functions at the new program input I_{k+1}. If all the branch predicates evaluate to their desired branches then I_{k+1} is a solution to the test data generation problem. Otherwise, the algorithm goes back to step 2 with I_{k+1} as the current program input for $(K+1)^{th}$ iteration.

As explained in the previous section, input dependency sets and the linear representations depend on the current input data. Therefore, they are computed again in the next iteration using I_{k+1}.

In the example considered, there are three linear constraints and six unknowns. Therefore, it is an underdetermined system of equations and can be considered as a system of three equations in three unknowns with the other three unknowns as free variables. If it is considered as a system of three equations in the three variables ΔX, ΔY and ΔZ and then Gaussian elimination is used to triangularize the coefficient matrix, we find that the third equation is dependent on first equation because the third row reduces to a row of zeros resulting in a singular matrix. Therefore, we consider it as a system of equations in ΔX, ΔZ and w:

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 2 & 0 & -1 \end{bmatrix} \begin{bmatrix} \Delta X \\ \Delta Z \\ w \end{bmatrix} = \begin{bmatrix} 1 + u + \Delta Y \\ 99 + v + 2\Delta Y \\ 2 + 2\Delta Y \end{bmatrix}$$

The values of free variables can be chosen arbitrarily such that u, v and w >0. Choosing the free variables u, v and Δy equal to 1, and solving for Δx, Δz and w, we get, $\Delta x = 3$, $\Delta z = 96$, $w = 2$. Adding above increments to I_0, we get

$$I_1 = (X_1 = 4, Y_1 = 3, Z_1 = 99).$$

Executing the predicate slices on path P using input I_1 and evaluating the corresponding predicate functions, we see that the three branch predicates evaluate to the desired branch leading to the traversal of P. Therefore, the algorithm terminates successfully in one iteration.

In this method, a new approximation of the program input is obtained from the previous approximation and its residuals. Therefore, it falls in the class of relaxation methods. The relaxation technique is used iteratively to obtain a new program input until all branch predicates evaluate to their desired outcomes by executing their corresponding predicate slices.

If it is found that the method does not terminate in a given time, then it is possible that either the time allotted for test data generation was insufficient or there was an accumulation of round off errors during the Gauss elimination method due to the finite precision arithmetic used. Gaussian elimination is a well established method for solving a system of linear equations. Its implementations with several pivoting strategies are available to avoid the accumulation of round off errors due to finite precision arithmetic. Besides these two possibilities, the only other reason for the method not terminating in a given time is that the path is infeasible.

It is clear from the construction of linear representations in step 3 that if the function of input computed by a predicate function is linear, then the corresponding linear arithmetic representation constructed by our method is the exact representation of the function computed by the predicate function. We prove that in this case, the desired program input is obtained in only one iteration.

Theorem 1
If the functions of input computed by all the predicate functions for a path P are linear, then either the desired program input for the traversal of the path P is obtained in one iteration or the path is guaranteed to be infeasible.

Proof
Let us assume that there are m input variables for the program containing the given path P and there are n branch predicates $BP1, BP2, ...BPn$ on the path P such that $n1$ of them use " $=$ ", $n2$ use " $>$ " and $n3$ use the relational operator " $\geq$ ". The linear representations for the predicate functions corresponding to the $n = (n1 + n2 + n3)$ branch predicates on P can be computed by the method described in Step 3 of the algorithm. Note that these representations will be exact because the functions of input computed by the predicate functions are linear. We can write the branch predicates on path P in terms of these representations as follows:

$$
\begin{array}{l}
a_{1,1}x_1 \ + a_{1,2}x_2 \ + ... \ + a_{1,m}x_m \ + a_{1,0} \ = \ 0 \\
... \\
a_{n1,1}x_1 + a_{n1,2}x_2 + ... \ + a_{n1,m}x_m + a_{n1,0} = \ 0 \\
b_{1,1}x_1 \ + b_{1,2}x_2 \ + ... \ + b_{1,m}x_m \ + b_{1,0} \ > \ 0 \\
... \\
b_{n2,1}x_1 + b_{n2,2}x_2 + ... \ + b_{n2,m}x_m + b_{n2,0} > \ 0 \\
c_{1,1}x_1 \ + c_{1,2}x_2 \ + ... \ + c_{1,m}x_m \ + c_{1,0} \ \geq \ 0 \\
... \\
c_{n3,1}x_1 + c_{n3,2}x_2 + ... \ + c_{n3,m}x_m + c_{n3,0} \geq \ 0
\end{array}
$$

eq. set 1

Note that the coefficients corresponding to input variables not in the input dependency set of a predicate function will be zero.

Let $I_0 = (x_{10}, x_{20}, ..., x_{m0})$ be an approximation to the solution of above set of equations. Then we have:

$$
\begin{array}{l}
a_{1,1}x_{10} \ + a_{1,2}x_{20} \ + ... \ + a_{1,m}x_{m0} \ + a_{1,0} \ = \ r_{1,1} \\
... \\
a_{n1,1}x_{10} + a_{n1,2}x_{20} + ... \ + a_{n1,m}x_{m0} + a_{n1,0} = \ r_{1,n1} \\
b_{1,1}x_{10} \ + b_{1,2}x_{20} \ + ... \ + b_{1,m}x_{m0} \ + b_{1,0} \ > \ r_{2,1} \\
... \\
b_{n2,1}x_{10} + b_{n2,2}x_{20} + ... \ + b_{n2,m}x_{m0} + b_{n2,0} > \ r_{2,n2} \\
c_{1,1}x_{10} \ + c_{1,2}x_{20} \ + ... \ + c_{1,m}x_{m0} \ + c_{1,0} \ \geq \ r_{3,1} \\
... \\
c_{n3,1}x_{10} + c_{n3,2}x_{20} + ... \ + c_{n3,m}x_{m0} + c_{n3,0} \geq \ r_{3,n3}
\end{array}
$$

eq. set 2.

where $r_{i,j}$ is the residual value obtained by executing the corresponding predicate slice using I_0 and evaluating the corresponding predicate function. Let $I_f = (x_{1f}, x_{2f}, ..., x_{mf})$ be a solution of the eq. set 1. Then, executing the given program at I_f would result in traversal of the path P. The goal is to compute this solution. Substituting I_f in eq. set 1, we get:

$$
\begin{array}{l}
a_{1,1}x_{1f} \ + a_{1,2}x_{2f} \ + ... \ + a_{1,m}x_{mf} \ + a_{1,0} \ = \ 0 \\
... \\
a_{n1,1}x_{1f} + a_{n1,2}x_{2f} + ... \ + a_{n1,m}x_{mf} + a_{n1,0} = \ 0 \\
b_{1,1}x_{1f} \ + b_{1,2}x_{2f} \ + ... \ + b_{1,m}x_{mf} \ + b_{1,0} \ > \ 0 \\
... \\
b_{n2,1}x_{1f} + b_{n2,2}x_{2f} + ... \ + b_{n2,m}x_{mf} + b_{n2,0} > \ 0 \\
c_{1,1}x_{1f} \ + c_{1,2}x_{2f} \ + ... \ + c_{1,m}x_{mf} \ + c_{1,0} \ \geq \ 0 \\
... \\
c_{n3,1}x_{1f} + c_{n3,2}x_{2f} + ... \ + c_{n3,m}x_{mf} + c_{n3,0} \geq \ 0
\end{array}
$$

eq. set 3

Now subtracting eq. set 2 from eq. set 3, we get:

$$
\begin{array}{l}
a_{1,1}\Delta x_1 \ + a_{1,2}\Delta x_2 \ + ... \ + a_{1,m}\Delta x_m \ = \ -r_{1,1} \\
... \\
a_{n1,1}\Delta x_1 + a_{n1,2}\Delta x_2 + ... \ + a_{n1,m}\Delta x_m = \ -r_{1,n1} \\
b_{1,1}\Delta x_1 \ + b_{1,2}\Delta x_2 \ + ... \ + b_{1,m}\Delta x_m \ > \ -r_{2,1} \\
... \\
b_{n2,1}\Delta x_1 + b_{n2,2}\Delta x_2 + ... \ + b_{n2,m}\Delta x_m > \ -r_{2,n2} \\
c_{1,1}\Delta x_1 \ + c_{1,2}\Delta x_2 \ + ... \ + c_{1,m}\Delta x_m \ \geq \ -r_{3,1} \\
... \\
c_{n3,1}\Delta x_1 + c_{n3,2}\Delta x_2 + ... \ + c_{n3,m}\Delta x_m \geq \ -r_{3,n3}
\end{array}
$$

where $\Delta x_i = (x_{if} - x_{i0})$. This is precisely the set of constraints on the increments to the input that must be satisfied so as to obtain the desired input. If the increment Δx_i for x_{i0} is computed from the above set of constraints then $\Delta x_i + x_{i0} = x_{if}$, for $i = 1, m$, gives the desired solution I_f. As illustrated above, this requires only one iteration. This indeed is the set of constraints used in Step 5 of our method for test data generation. Therefore, given any arbitrarily chosen input I_0 in the program domain, our method derives the desired input in one iteration.

While solving the constraints above, if it is found that the set is inconsistent then the given path P is infeasible. Therefore, our method either derives the desired solution in one iteration or guarantees that the given path is infeasible.

3.1 Paths with Nonlinear Predicate Slices.

If the function of input computed by a predicate function is nonlinear, the predicate function is locally approximated by its tangent plane in the neighborhood of the current input I_k. The residual value computed at I_k provides feedback to the tangent plane at I_k for the computation of increments to I_k so that if the tangent plane was an exact representation of the predicate function, the predicate function will evaluate to the desired outcome in the next iteration. Because the slope correspondence between the tangent plane and the predicate function is local to the current iteration point I_k, it may take more than one iteration to compute a program input at which the execution of predicate slice results in evaluation of the branch predicate to the desired branch outcome.

Let us now consider a **path with a predicate function computing a second degree function of the input,** for the example program in Figure 1,

$$P = \{0, 1, P1, 2, P2, P3, 7, 8, P4, 9\}$$

with initial program input $I_0 = (X_0 = 1, Y_0 = 2, Z_0 = 3)$. The path P is not traversed on I_0. Therefore, input I_0 is iteratively refined. The predicate slices and the input dependency sets of branch predicates $BP1, BP2$ and $BP4$ are the same as in the example on path with linear predicate slices. For $BP3$,

$$S(BP3, P) = \{0\} \text{ and}$$
$$ID(BP3, I_0, P) = \{X, Z\}.$$

Also, the linear representations for the predicate functions $F1, F2$ and $F4$ are the same as for the example in the previous section. For $F3$, we have

$$F3 : 3X + 7Z - 114.$$

The slope of F3 with respect to Z is computed by evaluating the divided difference at (1,2,3) and (1,2,4). The above linear function represents the **tangent plane at** I_0 of the nonlinear function computed by the predicate function corresponding to branch predicate $BP3$.

Converting each of these functions into an inequality with the relational operator that the branch predicate should evaluate to, we get:

$$F1 > 0 : X - Y > 0,$$
$$F2 \leq 0 : 2X - 2Y + Z - 100 \leq 0,$$
$$F3 \geq 0 : 3X + 7Z - 114 \geq 0,$$
$$F4 > 0 : 2X - 2Y > 0.$$

Note that the relational operator for the representation for $BP2$ *is different from that of the example in previous section because a different branch is taken.*

The predicate residuals at I_0 for the predicate slices on P are:

$$R(BP1, I_0, P) = -1, \qquad R(BP2, I_0, P) = -99,$$
$$R(BP3, I_0, P) = -90, \qquad R(BP4, I_0, P) = -2.$$

The set of linear constraints to be used for computing the increment for I_0 using the results of above two steps are:

$$F1 : \Delta X - \Delta Y > 1,$$
$$F2 : 2\Delta X - 2\Delta Y + \Delta Z \leq 99,$$
$$F3 : 3\Delta X + 7\Delta Z \geq 90,$$
$$F4 : 2\Delta X - 2\Delta Y > 2.$$

The inequalities in the above constraint set are converted to equalities by introducing new variables $s > 0, t \geq 0, u \geq 0$ and $v > 0$. The resulting system of equations in ΔX, ΔY, ΔZ and v is solved using Gaussian elimination.

$$\begin{bmatrix} 1 & -1 & 0 & 0 \\ 2 & -2 & 1 & 0 \\ 3 & 0 & 7 & 0 \\ 2 & -2 & 0 & -1 \end{bmatrix} \begin{bmatrix} \Delta X \\ \Delta Y \\ \Delta Z \\ v \end{bmatrix} = \begin{bmatrix} 1+s \\ 99-t \\ 90+u \\ 2 \end{bmatrix}$$

The free variables s, t and u are arbitrarily chosen equal to 1, and the system is solved for ΔX, ΔY, ΔZ, and v. The solution of the above system is:

$$\Delta X = -189, \quad \Delta Y = -191, \quad \Delta Z = 94, \quad v = 2.$$

These increments are added to I_0 to obtain a new input I_1.

$$I_1 = (1 + \Delta X, 2 + \Delta Y, 3 + \Delta Z) = (-188, -189, 97).$$

Executing the predicate slices on P using the program input I_1, we find that all the four branch predicates evaluate to their desired branches resulting in the traversal of P. Therefore, the algorithm terminates successfully in one iteration. We summarize the results in the following table.

Iteration #	X	Y	Z	BP1	BP2	BP3	BP4
0	1	2	3	F	F	F	F
1	-188	-189	97	T	F	T	T

This example illustrates that the tangent plane at the current input is a good enough linear approximation for the predicate function in the neighborhood of the current input.

Now we consider **a path with a predicate function computing the sine function of the input**. Let us consider the following path P for the program in Figure 1,

$$P = \{0, 1, P1, 3, P2, 4, 5, 6, P4, P5, 10\}$$

with initial program input $I_0 = (X_0 = 1, Y_0 = 2, Z_0 = 3)$. The path P is not traversed on I_0 because $BP2$ evaluates to an undesired branch on I_0. Therefore, the steps for iterative refinement of I_0 are executed. We summarize the results of execution of our test data generation algorithm for this example in the table given below.

Iter. #	X	Y	Z	BP1	BP2	BP4	BP5
0	1	2	3	F	F	F	T
1	-80.15	-79.15	180.5	F	T	F	F
2	-12.55	-11.55	112.55	F	T	F	F
3	-4.58	-3.58	104.58	F	T	F	F
4	-0.57	0.43	100.57	F	T	F	T

For path P to be traversed, the branch predicates $BP1$ and $BP4$ should evaluate to "false" and the branch predicates $BP2$ and $BP5$ should evaluate to "true". As shown in the table, through iterations 1 to 4 of the algorithm $BP1$, $BP2$, and $BP4$ continue to evaluate to their desired outcomes and the values of inputs X, Y and Z are incremented such that $F5$ moves closer to zero in each iteration. Finally in iteration 4, $F5$ becomes positive for program input I_4 and therefore $BP5$ becomes true.

We would like to point out that if the linear arithmetic representation of a branch predicate is exact, then the branch predicate evaluates to its desired outcome in the first iteration and continues to do so in the subsequent iterations. Whereas, if the linear arithmetic representation approximates the branch predicate in the neighborhood of current input (as in the case of $BP5$) by its tangent plane, then although in each iteration the refined input evaluates to the desired outcome with respect to the tangent plane, it may take several iterative refinements of the input for the corresponding branch predicate to evaluate to its desired outcome.

In this example, $BP5$ evaluated to "true"(its desired outcome) at I_0, but it evaluated to "false" at I_1. This is because the predicate residual provides the feedback to the linear arithmetic representation (i.e., the tangent plane to the sine function) of $BP5$ and the input is modified by stepping along this linear arithmetic representation. As a result, the linear representation evaluates to a positive value at I_1, but the change in the program input still falls short of making the predicate function $F5$ evaluate to a positive value at I_1. In the subsequent iterations, the input gets further refined and finally in the fourth iteration, the predicate function $F5$ evaluates to a positive value.

As illustrated by this example, after the first iteration, all the branch predicates computing linear functions of the input continue to evaluate to their desired outcomes as the input is further refined to satisfy the branch predicates computing nonlinear functions of input. During regression testing, a branch predicate or a statement on the given path may be changed. To generate test data so that the modified path is traversed, an input on which other branch predicates already evaluate to their desired outcomes will be a good initial input to be refined by our method. Therefore, during regression testing, we can use the existing test data as the initial input and refine it to generate new test data.

3.2 Arrays and Loops

When arrays are input in a procedure, one of the problems faced by a test data generation method is the size of the input arrays. Our test data generation method considers only those array elements that are referenced when the predicate slices for the branch predicates on the path are executed and the corresponding predicate functions are evaluated. The input dependency set for a given input identifies the input variables that are relevant for a predicate function. Therefore, the test data generation algorithm uses and refines only those array elements that are relevant. This makes the test data generation independent of the size of input arrays.

We illustrate how our method handles arrays and loops by generating test data for a program from [10] given in Figure 4. We take the same path and initial input as in [10] so that we can compare the performance of the two program execution based test data generation methods. Therefore,

$$P = \{1, 2, 3, 4, P1_1, P2_1, P3_1, 7, P1_2, P2_2, P3_2, 6, 7, P1_3, 8\}$$

where Pi_j denotes the j^{th} execution of the predicate Pi; with initial program input:

$$\begin{aligned} I_0 = (\ & Low = 39, High = 93, Step = 12, \\ & A[1] = 1, ..., A[100] = 100). \end{aligned}$$

The path P is not traversed on I_0. Therefore, the steps for iterative refinement of I_0 are executed. Let l, h, s denote low, $high$, $step$ respectively, and $x = A[l]$, $y = A[l+s]$, $z = A[l+2s]$, then the predicate slices and input dependency sets of the branch predicates on P are:

$$\begin{array}{ll} S(BP1_1, P) = \{1, 4\}, & ID(BP1_1, I_0, P) = \{l, s, h\}, \\ S(BP2_1, P) = \{1, 3, 4\}, & ID(BP2_1, I_0, P) = \{x, y\}, \\ S(BP3_1, P) = \{1, 2, 4\}, & ID(BP3_1, I_0, P) = \{x, y\}, \\ S(BP1_2, P) = \{1, 4, 7\}, & ID(BP1_2, I_0, P) = \{l, s, h\}, \\ S(BP2_2, P) = \{1, 3, 4, 7\}, & ID(BP2_2, I_0, P) = \{x, z\}, \\ S(BP3_2, P) = \{1, 2, 4, 7\}, & ID(BP3_2, I_0, P) = \{x, z\}, \\ S(BP1_3, P) = \{1, 4, 7, 6, 7\}, & ID(BP1_3, I_0, P) = \{l, s, h\}. \end{array}$$

The linear representations for the predicate functions corresponding to the branch predicates on P are:

$$\begin{array}{ll} F1_1 : l + s - h, & F2_1 : x - y, \\ F3_1 : x - y, & F1_2 : l + 2s - h, \\ F2_2 : x - z, & F3_2 : x - z, \\ F1_3 : l + 3s - h. & \end{array}$$

The predicate residuals at I_0 for the predicate functions of the branch predicates on P are:

$$\begin{array}{ll} R(BP1_1, I_0, P) = -42, & R(BP2_1, I_0, P) = -12, \\ R(BP3_1, I_0, P) = -12, & R(BP1_2, I_0, P) = -30, \\ R(BP2_2, I_0, P) = -24, & R(BP3_2, I_0, P) = -24, \\ R(BP1_3, I_0, P) = -18. & \end{array}$$

The set of linear constraints to be used for computing increment to I_0 using the results of above two steps are:

$$\begin{array}{ll} F1_1 : \Delta l + \Delta s - \Delta h < 42, & F2_1 : \Delta x - \Delta y \geq 12, \\ F3_1 : \Delta x - \Delta y \leq 12, & F1_2 : \Delta l + 2\Delta s - \Delta h < 30, \\ F2_2 : \Delta x - \Delta z \geq 24, & F3_2 : \Delta x - \Delta z > 24, \\ F1_3 : \Delta l + 3\Delta s - \Delta h \geq 18. & \end{array}$$

The above inequalities are converted to equalities by introducing seven new variables a, b, c, d, e, f and g. where a, d, $f > 0$; and b, c, e and $g \geq 0$. Considering it a system of equations expressed in unknowns Δl, Δs, d, Δx, Δy, b and e, we get:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 3 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta l \\ \Delta s \\ d \\ \Delta x \\ \Delta y \\ b \\ e \end{bmatrix} = \begin{bmatrix} 42 + \Delta h - a \\ 12 \\ 12 - c \\ 30 + \Delta h \\ 24 + \Delta z \\ 24 + f + \Delta z \\ 18 + g + \Delta h \end{bmatrix}$$

The unknowns and free variables are selected so as to obtain a nonsingular system of equations. The values of free variables can be chosen arbitrarily with the constraints that a, d, $f > 0$; and b, c, e and $g \geq 0$. The values of free variables f, Δz, Δh, and g are chosen as 1. The value of free variable a is 3 for integer arithmetic. Solving for the unknown variables using Gaussian elimination, we get:

$$\begin{array}{llll} b = 0, & c = 0, & e = 1, & \Delta y = 14, \\ \Delta x = 26, & d = 1, & \Delta s = -10, & \Delta l = 50, \\ f = 1, & \Delta z = 1, & \Delta h = 1, & g = 1, \\ a = 3. & & & \end{array}$$

The new input generated after the first iteration is:

$$\begin{aligned} I_1 = (& l = 89, h = 94, s = 2, A[89] = A[39] + \Delta x, \\ & A[91] = A[51] + \Delta y, A[93] = A[63] + \Delta z). \end{aligned}$$

The input values of $A[39], A[51]$ and $A[63]$ are copied into $A[89], A[91]$ and $A[93]$ respectively and then the increments computed in this iteration are added to $A[89], A[91]$ and $A[93]$, giving:

$$\begin{aligned} I_1 = (\ & Low = 89, High = 94, Step = 2, \\ & A[89] = 65, A[91] = 65, A[93] = 64). \end{aligned}$$

This step is important because the increments computed in the current iteration have to be added to the input used in the current iteration but the resulting values have to be copied into the array elements to be used in the next iteration. Only elements $A[89], A[91]$, and $A[93]$ are relevant for the next iteration.

By executing the predicate slices for P on the program input I_1 and evaluating the corresponding predicate functions, we see that all the branch predicates evaluate to their desired branch outcomes resulting in the traversal of P. All the predicate functions corresponding to branch predicates on P compute linear functions of input. Therefore, as expected the algorithm terminates successfully in one iteration. We summarize the results of this example in the table in Figure 4. Korel in [10] obtains test data for the above path in **21** program executions, whereas our method finds a solution in only one iteration with only **8** program executions. One program execution is used in the beginning to test whether path P is traversed on I_0, six additional program executions are required for computation of all the slope computations for linear representations and one more program execution is required to test whether path P is traversed on I_1.

If we choose the path

$$P = \{1, 2, 3, 4, P1_1, P2, 5, P3, 6, 7, P1_2, 7\},$$

the set of linear constraints obtained in step 3 will be inconsistent. Since, all the predicate functions for this path compute linear functions of input, from Theorem 1, we conclude that this path must be infeasible. It is easy to check that P is indeed an infeasible path.

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

1:   input(low,high,step,A);
2:   min := A[low];
3:   max := A[low];
4:   i := low + step;
P1   while i < high do
P2:      if max < A[i] then
5:           max := A[i];
P3:      if min > A[i] then
6:           min := A[i];
7:       i := i + step;
     endwhile;
8:   output(min,max);
```

Iteration #	low	high	step	A[39]	A[51]	A[63]	A[89]	A[91]	A[93]
0	39	93	12	39	51	63	89	91	93
1	89	94	2	39	51	63	65	65	64

Iteration #	$BP1_1$	$BP2_1$	$BP3_1$	$BP1_2$	$BP2_2$	$BP3_2$	$BP1_3$
0	T	T	F	T	T	F	T
1	T	F	F	T	F	T	F

Figure 4: An example using an array and a loop.

4 Related Work

The most popular approach to automated test data generation has been through path oriented methods such as symbolic evaluation and actual program execution. One of the earliest systems to automatically generate test data using symbolic evaluation only for linear path constraints was described in [4]. It can detect infeasible paths with linear path constraints but is limited in its ability to handle array references that depend on program input. A more recent attempt at using symbolic evaluation for test data generation for fault based criteria is described in [6]. In this work, a test data generation system based on a collection of heuristics for solving a system of constraints is developed. The constraints derived are often imprecise, resulting in an approximate solution on which the path may not be traversed. Since the test data is not refined further so as to eventually obtain the desired input, the method fails when the path is not traversed on the approximate solution.

A program execution based approach that requires a partial solution to test data generation problem to be computed by hand using values of integer input variables is described in [14]. There is no indication on how to automate the step requiring computation by hand. Program execution based approaches for automated test data generation have been described in [11, 8], but they have been developed for statement and branch testing criteria.

An approach to automatically generate test data for a given path using the actual execution of the program is presented in [10]. Another program execution based approach that uses program instrumentation for test data generation for a given path has been reported in [7]. These approaches consider only one branch predicate and one input variable at a time and use backtracking. Therefore, they may require a large number of iterations even if all the branch conditionals along the path are linear functions of the input. If several conditionals on the selected path depend on common input variables, a lot of effort can be wasted in backtracking. They cannot consider all the branch predicates on the path simultaneously because the path may not be traversed on an intermediate input. The concept of predicate slice allows us to evaluate each branch predicate on the path independently on an intermediate input even though the path may not be traversed on this input. This makes our technique more efficient than other execution based methods.

Our method is scalable to large programs since the number of program executions in each iteration is independent of the path length and at most equal to the number of input variables plus one. If there are m input variables, in each iteration, at most m executions of the input and assignment statements on the given path are required to compute the slope coefficients. The values of the predicate functions at input I_k are known from the previous iteration. One execution of the input and assignment statements on the path is required to test whether the path is traversed on I_{k+1}.

Our method uses Gaussian elimination to solve the system of linear equations, which is a well established and widely implemented technique to solve a system of linear equations. Therefore, our method is suitable for automation. The size of the system of linear equations to be solved increases with an increase in the number of branch conditionals on a path, but the increase in cost in solving a larger system is significantly less than that of existing execution based methods.

5 Conclusions

In this paper, we have presented a new program execution based method, using well established mathematical techniques, to automatically generate test data for a given path. The method is an innovative application of the traditional relaxation technique used in numerical analysis to obtain an exact solution of an equation by iterative improvement of an approximate solution. The results obtained from this method for test data generation are very promising. It provides a practical solution to the automated test data generation problem. It is easy to implement as the tools required are already available. It is more efficient than existing program execution based approaches as it requires fewer program executions because all the branch predicates on the path are considered simultaneously, and there is no backtracking. It can also detect infeasibility for a large class of paths in a single iteration. Because it is execution based, it can handle different programming language features. We are working on extending the technique for strings and pointers.

References

[1] J. Bauer and A. Finger, "Test Plan Generation using Formal Grammars," *Proc. 4th Int. Conf. Software Engineering,* pp. 425-432, 1979.

[2] D. Bird and C. Munoz, "Automatic Generation of Random Self-checking Test Cases," *IBM Syst. J.*, vol. 22, no. 3, pp. 229-245, 1983.

[3] C. Bischof, L. Roh, and A. M-Oats, "ADIC: An Extensible Automatic Differentiation Tool for ANSI-C," *to appear Software: Practice and Experience.*

[4] 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.

[5] W.H. Deason, D.B. Brown, K. Chang and J.H. Cross, "A Rule Based Software Test Data Generator," *IEEE Trans. Knowledge and Data Eng.,* vol. 3, no. 1, pp.108-116, Jan 1991.

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

[7] 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.

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

[9] W. Jessop, J. Kanem, S. Roy, and J. Scanlon, "ATLAS - An Automated Software Testing System," *2nd International Conference on Software Engineering*, 1976.

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

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

[12] N. Lyons, "An Automatic Data Generation System for Data Base Simulation and Testing," *Data Base*, vol. 8, no. 4, pp. 10-13, 1977.

[13] E. Miller, Jr. and R. Melton, "Automatic Generation of Testcase Datasets," *SIGPLAN Notices*, vol. 10, no. 6, pages 51-58, June 1975.

[14] W. Miller and D.L. Spooner, "Automatic Generation of Floating-Point Test Data," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 3, pages 223-226, September 1976.

[15] F. Scheid, "Numerical Analysis," *Schaum's Outline Series*, McGraw-Hill Book Company, 1968.