UNA Based Iterative Test Data Generation and its Evaluation

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
Univ. of Pittsburgh
Pittsburgh, PA 15260
soffa@cs.pitt.edu

Abstract

A number of approaches have been proposed to automatically generate test data to traverse a given path in a program. In this paper, we present a program execution based approach to generate test data for a given path. The technique derives a desired input for a test path by iteratively refining an arbitrarily chosen input. A set of linear constraints on the increments to the input are derived to refine the input. We solve this constraint set using a Unified Numerical Approach (UNA) developed in this paper. Our technique can generate both integer and floating point inputs as well as handle arrays and loops. We determine a basis set of paths for a program and use our technique to generate test data for this set. We implemented and experimentally evaluated our technique. We present results of generating input for scientific programs. The experimental results show that the technique is effective in that it generates input for most of the paths in the basis sets and also efficiently detects linear infeasible paths. Our experiments also show that our technique is efficient in the number of iterations required to generate test data. The time performance shows that it provides a practical method to automatically generate test data for scientific programs.

Keywords - *Path testing, scientific programs, iterative relaxation technique, testing tools.*

1 Introduction

Generating test data to exercise a given path in a program is an important but challenging phase of the software testing process. In an attempt to avoid the expense of manually generating input that exercises a path, a number of techniques that automatically generate input have been proposed in the literature. Symbolic evaluation based approaches [2, 3] have been developed but are limited in handling arrays and pointers since the element being referenced may depend on program input and therefore it may not be possible to determine the variable being referenced without executing the program. Program execution based approaches [7, 4] have also been proposed that consider one branch predicate on the given path at a time and use function minimization to force the branch predicate to evaluate to a desired outcome. While trying to modify the input to force execute the selected branch, if a preceeding branch evaluates to an undesired outcome, then the method backtracks to the preceeding branch predicate and the process is repeated by minimizing the function associated with that predicate. The function minimization for each branch predicate is carried out with respect to one input variable at a time and cycles through all the input variables in search for an input that forces the predicate to evaluate to the desired outcome. These methods are inefficient in handling infeasibile paths. If the selected path is infeasible and the infeasibility is not detected, a large number of attempts can be performed before the search procedure terminates and a lot of effort can be wasted [7].

In a previous paper [5], we proposed a method for automatically generating test data using an iterative relaxation technique. 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 are computed by solving the set of linear constraints. We proposed converting the inequalities in the constraint set to equalities by introducing free variables and solving the resulting system using Gaussian elimination. The increments thus computed 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 using a relaxation technique to obtain the desired input.

In this paper, we improve on our previous technique [5] and also present experimental results demonstrating that the new technique is viable, practical and effective. An important step in our previous method is solving the set of linear inequalities obtained from the branch predicates and then using the solution to refine the current input. If the system of inequalities is solved using the Gaussian elimination technique, very often the ad hoc choice of values for the free variables results in an inconsistent system of linear equations. In that case, another choice has to be made for values of free variables and the procedure repeated to obtain a consistent system of linear equations.

The **Unified Numerical Approach (UNA)** presented in this paper improves on our previous scheme by using the Least Square Error Solution for over determined systems. The value of this technique is that even when a system of linear equations resulting from the choice of values for free variables is inconsistent, the least squared error solution can systematically guide the next choice of values for free variables so as to obtain a solution for the set of inequalities. Once we have a solution for the system of inequalities, we can refine the current input to obtain a new input that exercises the path. The technique is powerful in that it is able to work well with both integer and real input values. As was the case with our previous technique, we can detect if a path is infeasible when the conditionals on the path are linear functions of the input. Also, when the conditionals are linear functions of the input (and path is feasible) our method derives the input for the path in one iteration. When the conditionals are non linear functions of the input, we are also able to generate input in many cases but the technique may fail to produce the input in some cases. In these cases, the path may be infeasible.

The technique as presented in [5] was developed for programs with numeric inputs such as real and integer variables. The functions used in scientific computing represent an important class of routines that take numeric input and perform complex manipulations of the input. The number of iterations needed by our test data generation technique depend on the semantic complexity of the functions of input computed by the branch predicates on the path. Therefore, we selected some functions used in scientific computing from *Numerical Recipes in C* [9] with a wide range of semantically complex branch predicates for experimental evaluation of our test data generation technique. We determine a basis set of paths for each of the benchmark programs and derive inputs for the paths in this set. We also demonstrate that our technique can be used for programs with character inputs since the character variables are also numeric variables with the restriction that the integer values representing them should be within a valid range. Our experimental results show that the technique was able to generate input for most of the paths in the basis sets or determine they were infeasible.

Thus, the important contributions of this paper are:

- A new method called Unified Numerical Approach (UNA) is developed for solving a system of linear inequalities. It is used for generating an input to exercise a test path by iteratively refining an arbitrarily chosen input. The technique can handle integer as well as real variables.

- This method inherits the characteristics of our previously presented method: (1) when all the conditionals on a path are linear functions of the input, either the input is derived or the path is determined to be infeasible in one iteration; (2) when at least one conditional is a non linear function of the input, the technique may take more than one iteration to generate the input. When a desired input is not generated after a threshold number of iterations, the process can be halted with no input or conclusions as to whether the path is likely to be infeasible.

- Our experimental results show that technique is effective in that it is able to generate input most of the time.

- Our experimental results also show that the technique is efficient and practical.

The organization of this paper is as follows. An overview of the test data generation method is presented in the next section. Our novel technique, UNA, that systematically solves a set of linear inequalities is presented in section 3. The extensions to the basic test data generation technique are explained in section 4. The evaluation methodology, benchmarks used and the results of experiments are discussed in section 5. Finally, the important features of the work presented in this paper are summarized in section 6.

2 Overview of the Method

We now briefly review our iterative relaxation method for generating test data to exercise a given path [5]. The problem is to generate an input such that the branch predicates on the path evaluate to their desired outcomes when the program is executed using this input. 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 $\{<, \leq, >, \geq, =, \neq\}$.[1] Each branch predicate $E_1 \ op \ E_2$ can be equivalently written as $(E_1 - E_2) \ op \ 0$. Along a given path, $E_1 - E_2$ represents a real valued function of the input computed by the corresponding branch predicate. We call the expression $E_1 - E_2$ the **Predicate Function** of the branch predicate.

We derive a **Linear Arithmetic Representation** of the predicate function for each branch predicate on the given path. 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 predicate slice for a branch predicate on the path essentially consists of all the statements along the path whose execution is necessary to evaluate the branch predicate. 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 value of the predicate function of a branch predicate for an input can be computed by executing its predicate slice using the input. By executing the predicate slice and taking a slice over the dynamic data dependence graph starting from the branch predicate, the set of input variables on which the value of the predicate function depends are identified.

To derive the linear arithmetic representation for a predicate function of branch predicate, we first write a general linear expression in terms of the relevant input variables. For example, if the value of a branch predicate $E_1 \ op \ E_2$ depends on inputs x, y and z, then a general linear formulation in x, y and z is given by

$$ax + by + cz + d.$$

In the above linear expression, if a, b and c respectively are the derivatives of the predicate function with respect to x, y and z, then it represents the tangent plane to the predicate function. We approximate the values of these derivatives by their respective divided differences, which are computed by executing the predicate slice using an input I and then using inputs $I+(\Delta x, 0, 0)$, $I+(0, \Delta y, 0)$ and $I+(0, 0, \Delta X)$. For the above example, the linear arithmetic representation is

$$ax + by + cz + d \ op \ 0$$

[1] The predicates involving boolean operators and boolean variables can be handled using the above simple arithmetic relational expressions as explained in [5]

where *op* is the relational operator in the corresponding branch predicate. Using this method, we derive a linear arithmetic representation at the current input for each predicate function on the given path.

Our test data generation for the given path begins with an arbitrarily chosen input I_0. We then iteratively refine this input to derive an input for the traversal of the path. To refine the current input, we derive a set of linear constraints on the increments to the current input using the linear arithmetic representations. The linear arithmetic representation of a branch predicate for an input essentially allows us to map the change in the value of the predicate function to changes in the program input.

For example if the predicate function of a branch predicate evaluates to a positive value r at the current input, whereas the predicate function should evaluate to a negative value for the traversal of the path, then the constraint on the increments to the current input is given as:

$$a\Delta x + b\Delta y + c\Delta z < -r.$$

The above constraint means that the increments Δx, Δy, and Δz to the current input should be such that the value of the predicate function is decreased by $-r$. As a result when the predicate function is evaluated at $I + (\Delta x, \Delta y, \Delta z)$, it evaluates to a negative value because the incremented input decreases the value of the predicate function by $-r$. Note that the constant term d does not appear in the constraint.

We derive a linear constraint on the increments to the input for each branch predicate on the path. We then solve the resulting system of constraints to compute the increments for the current input. We add these increments to the current input to obtain a refined input. If the given path is traversed on the refined input then the algorithm terminates; otherwise it is further refined to obtain a desired input.

While implementing our method for test data generation, we needed a method to solve a system of linear constraints to compute the increments to the current input. We have developed a systematic method called Unified Numerical Approach (UNA) to solve a system of linear inequalities that works for both real and integer variables. In the next section, we describe our method to solve a system of linear inequalities.

3 Solving the set of linear constraints

An important step in the method explained in section 2 is solving the set of linear inequalities obtained to refine the current input. The existing techniques to solve a system of linear inequalities formulate the problem as a linear programming problem with a fictitious objective function to be minimized [1]. Moreover these techniques work only for real variables and to compute integer solutions integer programming has to be used. Actually, the problem that we need to solve is to find a feasible solution and not an optimal solution. Also the program may contain a mix of real and integer input variables. Therefore, instead of using linear programming techniques that search for an optimum solution at the vertices of the convex space of feasible solutions, we developed a new interior point numerical method called **Unified Numerical Approach (UNA)** that iteratively moves a point from an infeasible region to the feasible region. The importance of this technique is that it can be used to systematically compute a real, integer and mix of real and integer feasible solution of a set of linear constraints consisting of inequalities and equalities. The technique is based on computing a least squared error solution of an over determined system of linear equations and using the residual values obtained for the linear constraints to move towards a feasible solution. We now explain our technique for solving a set of inequalities with an example.

Step 1: We first convert all the inequalities to a standard form $expr\ relop\ 0$, where the $relop$ is $>$ or $\geq$. This ensures that substituting a feasible solution in the $expr$ of these inequalities results in a nonnegative value. We then convert the linear inequalities in the above standard form to equalities by introducing new variables called artificial variables. For example, an inequality of the form $expr \geq 0$ is converted into an equation $expr - u = 0$ by introducing an artificial variable $u \geq 0$. To illustrate the method, let us consider the following three inequalities in two variables x and y plotted in Figure 1.

$$x \leq y; \qquad x + y \geq 0; \qquad y \geq -2.$$

Converting them into the form $expr \geq 0$ we obtain

$$-x + y \geq 0; \qquad x + y \geq 0; \qquad y + 2 \geq 0. \qquad (1).$$

Now, adding artificial variables, we obtain

$$-x + y - u = 0; x + y - v = 0; \quad y + 2 - w = 0,$$

where we require that u, v and $w \geq 0$.

Step 2: We begin by choosing all the artificial variables to be zero and compute the least squared error solution of the resulting system of equations. For the example being considered, we begin by choosing u, v, $w = 0$. We have a system of linear equations $A * X = B$, where

$$A = \begin{pmatrix} -1 & 1 \\ 1 & 1 \\ 0 & 1 \end{pmatrix}, B = \begin{pmatrix} 0 \\ 0 \\ -2 \end{pmatrix} \ and\ X = \begin{pmatrix} x \\ y \end{pmatrix}.$$

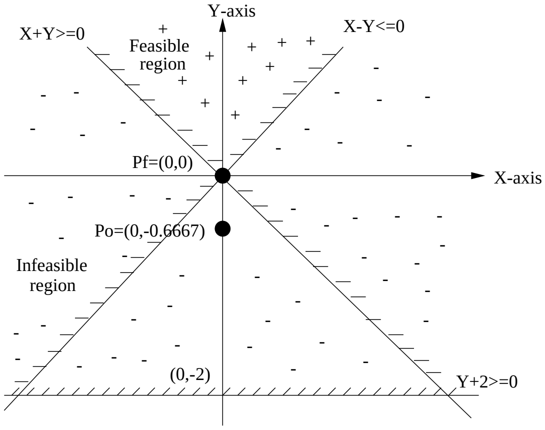

Figure 1. Movement of least squared error solution from an infeasible region to the feasible region

Now computing the least squared solution of the system of linear equations $A * X = B$, we get $x = 0$ and $y = -0.6667$. This point is shown by P_0 in Figure 1, which is clearly in an infeasible region.

Step 3: We compute the residual value for each of the linear constraint in the system using the least squared error solution computed in Step 2. This is done by substituting the least squared solution obtained above in equation set (1). For our example, the residual vector R is given by:

$$R = \begin{pmatrix} -0.6667 \\ -0.6667 \\ 1.3333 \end{pmatrix}$$

If the residuals of all the constraints with $relop \geq$ are nonnegative, those with $relop >$ are positive and the ones with $relop =$ are equal to zero, then the least squared error solution is a feasible solution of the original system of linear constraints with inequalities and equalities, and the method terminates. Otherwise, for each linear inequality with negative residual value and for each equality with a nonzero residual value, we increase the value of the corresponding artificial variable by the negative of the residual value for the constraint and recompute the least squared error solution of the system. Increasing the value of the artificial variable of an inequality ensures that the new least squared error solution makes the new residual more positive. Therefore, the least squared error solution moves towards the feasible region. This process is repeated until a feasible solution is obtained or it is concluded that the original set of inequalities are unsatisfiable. In our example, only the residual values for the first and second inequalities are negative. Therefore, incrementing the value of u by 0.6667 and of v by 0.6667, we get a new system of linear equations $A * X = B$ where now B is given by

$$B = \begin{pmatrix} 0.6667 \\ 0.6667 \\ -2 \end{pmatrix}$$

The least squared solution of this new system is $x = 0$ and $y = -0.2222$. Again computing the residuals by substituting this least squared solution in the l.h.s of original constraints in equation set (1), we obtain:

$$R = \begin{pmatrix} -0.2222 \\ -0.2222 \\ 1.7778 \end{pmatrix}$$

We see that the residuals corresponding to the first two constraints are still negative. Therefore, the steps for incrementing the values of artificial variables and recomputing the least squared error solution and corresponding residual are repeated until the residual value for each of the inequality constraint is nonnegative and for each of the equality constraint is zero. *Note that the residuals are always computed by substituting the current least squared error solution in the original inequalities given in equation set (1).* For our example, Table 1 illustrates how the least squared solution steadily moves towards a feasible solution by increasing the values of artificial variables by the magnitudes of the corresponding negative residuals. Finally, in the 11th iteration, a feasible solution shown by the point $P_f = (x = 0, y = 0)$ is obtained.

It was also experimentally observed that the residuals oscillate in the case of an unsatisfiable set of constraints. In the example considered above, if the last inequality is changed to $y + 2 <= 0$, then we see that the residuals oscillate about the residual obtained in the first least square solution and there is no improvement. If a certain threshold number of iterations have been performed and the residuals do not improve i.e., their values oscillate, then our implementation concludes that the set of constraints is likely to be unsatisfiable. In this case, we move to next iteration with the approximate solution found in the current iteration and attempt to refine it further to force execute the path.

For integer variables, we solve the set of linear constraints assuming all the variable are real. We then use rounding to obtain integer increments for the integer inputs. Using rounding to compute the increments for the integer inputs is justified because we are only computing a feasible (and not an optimal) solution of the set of linear constraints. Also, since the variables in the constraints are the increments to the original input variables rather than encoded variables, i.e, we are using a direct integer model for the integer inputs, rounding can be used to compute the desired values of the integer inputs [11]. If the inequalities are not satisfied after rounding the real solution, we compute the residual values for the inequalities using the rounded solution and use them to compute a new refined integer solution.

4 Other extensions

In order for our technique to be applicable to real programs, we developed the following extensions to the basic technique explained in section 2.

Handling Error Conditions.

When the predicate slices are executed on an intermediate input, some execution errors such as array index overflow or divide by zero may occur. In addition, some inputs may have range constraints. We introduce additional constraints in the predicate slices to trap these errors. We also add these constraints to the set of constraints used for refining the current input. For example, let us consider the code segment of Figure 2(a).

1: x=y/z;	0: if (z<>0)
2: sum +=x;	1: x=y/z;
P1:if (sum)	2: sum +=x;
....	P1:printf(sum);
(a) Original code.	(b) Modified code.

Figure 2. Handling division by zero.

The predicate slice of the branch predicate $P1$ will consist of both the statements numbered 1 and 2. If $z = 0$ for an intermediate input, then an exception will occur and it will be not be possible to execute the slice. Therefore, we modify the slice as shown in Figure 2(b). Now, if $z = 0$ for an input, then statement numbered 1 will not be executed and therefore, the exception will be avoided. In this case, the predicate residual of the branch predicate $P1$ will also not be correct. But the constraint corresponding to the statement 0, in the set of linear constraints used to refine the current input, may force z to have a nonzero value in the next iteration. Therefore, the current iteration will be used only to refine the input so that the divide by zero error does not occur. Once the current input is refined so that this exception does not occur, statement 1 will be executed and the residual for the branch predicate $P1$ will be computed accurately. This iteration will refine the input in order to force execute the test path. Therefore, some iterations in our method may be used only to refine the input so that the errors mentioned above

Iter#	1	2	3	4	5	6	7	8	9	10	11
Res.1	-.6667	-.2222	-.0741	-.0247	-.0082	-.0027	-.0009	-.0003	-.0001	-.0001	0
Res.2	-.6667	-.2222	-.0741	-.0247	-.0082	-.0027	-.0009	-.0003	-.0001	-.0001	0
Res.3	1.3333	1.7778	1.9259	1.9753	1.9918	1.9973	1.9991	1.9997	1.9999	1.9999	2

Table 1. The residuals for the example in Figure 1.

do not lead to an exception and others will be used to refine the input so as to force execute the test path.

Handling Function Calls.

We treat a function call in a program statement as an expression in its parameters and its global variables. We assume that the global variables used by each function are known. If a predicate function is data dependent on a function call, we compute the data dependencies of its parameters and the global variables starting at the point of function call. We compute the slope coefficients of the predicate function with respect to the input variables on which these parameters and the global variables are data dependent.

Handling Other Input Types.

Although the method presented in [5] has been developed for numeric inputs such as real and integer variables, it can also handle character inputs. We use the existing ASCII character set mappings to integers for character variables. The character variables are essentially numeric variables that take integer values within a given range. For each character variable, we introduce additional constraints to restrict its integer solution to be within valid integer range for the character. The integer solution is mapped back to the character using the ASCII character set.

Although pointer inputs for programs are not meaningful, they can be input parameters to the functions to be tested at unit level testing. After studying the functions in [9] it was found that inputs to these functions were almost always numeric such as integers and floating point numbers or pointers to integers and floating point numbers. Mostly arrays are passed as parameter by passing the address of the first element of the array, but we found that always the size of the array was also passed as an integer parameter. So, the size and shape of the data structure was determined by the input. Since the size n is also passed as input, it can be easily treated as n inputs of basic type int or $float$ as specified in the type definition of the argument. Therefore, for our current evaluation using scientific programs, it suffices not to consider test data generation for pointer inputs that point to data structures such as linked lists, trees etc. Please refer to [6] for a description of how our technique can be used to generate test data for functions with input parameters that are records or pointers to dynamic data strucures.

5 Evaluation

We selected functions from the [9] for experimental evaluation of our technique. The functions have numeric inputs that can be integer or floating point numbers or pointers to integers and floating point numbers. In each program, we selected a basis set of paths [10] for evaluating our test data generation method.

Definition: A **Basis set** of execution paths is an ordered set of paths in a program such that each path in the set introduces at least one new statement or a conditional edge that has not been covered before the path is defined.

If test cases can be derived to force execution of paths in a basis set, every statement in the program will have been guaranteed to be executed at least once and every branch predicate will have been executed on its true and false side. It should be noted that the basis set is not unique. The importance of using a basis set for test case generation is that its size is bounded by the number of predicate nodes in the control flow graph of the program plus one.

To derive a basis set of paths for a given program we construct the control flow graph of the program and number the edges at each node from left to right in a breadth first order. Initially, all the edges in the flow graph are unmarked. We begin by choosing the lowest numbered unmarked edge in the program. We consider any path from the start node through the selected edge and mark all the unmarked edges on this path. We then select the lowest numbered unmarked edge in the program and repeat the process until all the edges in the control flow graph of the program are marked. We pick a path in this set and generate an input that forces program execution through this path. If the path turns out be infeasible and there is a branch on the path that is not covered by any other path in the set, then we

Benchmark	Predicate Types	Inputs Types	Function calls	Basis Set Size	Cyclomatic Complexity
Fourier	Linear	int, real	math lib	6	8
Quicksort	Linear	int, real	none	14	14
Expint	NonLinear	int, real	math library	15	16
Ran0	NonLinear	int	none	2	2
Gamdev	NonLinear	int	ran1, math library	7	7
Getword	NonLinear	char	getchar,isletter	3	5

Table 2. Characteristics of Benchmark Programs.

pick another path though the branch and attempt to generate input for its traversal. If the branches on the infeasible path are covered by other paths in the set then we pick next path from the set. The set of paths generated in this process constitute a basis set of paths.

Evaluation Measures

We used the number of iterations needed to generate test input as a measure of the efficiency of the technique. The number of program executions in each iteration in our technique is bounded by the number of inputs and are independent of the number of predicates on the path. Therefore, the number of iterations required to generate an input for a given path also effectively measures the maximum number of program executions required to generate the input. We evaluated the effectiveness of the technique by measuring its coverage for a basis set of paths for a given program. We also evaluated its power in detecting linear infeasible paths and efficiently generating input for linear feasible paths. We evaluated our technique for generating input for nonlinear paths. We conducted experiments to evaluate the effectiveness of our technique in generating real and integer inputs. We solved the system of linear constraints using our Unified Numerical Approach (UNA) described in section 3 to generate integer inputs. We measured the time performance of our technique for generating linear constraints as well as for solving these constraints.

Benchmarks

To evaluate the effectiveness and performance of our technique, we selected the functions used in scientific computing from *Numerical Recipes in C* [9]that exhibit a range of characteristics with respect to the inputs and semantic complexity of their predicate functions. The properties of these functions and programs are summarized in Table 2. These functions have branch predicates that compute linear functions of input, piecewise linear functions of input, XORing of input bits with a MASK, square and logarithmic function of a sequence of random variates being generated by a random number generator with different integer seed values and absolute value of nonlinear functions of input. These functions were selected for evaluation of our technique because the number of iterations in our technique depend on the semantic complexity of the predicates on the path. *Note that if there are a large number of predicates on a path, but they all compute linear functions of the input, the desired input will be generated in a single iteration. If at least one of the predicates computes a nonlinear function of the input, the predicates computing linear functions of input will evaluate to desired outcomes after first iteration and the total number of iterations will be governed by the nonlinear predicate. Therefore, the selection of functions with nonlinear predicates was based on the complexity of their branch predicates instead of the size of the function.* To illustrate that our technique extends to character inputs, we considered a program that extracts words from an input string and calls on 2 other routines that take a character as an input.

5.1 Results and Discussion

We conducted the experiments on Windows NT platform using a 400MHz Pentium II machine with 128 MB RAM.

Fourier: Out of a total 6 linear paths in the basis set, three were found to be linear infeasible whereas the test data for the remaining 3 paths was generated in a single iteration each. The branches of the infeasible path were exercised by the test data generated for the other paths in the basis set and thus all the statemetns and branches in the program were covered. The max time taken to generate test data for a path in the basis set was 5.82 seconds.

Quicksort: This program was chosen because it used an array as input. Out of 14 linear paths in the basis set, 5 were detected linear infeasible. The branches in these infeasible paths were executed by the test data generated for other paths in the basis set. The maximum time taken to generate test data for a path in the basis set was 13.48 seconds.

Expint: The basis set consisted of 15 paths. The classification of paths in this basis set is shown in Table 3. The nonlinear infeasible paths were detected in a single iteration each because the constraints were determined to be infeasible and the conflicting constraints corresponded to branch predicates computing linear functions of input. Each of the feasible linear paths took a single iteration each for generation of test data. The detection of infeasible linear path also took a single iteration. The time performance and the number of iterations taken to generate input for 5 feasible nonlinear paths is shown in Table 4. One of the nonlinear feasible paths was executed with the initial input.

An interesting observation was made while conducting the experiments. A predicate computing a nonlinear function on a feasible nonlinear feasible was modified so that the modified predicate still computed a nonlinear function of the input. The set of linear constraints in the first iteration were found to be consistent, but found to be unsatisfiable in the subsequent 3 iterations. It was suspected that the nonlinear path is likely to be infeasible. Interestingly, it was possible to prove analytically that the nonlinear path was indeed infeasible.

Ran0: Even though the predicate function depended on a nonlinear boolean operation i.e., XORing of the input bits with a mask, the desired input for execution of both the paths in the basis set was obtained in a single iteration. The time performance for generation of input for each of these paths is shown in Table 4.

Gamdev: There were 7 paths in the basis set. Three of these were linear. One of the linear path was detected infeasible and input for the two feasible linear paths was generated in one iteration each. Three of the nonlinear paths were detected to be feasible. The time performance and the number of iterations taken to generate input for 3 feasible nonlinear paths are shown in Table 4. Input could not be generated for one of the nonlinear paths. The residuals improved for two iterations when they reached a point from where no further improvement was possible. When the inputs generated and the program were analyzed, it was found that the seed value passed to *rand1* function that was invoked by *gamdev* to generate a sequence of random variates had become positive. In the subsequent iterations, the seed value continued to remain positive and increased in magnitude. It was concluded that a local minima was reached which did not give the desired input. Therefore, input switched to another point with a positive seed and moved in the direction of overflow of the seed value. For positive seed values, *rand1* generates same sequence of random variates. Therefore a desired input could not be generated for this path. Since the system of linear constraints was inconsistent in a number of consequtive iterations(7 in this case), it was concluded that the path was likely to be infeasible.

Benchmark	Lin. Feas.	Lin. Inf.	Nonl. Feas.	Nonl. Inf.	Likely Inf.	Total
Fourier	3	3	-	-	-	6
Quicksort	9	5	-	-	-	14
Expint	7	1	5	2	-	15
Ran0	-	-	2	-	-	2
Gamdev	2	1	3	-	1	7
Getword	-	-	3	-	-	3

Table 3. Classification of paths in the basis sets of programs.

Benchmark	# of Iterations	Total time(sec)
Expint	8	216.792
	5	56.912
	2	3.385
	1	1.502
Ran0	1	1.352
	1	1.282
Gamdev	4	14.743
	3	179.950
	1	1.352

Table 4. Performance for Nonlinear feasible paths in basis set of programs.

Getword: There were 3 paths in the basis set. They all computed nonlinear functions of input because the calls to *isletter* introduced nonlinearities. The range checking in *isletter* made it a piecewise discrete valued function of integer inputs. The generation of input for all the 3 paths in the basis set took a single iteration each.

The experiments indicate that the test data generation method developed is efficient in number of iterations as well as the time performance. The experiments also show that the test data generation method is powerful in handling infeasible paths. The number of iterations to generate test data for a path varied from 1 to a max of 8. The time performance of the method in each iteration is determined by the time taken to comput the linear arithmetic representations and to solve the set of linear constraints. The time taken to compute the linear arithmetic representations in an iteration was found to be less than a second in all the paths considered in the experiments. The major portion of the time was taken to compute a feasible solution of the linear constraints since in the current implementation, the MATLAB package is invoked in each iteration to compute the least squared error solution of the over determined system of linear equations. MATLAB does not operate in silent mode; instead it displays all the computations being performed in the MATLAB window and hence affects the time performance of the method. A direct call to a function that computes least squared error solution of a system of equations would reduce the time taken to solve the linear constraints significantly.

The number of iterations in our technique is governed by the complexity of the functions of input computed by the branch predicates. In our test data generation technique [5], the size of the system to be solved increases with the number of inputs and the cyclomatic complexity of the program, but the number of program executions in each iteration is bounded by the number of input parameters in the function being tested. Therefore, the maximum number of program executions required by the method is bounded by the number of parameters of the function times the number of iterations taken. In our prior work [5], we showed that our approach has several advantages over the existing program execution based approaches [7, 4, 8]. The experimental evaluation of the existing program execution based techniques for different types of paths is not available for the quantitative comparison of our method with the existing approaches.

6 Conclusions

In this paper, we have presented an iterative relaxation technique to generate test data for path testing and evaluated its performance. The test data generation method requires a systematic technique to solve a system of linear inequalities for real and integer variables. We presented a numerical technique called UNA to solve the set of linear inequalities. We evaluated our test data generation technique on functions used in scientific computing. Our experimental evaluation indicated that the technique is effective in generating input data or identifying infeasible paths for 46 of the 47 paths in the programs considered. The technique is also efficient in terms of the number of iterations performed as well as its time performance.

References

[1] V. Chvatal, "Linear Programming," W.H. Freeman and Company, New York, 1983.

[2] 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.

[3] 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.

[4] , 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.

[5] N. Gupta, A. P. Mathur, and M. L. Soffa, "Automated Test Data Generation using An Iterative Relaxation Method" *ACM SIGSOFT Sixth International Symposium on Foundations of Software Engine ering(FSE-6),* pages 231-244, Orlando, Florida, November 1998.

[6] Neelam Gupta, "Automated Test Data Generation Using Iterative Relaxation Methods." PhD. Thesis, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, August 1999 (expected).

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

[8] 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.

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

[10] R.S.Pressman, "Software Engineering: A practitioner's approach" Third Edition, McGraw Hill.

[11] H. A. Taha, "Integer Programming: Theory, Applications, and Computations" Academic Press, 1975.