The Dynamic Domain Reduction Procedure for Test Data Generation[1]

A. JEFFERSON OFFUTT
ZHENYI JIN
Department of Information and Software Engineering, 4A4, George Mason University, Fairfax, VA 22030 U.S.A.
(email: {ofut,zjin}@ise.gmu.edu)

JIE PAN
Template Software, Inc., 45365 Vintage Park Plaza, Suite 100, Dulles, VA 20166, U.S.A.
(email: jenny.pan@template.com)

Software Practice and Experience, 29(2):167–193, January 1997.

SUMMARY

Test data generation is one of the most technically challenging steps of testing software, but most commercial systems currently incorporate very little automation for this step. This paper presents results from a project that is trying to find ways to incorporate test data generation into practical test processes. The results include a new procedure for automatically generating test data that incorporates ideas from symbolic evaluation, constraint-based testing, and dynamic test data generation. It takes an initial set of values for each input, and dynamically "pushes" the values through the control-flow graph of the program, modifying the sets of values as branches in the program are taken. The result is usually a set of values for each input parameter that has the property that any choice from the sets will cause the path to be traversed. This procedure uses new analysis techniques, offers improvements over previous research results in constraint-based testing, and combines several steps into one coherent process. The dynamic nature of this procedure yields several benefits. Moving through the control flow graph dynamically allows path constraints to be resolved immediately, which is more efficient both in space and time, and more often successful than constraint-based testing. This new procedure also incorporates an intelligent search technique based on bisection. The dynamic nature of this procedure also allows certain improvements to be made in the handling of arrays, loops, and expressions; language features that are traditionally difficult to handle in test data generation systems. The paper presents the test data generation procedure, examples to explain the working of the procedure, and results from a proof-of-concept implementation.

KEY WORDS: Automated test generation; Software testing; Symbolic evaluation

[1]Supported by the National Science Foundation under grant CCR-93-11967.

INTRODUCTION

Software testing is an expensive and labor-intensive task. It has been estimated that software testing accounts for up to 50% of software development [1, 2], and even more in safety-critical systems. If most of software testing could be automated, the cost of software development could be significantly reduced. Because the entire input domain of the program (which in most cases is effectively infinite) cannot be exhaustively searched, formal coverage criteria are sometimes used to decide what test inputs to use. Many test engineers and researchers believe that coverage criteria make it more likely that the tester will find faults in the program and provide greater assurance that the software is of high quality and reliability. Such criteria also provide rules for when to stop as well as repeatability of the test process. Adequacy criteria are therefore defined for testers to decide whether software has been adequately tested for a specific testing criterion [3].

In this paper, *test requirements* are specific software artifacts that must be satisfied or covered. As examples, reaching statements are the requirements for statement coverage, killing mutants are requirements for mutation, and executing DU pairs are requirements in data flow testing [4]. A *testing criterion* is a rule or collection of rules that impose requirements on a set of test cases[2]. Test engineers measure the extent to which a criterion is satisfied in terms of *coverage*, which is the percent of requirements that are satisfied.

One of the most difficult and expensive technical problems of software testing is the actual generation of test data values — which is traditionally done by hand. Test data generation is the process of creating program inputs that satisfy some testing criterion. The problem of automatic test data generation has been examined by a number of researchers. The general problem is undecidable, thus research has focused on partial solutions and heuristics. Korel [5] gives a formal description of the problem in terms of finding inputs to execute a particular path in the program. Offutt and DeMillo [6] described this problem in terms of killing a mutant. Test data generation problems are normally based on some kind of adequacy criterion. Adequacy criteria are therefore defined for testers to decide whether software has been adequately tested for a specific testing criterion [3]. Test data generators can be categorized into three groups: structural-oriented test data generators attempt to cover certain structural elements in the program [5, 6, 7, 8, 9, 10, 11, 12], data specification generators generate test data from a formal description of the input domain [13, 14, 15], and random test data generators [16, 17, 18] create test data according to some distribution of the inputs without satisfying any test criterion. Because the analysis of the software is so complex, structural test data generators are intended to be used during unit testing. The ideas that have been presented in the past have not come into general use because they were not practical or cost-effective for real programs. This paper presents a new procedure for use in structural-oriented generators. It attempts to generate test data for individual program units to meet a testing criterion such as branch coverage, dataflow coverage, or mutation. The goal of this research is to develop analysis methods that will be practical and cost-effective for real programs.

Structural-oriented test data generators typically use an abstract representation of the program (such as a control flow graph) and some form of symbolic evaluation. Symbolic evaluation [19, 20, 21] executes a program using symbolic values for variables instead of actual values. Symbolic evaluation derives a *path constraint* (also called *path condition*), which is a constraint system that describes conditions under which a path or set of paths is traversed. The path constraint must be satisfied for the path to be traversed; the path constraint is usually derived first, then an attempt is made to

[2]Note that this definition does not include all test criteria. For example, testing can be based on mean-time-to-failure, which does not use test requirements or a coverage measure. Non-structural criteria are not considered in this research.

satisfy it. Although symbolic evaluation is a powerful analysis tool, it has several practical problems, including aliasing, solving for indeterminate loops, and the size of the symbolic expressions.

Constraint-based test data generation
Previous work [6, 22] presented an approach to test data generation that uses control-flow analysis, symbolic evaluation, and information about mutants to automatically generate test data to satisfy the mutation testing criterion [23, 24, 25]. This approach, called *constraint-based testing (CBT)*, uses a constraint satisfaction technique called *domain reduction*.

CBT suffers from several shortcomings that prevent it from working in some situations and hamper its applicability in practical situations. These include problems handling arrays, loops, and nested expressions. Tools based on CBT (e.g., the Godzilla system [6]) occasionally fail to find test cases, and for some programs fail a large percentage of the time. This is partly because of problems with the technique, partly because of insufficiently general approaches to handling expressions, and partly because of unsophisticated search procedures.

Dynamic domain reduction test data generation
This paper presents a novel approach to test data generation, called the *dynamic domain reduction procedure*, that addresses most of these problems. The dynamic domain reduction procedure (DDR) is a new procedure that uses part of the CBT approach, and also draws from Korel's dynamic test data generation approach [5, 26] and symbolic evaluation. It uses a direct "domain reduction" method for deriving values, rather than function minimization methods as used by Korel [5, 11, 26] or linear programming-like methods as used by Clarke [9]. Korel's dynamic method [26] executes a program along one specific path by starting with a particular input. When a branching point is reached, if the current inputs will cause the the appropriate branch to be taken, the inputs will remain the same. If a different branch is required, then the inputs are dynamically modified to take the correct branch using function minimization. Although the dynamic domain reduction process presented here also works by choosing a specific path, there are no initial values, and the values are derived in-process from initial input domains.

Note that this is not the same as dynamic symbolic evaluation [20, 27]. Dynamic symbolic evaluation creates symbolic representations of results from executions on specific paths in a program. Dynamic domain reduction creates sets of values that represent conditions under which a path will be executed. Thus, the results of dynamic symbolic evaluation attempt to represent all possible values that will execute a given path, while dynamic domain reduction only results in a small set of possible values. While this is more limited, it is also more practical for real programs.

The solutions presented here allow the DDR procedure to succeed in many situations in which others fail. Its dynamic nature, which combines analysis of the software with satisfaction of constraints and test data generation, allows better handling of arrays and expressions. DDR also incorporates a sophisticated back-tracking search procedure to partially solve a problem that caused previous methods to fail. Unfortunately it is difficult to compare the DDR procedure with other techniques, because tools are not available for the few techniques that address this problem. Because of the historical basis, the DDR procedure will always work when CBT does, and also in many cases when CBT does not.

The DDR procedure walks through the program control flow graph, generating test data along the way. Each input variable is initially given a large set of potential values (its *domain*), and as branches are taken in the control flow graph, the domains for the variables involved in the predicates are reduced so that the appropriate predicates would be true for any assignment of values from the domain. When choices for how to reduce the domains must be made, a search process is initiated

and choices are systematically made to try to find a choice that allows the subsequent edges on the path to be executed. When the procedure is finished, the remaining values for the variables' domains represent sets of test cases that will cause execution of the path. If any variable's domain is empty, the search process failed, which indicates one of two things. One, the path is infeasible, so no satisfying values could be found. Two, it was very difficult to find values that execute the path. This could be because the constraints were too complicated, or there are relatively few inputs that will execute the path.

The next section introduces some background terminology and concepts, presents a formal description of the test data generation problem, and describes the domain reduction procedure used in CBT. The dynamic domain reduction procedure for generating test data is described under **A test data generation problem**. A discussion of how arrays, loops, and expressions are handled by the dynamic domain reduction is given under ARRAYS, LOOPS, AND EXPRESSIONS, and results from a proof-of-concept test data generator are given under EVALUATION.

BACKGROUND

A *basic block* is a maximum sequence of program statements such that if any statement of the block is executed, all statements in the block will be executed. Basic blocks contain only one entry point and one exit point. A *decision* is a point in a program where control flow can diverge. `IF`, `DO`, `WHILE`, `GOTO` and `CASE` statements are decision points. A *junction* is a point in the program where control flows merge. For instance, the `ENDIF` of an `IF` statement is a junction.

A *control flow graph (CFG)* of a program is a directed graph that represents the control structure of the program. Each node is either a basic block, a junction, or a decision node. The edges represent potential control flow among nodes. A *control path* is a directed path from an entry node to a terminal node of the CFG. A *predicate* is a boolean expression associated with a decision node that determines which edge from the node will be traversed. A *constraint* is an algebraic expression that restricts the space of program variables to certain domains. For example, the constraint $A > 0$ describes the portion of the input domain where A is positive.

Paths can be represented by systems of constraints; one constraint for each predicate on the control path. Although there can effectively be an infinite number of paths in a program, one open-ended constraint system (e.g., $X > 0$) can represent an infinite number of paths by describing all possible iterations of a loop. The predicates are initially expressed in terms of program variables; since each of these program variables can be ultimately expressed in terms of the input variables using assignment statements along the control path, it is possible to re-express the predicates as constraints in terms of only the input variables.

If input data that satisfy the path constraint exist, the control path is also an *execution path* and can be used to test the program. If the path constraint cannot be satisfied, the control path is said to be *infeasible*.

A *test case* is a set of input data that is used to evaluate the software. Test data generation is the process of identifying a set of test data that satisfies a testing criterion. The *domain* of a variable is the set of its possible values.

A *constraint system* is a hierarchical structure composed of expressions, constraints, and clauses. An *expression* is composed of variables, parentheses, and programming language operators. Expressions are taken directly from the test program and derived from predicates within

decision statements and right-hand sides of assignment statements during symbolic evaluation. A *constraint* is a pair of expressions related by one of the relational operators $\{>, <, =, \geq, \leq, \neq\}$. Constraints evaluate to one of the binary values TRUE or FALSE and can be modified by the negation operator NOT ($\neg$). A *clause* is a list of constraints connected by the two logical operators AND ($\bigwedge$) and OR ($\bigvee$). A *conjunctive clause* uses only the logical AND and a *disjunctive clause* uses only the logical OR. A *constraint system* is considered to be a constraint or clause that represents one complete test case. In this work, all constraint systems are kept in *disjunctive normal form* (DNF), a list of conjunctive clauses connected by logical ORs.

A test data generation problem

This paper presents the test data generation problem in terms of reaching a particular node using an arbitrary path. Executing a particular path is a special case of this presentation, and it is easily extended to incorporate testing criteria such as data flow and mutation. Thus this paper treats this problem in a very general way. Automatic test data generation tools work by searching for values that satisfy individual test requirements for some criterion.

Let n_g be a node in the CFG of a program P with input domain D; n_g is called the *goal* node. The test data generation problem is: *find a program input $t \in D$ such that when P is executed on t, n_g will be reached.* To express the specific path version of this problem, for a given path $p =< n_1, n_2, ..., n_g >$, t must cause that path to be executed. For mutation testing, the goal node n_g contains the statement that is mutated, and the additional requirement is imposed that after n_g is executed, the *necessity condition* must be true [6]. The necessity condition is the condition that expresses what is necessary for a test case to kill the mutant. The test data generation problem statement can also be extended to include data flow testing criteria. For example, the *all-uses* data flow criterion [3] requires that each definition of a variable reach all possible uses of that variable. Thus, the goal node n_g in the test data generation problem becomes the node that contains a definition of the variable x, and the requirement is added that after n_g is reached, the node containing the use of x (n_u) must also be reached, with the further restriction that the subpath from n_g to n_u must not contain another definition of x.

It should be noted that the test data generation problem is formally unsolvable. If we consider the goal node to be a terminating statement, then this is a modified version of the halting problem. Nonetheless, this is a situation where partial solutions can be valuable to practicing engineers.

The CBT satisfaction procedure

Constraint-based testing was designed [6, 22] as a technique to develop test data for mutation testing. CBT works by developing constraints to represent conditions under which mutants will be killed, then solves those constraints to yield test case values.

The CBT approach uses four separate procedures to generate tests; one for constraints that represent conditions under which a particular statement will be reached (*reachability constraints*), another for constraints that represent conditions under which a mutant will be killed (*necessity constraints*), a third that applies symbolic evaluation to rewrite the constraint systems to be in terms of input variables, and a fourth to find test case input values that will satisfy the constraints (*constraint satisfaction*). These procedures are described in detail elsewhere [6, 28] and have been implemented in the test data generation tool Godzilla [25].

The constraint satisfaction procedure used by Godzilla is known as domain reduction [28] and is based on the topological sort algorithm. The domain reduction procedure uses local information in the constraint systems to find values for variables, then uses back-substitution to simplify the remaining constraints in the constraint system. Initially, each variable is given a domain of values.

This domain can be supplied by the tester or derived automatically from specifications or preconditions. Each individual constraint is viewed as a statement that reduces the domain of values for the variable(s) in the constraint. Constraints of the form $x \Re c$, where x is a variable, c a constant and $\Re$ a relational operator, are *ground terms* and are used to reduce the current domain of values for x. Constraints of the form $x \Re y$, where both x and y are variables, are used to reduce the domain of values for both x and y. More complicated constraints, where one or both sides are expressions, are simplified by back-substitution when values are selected.

When no additional simplification can be done, a heuristic is employed to choose a value for one of the remaining variables in the constraints. The variable chosen is the variable with the smallest current domain. The value for this variable is chosen arbitrarily from its current domain of values and is then back-substituted into the remaining constraints. This process is repeated until all variables have been assigned a value. The expectation is that by choosing a variable with the smallest current domain size, there is less chance of making a mistake (that is, choosing a value that will cause a solvable constraint system to become unsolvable).

Each time a variable is assigned a value, the input space for the program is reduced by one dimension. As the number of dimensions is reduced, the constraints in the system become progressively simpler. Another way to state this is that each variable assignment implicitly introduces a new constraint into the system of the form $(x = c)$, where c is a constant. If chosen poorly, the constraint system may no longer be feasible, and the procedure will have to make a new choice. The underlying assumption is that because of the simple form of the test case constraints, these dimension-reducing constraints will rarely make the region infeasible. Experience with Godzilla [6, 22] has shown this assumption to be valid for most cases, although how often depends on many factors that are specific to the program being analyzed. When a constraint system becomes infeasible because of a value chosen for a variable, CBT employs a very simple search procedure; a new value is chosen for the same variable and the system is re-evaluated.

Weaknesses of the CBT domain reduction procedure

The domain reduction constraint satisfaction procedure used in constraint-based testing has five general problems. First, it requires the entire constraint system to be present before satisfaction starts. The constraint systems are often large and use a lot of memory, and the design of the symbolic evaluation and satisfaction software causes the constraint system to be copied many times, imposing great memory demands on the system. This is a major limiting factor in terms of the size of the programs that the Godzilla system can handle. An underlying cause of the problem is the *static* nature of the algorithms — all of the constraints have to be computed before they are analyzed. Second, since the domain reduction procedure randomly guesses values in its search procedure, the search procedure is poorly organized and some values may be chosen more than once. A third problem is that the domain reduction procedure has very simple expression handling mechanisms. The general problem of finding values for variables that will give the expression a specific result has not been addressed in the literature. Most real world programs have more complicated expressions, which the domain reduction procedure has to either skip or simplify into a format that can be handled. This limits the scope of the programs that can be tested by the procedure. Fourth, the domain reduction procedure has great difficulty with loops. Constraints are generated to represent weakest preconditions going into and out of loops, but when the details of a loop's execution affects the constraint systems, the symbolic information from the loop's execution is lost. Finally, the domain reduction procedure views an array as one variable, and does not differentiate values between individual elements in the array. This impacts the power of the test data generation process on programs that make heavy use of arrays. Other types of aliasing are not handled at all.

THE DYNAMIC DOMAIN REDUCTION PROCEDURE

The dynamic domain constraint satisfaction procedure represents an advance over the **static** domain procedure, by addressing the five problems discussed under **Weaknesses of the CBT domain reduction procedure**. It uses constraints derived from the test program to progressively reduce domains of variables until test data that satisfy these constraints are found. This method finds values by walking through the control flow graph, using one predicate at a time and reducing domains of variables step by step.

The DDR procedure introduces several new techniques. First, it uses *dynamic* analysis. Static analysis techniques analyze the software without any execution; the analysis is obtained strictly from the source code. Dynamic analysis techniques rely on some sort of execution of the software. In general, dynamic analysis techniques are more expensive, but are also able to provide more information. The DDR procedure applies a kind of slicing technique [29, 30] by looking at specific execution paths through the program. These paths are not fully executed, but executed *symbolically.*

The second technique new to this paper is *domain-based* symbolic execution. This is different from traditional symbolic execution techniques [27, 31], which tried to compute a symbolic representation of the output. *Domain-based* symbolic execution works in the other direction; it starts with a symbolic representation of the *input* and walks through the program, modifying the inputs so that they conform to the path that is taken. This is based on variable *input domains*, which represent a possible set of values. Initially, a domain for an input variable contains all possible values for the variable. Theoretically, this may be infinite for numeric variables; in reality, the domain is limited by the computer's word size. In its simplest form, a domain is represented by an upper and lower bound (for example, MININT and MAXINT). As a path is symbolically executed, the domains are reduced to reflect decisions and computations; the new domains represent values under which the current path will be taken. The initial upper and lower values for a variable may be MININT and MAXINT, or the domain may be restricted based on program input specifications or the test engineer's knowledge.

Of course, the values under which a particular path will be taken cannot be precisely determined. The set of values will either contain some values that will not execute the path, the set will not contain some values that will execute the path, or both. The approach taken is to be conservative, that is, we prefer to have values such that the path will be taken for every value, even though there are some values that will execute the path that will not be included. The third new technique is introduced to handle situations when important values that will execute a path are not included. When the procedure makes choices for values to exclude, the choices may lead to situations where later constraints cause domains to become empty, and values cannot be generated. When this happens, a backtrack search process is used to make different choices.

The fourth new technique is introduced to handle complicated expressions. Expressions are always difficult for test case generators to handle, and previous automatic test data generation tools have either ignored expressions or solved for expressions by generating values randomly. The essential problem is, given an expression and a value, to find a set of values for the variables in the expression such that when evaluated, the expression has the required value. This paper introduces a domain-symbolic expression handling technique that does two things. First, it takes domains for values at the leaves of an expression tree and propagates those domains up through the operations to derive a domain for the entire expression. Second, it takes a (modified) domain for an expression and propagates the domain down through the operations to derive domains for the base variables that are consistent with the domain for the entire expression.

Before applying the dynamic domain procedure, the predicates and constraints in the test program must be put into the following forms: (1) predicates must be in disjunctive normal form, and (2) constraints and expressions are put in a canonical form where constants are always on the right.

Overview of the dynamic domain constraint satisfaction procedure
The dynamic domain procedure is quite difficult to clearly describe. We proceed by first giving a high level overview of the procedure, followed by a small illustrative example, then a detailed description that uses a textual description of a flowchart. Finally, several more examples are given to illustrate some of the subtle aspects of the algorithm.

DDR starts with several pieces of information about the test procedure: a control flow graph (CFG), two nodes representing the initial and goal nodes, and initial domains for all input variables. The first step is that a finite set of paths from the initial to the goal node is determined. Then each path is analyzed in turn. The path is traversed, and symbolic evaluation is used to progressively reduce the domains of values for the input variables. When choices must be made that do not directly reflect the symbolic evaluation, a search process is used to split the domain of some variable in an attempt to find a set of values that allow the constraints to be satisfied. The procedure proceeds until the goal node is reached and all input variables have current domains that can still be evaluated, or all paths have been unsuccessfully searched.

Example 1
Assume a program function `mid (x, y, z)` that determines the middle value of three integers. A C version of the function and its control flow graph are shown in Figure 1. The predicates are shown on their associated edges, and the assignment statements are shown beside the nodes. Assume that the initial domains of input variables `x`, `y` and `z` are given as follows:

```
x:  < -10 .. 10 >
y:  < -10 .. 10 >
z:  < -10 .. 10 >
```

Assume that the goal node N_g is node 10, the exit node. To generate test cases for N_g, a control path in the CFG needs to be selected. If path 1-2-3-5-10 (shown in dashed lines) is chosen, then three predicates are encountered on edges 1-2, 2-3 and 3-5.

The procedure starts on node 1. Since node 1 is a decision node, the predicate on edge 1-2 (y $<$ z) is used to reduce the domains for y and z. The domains for x and y are split at 0 (0 is called the "split point"), leaving the domain for y to be $<$ -10 .. 0 $>$ and for z to be $<$ 1 .. 10 $>$. This is interpreted to mean that all possible values of y are less than all possible values of z. Thus, this represents a portion of the input space that will take this branch. Note that this also eliminates part of the valid input space. The best way to understand this is graphically. Figure 2 shows the valid values for the original domains and the constraint (y $<$ z) on the left, and the regions with a split point of 0, then of -5, to the right.

After traversing edge 1-2, the decision node 2 is reached. The predicate on edge 2-3 is (x $\geq$ y) and is used to reduce the domains for variables x and y. The split point is chosen to be -5, leaving the domain for x to be $<$ -5 .. 0 $>$ and for y to be $<$ -10 .. -5 $>$. That is, all possible values for x are greater than or equal to all possible values for y.

Next, edge 3-5 is traversed, and the constraint (x $<$ z) is used to reduce the domains for x and z. The split point chosen is 2, leaving the domain for x to be $<$ -5 .. 2 $>$ and for z to be $<$ 3 .. 10 $>$. That is, all possible values for x are less than all possible values for z. The domains of each

```
int mid (x, y, z)
int x, y, z;
{
   int mid;
   mid = z;
   if (y < z)
   {
      if (x < y)
         mid = y;
      else if (x < z)
         mid = x;
   }
   else
   {
      if (x > y)
         mid = y;
      else if (x > z)
         mid = x;
   }
   return (mid);
}
```

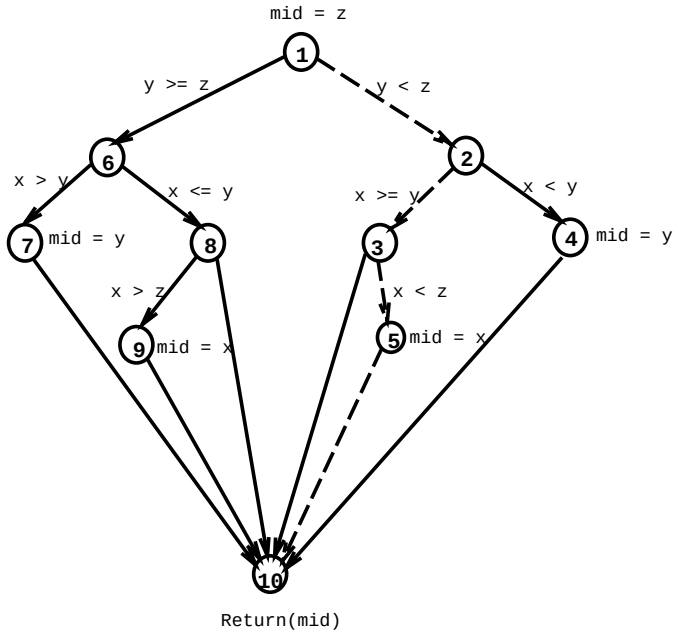

Figure 1: **Function Mid and its Control Flow Graph**

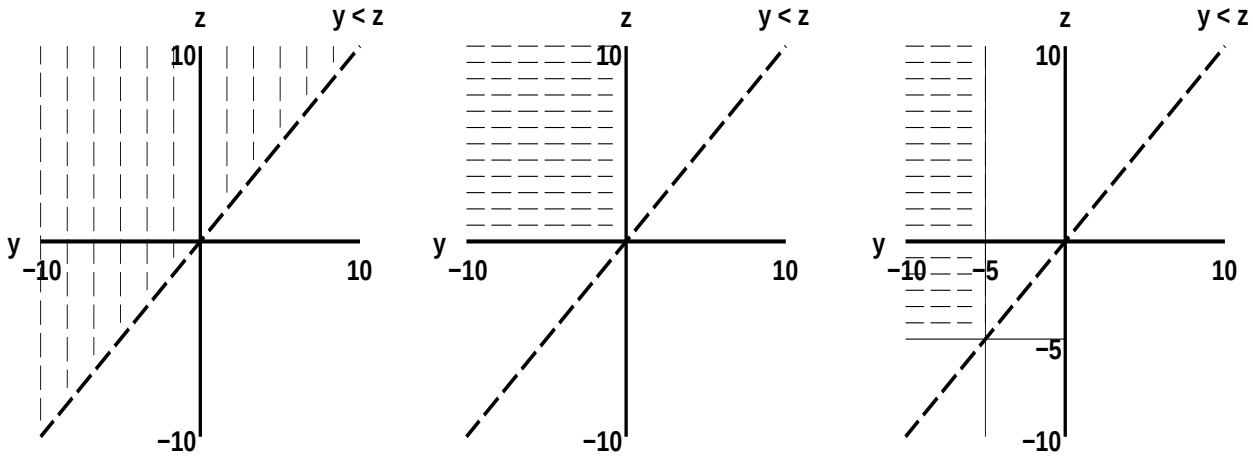

Figure 2: **Graphical View of Domains Before and After Splitting**

input variable after each constraint has been used are as follows:

	x	y	z
1. Start:	< -10 .. 10 >	< -10 .. 10 >	< -10 .. 10 >
2. y < z:	< -10 .. 10 >	< -10 .. 0 >	< 1 .. 10 >
3. x ≥ y:	< -5 .. 10 >	< -10 .. -5 >	< 0 .. 10 >
4. x < z:	< -5 .. 2 >	< -10 .. -5 >	< 3 .. 10 >

After all constraints have been used, a test case is chosen arbitrarily from within the input domains. For example, one valid test case is (x=0, y=-10, and z=8). In some cases, a poor split point may be chosen and a later predicate cannot be satisfied. When this happens, the backtracking search procedure is used to choose a different split point. This is elaborated in the detailed discussion below and illustrated in Example 2.

Detailed description of the procedure

Figure 3 shows the overall flow of operations within the dynamic domain reduction procedure. In the diagram, bubbles represent inputs to and outputs from the procedure, rectangles represent process steps, and diamonds represent branches in the execution. Three inputs are needed: the initial value domains of the input variables; the control flow graph of the program; and the starting node N_1 and goal node N_g.

`FindPath` selects a set of paths P from N_1 to N_g. Which paths, how many, and how the paths are chosen depends on the testing criterion being applied. A path-coverage testing technique might choose a single, complete path. If a data flow criterion is used, N_g will be a node that contains a def, and any path from N_1 to N_g can be selected. In this case, `FindPath` will return a finite set of paths. There may be loops, so `FindPath` is expected to select from among the theoretically infinite number of paths represented by the loops.

The `IsPEempty?` decision checks whether all paths in the path set P have been searched. If all paths in P have been checked and test data have not been found, the procedure fails; either there is no feasible path from N_1 to N_g or a test case that will execute a feasible path is too difficult to find. The `SelectOnePath` procedure selects a current subpath P_i to traverse, and removes it from the path set P so that it will not be chosen again.

The rest of the procedure walks through the CFG along P_i, attempting to find a test case that will execute it. If the current node is a decision node (`IsNodeADecision?`) then the constraint associated with the appropriate outgoing branch is used to reduce the corresponding variable domains. If the current node is not a decision node, then the statements associated with this node are symbolically evaluated, which also may modify the variable domains.

`ReduceDomains` is the key step in the dynamic domain reduction procedure. The predicates (from the `yes` branch from `IsNodeADecision?` in Figure 3) or information from symbolic evaluation (from the `no` branch) is used to form new constraints. These constraints are then used to update the current domains for all variables that appear in the constraints.

When a constraint of the form $x \Re y$ is encountered, it cannot be fully determined how to reduce the value domains for x and y. For example, suppose the domains for x and y are both (1 .. 100) and the constraint $x > y$ is encountered. CBT would have chosen an arbitrary value from within the domain for one of the variables, then adjusted the domain of the other variables

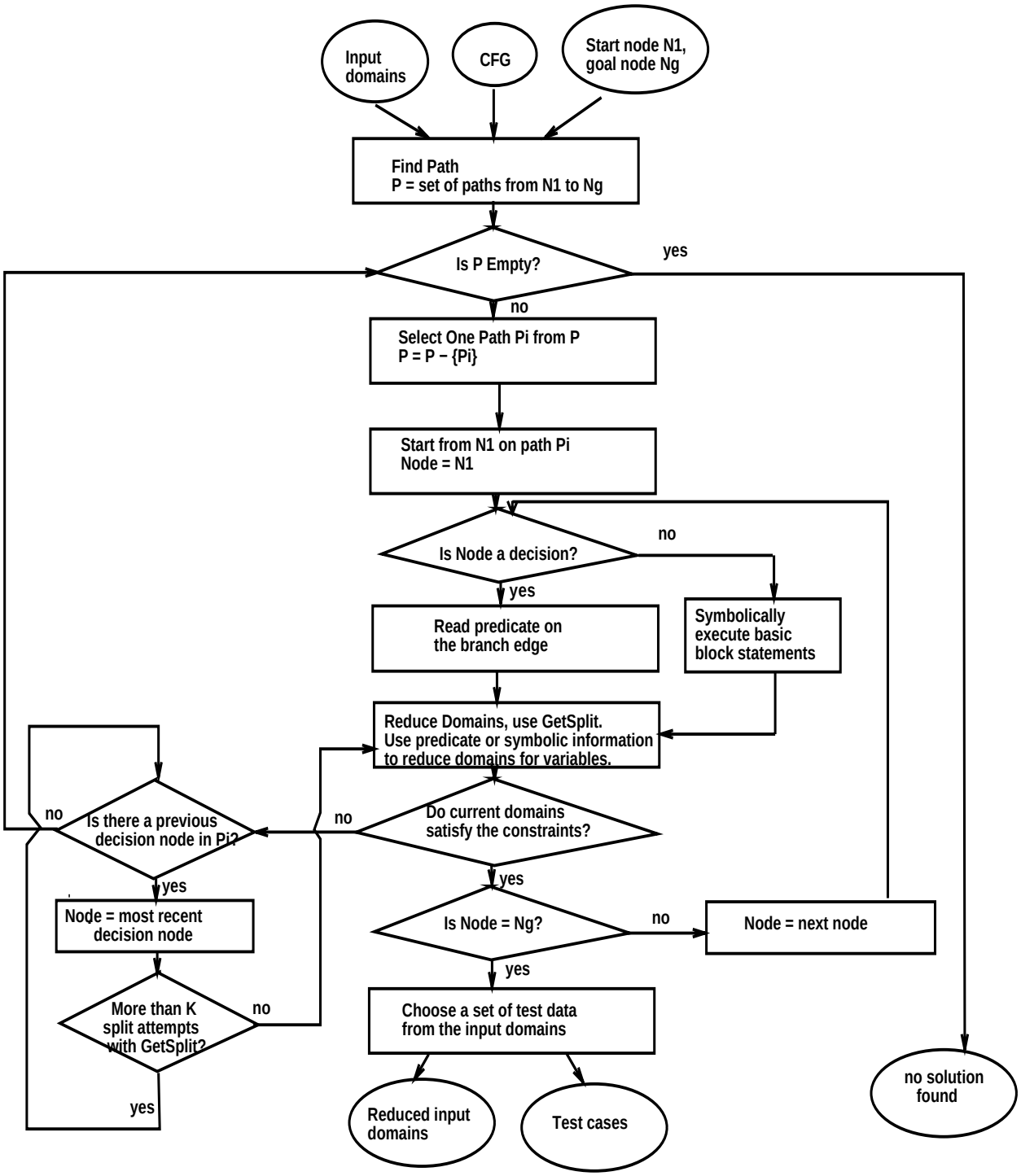

Figure 3: **The Dynamic Domain Reduction Procedure**

accordingly. So if the value for x was chosen to be 25, the domain for y would become (0 .. 24). In DDR, this situation spawns a search procedure using `GetSplit`. `GetSplit` is a novel algorithm that modifies the domains for two variables in a constraint so that (1) the new domains satisfy the constraint, and (2) the size of the two domains are balanced. So in the $x > y$ example, the first attempt would be to make the domain for x to be (51 .. 100), and for y, (0 .. 50).

`GetSplit`, shown in Figure 4, encodes the major heuristics for selecting test case values and is the key to the searching process. When two variables or expressions have a relation defined by a constraint, there are two cases, defined by the relationships between the two variables' domains. If the two domains define non-intersecting sets of values, then the constraint is already either satisfied, or is infeasible. If the two domains define sets of values that intersect, then some of the values can satisfy the constraint, and some cannot. The purpose of `GetSplit` is to modify the two domains such that the constraint is satisfied for all pairs of values from the two domains.

`GetSplit` accepts two domains and a searching index, and finds new domains such that the constraint will be true for all values in the new domains. Initially, a value in the two domains is chosen to be a point at which the two domains are *split*. This split point is chosen such that the two domains are non-intersecting, and each domain is reduced by approximately the same amount. Sometimes a split point may be chosen that is invalid in the sense that the resulting domains are incompatible with constraints that are derived later. If the split that is chosen causes a later constraint to be infeasible, `GetSplit` is used to search for a better split. During the search process, the split point is successively reevaluated using bisection — the split point is moved halfway in one direction, then the other, and so on until (1) the choice succeeds in allowing a test case to be found, (2) all choices have been exhausted, or (3) a predetermined constant number of choices have been made (to avoid an infinite search).

The inputs to `GetSplit` are domains for two expressions (`LeftDom` and `RightDom`) and an integer that indicates what iteration of the search is being performed (`SrchIndx` = 1, 2, 3, ...). Each expression has a domain for the left side of the expression (`LeftDom.`$_{Bot}$ .. `LeftDom.`$_{Top}$) and a domain for the right side (`RightDom.`$_{Bot}$ .. `RightDom.`$_{Top}$). A *split point* is found based on the given domains, and returned to the caller to reduce the domains. The algorithm uses four cases that depend on the relationships between `LeftDom` and `RightDom`.

The four cases are presented algorithmically in Figure 4, but are perhaps clearer pictorially. Figure 5 shows the same four cases, in the same order they appear in the `GetSplit` algorithm. In the first case, the domain of the left expression is wholly contained in the domain of the right expression, and in the second case, the right expression is wholly contained in the left. In the last two cases, the domains overlap, but neither is contained in the other. Note that if the two domains do not overlap, `GetSplit` is not needed.

For example, `LeftDom` = (-20 .. 20), `RightDom` = (-40 .. 30), and `SrchIndx` = 1 satisfies the first case where (`RightDom.`$_{Bot}$ < `LeftDom.`$_{Bot}$) and (`RightDom.`$_{Top}$ > `LeftDom.`$_{Top}$). So `SrchPt` = 1/2, and `SplitPoint` = (`LeftDom.`$_{Top}$ - `LeftDom.`$_{Bot}$)*srchPt + `LeftDom.`$_{Bot}$ = (20 - (-20))/2 + (-20) = 0. Thus, the domains would be changed to be: `LeftDom` = (-20 .. -1) and `RightDom` = (0 .. 30) (assuming integer arithmetic). On the other hand, if `SrchIndx` = 5, then `SrchPt` = 3/8, and `SplitPoint` = (20 - (-20)) * (3/8) + (-20) = -5. This would cause the domains to be changed to be: `LeftDom` = (-20 .. -6) and `RightDom` = (-5 .. 30).

After the domains have been reduced by the `ReduceDomains` algorithm, the status of the domains are reevaluated. If the new domain values satisfy the predicate, the procedure either goes to the next node, or if the current node is the goal node, the procedure is finished. If there are

```
algorithm      GetSplit (LeftDom, RightDom, SrchIndx)
precondition   LeftDom and RightDom are initialized appropriately
               and SrchIndx is one more than the last time GetSplit was called
               with these domains for this expression.
postcondition  split value >= (LeftDom.Bot AND RightDom.Bot) and
               split value <= (LeftDom.Top AND RightDom.Top)
input          LeftDom:  left expr's domain with Bot and Top values
               RightDom: right expr's domain with Bot and Top values
output         split -- a value that divides a domain of values into two subdomains.

BEGIN
    -- Compute the current search point.
    -- srchPt = (1/2, 1/4, 3/4, 1/8, 3/8, ...)
    Choose exp such that 2^exp <= SrchIndx <= 2^exp + 1
    srchPt = (2^exp - (2 * (2^exp - 1) - 1))/2^exp

    -- Try to equally split the left and right expression's domains.
    IF (LeftDom.Bot >= RightDom.Bot AND LeftDom.Top <= RightDom.Top)
        split = (LeftDom.Top - LeftDom.Bot)*srchPt + LeftDom.Bot
    ELSE IF (LeftDom.Bot <= RightDom.Bot AND LeftDom.Top >= RightDom.Top)
        split = (RightDom.Top - RightDom.Bot)*srchPt + RightDom.Bot
    ELSE IF (LeftDom.Bot >= RightDom.Bot AND LeftDom.Top >= RightDom.Top)
        split = (RightDom.Top - LeftDom.Bot)*srchPt + LeftDom.Bot
    ELSE -- LeftDom.Bot <= RightDom.Bot AND LeftDom.Top <= RightDom.Top
        split = (LeftDom.Top - RightDom.Bot)*srchPt + RightDom.Bot
    END IF
    RETURN split
END GetSplit
```

Figure 4: **The GetSplit Algorithm – Computes the Next Search Point For the Two Input Domains.**

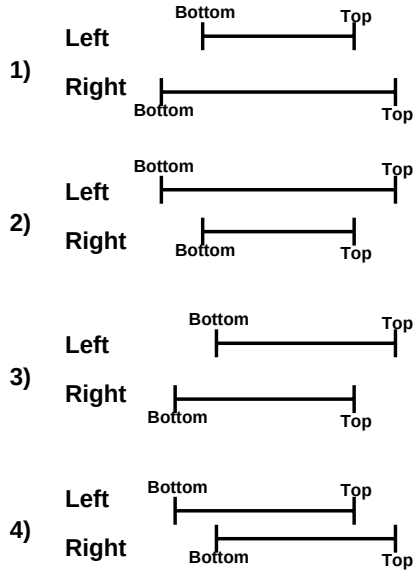

Figure 5: **Cases for the GetSplit Algorithm**

any variables left with domains containing more than one value, it is certain that any value from within the domain will satisfy the test requirement, and a value can be chosen arbitrarily.

If the new domain values do not satisfy the constraint (`Do current domains satisfy the constraints?`), the search process is triggered. The procedure goes back to the most recent decision node in P_i (`Is there a previous decision node in Pi?`) and tries to satisfy the predicate again using a different split point. If there have been too many attempts to find a feasible split point at the most recent decision node (`more than K split attempts with GetSplit?`), the procedure goes to the previous decision node in the CFG (K is a predefined constant). If there are no previous decision nodes to evaluate, the procedure gives up on this path and goes to the next path in P.

The next two subsections present two examples that illustrate the operation of the dynamic domain reduction procedure. The examples are chosen to be small enough to fit within a paper, but the procedure works on arbitrary predicates. The first shows how the dynamic domain constraint satisfaction procedure is used to successfully reduce the input domains and to generate test data. The second illustrates a case when the reduced domain does not satisfy later constraints, so the domains have to be re-chosen at a previous decision node. Neither of these examples contain loops or arrays; this discussion is deferred until later.

Example 2
The function *Value* in Figure 6 takes three integer inputs A, B, and C, and returns an integer, V. Assume that the path 1-2-4-6-8 (shown in dashed lines) is selected and the initial domains of the input variables are:

```
A: < 0  ..  20 >
B: < 10 ..  40 >
C: < 0  .. 100 >
```

Node 1 is a decision node and branch 1-2 has the predicate (A < B). The split point chosen is 15, leaving the domain for A to be < 0 .. 14 > and for B to be < 15 .. 40 >. This forces all possible

```
int Value (A, B, C)
int A, B, C;
{
    int V;
    V = 0;
    if (A < B)
    {
        C = 16 ;
        if (A < C)
            V = A + 30 ;
        else
            V = A;
    }
    else
    {
        C = 30 ;
        V = C + B + A ;
    }
    return (V);
}
```

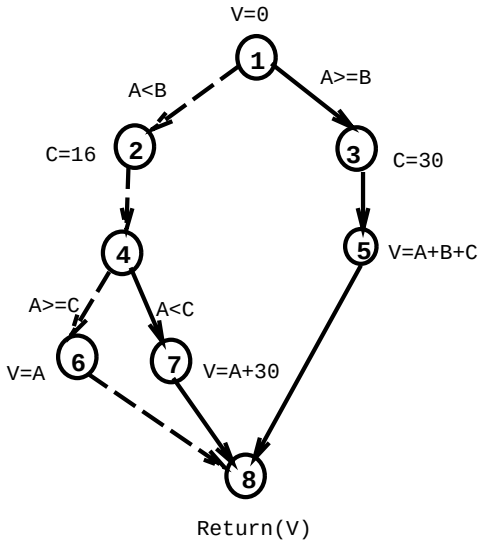

Figure 6: **Function Value and its Control Flow Graph**

values for A to be less than all possible values for B. Node 2 is a basic block node containing the assignment C = 16, therefore the variable C is given a value 16, and its domain is modified to be $< 16 .. 16 >$. That is, C's domain now contains only one value.

The next node, 4, is a decision node, and the branch 4-6 is to be traversed. The constraint associated with this branch is $(A \geq C)$. Unfortunately, the domain for A is currently $< 0 .. 14 >$, so A cannot be greater than or equal to C, and the branch is currently infeasible. This means that the split point chosen at the previous decision node 1 was a poor choice, and another split point must be calculated.

The domains for A, B, and C are reset to their initial values, and the procedure returns to node 1. This time, the split point for the constraint $(A < B)$ is chosen to be 12, leaving the domain for A to be $< 0 .. 12 >$ and for B to be $< 13 .. 40 >$. Again, A cannot be greater than or equal to C, so the procedure returns to node 1.

The third time, the split point for the constraint $(A < B)$ is chosen to be 17, leaving the domain for A to be $< 0 .. 17 >$ and for B to be $< 18 .. 40 >$. These domains allow the constraint $(A > C)$ to be satisfied when the edge 4-6 is reached. The split point for $(A > C)$ is 16, A's domain becomes $< 17 .. 17 >$, and C's domain remains $< 16 .. 16 >$. The value for B can be generated by arbitrarily choosing data from its domain. Thus, a test case such as (A=17, B=25, C=16) will execute the path 1-2-4-6-8.

ARRAYS, LOOPS, AND EXPRESSIONS

Arrays and loops are language features that are traditionally difficult to handle in test data generation systems. They have an inherently dynamic nature, and thus are difficult to resolve

using purely static analysis. Most testing systems (such as Godzilla [6], Atac [32], and Asset [33, 3]) have to make simplifications for arrays, usually by treating a reference to any element as a reference to all elements in the array. Pointers are often treated in a similar way — references are either not resolved and the objects of the pointers are ignored, or a reference through a pointer is considered to be a reference to the entire object. The problems with arrays and pointers are really problems with aliasing and dynamic reference resolution. Aliasing refers to the situation when two names refer to the same address location. Dynamic reference resolution refers to the situation when the address location of a reference cannot be determined statically. If the references can be resolved, aliasing is a much simpler problem to handle, but if the names cannot be resolved, it is impossible to deterministically recognize aliasing.

Loops and arrays
Loops are often either ignored or simplified. For example, Godzilla handles loops by only considering entry and exit conditions; the number of iterations of the loop is ignored. Although these simplifications work well in many cases, they obviously do not always work. A major innovation of this research is that it employs a partially dynamic procedure to facilitate handling of arrays and loops.

The dynamic domain reduction procedure allows references to be fully resolved during execution. Each element in an array can then be treated as a distinct variable. To do this, the index of an array is used as an expression, and then used to differentiate each element of the array and perform domain reduction on these elements the same way as the other variables. Pointers are also handled as variables when we read pointers as expressions and find their corresponding domains. Although the dynamic procedure allows references to be resolved deterministically, this entails making decisions about pointer de-referencing and array indexing.

The dynamic domain reduction procedure handles loops in a novel way. Most existing techniques handle loops by discovering all possible paths from the given start node to the goal node. If there is a loop structure in the control flow graph, the loops must be unrolled and the number of paths between these two nodes is potentially infinite. Therefore, constraints on the decision nodes and control variables of the loop structures need to be checked and updated to decide which path to take. This method is obviously not efficient because among all the possible paths found, many do not satisfy the loop constraint and have to be thrown away.

In the dynamic domain reduction procedure, loops are handled dynamically. Instead of finding all possible paths, the procedure finds all the paths that contain at most one loop structure. It then marks those decision nodes that affect whether another iteration of the loop is made. Then as the path is traversed, when the decision node is encountered, the loop constraint and control variables are checked dynamically to decide whether to continue with another iteration or to exit the loop. If the control variable satisfies the constraint, another iteration is carried out and the loop control variable is updated, otherwise the procedure exits the loop and continues traversing the path on the node after the loop. This eliminates the need for loop unrolling, which allows more realistic programs to be handled.

Example 3
As an example, consider the program function `BSearch` shown in Figure 7. The predicates are shown on their associated edges, and the assignment statements are shown by the nodes. Assume that the initial domains of input variables `N` and `A` are:

`N:` $< -10 \ldots 10 >$
`A[i]:` $< -10 \ldots 10 > \forall i, 1 \leq i \leq Len(A)$

```
int BSearch (A, N)
int N, A();
{
   int i;
   for i = 1 to Len (A)
   {
      if (N == A(i))
         return (i);
      endif;
   }
   return (-1);
}
```

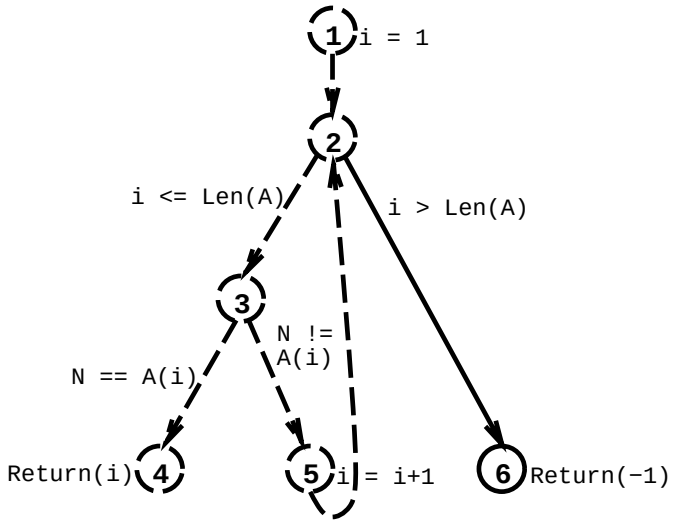

Figure 7: **Function BSearch and its Control Flow Graph**

Assume that the goal path is: 1-2-3-5-2-3-5-2-3-4. The subpath of 1-2-3 introduces no change to the input variables, and the value of i is 1. To take the branch from node 3 to 5, the predicate $N \neq A(i)$ must be true. i is 1, therefore the first element of A is changed using `GetSplit`:

$$\begin{aligned} \text{A(1)} \neq \text{N} \Longrightarrow\ & \text{A(1): } < \text{-10 .. 0} > \\ & \text{N: } < 1 \text{ .. } 10 > \end{aligned}$$

The next time through the loop, we have the same predicate, but i is 2, thus:

$$\begin{aligned} \text{A(2)} \neq \text{N} \Longrightarrow\ & \text{A(2): } < \text{-10 .. 4} > \\ & \text{N: } < 5 \text{ .. } 10 > \end{aligned}$$

The final time through the loop, the branch from 3 to 4 is taken, thus the predicate $N = A(i)$ must be true. i is 3, thus:

$$\begin{aligned} \text{A(3)} = \text{N} \Longrightarrow\ & \text{A(3): } < 5 \text{ .. } 10 > \\ & \text{N: } < 5 \text{ .. } 10 > \end{aligned}$$

with the additional notation that A(3) and N are now "aliased", that is their domains should be exactly the same. This means that if one is later changed, the other will also be changed, and when actual values are chosen at the end, the same values will be chosen for both variables.

At this point, the dynamic part of the procedure is finished, and values can be chosen arbitrarily from the remaining domain. The dynamic nature of this procedure allows each element of an array to be viewed separately, and which element is being referred to is decided *deterministically*.

Expressions
Expressions are notoriously difficult to handle during symbolic evaluation, and even harder during test case generation. In fact, we have not been able to find a published description of a method that successfully finds values to satisfy complicated expressions. The essential problem is, given an expression and a value, to find a set of values for the variables in the expression such that when evaluated, the expression has the required value. In effect, we want to "de-evaluate" the expression. Previous research papers in automatic test data generation have either not discussed how the problem is solved or solved it by repeatedly generating random values until the expression yields the correct value. This method is slow and often fails.

The `ExprDomain` algorithm, shown in Figure 8, works with the `Update` algorithm in Figure 9 to address this problem. `ExprDomain` accepts an expression to evaluate and a Domain Data Store (DDS), which stores domains for variables and expressions. `ExprDomain` uses the domains to symbolically evaluate the expression. `Update` attempts to "de-evaluate" an expression, so that the value of the expression is consistent with a given domain. The problem is somewhat relaxed because of the dynamic nature of the DDR procedure. Specifically, we do not need to find variable value assignments for a specific value, rather, we need variable value assignments that are consistent with the domain of values needed by the expression. In addition, the values must be consistent with the current domains of the variables involved in the expression.

Since expressions are defined recursively, `ExprDomain` runs recursively until domains for each variable are found. `ExprDomain` proceeds by determining the domains of the variables and constants at the leaves of the expression and then propagating these domains up by applying the operations. When changes are necessitated by decisions made when evaluating the constraints, `Update` is called after `ExprDomain` to propagate the changes back down to the leaves of the expressions. An expression could be an algebraic expression containing several variables, or it could be a single variable or constant. The process will be the same except that the domains for algebraic expressions need expression evaluation techniques.

Example 4
As an example of expression handling, assume the following expression and associated domains:

```
A + B
A: < 0 .. 20 >
B: < 10 .. 50 >
```

The "`WHEN EXPRESSION`" case is taken in Figure 8, with `L = A`, `R = B`, and `aop = '+'`. When `ExprDomain` is called recursively within the `WHEN "+"` case, `exprdom.Top` is assigned `A.Top + B.Top = 20 + 50 = 70`, and `exprdom.Bot` is assigned `A.Bot + B.Bot = 0 + 10 = 10`. Thus, `ExprDomain` returns the domain $< 10 .. 70 >$.

Now let us consider a more complicated expression:

```
(A + B) * (C - D)
A: < 0 .. 20 >
B: < -50 .. 50 >
C: < 10 .. 50 >
D: < 30 .. 100 >
```

When `ExprDomain` is called, the "`WHEN EXPRESSION`" case is taken, with `L = A + B`, `R = C - D`, and `aop = '*'`. The `WHEN "*"` case checks all four combinations of `L.Bot*R.Bot`, `L.Bot*R.Top`, `L.Top*R.Bot`, and `L.Top*R.Top`, and uses the maximum of the four for the top, and the minimum

```
algorithm      ExprDomain (Expr, TmpDDS) : Domain
precondition   The variable's domains exist.
postcondition  The output exprdom is feasible, that is, exprdom.Top >= exprdom.Bot.
input          Expr:           an expression
               TmpDDS:         Domain Data Store
output         exprdom:        an expression domain with Bot and Top values as bounds
declare        L:              left side of Expr
               R:              right side of Expr
               aop:            arithmetic operator in Expr
               tmpBot, tmpTop: var to temporarily hold Bot and Top values


BEGIN
   CASE (Expr)
   WHEN CONSTANT:
      exprdom.Bot = exprdom.Top = constant
      RETURN exprdom
   WHEN (VARIABLE)
      exprdom.Top = Top_n in this var's domain in DDS
      exprdom.Bot = Bot_1 in this var's domain in DDS
      RETURN exprdom
   WHEN (EXPRESSION)
      L = GetLExpr (Expr)
      R = GetRExpr (Expr)
      aop = GetAop (Expr)
      CASE (aop)
         WHEN "+":
            exprdom.Top = ExprDomain (L, TmpDDS).Top + ExprDomain (R, TmpDDS ).Top
            exprdom.Bot = ExprDomain (L, TmpDDS).Bot + ExprDomain (R, TmpDDS ).Bot
         WHEN "-":
            exprdom.Top = ExprDomain (L, TmpDDS).Top - ExprDomain (R, TmpDDS ).Bot
            exprdom.Bot = ExprDomain (L, TmpDDS).Bot - ExprDomain (R, TmpDDS ).Top
         WHEN "*":  -- Must check all 4 combinations because of negative numbers.
            exprdom.Top = MAX (ExprDomain (L, TmpDDS).Bot * ExprDomain (R, TmpDDS).Bot,
                               ExprDomain (L, TmpDDS).Bot * ExprDomain (R, TmpDDS).Top,
                               ExprDomain (L, TmpDDS).Top * ExprDomain (R, TmpDDS).Bot,
                               ExprDomain (L, TmpDDS).Top * ExprDomain (R, TmpDDS).Top)
            exprdom.Bot = MIN (ExprDomain (L, TmpDDS).Bot * ExprDomain (R, TmpDDS).Bot,
                               ExprDomain (L, TmpDDS).Bot * ExprDomain (R, TmpDDS).Top,
                               ExprDomain (L, TmpDDS).Top * ExprDomain (R, TmpDDS).Bot,
                               ExprDomain (L, TmpDDS).Top * ExprDomain (R, TmpDDS).Top)
         WHEN "/":
            tmpTop = ExprDomain (R, TmpDDS).Top
            tmpBot = ExprDomain (R, TmpDDS).Bot
            IF (tmpTop == 0) -- Avoid division by zero.
               tmpTop = -1
            END IF
            IF (tmpBot == 0)
               tmpBot == 1
            END IF
            exprdom.Top = ExprDomain (L, TmpDDS).Top / tmpBot
            exprdom.Bot = ExprDomain (L, TmpDDS).Bot / tmpTop
         END CASE
      IF (exprdom.Top < exprdom.Bot)
         exprdom = Flip (TmpDDS, Expr, exprdom)
      END IF
      Add (TmpDDS, Expr, exprdom)
      RETURN exprdom
   END CASE
END ExprDomain
```

Figure 8: **The ExprDomain Algorithm – Finds a Possible Domain For an Expression.**

of the four for the bottom. This case analysis is required to handle negative values[3].

For the `A + B` expression, the domain < -50 .. 70 > is returned. The right side expression illustrates the `Flip` procedure. When `ExprDomain` is called on `C - D`, the domain < -20 .. -50 > is initially computed. Because the top is now less than the bottom, `Flip` is called to make this domain < -50 .. -20 >.

After the left and right side expressions are computed, `exprdom.Top = MAX (2500, 1000, -3500, -1400) = 2500`, and `exprdom.Bot = MIN (2500, 1000, -3500, -1400) = -3500`. Thus, `ExprDomain` returns the domain < -3500 .. 2500 >.

Downward propagation of expression domains
`Update` is used to propagate changes to an expression's domain back down the expression tree to the variables. It is a recursive procedure that attempts to "balance" changes to a domain between the two sides of the expression. Not all of `Update` is shown; for brevity, only the part that relates to expressions is included in this paper. Full details are available in the technical report [34]. `Update` implements an inherently nondeterministic procedure; that is, for a given expression and domain, there are many left and right domains that could result in the domain. Its decisions are designed to mirror `ExprDomain` by being as close as possible to an inverse function.

Example 5
As an example of the `Update` procedure, assume that the final domain from example 4, < -3500 .. 2500 >, was changed to < -500 .. 500 >. When `Update` is called, `Bot < 0` and `Top ≥ 0`, so the third branch in the "*" case is taken. `ldomain.Bot = -SquareRoot (Abs (Bot)) = -22.3`, `ldomain.Top = MIN (FLOOR (SquareRoot (Abs (Bot))), FLOOR (Top / SquareRoot (Abs (Bot)))) = 22.3`, `rdomain.Bot = - MIN (FLOOR (SquareRoot (Abs (Bot))), FLOOR (Top / SquareRoot (Abs (Bot)))) = -22.3`, and `rdomain.Top = SquareRoot (Abs (Bot)) = 22.3`. So the new domains are < -22 .. 22 > for both `A + B` and `C - D`.

Then `Update` is called recursively for `A + B` with the domains < -22 .. 22 >. This results in the domains `A: < -11 .. 11 >` and `B: < -11 .. 11 >`. `Update` is also called recursively for `C - D` with the domains < -22 .. 22 >, which results in the domains `C: < 22 .. 44 >` and `D: < 22 .. 44 >`. Thus, the modification to the expression's domain is used to modify the domains for the constituent variables.

Limitations
The DDR procedure currently has several limitations. This research has not yet addressed the interprocedural case, specifically, where a desired test path may extend through several procedures. Although we plan to explore inter-procedural test data generation in the future, the memory and execution requirements may simply be too high for structural-based coverage automatic test data generation. Currently, this is intended to be a unit testing technique. Thus far, dynamic domain reduction has only been applied to numeric software. This is primarily a limitation of the current proof-of-concept implementation (described in the next section), not the technique. Since the implementation must be able to handle all operations on all data objects, we restricted ourselves to numeric operations. There is no theoretical reason why appropriate extensions could not be made to handle other operations. Finally, the aliasing problem has still not been fully addressed. Although the solution for arrays works well, the proof-of-concept implementation does not fully handle pointers, so the ability to handle pointers has not been evaluated. Because the analysis is

[3]Note that the algorithm calls ExprDomain recursively four times for each bottom and top. This is of course inefficient, and the implementation uses temporary variables throughout the `ExprDomain` function to avoid repeated calls.

```
algorithm      Update (Expr, Bot, Top, TmpDDS)
precondition   Top >= Bot
postcondition  All domains of expr and var in TmpDDS are feasible,
               which is Top_i >= Bot_i for all vars' subdomains,
               and Top >= Bot for all exprs' domains.
input          Expr:       an expression
               Bot:        a bottom value of Expr's domain
               Top:        a top value of Expr's domain
               TmpDDS:     Domain Data Store
output         TmpDDS:     an updated DDS
return         Boolean
declare        L:          left side of Expr
               R:          right side of Expr
               aop:        arithmetic operator in Expr
               rdomain:    right expr's domain with Bot and Top
               ldomain:    left expr's domain with Bot and Top
               s:          subdomain


BEGIN
   L   = GetLExpr (Expr)
   R   = GetRExpr (Expr)
   aop = GetAop (Expr)
   CASE (aop)
      WHEN "+":  -- Divide the domains in half.
         ldomain.Bot = rdomain.Bot = Bot/2
         ldomain.Top = rdomain.Top = Top/2
      WHEN "-":
         -- There is a general form to get the domains of LExpr and RExpr:
         -- ldomain.Bot = Top + n*(Top - Bot)/2,
         -- ldomain.Top = (Top-Bot)/2 + Top + n*(Top - Bot)/2,
         -- rdomain.Bot = (Top-Bot)/2 + n*(Top - Bot)/2,
         -- rdomain.Top = Top - Bot + n*(Top - Bot)/2,
         -- (where n = ...  -3, -2, -1, 0, 1, 2, 3, ...)
         -- For this algorithm, n=0; in the implementation n is varied as part of a search process.
         ldomain.Bot = Top
         ldomain.Top = (Top-Bot)/2 + Top
         rdomain.Bot = (Top-Bot)/2
         rdomain.Top = Top -Bot
      WHEN "*":
         IF (Top ≥ 0 AND Bot ≥ 0)
            IF (CheckStatus (TmpDDS, Expr).flipped == TRUE)
               ldomain.Top = rdomain.Top = - SquareRoot (Bot)
               ldomain.Bot = rdomain.Bot = - SquareRoot (Top)
            ELSE
               ldomain.Top = rdomain.Top = SquareRoot (Top)
               ldomain.Bot = rdomain.Bot = SquareRoot (Bot)
         ELSE IF (Top < 0 AND Bot < 0)
            ldomain.Top = ABS (Top)
            ldomain.Bot = 1
            rdomain.Top = -1
            rdomain.Bot = Bot/ABS (Top)
         ELSE IF -- Bot < 0 AND Top ≥ 0
            ldomain.Bot = - SquareRoot (ABS (Bot))
            ldomain.Top = MIN (FLOOR (SquareRoot (ABS (Bot))),
                               FLOOR (Top/SquareRoot (ABS (Bot))))
            rdomain.Bot = - MIN (FLOOR (SquareRoot (ABS (Bot))),
                                 FLOOR (Top/SquareRoot (ABS (Bot))))
            rdomain.Top = SquareRoot (ABS (Bot))
```

```
        WHEN "/":
            -- There is a general form to get the domains of LExpr and RExpr:
            -- ldomain.Bot = (Top+1)/(Bot+1) * Bot**(i+1) * Top**i,
            -- ldomain.Top = Bot**i * Top**(i+1),
            -- rdomain.Bot = Bot**i * Top**i,
            -- rdomain.Top = (Top+1)/(Bot+1) * Bot**i * Top**(i+1),
            -- (where n = ...  -3, -2, -1, 0, 1, 2, 3, ...)
            -- For this algorithm, n=0; in the implementation n is varied
            -- as part of a search process.
            ldomain.Bot = (Top+1)/(Bot+1) * Bot
            ldomain.Top = Top
            rdomain.Bot = 1
            rdomain.Top = (Top+1)/(Bot+1) * Top
    END CASE
    IF (ldomain.Top < ldomain.Bot)
        ldomain = Flip (TmpDDS, L, ldomain)
    IF (rdomain.Top < rdomain.Bot)
        rdomain = Flip (TmpDDS, R, rdomain)
    RETURN (Update (L, ldomain.Bot, ldomain.Top, TmpDDS) AND
            Update (R, rdomain.Bot, rdomain.Top, TmpDDS))
END Update
```

Figure 9: **The Update Algorithm – Propagates Expression Domains Back Down To Variables.**

done dynamically, all pointers (and therefore pointer aliasing) will be resolved "on-the-fly", so it is expected that pointer aliasing will not be a problem. Aliasing through parameters arises during inter-procedural testing [35, 36], but not in the case of unit testing.

PRESENT STATUS

To observe the effectiveness of the dynamic domain reduction procedure on *C* programs, we have constructed a proof-of-concept tool. The tool currently does not handle pointers and the expression handling is limited to expressions that use numeric operators. The results focus on two questions: (1) does the dynamic domain reduction procedure work, and (2) does it work better than constraint-based testing? For the first question, test cases were automatically generated to satisfy the all-uses criterion, and for the second, test cases generated by the tool were directly compared with test cases generated by Godzilla. We know of no other tools that use structural-based coverage information to generate test cases.

All-uses evaluation

To evaluate the DDR procedure's capability to generate test cases, test cases were automatically generated to cover the all-uses data flow criterion for a number of program units. The coverage was measured using Bellcore's Atac [32].

Results are shown in Table 1. The number of functions and basic blocks in each program are provided, the total number of decisions, and the percent of decisions covered by dynamic domain reduction. The total number of DU-pairs required to achieve all-uses coverage, and the number of DU-pairs that are infeasible are also given. The last column gives the percent of DU-pairs that were covered by dynamic domain reduction, not including the infeasible DU-pairs. The infeasible DU-pairs were determined by hand analysis. For these programs, the DDR procedure was able to

cover almost every decision and DU-pair, even in the presence of loops and arrays. Generating the test cases took a few milliseconds for each; most of the execution was in creating the CFGs, and the time to generate tests was swamped by the time to execute and check the results. We know of no other automatic test data generator that can achieve a high degree of coverage for all-uses.

Program	Functions	Blocks	Decisions	% Decisions Covered	DU-Pairs	Infeasible DU-Pairs	% DU-Pairs Covered
Stats	4	70	32	100	154	15	92
Twenty-four	2	143	66	100	545	45	93
Conversions	8	523	247	97	1150	97	94
Binom	6	444	217	99	1110	151	93
Operators	4	1025	650	98	1438	233	97
Bub	1	9	5	100	29	1	100
Euclid	1	6	2	100	10	1	100
Insert	1	13	6	100	29	1	100
Mid	1	11	10	100	30	0	100
Quad	1	9	2	100	15	0	93
Trityp	1	31	34	100	101	14	99
Warshall	1	12	7	100	44	2	100

Table 1: **All-uses Data.**
Coverages are percentages of feasible paths.

Of course, these are relatively small programs, but it should be emphasized that DDR is designed to work on program **units**, not **integrated** software systems. Since the DDR procedure does not address the issue of inter-procedural testing, no claims can be made about its efficacy on integrated software modules or components. However, we have tried to choose a variety of subject functions and subroutines. Of course, the knowledge needed to choose a statistically representative sample is not available at this time, but the coverage results are good enough to merit consideration.

Constraint-based testing comparison
This section presents results from an empirical comparison of DDR with CBT. There are several practical differences between the new DDR tool and the constraint-based testing tool Godzilla that must be handled in any empirical comparison. The DDR tool tests programs in the language C, and generates test cases to satisfy data flow criteria. Godzilla, on the other hand, tests Fortran programs and generates test cases to satisfy mutation. To handle the language issue, Fortran programs were translated into C. Care was taken to use as direct a translation as possible so as not to introduce any variance into the results by using different programs. The control flow graphs, executable statements, and DU-pairs are exactly the same for all pairs of programs.

To get a valid comparison, test data were needed to satisfy the same criterion. Since both mutation and all-uses data flow subsume statement coverage, both tools can be used to satisfy statement coverage. Thus, statement coverage was used. Godzilla was used to generate test cases for statement coverage by only satisfying the reachability constraints, ignoring necessity constraints. The coverage of both sets of test cases was measured using Atac on the C versions of the program.

Results are shown in Table 2. The numbers of basic blocks in the programs are shown, followed by the percent of basic blocks covered by the CBT test cases and by the DDR test cases. As can be seen, the dynamic domain reduction procedure generated test cases that covered all blocks in all cases, and constraint-based testing missed a number of blocks, particularly in the

larger programs. In previous research [22], it was found that when values were generated purely randomly, block coverage of 70% to 80% was typically obtained, so it is clear that the DDR test cases had significantly more coverage than the CBT test cases. Hand analysis showed that the differences in coverage was due to two things. One was the fact that CBT treats all elements in an array as a single element, whereas DDR is able to treat array elements individually. The second was that the CBT-based tool ran out of memory and crashed on several occasions (particularly with the bigger procedures of Conversions, Binom, and Operators).

Program	Basic Blocks	CBT Coverage %	DDR Coverage %
Stats	70	81	100
Twenty-four	9	81	100
Conversions	523	79	100
Binom	444	80	100
Operators	1025	84	100
Bub	9	100	100
Euclid	6	91	100
Insert	13	100	100
Mid	11	100	100
Quad	11	100	100
Trityp	31	84	100
Warshall	12	100	100

Table 2: **Comparison with Constraint-based Testing. Coverages are given in percentages.**

Execution time
Algorithmic analysis of algorithms that are this complicated is exceedingly difficult. The execution time of the DDR procedure depends upon the number of decisions in the program (D), the number of paths (P), and the constant K. Moreover, if the the DDR procedure has to go through K attempts at each decision point, then that is K split attempts at the first decision, then K splits at the second decision for every attempt at the second decision, and so on, for a total of K^D split attempts. So the running time is $\Theta(P * K^D)$.

Although the worst-case running time is exponential, the worst case can seldom be expected to be achieved in practice. This upper bound will almost never be reached, and whether it is depends on aspects of the program that we have no idea how to quantify or measure. A more useful measure is the amount of time it takes for the tool to run. Neither of the CBT and DDR proof-of-concept tools were built with the idea of efficiency in mind, so they are not particularly efficient. Nevertheless, a comparison is useful. When generating test data, it took the CBT tool from a low of 9 seconds for the smallest procedure (Euclid), to over 12 hours for the largest set of procedures (Operators). For Operators, the tool had to be manually run several times on subsets of the constraints, because it kept running out of memory and crashing. For the DDR tool, it took from 5 seconds (with Euclid) to over 30 minutes for Operators, and never ran out of memory. Any of these data sets would take days, if not weeks, to generate by hand.

Even at this, the time to generate test cases is completely swamped by the time to execute the test cases (over 10 hours for Operators), which in turn is swamped by the time to manually check the result of each test case individually. In practical software testing, the dominant time is typically that of checking the outputs of test cases. Thus, the execution time of a DDR-based automatic test data generator can be expected to be only a tiny part of the time for the whole test process.

CONCLUSIONS

This paper presents a new method for automatically generating test data to test program units. It uses elements from previous test data generation methods and offers novel solutions to problems they encountered. The dynamic domain reduction procedure incorporates elements from the constraint-based testing domain reduction procedure, symbolic evaluation, and the dynamic test data generation approach. It integrates constraint satisfaction, symbolic evaluation, and a novel search process into one dynamic process. As compared with previous automatic test data generation procedures, we believe that the dynamic domain reduction procedure can be expected to be more likely to find a test case when a test case exists, and that implementations can be more effective and efficient. In this approach, array indexes can be calculated symbolically at the same time that values are being found, allowing the test data generation to overcome previous difficulties with arrays. In addition, the dynamic nature allows a space savings over previous methods, because the large constraint systems do not have to be stored and manipulated in their complete form.

The search process uses a new technique, *domain splitting*, to make choices at certain steps in the process. This allows flexibility in the values being chosen, and allows an efficient search procedure (based on bisection) to be used, which in turn increases the chances for success. This process also allows complicated expressions to be handled uniformly, as described previously. The algorithms developed for DDR are too numerous and long to all be presented here; they can be found in a technical report [34].

This paper also presents results from a proof-of-concept implementation of the DDR procedure.

On small programs, the procedure comes very close to completely satisfying the all-uses data flow criterion, and performs better than constraint-based testing for statement coverage. We know of no other automatic test data generator that can achieve a high degree of coverage for all-uses. This procedure can also work for procedures that are too big for existing techniques.

Of course, any technique for automated test data generation has inherent limitations. Some of the problems of arrays and loops are formally undecidable and cannot be completely solved. But this technique uses more information about such constructs than previous methods, and requires significantly less space, allowing more test cases to be automatically generated. In the empirical study, it was found that arrays and loops were handled both more accurately and more efficiently than with the constraint-based approach. This paper also makes no claims about optimality. Test data generation is an extremely complex problem and it can only be hoped to find partial solutions that are general and robust enough to work most of the time in the real world.

Future work

One common problem in test data generation is that of detecting infeasible paths. This shows up in various testing criteria in different forms — in mutation it is part of the equivalent mutant problem, and in data flow testing the term infeasible DU-pairs has been used [3]. The problem was considered directly by Jasper et al. [37] and Goldberg et al. [38] in the context of branch testing. Offutt and Pan [39] presented a technique for detecting equivalent mutants that is based on recognizing infeasible systems of constraints. Whereas the DDR procedure will fail in the presence of infeasible paths, the fact that the path is infeasible is not explicitly known. We hope to modify the results of Pan's thesis to work with the DDR procedure to explicitly recognize most infeasible paths (and infeasible DU-pairs).

Most of the work in automated test data generation has been intra-procedural rather than inter-procedural. One implication of this fact is that the generators are useful for unit testing, but generally do not perform well for integration or system testing. We believe that the DDR procedure could be applied in a limited manner to inter-procedural problems. Because the constraint systems are analyzed and disposed of in-process, the combinatorial explosion of constraints that happens with traditional techniques can be avoided. Hopefully, this will allow test data to be generated inter-procedurally, during module and integration testing.

This research is part of a long term project to provide practical, powerful automated test environments to testers, so that highly reliable software can be produced at reasonable cost. We envision an eventual system that provides almost complete automation to the tester. This type of system would allow a programmer to submit a software module, and after a few minutes of computation, respond with a set of test cases that are assured of providing the software with a very effective test, and a set of outputs that can be examined to find failures in the software. Furthermore, these input-output pairs can be used as a basis for debugging when failures are found. It is hoped that eventually the unit testing process will become part of compilation.

ACKNOWLEDGMENTS

It is a pleasure to acknowledge Bellcore, and specifically Bob Horgan and Saul London, for the use of the testing tool ATAC and for support using it. We would also like to thank Zhen Zhou for help with the implementation.

*

References

[1] G. Myers. *The Art of Software Testing*. John Wiley and Sons, New York NY, 1979.

[2] I. Sommerville. *Software Engineering*. Addison-Wesley Publishing Company Inc., 4th edition, 1992.

[3] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[4] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

[5] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[6] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[7] R. S. Boyer, B. Elpas, and K. N. Levitt. Select–a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, June 1975. SIGPLAN Notices, vol. 10, no. 6.

[8] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4):293–300, December 1976.

[9] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, September 1976.

[10] W. E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, 3(4), July 1977.

[11] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering Methodology*, 5(1):63–86, January 1996.

[12] Juris Borzovs, Audris Kalniņš, and Inga Medvedis. Automatic construction of test sets: Practical approach. In *Lecture Notes in Computer Science, Vol 502*, pages 360–432. Springer Verlag, 1991.

[13] E. F. Miller and R. A. Melton. Automated generation of testcase datasets. In *Proceedings of the International Conference on Reliable Software*, pages 51–58, April 1975.

[14] J. A. Bauer and A. B. Finger. Test plan generation using formal grammars. In *Proceedings of the 4th International Conference on Software Engineering*, pages 425–432, San Diego CA, September 1979. IEEE Computer Society Press.

[15] Peter M. Maurer. Generating testing data with enhanced context-free grammars. *IEEE Software*, 7(4), July 1990.

[16] H. Sturgis. An effective test strategy. Technical report CSL-85-8, Xerox Parc, November 1985.

[17] H. D. Mills, M. D. Dyer, and R. C. Linger. Cleanroom software engineering. *IEEE Software*, September 1987.

[18] J. M. Voas, L. Morell, and K. W. Miller. Predicting where faults can hide from testing. *IEEE Software*, 8(2):41–58, March 1991.

[19] W. E. Howden. Methodology for the generation of program test data. *IEEE Transactions on Software Engineering*, (SE-24), May 1975.

[20] L. A. Clarke and D. J. Richardson. Applications of symbolic evaluation. *The Journal of Systems and Software*, 5(1):15–35, January 1985.

[21] J. A. Darringer and J. C. King. Applications of symbolic execution to program testing. *IEEE Computer*, 11(4), April 1978.

[22] R. A. DeMillo and A. J. Offutt. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering Methodology*, 2(2):109–127, April 1993.

[23] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[24] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[25] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Banff Alberta, July 1988. IEEE Computer Society Press.

[26] B. Korel. Dynamic method for software test data generation. *The Journal of Software Testing, Verification, and Reliability*, 2(4):203–213, 1992.

[27] R. E. Fairley. An experimental program testing facility. *IEEE Transactions on Software Engineering*, SE-1:350–3571, December 1975.

[28] A. J. Offutt. An integrated automatic test data generation system. *Journal of Systems Integration*, 1(3):391–409, November 1991.

[29] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[30] F. Tip. A survey of program slicing techniques. Technical report CS-R-9438, Computer Science/Department of Software Technology, Centrum voor Wiskunde en Informatica, 1994.

[31] T. E. Cheatham, G. H. Holloway, and J. A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Transactions on Software Engineering*, 5(4), July 1979.

[32] J. R. Horgan and S. London. ATAC: A data flow coverage testing tool for C. In *Proceedings of the Symposium of Quality Software Development Tools*, pages 2–10, New Orleans LA, May 1992.

[33] P. G. Frankl, S. N. Weiss, and E. J. Weyuker. ASSET: A system to select and evaluate tests. In *Proceedings of the Conference on Software Tools*, New York NY, April 1985. IEEE Computer Society Press.

[34] J. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction approach for test data generation: Design and algorithms. Technical report ISSE-TR-94-110, Department of Information and Software Systems Engineering, George Mason University, Fairfax VA, September 1994.

[35] H. D. Pande, W. A. Landi, and B. G. Ryder. Interprocedural def-use associations for c systems with single level pointers. *IEEE Transactions of Software Engineering*, 20(5):385–403, May 1994.

[36] M. J. Harrold and M. L. Soffa. Selecting and using data for integration testing. *IEEE Software*, 8(2):58–65, March 1991.

[37] R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman. Test data generation and feasible path analysis. In *Proceedings of the 1994 International Symposium on Software Testing, and Analysis*, pages 95–107, Seattle WA, August 1994.

[38] A. Goldberg, T. C. Wang, and D. Zimmerman. Applications of feasible path analysis to program testing. In *Proceedings of the 1994 International Symposium on Software Testing, and Analysis*, pages 80–94, Seattle WA, August 1994.

[39] A. J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. *The Journal of Software Testing, Verification, and Reliability*, 7(3):165–192, September 1997.