The Dynamic Domain Reduction Procedure for Test Data Generation: Design and Algorithms*

A. Jefferson Offutt
Zhenyi Jin
Jie Pan
ISSE Department
George Mason University
Fairfax, VA 22030
phone: 703-993-1654
fax: 703-993-1638
email: {ofut,jpan,zjin}@isse.gmu.edu

August 1994

ISSE Technical Report ISSE-TR-94-110

Abstract

Although there are many techniques and tools available to support the software testing process, one of the most crucial parts of testing, generating the test data, is usually done by hand. Test data generation is one of the most technically challenging steps in testing software, but unfortunately, most practical systems do not incorporate any automation in this step. This paper presents a new method 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 "pushes" the values through the control-flow graph of the program in a dynamic way, modifying the sets of values as branches in the program are taken. This method is an outgrowth of previous research in constraint-based testing, and combines several of the steps that were previously separate into one coherent process. The major difference is that this method is dynamic, and resolves path constraints immediately; it also includes an intelligent search technique, and improved handling of arrays, loops, and pointers.

Keywords: Automated test generation, software testing, symbolic evaluation.

*Supported by the National Science Foundation under grant CCR-93-11967.

1 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 [Mye79, Som92]. If most of software testing could be automated, then the cost of software development could be greatly reduced. One of the most difficult and expensive technical problems of software testing has been the actual generation of test data—which has traditionally been done by hand. Test data generation is the process of creating program input data that satisfy some testing criterion. An automatic test data generator is a tool that helps the tester create test data. Test data generators have been categorized into three groups: structural-oriented test data generators [BKM91, BEL75, Cla76, DO91, How77, Kor90, RHC76], data specification generators [BF79, MM75, Mau90], and random test data generators [Stu85, VMM91, MDL87]. This paper focuses on structural-oriented generators, which are based on covering certain structural elements in the program. Usually, these generators attempt to generate test data to meet a testing criterion such as path coverage, branch coverage, mutation, etc.

Structural-oriented test data generators typically use the CFG of the program and some form of symbolic evaluation [DK78]. Symbolic evaluation executes a program using symbolic values for variables instead of actual values. Symbolic evaluation derives a *path constraint* for a path in the program, which is a constraint system on the program input variables. The path constraint must be satisfied for the path to be traversed; the path constraint is usually derived first, then an attempt to satisfy it is made. Although symbolic evaluation is a very powerful analysis tool, it has several problems such as determining array indexes and handling pointers and indeterminate loops.

In previous work [DO91, DO93], we presented an approach to test data generation that uses symbolic evaluation and other information about the program to automatically generate test data to satisfy the mutation testing criterion [DLS78, DGK+88]. This approach, called *constraint-based testing (CBT)*, uses a novel constraint satisfaction technique called the "domain reduction procedure". CBT suffers from several problems, including problems handling arrays, loops, and nested expressions. In this report, we present a new approach to test data generation, called the *dynamic domain reduction procedure*, which combines several of the previously separated steps into one smooth process. The dynamic domain reduction procedure (DDR) uses elements of the CBT approach, but also draws from Korel's dynamic test data generation approach [Kor90] and symbolic evaluation. It uses a direct "domain reduction" method for deriving values, rather than function minimization methods [Kor90] or linear programming-like methods [Cla76].

Previous approaches occasionally failed to find test cases, and for some programs failed a large percentage of the time. This is partly because of problems with handling arrays and loops, partly because of insufficiently general approaches to handling expressions, and partly because of unsophisticated search procedures. The DDR procedure is capable of succeeding where others fail because of several improvements. Its dynamic nature, combining symbolic evaluation with domain reduction, allows better handling of arrays and expressions. DDR also incorporates a sophisticated back-tracking search procedure that allows test data to be generated when previous methods would fail.

The DDR procedure walks through the program CFG, generating test data along the way. Each variable is initially given a large set of possible values (its *domain*), and as branches are taken in the CFG, the domains for the variables involved in the predicates are reduced to reflect the truth values of the predicates. When choices for how to reduce the domains must be made, a search process is initiated to try to make a choice that allows the subsequent edges on the path to be executed. When the procedure is finished, the remaining values for the variable's domains represent the set of test cases that will cause execution of the path. If any variable's domain is empty, the search process failed, indicating that either the path was infeasible or there are relatively few inputs that will execute the path.

The next section introduces some background terminology and concepts. In Section 3, a formal description of the test data generation problem is given, and Section 3.1 describes the domain reduction procedure. Our new procedure for generating test data is presented in Section 3.2. A discussion of how arrays, loops, and pointers are handled in this procedure is given in Section 4 and future research is suggested in the Conclusions.

2 Background

In this section, some basic concepts used in this report are introduced, a formal description of the test data generation problem is given, and finally the domain reduction procedure is introduced.

2.1 Concepts

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

A **control flow graph (CFG)** of a program is a directed graph that represents the structure of the program. Each node is either a basic block node, a junction node or a decision node. The edges represent potential control flow from node to node. A *control path* is a directed path from an entry node to a terminal node of a CFG. A *predicate* is a controlling function associated with a decision node whose value evaluates to TRUE or FALSE, which determines which branch will be followed. The flow of control in the CFG will change according to the evaluation of the predicate expression. A *constraint* is a mathematical algebraic expression that restricts the space of the input program variables to certain input domains so that the constraint can be satisfied. For example, a constraint $A > 0$ describes the portion of the input domain where A is positive.

Each path is represented by a list of constraints, one constraint for each predicate along the control path. 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, the predicates can be re-expressed as constraints in terms of only the input variables.

If input data that satisfy the path condition exist, the control path is also an *execution path* and can be used to test the program. If the path condition can not be satisfied by any input values, the control path is said to be *infeasible*.

For example, given a program that calculates GCD of integer A and B using Euclid's algorithm as follows:

```
   int euclid(A,B)
   int A, B;
   {
     int div, rem;
1.   rem = 1;
2.   WHILE (rem > 0)
     {
4.     div = A/B;
5.     rem = A - Div * B;
6.     A = B ;
7.     B = rem;
     };
8.   return (A)
   }
```

The CFG of this program is shown in Figure 1.

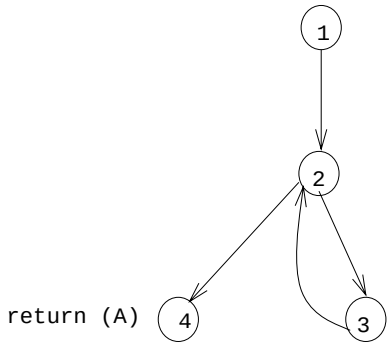

Figure 1: **The CFG of program** *euclid*

In Figure 1, node 1 is a basic block node; it contains statement 1 of the program: rem = 1. Node 2 is a decision node because the control flow diverges into two directions: 2-4 and 2-3. Node 2 is also a junction node since control flow from node 3 merges to node 2. The predicate associated with decision node 2 is (rem > 0) in the WHILE statement. If this predicate evaluates to TRUE, then the program will proceed to node 3; otherwise node 4. Node 4 is the exit node, return(A). Path 1-2-4 is a control path, 1-2-3-4 is another control path.

A *test case* is a set of input data that is used in the software to execute on so as to determine if the software is correct on these inputs. Test data generation in software testing is the process of

identifying a set of test data that satisfies a selected testing criterion. The *domain* of a variable is the set of possible values that the variable can have. The domain does not have to be continuous, there can be many discrete sets of values.

A *constraint system* is a hierarchical structure composed of expressions, constraints, and clauses. An *algebraic expression* is composed of variables, parentheses, and programming language operators. Expressions are taken directly from the test program and are derived from predicates within decision statements and right-hand sides of assignment statements during symbolic evaluation. A *constraint* is a pair of algebraic expressions related by one of the conditional 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 CBT systems, all constraint systems are kept in *disjunctive normal form* (DNF), which is a list of conjunctive clauses connected by logical ORs.

3 A TEST DATA GENERATION PROBLEM

The problem of automatic test data generation has been looked at by a number of researchers. Korel [Kor90] gives a formal description of the problem in terms of finding inputs to execute a particular path in the program. In earlier work [DO91], we described this problem in terms of killing a mutant. In this paper, we present this problem in terms of reaching a particular node using an arbitrary path. Executing a particular path is a special case of this treatment, and it is easily extended to incorporate testing criteria such as data flow or mutation.

Let n_g be a node in the CFG of a program P with input domain D; we call n_g the "goal" node. The test data generation problem is *to 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, we say that given a path in the program $p =< n_1, n_2, ..., n_g >$, t must cause that path to be executed. For mutation testing, the goal node n_g contains the statement that is mutated, and we impose the additional requirement that after n_g is executed, the *necessity condition* must be true. The necessity condition [DO91] is the condition that expresses what is necessary for a test case to kill the mutant. Our problem statement can also be extended to include data flow testing criteria. The *All-uses* data flow criterion [FW88] 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 we add the requirement that after n_g is reached, the node containing the use of x must also be reached (n_u), with the restriction that the subpath from n_g to n_u must not contain another definition of x.

3.1 THE DOMAIN REDUCTION PROCEDURE

The CBT approach uses separate procedures to derive constraints that represent conditions under which a particular statement will be reached (*reachability constraints*), derive constraints that represent conditions under which a mutant will be killed (*necessity constraints*), apply symbolic evaluation to rewrite the constraint systems to be in terms of input variables, and finally to find test case input values that will satisfy the constraints (*constraint satisfaction*). These procedures are described in detail elsewhere [DO91, Off91] and have been implemented in a test data generation tool Godzilla [DGK+88].

The constraint satisfaction procedure used by Godzilla is known as "domain reduction" [Off91] 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 constraint within the system 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 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.

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 effectively reduced by one dimension. As the number of dimensions is reduced, the constraints in the system become progressively simpler. Each variable assignment implicitly introduces a new constraint into the system of the form $(x = c)$, where c is a constant. If chosen poorly, this new constraint may make the constraint system infeasible, and the procedure will have to make a new choice. The basic assumption is that because of the simple form of the test case constraints, these dimension reducing constraints will rarely make the region infeasible. When they do, CBT employs a very simple search procedure; a different value is chosen for the same variable and the system is re-evaluated.

Weaknesses of the Domain Reduction Procedure

The domain reduction procedure uses the entire constraint system, which contains many constraints. Each time a constraint is used, the procedure reduces the domains by choosing one definite value from that domain and then uses this value throughout the reduction procedure on other constraints. This does not give much flexibility in the values chosen, which in turn may decrease the chance for other constraints to be satisfied. As a result, the procedure may have to re-choose

values more often, which decreases the efficiency and effectiveness of the procedure. Also, since the domain reduction procedure uses random guessing of variable values in its searching procedure, this leaves the search procedure poorly organized and same values may be chosen more than once. The domain reduction procedure also has very simple expression handling mechanisms. Most real world programs have more complicated expressions, which the domain reduction procedure has to either skip or modify into a format that can be handled. This limits the scope of the programs that can be tested by the procedure.

Also, the domain reduction procedure views an array as one variable, and does not differentiate values between individual elements in the array. This will impact the power of the test data generation process on programs that make heavy use of arrays.

3.2 The Dynamic Domain Reduction Procedure

DDR procedure is an automatic test case generating method that uses constraints derived from the test program that control the execution paths in a CFG to reduce domains of variables progressively until test data that satisfy these constraints are found for the test program. This method finds variable values by walking through the CFG, using one predicate at a time and reducing domains step by step. DDR procedure uses certain mechanisms to generate test cases. These mechanisms include reducing domain spaces by splitting variable domains, using a binary search algorithm when bad choices were made, and using a improved expression handling technique.

3.3 Representations and Assumptions

Predicates and constraints are used to reduce variable domains in the dynamic domain constraint satisfaction procedure. DDR procedure handles certain types of predicates and constraints in the test program and therefore makes the following assumptions:

- Predicates are always in disjunctive normal form.
- Constraints and expressions are put in a canonical form where constants are always on the right.

```
Predicate     = ORClause
OrClause      = AndClause1 ∨ AndClause2 ∨...∨ AndClausen
AndClause     = Constraint1 ∧ Constraint2 ∧ ...  ∧ Constraintm
Constraint    = LeftExpr Relation RightExpr
LeftExpr      = Expr
RightExpr     = Expr | Const
Expr          = Var aop Var | Var aop Const | Expr aop Expr |
                Expr aop Var | Expr aop Const | Var
aop           = + | - | * | /
relation      = < | > | >= | <= | = | !=
```

3.4 Overview of the Dynamic Domain Reduction Procedure

DDR procedure flow diagram is shown in Figure 2. In the diagram, bubbles represent inputs and outputs of the procedure, rectangles represent process steps, and diamonds represent branches in the execution. Three kinds of input data are needed in this procedure: the value domains of the input variables to the test program; the CFG of the program; and the starting node N_1 and goal node N_g in the CFG.

In `Find Path`, a path that starts from node N_1 and ends at N_g in the CFG is chosen. How this is done will vary depending on the testing technique. For instance, in path-coverage testing, the input will be a complete path, while in data flow testing N_g may be a def node and there may be many possible paths from N_1 to N_g. The `Is P empty?` decision checks whether all paths in path set P have already been searched. If all paths in path set P have been checked and no test data has been found, the procedure fails; either there are no feasible paths from N_1 to N_g or the test case that will execute a feasible path is too difficult to find. The `Select one path` procedure selects a subpath P_i to satisfy from the subpath set P. This subpath is removed from the path set P so that it will not be chosen again.

The rest of the procedure walks through the CFG along the subpath P_i, attempting to find a test case that will execute the subpath. If the current node is a decision node 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 used to do symbolic evaluation, which also modifies the variable domains.

The `Reduce domains` step is the key process of DDR procedure. Variables related to the constraint are used to check if they satisfy the constraint or not. The constraint is then used to reduce the variable domains. An important function *SplitPoint* is used here to reduce the domains of the variables.

SplitPoint encodes our major heuristics for selecting test case values and is the key for our searching process. *SplitPoint* takes a constraint and two expressions with overlapping domains and finds a point at which the two domains can be split so that the constraint will be true for all values in the domains. Initially, the domains are split so that each domain "loses" approximately the same number of values. During the search process, the split point is successively reevaluated with the split point moved halfway in one direction, then the other, and so on until the choice succeeds in resulting in a test case, all choices have been exhausted, or a predetermined constant number of choices have been made. After the domains have been reduced, 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 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 is chosen arbitrarily.

If the new domain values do not satisfy the predicate, the search process is triggered. The procedure goes back to the most recent decision node that was encountered, and tries to satisfy the predicate again using different split points. If there are no previous decision nodes to evaluate, the procedure gives up on this path and goes to the next one. If there have been too many attempts to

find a feasible split point at the most recent decision node (more than a some constant value K), then the procedure goes to the previous decision node.

3.4.1 Functions

Six key functions used in DDR procedure are described in this section. Functions are first described, and then the algorithm name and variables are declared, finally the pseudo code of the function is listed.

1. **Function TCGenerator:** The main function of the DDR procedure.

   The inputs to this function are the variable domains, which are a list of bottom and top values for each variable's domain. Domain Data Stores (DDS) stores domains of expressions of the program, and is initialized as the domains of input variables. The TCGenerator first initializes DDS using domains of input variables. Then for a given start node and an end node of the CFG (Defined as def-use in the function), the function gets all the possible subpaths that starts and ends at the given nodes. For each subpath chosen from the SubPathSet, predicate on each edge e in the subpath is read and used to check if the current domains satisfy this predicate or not, this is processed in the function FoundSuitableDom(DDS,P). If current domains satisfy the predicate, then next edge on the subpath is checked until all edges on the subpath has been checked or a test case is found where test-case-found flag TCFound is assigned TRUE, otherwise, another subpath is chosen to generate the test cases.

   Declarations and pseudo code of this function are given as follows:

```
algorithm      TCGenerator ()
declare        DefUse:          DefUse_Type
               SubPathSet:      the set of all def clear subpaths from a def to a use
               SubPath:         a subpath in the SubPathSet
               Pred[i]:         the predicate on the i_th edge of a subpath
               DomSatisfyPred:  Boolean
               TCFound:         Boolean
               SubPathSetEmpty: Boolean
               p:               a predicate


BEGIN
    Get domain_values and initialize the DDS.
    FOR each var
        Init (DDS, Var, Domain)
    END FOR

    FOR each DefUse
        Get SubPathSet
        TCFound         = FALSE
        DomSatisfyPred  = TRUE
        SubPathSetEmpty = FALSE
        WHILE (¬TCFound AND ¬SubPathSetEmpty)
            Get one SubPath from SubPathSet
```

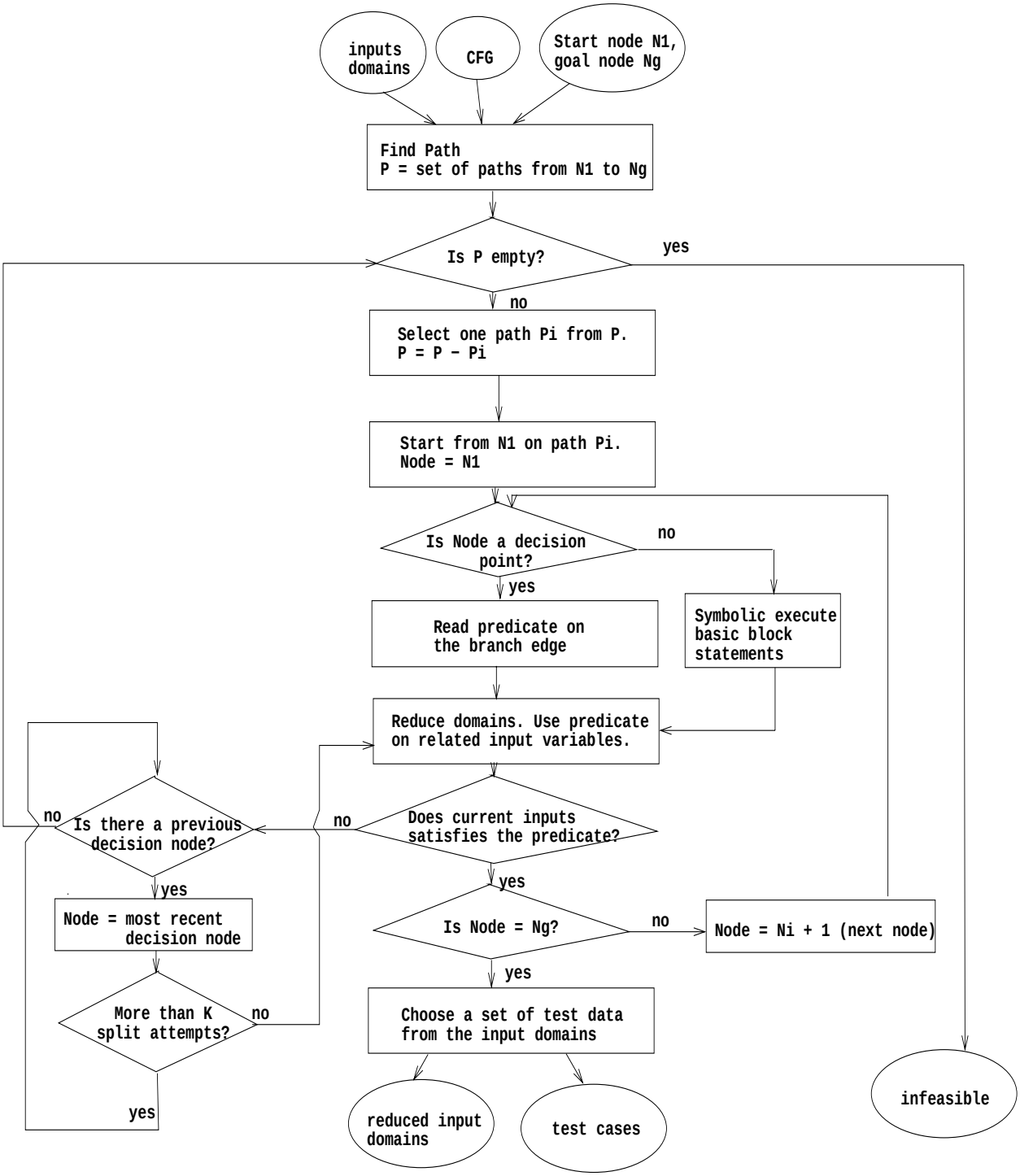

Figure 2: **The dataflow diagram**

```
            FOR each edge e on SubPath
                p = Pred[e]
                IF (¬FoundSuitableDom (DDS, p)) – Modify DDS to satisfy p.
                    DomSatisfyPred = FALSE
                    BREAK FOR LOOP
                END IF
            END FOR
            IF (DomSatisfyPred) – Entire subpath is satisfied
                Generate test case by using DDS -- a function call in DDS that also saves TC.
                TCFound = TRUE
                Reset DDS
            ELSE – Did not satisfy that subpath, try another.
                SubPathSet = SubPathSet - SubPath
        END WHILE
    END FOR
END TCGenerator
```

2. **Function FoundSuitableDom:** The function finds suitable domains for variables that satisfy the predicates.

   The input to this function is a predicate and DDS. This function outputs a boolean value indicating whether current DDS satisfy the predicate or not. The function checks each AND clause in the predicate, makes a temporary copy of DDS to TempDDS, and then check each constraint in the AND clause. If the domains fit each constraint in the AND clause, then the function terminates and returns boolean value TRUE to the caller function. When the function returns, TempDDS is copied to DDS to update any modifications. If any one of the constraints is not satisfied by current domains, the function returns FALSE notifying that the constraints are not satisfied.

   Declarations and pseudo code of this function are as follows:

```
algorithm     FindSuitableDom (DDS, p) :  Boolean
declare       AndCl[i]:  the i_th AndClause in p
              Cnst[i]:   the i_th Constraint in an AndClause
              TmpDDS:    DDS_Type -- a temporary DDS
              Infeasible:boolean
              cl:        a clause
              c:         a constraint

BEGIN
    FOR each AndClause cl in p
        Infeasible = FALSE
        Copy (DDS, TmpDDS)
        FOR each constraint c in cl
            IF (¬DomFitCnst (c, TmpDDS)) – Attempt to modify TmpDDS to satisfy c
                Infeasible = TRUE
                BREAK FOR LOOP
            END IF
        END FOR
        IF (¬Infeasible)
            Copy (TmpDDS, DDS)
            RETURN TRUE
        ELSE
```

```
            RETURN FALSE
        END IF
    END FOR
END FoundSuitableDom
```

3. **Function DomFitCnst:** The function that checks whether an expression's domain fits a constraint.

   The input to this function is a copy of DDS, named TempDDS and a constraint. The constraint contains two expressions and a relational operator that connects the two expressions. The expression on the left-hand side of the relational operator is called left-expression and the right-hand side right-expression. The function reads the left-expression, right-expression and the relational operator, then assign them to corresponding variables. There are two cases in this constraint, the first is when left-expression is a variable and right-expression is a constant; the second case is when both expressions are variables. These two cases are dealt differently. GetSplit function is called to find a split point for the two expressions. Getsplit function is to be discussed next.

   Pseudo code of this function are listed as follows:

```
algorithm     DomFitCnst (Cnst, TmpDDS) : Boolean
declare       rel:      a relation in Cnst
              fit:      Boolean
              lexpr:    left side expr of Cnst
              rexpr:    right side expr of Cnst
              ldomain:  left expr's domain with Bot and Top values
              rdomain:  right expr's domain with Bot and Top values
              const:    a constant value
              split:    a split point of a domain
              lsucceed: Boolean -- lexpr update TmpDDS succeeded
              rsucceed: Boolean -- rexpr update TmpDDS succeeded


BEGIN
    lexpr = GetLExpr (Cnst)
    rexpr = GetRExpr (Cnst)
    rel   = GetRel (Cnst)
    IF (rexpr is a constant)
        ldomain = ExprDomain (lexpr)
        const   = GetValue (rexpr)
        CASE (rel)
            WHEN ">":
                IF (ldomain.Bot ¿ const)
                    RETURN TRUE – TmpDDS stays the same.
                END IF
                IF (ldomain.Top ≤ const)
                    RETURN FALSE
                END IF
                lsucceed = Update (lexpr, const+1, ldomain.Top, TmpDDS)

    ELSE
        ldomain = ExprDomain (lexpr)
        rdomain = ExprDomain (rexpr)
```

```
        CASE (rel)
            WHEN ">":
                IF (ldomain.Bot ¿ rdomain.Top)
                    RETURN TRUE – TmpDDS stays the same.
                END IF
                IF (ldomain.Top ¡= rdomain.Bot)
                    RETURN FALSE
                END IF
                -- find a split point to split the domain.
                split = GetSplit (ldomain, rdomain)
                -- Now we might have a loop here, if one of following updates is
                -- unsuccessful, then adjust split, do again.
                lsucceed = Update (lexpr, split+1, ldomain.Top, TmpDDS)
                rsucceed = Update (rexpr, rdomain.Bot, split, TmpDDS)

    END IF
    IF (lsucceed AND rsucceed)
        RETURN TRUE
    ELSE
        RETURN FALSE
    END IF
END DomFitCnst
```

4. **Function GetSplit:** The function that finds a split point for variable domains.

   The input to this function are two domains for two expressions. Each expressions has domain (ldomain.$_{Bot}$ .. ldomain.$_{Top}$) and (rdomain.$_{Bot}$ .. rdomain.$_{Top}$) respectively. A split point is found based on the given domains. The split point found is then returned to the caller functions to reduce the domains. The split point is calculated based on the two given domains. There are four cases described as follows:

   (a) (ldomain.$_{Bot}$ $\geq$ rdomain.$_{Bot}$) and (ldomain.$_{Top}$ $\leq$ rdomian.$_{Top}$) :
   SplitPoint = (ldomain.$_{Top}$ - ldomain.$_{Bot}$)*i + ldomain.$_{Bot}$

   (b) (ldomain.$_{Bot}$ $\leq$ rdomain.$_{Bot}$) and (ldomain.$_{Top}$ $\geq$ rdomain.$_{Top}$) :
   SplitPoint = (rdomain.$_{Top}$ - rdomain.$_{Bot}$)*i + rdomain.$_{Bot}$

   (c) (ldomain.$_{Bot}$ $\geq$ rdomain.$_{Bot}$) and (ldomain.$_{Top}$ $\geq$ rdomain.$_{Top}$) :
   SplitPoint = (ldomain.$_{Top}$ - rdomain.$_{Bot}$)*i + rdomain.$_{Bot}$

   (d) (ldomain.$_{Bot}$ $\leq$ rdomain.$_{Bot}$) and (ldomain.$_{Top}$ $\leq$ rdomain.$_{Top}$) :
   SplitPoint = (rdomain.$_{Top}$ - ldomain.$_{Bot}$)*i + ldomain.$_{Bot}$

   i is a value from a set (1/2, 1/4, 3/4, 1/8, 7/8, ...).

   If new domain values do not satisfy the predicate, the the most recent decision node that was encountered has to be used to find different split points. The value of i changes sequentially according to the values in the given set each time this redo happens.

   For example, given ldomain (-20, 20), rdomain (-40, 30),

   this fits case 1 where (rdomain.$_{Bot}$ < ldomain.$_{Bot}$) and (rdomain.$_{Top}$ > ldomian.$_{Top}$),

   so, SplitPoint = (ldomain.$_{Top}$ - ldomain.$_{Bot}$)*i + ldomain.$_{Bot}$ = (20 - (-20))/2 + (-20) = 0

If the domains reduced based on this split point do not satisfy later constraints, then this split point needs to be calculated again.

In this case, i=1/4,

SplitPoint = (ldomain.$_{Top}$ - ldomain.$_{Bot}$)*i + ldomain.$_{Bot}$ = (20 - (-20))/4 + (-20) = -10

Declarations and the algorithm are listed as follows:

```
algorithm      GetSplit (ldomain, rdomain)


BEGIN
    -- Try to equally split leftexpr's and rightexpr's domain.
    IF (ldomain.Bot >= rdomain.Bot AND ldomain.Top <= rdomain.Top)
        split = (ldomain.Top - ldomain.Bot)/2 + ldomain.Bot
    ELSE IF (ldomain.Bot <= rdomain.Bot AND ldomain.Top >= rdomain.Top)
        split = (rdomain.Top - rdomain.Bot)/2 + rdomain.Bot
    ELSE IF (ldomain.Bot >= rdomain.Bot AND ldomain.Top >= rdomain.Top)
        split = (ldomain.Top - rdomain.Bot)/2 + rdomain.Bot
    ELSE
        split = (rdomain.Top - ldomain.Bot)/2 + ldomain.Bot
    END IF
    RETURN split
END GetSplit
```

5. **Function ExprDomain:** The function that finds a possible domain for an expression.

   The inputs to this function are an expression and DDS. Since expression is defined recursively, this domain reduction process may run recursively until domains for each specific variables are found. This function processes recursively by determining the domains of the variables and constants at the leaves of the expression and propagating these domains up by applying the operations. If there are any changes that are necessitated by decisions, then they are propagated back down to the leaves of the expressions. An expression could be algebraic expressions consists several variables, or it could be a single variable or constant, this depends on the content of the CFG. The process will be the same except that domain range for algebraic expressions will need expression evaluation techniques.

   Declarationa and pseudo code are as follows:

```
algorithm      ExprDomain (Expr, TmpDDS) : Domain
declare        L:          left side of Expr
               R:          right side of Expr
               aop:        arithmetic operator in Expr
               tmpBot:     a var to hold Bot value temporarily
               tmpTop:     a var to hold Top value temporarily


BEGIN
    IF (Expr is a constant)
        exprdom.Bot = exprdom.Top = constant
        RETURN exprdom
```

```
    END IF
    IF (Expr is a var)
        exprdom.Top = Top_n in this var's domain in DDS
        exprdom.Bot = Bot_1 in this var's domain in DDS
        RETURN exprdom
    END IF
    IF (Expr is an expr)
        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 "*":
                exprdom.Top = ExprDomain (L, TmpDDS).Top * ExprDomain (R, TmpDDS ).Top
                exprdom.Bot = ExprDomain (L, TmpDDS).Bot * ExprDomain (R, TmpDDS ).Bot
            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
        IF (exprdom.Top < exprdom.Bot)
        exprdom = Flip (TmpDDS, Expr, exprdom)
        Add (TmpDDS, Expr, exprdom)
        RETURN exprdom
    END IF
END ExprDomain
```

6. **Function Update:** The function that updates the temporary domains data stores (TempDDS).

   TempDDS is modified if the domain of an expression has no values less than the bottom of the domains or no greater than the top of the domains. Domains of variables in the expression are also modified, they are processed recursively. If update is feasible, return TRUE, otherwise return FALSE.

   This function has the following declaration and pseudo code:

```
algorithm   Update (Expr, Bot, Top, TmpDDS)
declare     L:          left expr
            R:          right expr
            aop:        arithmetic operator
            rdomain:    right expr's domain with Bot and Top
            ldomain:    left expr's domain with Bot and Top
```

s: subdomain

```
BEGIN
    IF (Expr is Const)
        RETURN TRUE
    END IF
    IF (Expr is Var)
        – domain of Var = < (Bot1:Top1), ... (Botn:Topn) >
        IF (Bot ≤ Bot1 AND Top ≥ Topn)
            RETURN TRUE
        ELSE IF (Bot > Topn OR Top < Bot1)
            RETURN FALSE
        ELSE
            FOR each domain (Boti:Topi) in Var's domain
            – Handle Top first. Processing is done from Topn to Top1
                IF (Top ≥ Topn)
                    BREAK FOR LOOP– Don't bother to check any more.
                                  – Topn is the biggest in the domain.
                ELSE IF (Top ≥ Boti AND Top ≤ Topi)
                    – Remove < (Boti+1:Topi + 1), ... (Botn:Topn) > from Var's d omain.
                    FOR each subdomain s from (Boti + 1:Topi), ... (Botn:Topn)
                      Remove (TmpDDS, Var, s)
                    END FOR
                    – Replace subdomain (Boti:Topi) with (Boti:Top)
                    Replace (TmpDDS, Var, (Boti:Top), (Boti:Topi))
                    BREAK FOR LOOP
                ELSE IF (Top > Topi AND Top < Boti + 1)
                    – Remove < (Boti + 1:Topi + 1), ... (Botn:Topn) > from Var's d omain.
                    FOR each subdomain s from (Boti + 1:Topi), ... (Botn:Topn)
                      Remove (TmpDDS, Var, s)
                    END FOR
                    BREAK FOR LOOP
                END IF
            END FOR
            – Now do Bot
            – Note that the domain of Var might be modified now.
            – domain = < (Bot1:Top1), ... (Botm:Topm) > .
            – m ≤ n
            FOR each subdomain (Boti:Topi) in Var's domain
                IF (Bot ≤ Bot1)
                    RETURN TRUE – Don't bother to check the rest, and
                                – since top part is done, we can return.
                ELSE IF (Bot ≥ Boti AND Bot ≤ Topi)
                    – Replace subdomain (Boti:Topi) with (Bot:Topi)
                    Replace (TmpDDS, Var, (Bot:Topi), (Boti:Topi))
                    – Remove < (Bot1:Top1), ... (Boti − 1:Topi − 1) > from
                    – Var's domain and shift the domain list (the i_th subdomain
                    – becomes the 1st subdomain).
                    FOR each subdomain s from (Bot1:Top1) to (Boti − 1:Topi − 1)
                      Remove (TmpDDS, Var, s)
                    RETURN TRUE
                ELSE IF (Bot > Topi AND (Bot < Boti + 1)
                    – Remove < (Bot1:Top1), ... (Boti:Topi) > from
                    – Var's domain and shift the domain list (the i+1_th
                    – subdomain becomes the 1st subdomain).
```

```
                    FOR each subdomain s from (Bot_1:Top_1) to (Bot_i:Top_i)
                      Remove (TmpDDS, Var, s)
                    RETURN TRUE
                END IF
            END FOR
            – if execution reaches here, the (Bot:Top) is in a gap
            – between subdomains, so TmpDDS does not contain any values
            – between Bot and Top.
            RETURN FALSE
        END IF
END IF – (Expr is Var)

– Now it is the only case left: Expr is expr.
– First modify the domain for Expr.
Bot = Max (Bot, TmpDDS (Expr).Bot)
Top = Min (Top, TmpDDS (Expr).Top)
Replace (TmpDDS, Expr, (Bot:Top), (TmpDDS (Expr).Bot:TmpDDS (Expr).Top))

– Then determine the domains for left and right side exprs of the Expr.
L   = GetLExpr (Expr)
R   = GetRExpr (Expr)
aop = GetAop (Expr)
aop = GetAop (Expr)
CASE (aop)
        WHEN "+":
            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, ...)
            – In our algorithm here, we choose n=0 now.
            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, we choose n=0.
            ldomain.Bot = (Top+1)/(Bot+1) * Bot
            ldomain.Top = Top
            rdomain.Bot = 1
            rdomain.Top = (Top+1)/(Bot+1) * Top
    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
```

3.5 Examples

Two examples are given in this section to show how DDR procedure is used to reduce the input domains and to generate test data. The first example shows a case when input domains can be successfully reduced. The second example illustrates a case when the reduced domain at certain node do not satisfy the later constraints, then the domains have to be re-chosen at previous decision point. Test cases are generated in both examples.

Example 1

Given a program that shows a function mid(x, y, z) which determines the middle value of three given integers x, y and z.

The program is given as follows:

```
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);
}
```

The CFG of this program is shown in Figure 3.

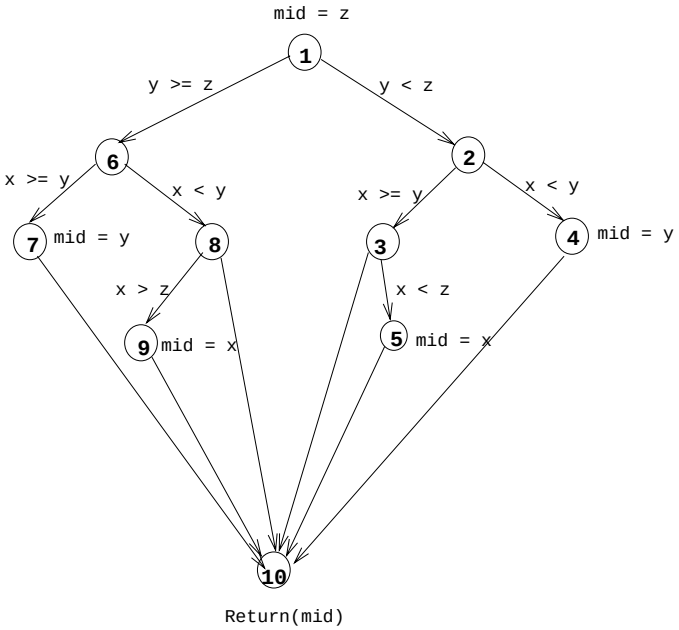

Figure 3: **The Control Flow Graph of** *mid*

Assume that the domains of input variables x, y and z are given as follows:

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

In order to generate test cases for the program, a control path in the CFG needs to be selected. If path 1-2-3-5-10 is chosen in this case, then three predicates encountered in this path are the ones on edge 1-2, 2-3 and 3-5 respectively.

At the start point, the input domains are the given ones.

Next node on the given path is node 2. Since node 1 is a decision node, the constraint on edge 1-2 (y < z) is used to reduce domains of variables y and z. According to function **GetSplit**, it is case 1 where

(ldomain.$_{Bot}$ $\geq$ rdomain.$_{Bot}$) and (ldomain.$_{Top}$ $\leq$ rdomian.$_{Top}$),

SplitPoint = (ldomain.$_{Top}$ - ldomain.$_{Bot}$)*i + ldomain.$_{Bot}$ = (10 - (-10))/2 + (-10) = 0

Therefore the domains of variables y and z are reduced. Domain of y is now < -10 .. 0 > and z < 1 .. 10 >. The reduced domains are then substituted into all remaining constraints that contain the corresponding variables.

After edge 1-2 is traversed, now node 2 is a decision node, constraint on edge 2-3 (x $\geq$ y) is used to reduce domains of variable x and y. It is case 2 in the **GetSplit** function, where

(ldomain.$_{Bot}$ $\leq$ rdomain.$_{Bot}$) and (ldomain.$_{Top}$ $\geq$ rdomain.$_{Top}$) :

SplitPoint = (rdomain.$_{Top}$ - rdomain.$_{Bot}$)*i + rdomain.$_{Bot}$ = (0 - (-10))/2 + (-10) = -5

So the domain for variable x is < -5 .. 10 >, and y is < -10 .. -5 >.

Next, edge 3-5 is to be traversed, since node 3 is a decision node branch 3-5 is on the given path, next constraint to be used is (x < z). Current domains for these two variables fit case 1 in **GetSplit** function, where

(ldomain.$_{Bot}$ $\geq$ rdomain.$_{Bot}$) and (ldomain.$_{Top}$ $\leq$ rdomian.$_{Top}$) :

SplitPoint = (ldomain.$_{Top}$ - ldomain.$_{Bot}$)*i + ldomain.$_{Bot}$= (10- (-5))/2 +(-5) = 2

Domain of x is now < -5 .. 2 >, and z is < 3 .. 10 >.

The domains of each input variable after each constraint has been used are reduced progressively shown as follows:

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

After these three constraints on path 1-2-3-5-10 are used to reduce the domain of the inputs, the input domains finally become: x: -5 .. 2; y: -10 .. -5 ; and z: 3 .. 10.

Test data can be chosen randomly from within these input domains and they should satisfy all the constraints on path 1-2-3-5-10. For instance, one test case is generated by randomly selecting one value for each input variables from their domains: (x=0, y=-10, and z=8); this test case is executed as follows:

Start :	mid = z;	(mid = 8)
edge 1-2 :	y<z;	(-10 < 8)
edge 2-3 :	x≥y;	(0 ≥ -10)
edge 3-5 :	x< z;	(0 < 8)
node 5:	mid = x;	(mid = 0)
node 10:	Return(4).	

From this example, it can be seen that DDR Procedure reduces domains of inputs by traversing paths in the CFG, using one constraint at a time. Then it generates test cases for each from the reduced input domains. Therefore, test cases are generated based on the given path.

Example 2

Given a program *Value* that determines the value of a variable V based on the values of variable A, B and C. The program is as follows:

```
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);
}
```

The CFG of this program is shown in Figure 4.

Assume that domains of the input variables are:

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

Path 1-2-4-6-8 is selected to generate test cases for the program.

Node 1 is a decision node, The constraint associated with branch 1-2 on the given path is (A < B). Domains of variables A and B satisfy case 4 in the **GetSplit** function, where

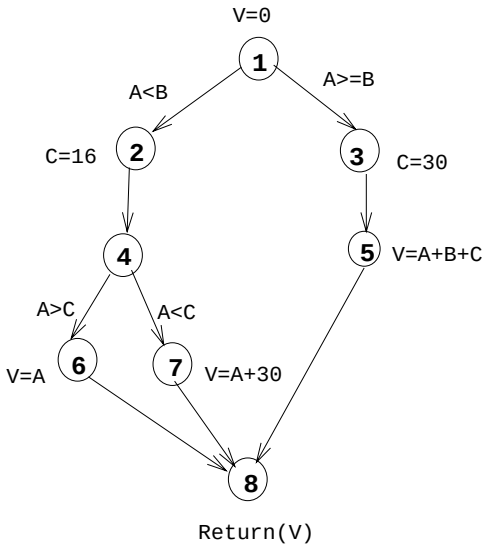

Figure 4: **The Control Flow Graph of** *value*

(rdomain.$_{Bot}$ $\geq$ ldomain.$_{Bot}$) and (rdomain.$_{Top}$ $\geq$ ldomain.$_{Top}$)

so, SplitPoint = (ldomain.$_{Top}$ - rdomain.$_{Bot}$)*i + rdomain.$_{Bot}$ = (20 - 10)* 1/2 + 10 = 15

Domain of A is reduced to $<$ 0 .. 15 $>$, and B is $<$ 15 .. 40 $>$.

Next node in the given path is node 2. It contains a statement C = 16. Therefore, variable C is given a value 16. Its domain is $<$ 16 .. 16 $>$.

Next, the procedure reaches node 4. It is a decision node, Constraint associated with branch 4-6 is (A $>$ C). The current domains of A and C do not satisfy constraint (A $>$ C). This means that the split point found at the previous decision node was not chosen appropriately. A different split point should be calculated to reduce the domains.

Now, back to the most recent decision node $--$ node 1. Apply the same **GetSplit** function except that the value of i is 1/4 this time. so, new SplitPoint is : (ldomain.$_{Top}$ - rdomain.$_{Bot}$)*i + rdomain.$_{Bot}$ = (20 - 10)* 1/4 + 10 = 12

The reduced domains for A and B are $<$ 0 .. 12 $>$ and $<$ 12 .. 40 $>$ respectively. But these domains still do not satisfy constraint (A $>$ C) as the procedure proceeds on the given path. so, another split point needs to be calculated. And the procedure is now back to node 1 again. The value of i is given 3/4 this time.

so, SplitPoint = (ldomain.$_{Top}$ - rdomain.$_{Bot}$)*i + rdomain.$_{Bot}$ = (20 - 10)* 3/4 + 10 = 17

A and B are now reduced to $<$ 0.. 17 $>$ and $<$ 17 .. 40 $>$.

These domains satisfy constraint (A $>$ C) as the procedure reaches branch 4-6. For this con-

straint, domains of A and C satisfy case 2 where (ldomain.$_{Bot}$ $\leq$ rdomain.$_{Bot}$) and (ldomain.$_{Top}$ $\geq$ rdomain.$_{Top}$) :

SplitPoint = (rdomain.$_{Top}$ - rdomain.$_{Bot}$)*i + rdomain.$_{Bot}$ = (16 - 16)*1/2 + 16 = 16

Therefore A domain is now < 17 .. 17 >. C remains value 16.

Finally, the procedure reaches at node 6, where variable v is given value of A and exits at node 8. At this point, the domains of variables A, B and C are: A < 17 .. 17 >, < 17 .. 40 >, and 16 respectively.

The result of each procedure sequence is listed as follows:

	A	B	C
1. Start :	0 .. 20;	10 .. 40;	0 .. 100;
2. A< B:	0 .. 15;	15 .. 40;	0 .. 100;
3. C=16;	0 .. 15;	15 .. 40;	16 .. 16;
4. A> C:	A: constraint is not satisfied		

Go back to decision node 1 where the SplitPoint was chosen:

	A	B	C
2. A< B:	0 .. 12;	12 .. 40;	0 .. 100;
3. C=16;	0 .. 12;	12 .. 40;	16 .. 16;
4. A> C:	constraint is not satisfied.		

Go back the third time to decision node 1 where the SplitPoint was chosen:

	A	B	C
2. A < B:	0 .. 17;	17 .. 40;	0 .. 100;
3. C=16;	0 .. 17;	17 .. 40;	16 .. 16;
4. A> C:	17 .. 17;	17 .. 40;	16 ..16;

Test cases can be generated by randomly choosing one set of data in the domains of the variables. A set of inputs such as (A=17, B=25, c=16) will execute on path 1-2-4-6-8. This test case is executed in the program as follows:

Start : v = 0 ;

edge 1-2 :	if (A<B;	(17 < 25) —- true
node 2 :	C = 16;	
edge 4-6:	if (A<C)	(17 < 16) —- false
node 6:	else V = A;	(V = 17)
node 8:	Return(V);	(return(17)

4 ARRAYS, POINTERS, AND LOOPS

Arrays are handled in the dynamic domain reduction procedure in such a way that each element in an array is treated as a distinct variable. To do this, we use the index of an array as an expression, and it is then used to differentiate each element of the array and perform domain reduction on these elements the same way as the other variables. This is of course limited to the extent that array indexes can be fully analyzed during a static execution of the program. Pointers are also handled as variables when we read pointers as expressions and find their corresponding domains.

Loop handling is done in a novel way by the dynamic domain reduction procedure. There are two general ways to handle loop structures. The first is to discover all possible paths that start from the given start node to the goal node. If there is a loop structure in the CFG, 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 of them do not satisfy the loop constraint and have to be thrown away.

In the dynamic domain reduction procedure, loops are handled more 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 cycle on 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 loop 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.

5 CONCLUSIONS

This paper presents a new method for automatically generating test data. It uses elements from several other test data generation methods and offers novel solutions to problems encountered by previous methods. The dynamic domain reduction procedure is based on the domain reduction procedure, symbolic evaluation, and the dynamic test data generation approach. It integrates constraint satisfaction, symbolic evaluation, and a robust search process into one dynamic process. These advantages mean that the dynamic domain reduction procedure is less likely to fail to find a test case when a test case exists, and that implementations can be more efficient. In this approach,

array indexes and pointers can be calculated symbolically at the same time that values are being found, allowing the test data generation to overcome previous difficulties with arrays and pointers.

The search process uses a novel technique, domain splitting,to make intelligent choices at certain steps in the process.This allows maximum flexibility in the values being chosen, and allows an efficient search procedure (binary search) to be used, which in turn increases the chances for success. This process also allows complicated expressions to be handled uniformly.

Of course, any technique for automated test data generation has inherent limitations. The problems of arrays, loops, and pointers cannot be completely solved. But this technique uses more information about such constructs than previous methods, allowing for more power in the test data generation process.This paper also makes no claims about optimality. Test data generation is an extremely complex problem and we can only hope to find partial solutions that are general and robust enough to work most of the time in the real world. Moreover, this is an ongoing research program and we anticipate future research using this method.

5.1 Future Work

We are currently working on an implementation of the dynamic domain reduction procedure. Algorithms have been designed and software is currently being built. This implementation will be used to compare the dynamic domain reduction procedure with other test data generation procedures, and as a tool for comparing testing criteria such as mutation and data flow.

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, in data flow testing the term infeasible DU-pairs has been used [FW88]. In Pan's thesis [Pan94], a technique for detecting equivalent mutants was presented that is based on recognizing infeasible systems of constraints. Whereas the procedure here 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 dynamic domain reduction procedure to explicitly recognize most infeasible paths.

Most of the work in automated test data generation has been intra-procedural rather than inter-procedural. There is no reason why the technique here could not be applied in an intra-procedural manner. 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 integration and even system testing.

References

[BEL75] 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.

[BF79] 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.

[BKM91] 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.

[Cla76] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, September 1976.

[DGK+88] 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.

[DK78] J. A. Darringer and J. C. King. Applications of symbolic execution to program testing. *IEEE Computer*, 11(4), April 1978.

[DLS78] 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.

[DO91] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[DO93] 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.

[FW88] 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.

[How77] W. E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, 3(4), July 1977.

[Kor90] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[Mau90] Peter M. Maurer. Generating testing data with enhanced context-free grammars. *IEEE Software*, 7(4), July 1990.

[MDL87] H. D. Mills, M. D. Dyer, and R. C. Linger. Cleanroom software engineering. *IEEE Software*, September 1987.

[MM75] 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.

[Mye79] G. Myers. *The Art of Software Testing*. John Wiley and Sons, New York NY, 1979.

[Off91] A. J. Offutt. An integrated automatic test data generation system. *Journal of Systems Integration*, 1(3):391–409, November 1991.

[Pan94] Jie Pan. Using constraints to detect equivalent mutants. Master's thesis, Department of Information and Software Systems Engineering, George Mason University, Fairfax VA, 1994. (Also released as technical report ISSE-TR-94-109).

[RHC76] 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.

[Som92] I. Sommerville. *Software Engineering*. Addison-Wesley Publishing Company Inc., 4th edition, 1992.

[Stu85] H. Sturgis. An effective test strategy. Technical report CSL-85-8, Xerox Parc, November 1985.

[VMM91] J. M. Voas, L. Morell, and K. W. Miller. Predicting where faults can hide from testing. *IEEE Software*, 8(2), March 1991.