Automatic Test Data Generation for Programs with Integer and Float Variables

Nguyen Tran Sy, Yves Deville

Université catholique de Louvain
Place Saint-Barbe 2
B-1348 Louvain-la-Neuve, Belgium
{tsn,yde}@info.ucl.ac.be

Abstract. This paper presents a novel approach for automated test data generation of imperative programs containing *integer*, *boolean* and/or *float* variables. Our approach is based on consistency techniques integrating integer and float variables. We handle statement, branch and path coverage criteria. Our purpose is thus to automatically generate test data that will cause the program to execute a statement, to traverse a branch or to traverse a specified path. For path coverage, the specified path is transformed into a *path constraint* which is solved by an interval-based constraint solving algorithm handling integer, boolean and real variables. A valid test input is then extracted from the interval solutions. For statement (and branch) coverage, a path reaching the specified statement or branch is dynamically constructed. Our algorithm for path coverage is then applied. The search for a suitable path and the solving of path constraints make an extensive use of consistency techniques. We propose here a simple consistency notion, called e-Box consistency, generalizing box consistency to integer and float variables. The eBox consistency is sufficient for our purpose. A prototype has been developed and experimental results show the feasibility of our approach.
Keywords software testing, test data generation, constraint satisfaction, consistency.

1 Introduction

1.1 Test data generation

Software verification is an expensive and difficult task, accounting for up to 50% of the cost of software development [KC00] and even more in critical systems. The objective of software testing is to detect faults in the program [DO93] and therefore provide more assurance for the quality of the software. If software testing phase could be automated, the cost of software development would be significantly reduced. As it is generally impossible to test the entire input domain of the program, structural test *coverage criteria* are used to identify a set of program elements and a *test input* is generated for each element from this set.

One usually distinguishes *control flow* criteria from *dataflow* criteria. Dataflow criteria are dealing with the dataflow dependencies in the program execution while control flow criteria are dealing with the control flow of the program execution.

In this paper, we are concerned with structural test data generation with control flow criteria. The control flow of a program is usually represented by a *control flow graph*, where the nodes are either a decision node or a block of instructions without decision statements. The edges represent the possible control flow between the nodes. *Statement coverage* requires exercising a given set of program statements (a set of nodes). The problem is thus to find, for each program statement, a program input on which this statement is executed. *Branch coverage* is the dual version where an input must be found, such that the execution traverses a specified edge of the control flow graph associated to the program. *Path coverage* is a generalization of branch coverage where the input causes the execution of a specified path (from the start to some statement). Structural testing thus includes :

- choice of a criteria (statement, branch or path),
- identification of a set of statements, branches or paths,
- generation of test data for each element of this set

The automation of the last phase is a vital challenge in software testing.

Among the difficulties in the generation of test data is the presence in the program of arrays, pointers, unstructured control statements (such as goto, break), and floating-point variables. In this paper, we specifically handle test data generation of *programs with both integer and float variables*, for path, branch and statement coverage criteria.

To generate test data, there exists three main approaches: *random*, *path-oriented*, and *goal-oriented* test data generation. *Random* test data generation [BM83,DN84] consist in trying test data generated randomly until the selected statement is reached. This approach is not appropriate for programs with float variables as the search space is large.

Path-oriented approaches [OJP97,GMS98] attack the problem by first selecting a set of paths that covers all the statements satisfying a given criterion and then generating a test input which executes each selected path. These approaches include, among others, *symbolic evaluation* and *program execution based* test data generation. Symbolic evaluation [Kin76] consists in replacing input variables by symbolic values. It then derives path constraints over these symbolic values, forming a constraint system describing conditions under which a path is traversed. The constraints are then symbolically solved. Although symbolic evaluation is promising, it still has several weaknesses including the handling of arrays, indeterminate loops, dynamic data structures and the size of symbolic expressions. In the *program execution based* approach [GMS98,GMS99], a first test data is initiated with a (randomly) chosen input. This input is then iteratively refined, by execution of the program, to obtain a final input executing the path. This approach exploits its dynamic nature to overcome some limitations of the approaches based on symbolic evaluation. However the number of iterations required before the finding of a final input depends much on the complexity of the constraints on a path. Moreover if the path is infeasible and the associated constraints nonlinear, this approach becomes difficult to apply.

In *goal-oriented* approaches, the generation of test data to execute the selected statement is carried out irrespectively of the executed path. The chaining approach presented in [FK96], a program execution based approach, starts by executing the program for an arbitrary test input. If an undesirable execution flow is observed at some branch in the program, then function minimization search algorithms are used to find a new input that will change the execution flow at this branch. Although the approach succeeds in handling arrays and dynamic data structures, it may require a great number of executions of the program. Other approaches, such as [PHP99], use genetic algorithms to guide the search process. In [Got00,GBR98,GBR00], a constraint-based approach is provided. It consists in translating the test data generation problem for a given statement into a set of constraints and solving it by an instance of the CLP scheme. This approach offers advantages such as the handling of arrays, loops and a restricted class of pointers. It is however limited to integer variables.

1.2 Results and approach

We propose a new method for test data generation of *imperative programs with integer, boolean and float variables*. Statement, branch and path coverage criteria are all handled. We however focus on imperative programs without arrays and/or dynamic data structures.

Problem statement *Given a statement s, a branch b or a path p of a program P (with integer, boolean and float variables), generate a test input i such that P when executed on i will cause s, b or p to be traversed.*

To solve this problem, we use a constraint solving approach based on a simple consistency notion, generalizing Box-consistency [HMD97] to integer, float and boolean variables. Path coverage criteria is the basic bloc of our method. For branch and statement criteria, paths reaching the specified branch or statement are dynamically constructed using consistency techniques, and the path coverage method is applied on these paths to find suitable test input.

In our approach, the program under analysis is first translated into a static single assignment (SSA) form. The search for a test input traversing a specified path of a program P can be summarized as follows :

– A path constraint on the input variables is derived from the specified path and the program in SSA form. Such a constraint involves integer, boolean and float variables and can be non linear.
– The path constraint is solved by an interval-based constraint solving algorithm handling integer, boolean and real variables.
– A test input is extracted from the interval solutions.

It is important to precise the specificities of solving a path constraint compared to classical interval-based constraint solving. First, a path constraint is under-constrained; there usually exists many test inputs traversing the specified path while we are interested by finding one of them. Existing systems, such as Numerica [HMD97] are not always appropriate for under-constrained systems as they try to generate all the solutions. Second, a path constraint involves both integer, boolean and *float* variables. There exists systems combining solvers. For instance, Prolog IV [BT95] and CLP(BNR) [BOV95] handle integer, boolean and *real* variables. Solving a path constraint with such solvers will produce a (small) interval containing the mathematical solution of the path constraint. Such a mathematical solution may involve real values which are not float numbers. Moreover, even if the mathematical solution only involves float numbers, this mathematical solution as test input is not guaranteed to traverse the specified path as the path constraint is executed using the programming language float operators, which are not sound. In practice however, a valid test input can usually be extracted as the path constraint is under-constrained. These differences make that existing constraint solving approaches cannot be used solely to generate test data for programs with integer, boolean and real variables.

Contribution The main contribution of this paper is a novel approach, based on consistency techniques, for automated test data generation of imperative programs containing *integer*, *boolean* and/or *float* variables. This approach handles *branch*, *statement* and *path* coverage criteria. We also propose a method, supported by a prototype, proving that this approach is feasible. This method uses a simple consistency notion generalizing box consistency to integer and float variables.

The other known method based on consistency [GBR00] is limited to integer variables (but handles arrays and a restrictied class of pointers). Compared to other approaches handling integer and real variables in the literature (e.g. [GMS99,PHP99]), our approach can be seen as an alternative or as a complement. Our consistency method could be combined with random or dynamic approaches when searching a test data exercising a specified statement of the program.

Organization The organization of this paper is as follows. The necessary background is presented in the next section. Section 3 gives an informal explanation of static single assignment (SSA) form needed in our approach. Generation of the path constraint for a given path is described in section 4. Section 5 proposes a simple constraint solving algorithm for test data generation, for the path coverage criteria. Section 6 describe an algorithm for branch and statement coverage criteria. A prototype implementation of the approach and experimental results are discussed in section 7. Conclusions are finally presented in section 8.

2 Background

Control flow graph A *control flow graph* (CFG) of a program P [Got00] is a directed graph G = (N, E, START, STOP) that represents the control structure of the program, where N is a set of nodes (each node is either a basic block or a decision node), E is a set of edges (each edge represents a possible control flow from one node to another), START is a unique entry node and STOP is a unique exit node of the program. Note that in this paper, we will talk about programs in terms of their CFGs. A *basic block* is a maximum sequence of statements without decision statements, i.e. it has only one entry point and one exit point. A *decision* is a point in the program where control flow can diverge. Edges from decision nodes are labelled with a *condition*. A *path* is a sequence of nodes from the entry node to another node of G. Examples of CFG are provided in the next sections.

Path constraint An (integer, boolean or float) *input variable* is either an input parameter or a variable (integer, boolean or float) in an input statement of P. The domain of a boolean variable is the set {0,1}, the domain of an integer variable is a set of consecutive integers, and the domain of a float variable is an interval of float numbers. Let $x_1, \ldots, x_n$ be n input variables of P, D_k is the domain of variable x_k $(1 \leq k \leq n)$ then a *test input* is a vector of values $(i_1, \ldots, i_n)$, where $i_k \in D_k \quad (1 \leq k \leq n)$.

A *basic constraint* is a simple relational expression of the form E_1 op E_2 or $NOT(E_1\ op\ E_2)$, where E_1 and E_2 are arithmetic expressions and op is one of the following relational operators $\{<, \leq, >, \geq, =, \neq\}$. A *constraint* is a basic constraint or a logical expression on basic constraints using the following logical operators $\{AND, OR, NOT\}$. A *conjunctive constraint* is a list of basic constraints connected by the logical AND. A *predicate* is a constraint associated with a decision node.

Note that a path can be represented by a list of constraints with one constraint for each predicate on the path. This list of constraints is called a *path constraint* where the constraints of the list are connected by the logical AND. The constraints on the path can always be expressed in terms of input variables because they are initially the constraints on program variables and those variables depend on input variables with assignment statements along the path.

CSP and Consistency Many important problems in areas like artificial intelligence and operations research can be viewed as constraint satisfaction problems (CSP). A CSP $P = (V, D, \mathcal{C})$ is defined by a finite set of variables V taking values from finite or continuous domains D and a set of constraints $\mathcal{C}$ between these variables. A solution to a CSP is an assignment of values to variables satisfying all constraints and the problem amounts to finding one or all solutions. Most problems in this class are $\mathcal{NP}$-complete, which means that backtracking search is an important technique in their solution.

Consistency techniques are constraint algorithms that reduce the search space by removing, from the domains and constraints, values that cannot appear in a solution. Consistency algorithms play an important role in the resolution of CSP [Tsa94].

Interval programming Interval methods aims at solving (continuous) constraints over the real numbers. The basic idea is to associate with each variable an interval representing its domain. Consistency techniques has been designed on continuous domains to reduce the size of the intervals without removing solutions of the constraints. Such consistency techniques are usually coupled in methods for solving such constraints [HMD97].

This paper uses rather standard notations of interval programming. The set of floating-point numbers (or F-numbers) is denoted F. The set of intervals is denoted by I. Capital letters denote intervals. Constraints involve reals and integers. Interval extension of a constraint is an important notion in interval programming. An interval extension of a constraint $c(x)$ is an interval constraint (i.e. a constraint on interval) $C(X)$ such that for all interval X, $(\exists x \in X : c(x)) \Rightarrow C(X)$. If a is a F-number, a^+ denotes the smallest F-number stricly greater than a and a^- the largest F-number stricly smaller than a. The lower and upper bounds of an interval X are denoted respectively by $left(X)$ and $right(X)$. The center of an interval X is a F-number denoted by $center(X)$. Note that if $X = [l, r]$ then $left(X) = l$, $right(X) = r$ and $center(X) = ((l + r)/2)^-$. If x is a real number, $\lfloor x \rfloor$ denotes the largest integer that is not larger than x and $\lceil x \rceil$ the smallest integer that is not smaller than x. Boldface letters denote vector of objects.

A *canonical interval* is an interval of the form $[a, a]$ or $[a, a^+]$, where a is a F-number. An interval X is an $\epsilon\_interval$ ($\epsilon > 0$) if X is canonical or $right(X) - left(X) \leq \epsilon$. A box $(X_1, \ldots, X_n)$ is an ϵ_box if X_i ($1 \leq i \leq n$) is an $\epsilon\_interval$ [HMD97].

Test cases The execution of the program (on the specified path) uses operators defined on F-number, integers and boolean. We assume here that the program under analysis is written is some fixed imperative language $\mathcal{L}$. It is therefore important to distinguish mathematical solutions of a path constraint from test case solutions of the path constraints.

Definition 1. *Let c be a path constraint, and $\mathbf{v}$ be F-numbers. The predicate* eval$(c, \mathbf{v})$ *holds if execution of c with $\mathbf{v}$ using the operators of the programming language $\mathcal{L}$ yields true.*

Definition 2. *A path constraint c is said to be* adequate *wrt its associated path p if for all test input $\mathbf{v}$, $eval(c, \mathbf{v})$ holds iff the execution of the program traverses the path p.*

Definition 3. *Given a path p, a* test case *for p is a (vector of) F-number(s) $\mathbf{v}$ such that $eval(c, \mathbf{v})$, where c is the path constraint associated with p.*

Definition 4. *Given a statement s, a (vector of) F-number(s) $\mathbf{v}$ is a* test case *for s if there exists a path p such that $\mathbf{v}$ is a test case for p.*

If path constraints are adequate wrt their associated paths, a test case is thus a test input traversing the specified path or reaching the specified statement. When no test case exists, the path is said to be *infeasible*.

The predicate *eval*$(c, \mathbf{v})$ can be realized in different ways. First, the program under analysis can be slightly modified such that when executed, if the execution traverses the specified path, the result is true, otherwise it is false. Second, a new program can be constructed in the language $\mathcal{L}$. This program contains all the constraints in c, each constraint being encapsulated in a assignment statement. The output is true if all the boolean expressions are evaluated to true.

As introduced in the preceding section, a float solution $\mathbf{v}$ of a path constraint may not traverse the specified path, i.e. $c(\mathbf{v}) \not\Rightarrow eval(c, \mathbf{v})$. Consider for instance the simple constraint, where x is real.

$$c(x) \triangleq x = \frac{x}{3} + \frac{x}{3} + \frac{x}{3}$$

It is clear that $c(x)$ is mathematically true for all floating-point number in F. However $eval(c, 1)$ may evaluate to false in some programming languages.

It is interesting to notice the duality between test data generation and constraint solving. In constraint solving (on reals), one is interested by mathematical solutions; hence the approximations of the float operators must be overcome and are handled through interval arithmetics. In test data generation, one is interested in the float operators on the specified path. A test data must fit these float operations with their approximation; not their mathematical counterpart. However, constraint solving is a useful framework to solve test data generation as it allows, in the solving step, to concentrate on the search of mathematical solutions, leaving the adequacy to the float operations in another step.

3 Static Single Assignment Form

The initial step of our approach is the transformation of the program into a SSA form. SSA form [AWZ98] is an equivalent intermediate representation of a program. It has been exploited effectively for optimization problems in many fields of computer science. The main properties of SSA form are:

- There is only one assignment to each variable in the entire program.
- Each use of a variable refers to only one assignment.

Thanks to this form, we can reason very easily about variables because if two variables have the same name, then they contain the same value wherever they occur in the program. An example of a simple sequence of assignments and its corresponding SSA form is presented in Example 3.1.

```
x = 0;                  x1 = 0;
y = x + 1;              y1 = x1 + 1;
x = x + y;              x2 = x1 + y1;
y = x + y;              y2 = x2 + y1;
```

Example 3.1: A simple program in normal form and its corresponding SSA form

Note that we use subscripted variables in order to make variable names unique and call these subscripted variables *values* of the original variables. More complicated statements, such as IF and WHILE statements, contain branches and *join nodes*. At a join node, several assignments to a variable along different branches can reach the node. In such a case multiple values of a variable can reach the same node and these values have to be grouped into one single value in order to reach further uses of this variable. This can be achieved by a special assignment with Φ-function on the right-hand side. Φ-functions have as many arguments as the number of branches into the join node and it returns the i-th argument if control reaches the join node by i-th branch. An example of IF and WHILE statements is illustrated in Example 3.2. Note that for a WHILE statement, Φ-functions are introduced before the condition of the WHILE statement to mean that they must be executed before this condition at every iteration. In the rest of the paper, for the sake of simplicity, a list of Φ-assignments is sometimes written as a single Φ-assignment:
$\mathbf{v}_2 = \Phi(\mathbf{v}_0, \mathbf{v}_1) \Longleftrightarrow x_2 = \Phi(x_0, x_1), \ldots, z_2 = \Phi(z_0, z_1)$

More details on SSA form can be found in [BM94,AWZ98].

```
x = 4;                  x1 = 4;
if x > 0 then            if (x1 > 0) then
  x = 5;                  x2 = 5;
  y = 1;                  y1 = 1;
else                     else
  y = 2;                  y2 = 2;
                         x3 = Φ(x2, x1);
                         y3 = Φ(y1, y2);

i=0;                    i1 = 0;
while i < 5 do           while (i3 := Φ(i1, i2); i3 < 5) do
  i = i + 1;              i2 = i3 + 1;
x = i;                  x1 = i3;
```

Example 3.2: SSA form of IF,WHILE statements

4 Generation of Path Constraints

Given a path of the program, we have to construct an adequate path constraint. A path is a sequence of nodes of the control flow graph.

Definition 5. *If P is a program and p a path in the CFG associated to the SSA form of this program, $\mathcal{T}(P, p)$ denotes the path constraint of path p in program P.*

The construction of a path constraint $\mathcal{T}(P, p)$ is defined inductively on the structure of the instructions on the path.

Variable declaration Integer variables are finite domain variables on some initial domain. Boolean variables are finite domain variables with initial domain {0,1}. Float variables are real variables with some initial interval. The initial domains and intervals may depend on the programming language $\mathcal{L}$, but can also be fixed by the user.

Assignment Let p be a single assignment *x = Expr*. In this case, $\mathcal{T}(P, p)$ is $x = Expr$

Condition Let p be a path traversing a conditional instruction

$$\textit{IF bexp THEN Inst1 ELSE Inst2 ;}\ \ \mathbf{v}_2 = \Phi(\mathbf{v}_0, \mathbf{v}_1)$$

If p traverses *Instr1*, $\mathcal{T}(P, p)$ is $bexp \wedge \mathcal{T}(Inst1, p) \wedge \mathbf{v}_2 = \mathbf{v}_0$
If p traverses *Instr2*, $\mathcal{T}(P, p)$ is $\neg bexp \wedge \mathcal{T}(Inst2, p) \wedge \mathbf{v}_2 = \mathbf{v}_1$

Path without Loop Let p be a path $p_1, \ldots, p_n$ not traversing a loop. In this case, $\mathcal{T}(P, p)$ is $\bigwedge_{1 \leq i \leq n} \mathcal{T}(P, p_i)$

Path with Loops Let p be a path $p_1, \ldots, p_n$ traversing a loop of the form

$$\textit{while } (\mathbf{v}_2 := \Phi(\mathbf{v}_0, \mathbf{v}_1);\ c) \textit{ do Inst}$$

where $\mathbf{v}_0$ is the vector of variables defined before the loop, $\mathbf{v}_1$ is the vector of variables defined inside the body of the loop, $\mathbf{v}_2$ is the vector of variables referred to inside and outside of the loop.

The path p traverses m times the body of the loop ($m \geq 0$). Hence the path p has the form

$$p_1, \ldots, p_{i_1}, \ldots, p_{i_2}, \ldots, p_{i_m}, \ldots, p_{i_{m+1}}, \ldots, p_n$$

where P_{i_k} are the passages in the loop control node ($i_k < i_{k+1}, 1 \leq k \leq m$)

In this case, $\mathcal{T}(P, p)$ is

$$\begin{array}{l}
\bigwedge_{1 \leq j < i_1} \mathcal{T}(P, p_j) \wedge \mathbf{v}_0 = \mathbf{w}_0 \\
\bigwedge_{1 \leq k \leq m} (c[\mathbf{v}_2/\mathbf{w}_{k-1}] \wedge \mathcal{T}(Inst[\mathbf{v}_2/\mathbf{w}_{k-1}, \mathbf{v}_1/\mathbf{w}_k], p_{i_k+1} \ldots p_{i_{k+1}-1})) \wedge \neg c[\mathbf{v}_2/\mathbf{w}_m] \\
\wedge \mathbf{v}_2 = \mathbf{w}_m \bigwedge_{i_{m+1} < j \leq n} \mathcal{T}(P, p_j)
\end{array}$$

It is easy to adapt this transformation for path ending in the body of the loop.

Example A simple program [GMS98] and its SSA form are given in Figure 4.1. Its associated CFG is depicted in Figure 4.2, where the path 1-3-4-6-7-9-10, represented in dashed lines. The corresponding path constraint is non linear and is the following

u_1 = (x_0-y_0)*2 $\wedge$ $\neg$ $x_0 > y_0$ $\wedge$ $w_2 = y_0$ $\wedge$ $w_3 = w_2$ $\wedge$ $\neg$(w_3+z_0 $>$ 100 $\wedge$ $x_0^2 + y_0^2 \geq 100$ $\wedge$ y_2 = x_0*z_0+1 $\wedge$ $x_3 = x_0$ $\wedge$ $y_3 = y_2$ $\wedge$ $u_1 \leq 0$ $\wedge$ y_3-sin(z_0) > 0

```
   program                            SSA form

float x, z                         float x0 , z0
int y                              int y0

read(x, y, z)                      read(x0, y0, z0)
u = (x - y)*2                      u1 = (x0 - y0)*2
if (x > y)                         if (x0 > y0)
  w = u                              w1 = u1
else                               else
  w = y                              w2 = y0
endif                              endif
                                   w3 = Φ(w1, w2)
if (w + z > 100)                   if (w3 + z0 > 100)
  x = x - 2                          x1 = x0 - 2
  y = y + w;                         y1 = y0 + w3
  write(‘‘Linear’’)                  write(‘‘Linear’’)
elseif (x^2 + y^2 ≥ 100)            elseif (x0^2 + y0^2 ≥ 100)
  y = x*z + 1                        y2 = x0 * z0 + 1
  write(‘‘Nonlinear:Quadratic’’)     write(‘‘Nonlinear:Quadratic’’)
endif                              endif
                                   x3 = Φ(x1, x0, x0)
                                   y3 = Φ(y1, y2, y0)
if (u > 0)                         if (u1 > 0)
  write(u)                           write(u1)
elseif (y - sin(z) > 0)            elseif (y3 - sin(z0) > 0)
  write(‘‘Nonlinear:Sine’’)          write(‘‘Nonlinear:Sine’’)
endif                              endif
```

Figure 4.1: Program 1 and its SSA form

Other examples of path constraints will be proposed in Section 7. From the above description, one can easily show that the generated path constraints are adequate.

Theorem 1. *Given a path p, the generated path constraint is adequate wrt p.*

5 Test Data Generation: Path Coverage

In this section, we describe a constraint solving algorithm for test data generation, under the path coverage criteria. We first define a local consistency, called *eBox consistency* that is simple enough to solve our test data generation problem. Of course, more sophisticated consistencies such as used in Prolog IV, CLP(BNR) or Numerica can be used.

Consistency The eBox consistency is an extension of the classical box consistency [HMD97] in order to handle both real and integer variables. The objective is to reduce the domains of the variables (i.e. their interval) without removing solutions.

Definition 6 (eBox consistency). *Let $P = (V, D, \mathcal{C})$ be a CSP where $V = (x_1, \ldots, x_n)$, a set of (real and integer) variables; $D = (X_1, \ldots, X_n)$ with $X_i = [l_i, r_i]$ the domain of x_i $(1 \leq i \leq n)$; $\mathcal{C} = (c_1, \ldots, c_m)$, a set of constraints defined on $x_1, \ldots, x_n$ and $c \in \mathcal{C}$ be a k-ary constraint on the variables $(x_1, \ldots, x_k)$.*

The constraint c is eBox-consistent *in D if for all x_i $(1 \leq i \leq k)$*

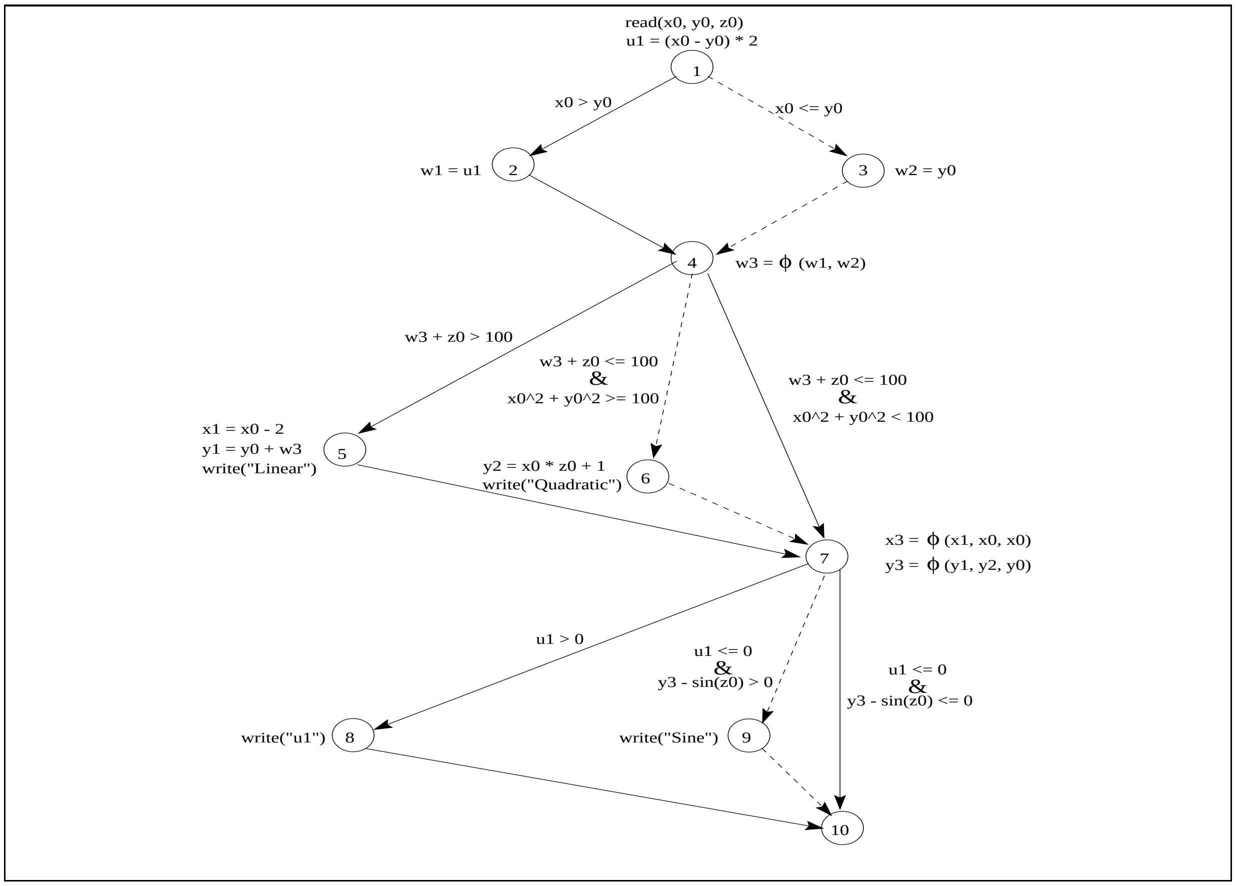

Figure 4.2: CFG of Program 1

– *if x_i is a real variable then*
$C(X_1,\ldots,X_{i-1},[l_i,l_i^+],X_{i+1},\ldots,X_k) \bigwedge C(X_1,\ldots,X_{i-1},[r_i^-,r_i],X_{i+1},\ldots,X_k)$ *when* $l_i \neq r_i$
or $C(X_1,\ldots,X_{i-1},[l_i,r_i],X_{i+1},\ldots,X_k)$ *when* $l_i = r_i$

– *if x_i is a integer variable then*
$C(X_1,\ldots,X_{i-1},[l_i,l_i],X_{i+1},\ldots,X_k) \bigwedge C(X_1,\ldots,X_{i-1},[r_i,r_i],X_{i+1},\ldots,X_k)$ *when* $l_i \neq r_i$
or $C(X_1,\ldots,X_{i-1},[l_i,r_i],X_{i+1},\ldots,X_k)$ *when* $l_i = r_i$

where C is an interval extension of constraint c.

The CSP P is eBox-consistent *in D if for all $c \in \mathcal{C}$, c is eBox-consistent in D.*

The purpose of filtering is to reduce as much as possible the domains of variables without removing any solution from the initial domains.

Definition 7 (Filtering by eBox consistency). *Filtering by eBox consistency of a CSP $P = (V, D, \mathcal{C})$ is a CSP $P' = (V, D', \mathcal{C})$ such that :*

1. *$D' \subseteq D$*
2. *P and P' have the same solutions*
3. *P' is eBox-consistent in D*

We denote $\Phi_{eBox}(P)$, the filtering by eBox consistency of P. Note that the filtering by eBox consistency of a CSP, by its definition, always exists and is unique. An incremental filtering algorithm implementing eBox consistency can easily be constructed. This is an extension of [HMD97,Del00] to handle integer and float variables. It consists in applying a filtering operation on each pair <*constraint, variable*> incrementally until an eBox-consistent CSP is obtained. The filtering operation on each pair <*constraint, variable*> in turn is carried out by 2 functions `LeftNarrow, RightNarrow` for finding the leftmost and rightmost zero canonical intervals, that is intervals $L \subseteq X$ and $R \subseteq X$ such that $C_x(L) \bigwedge C_x(R)$. A filtering algorithm is presented in Figure 5.1.

Algorithm 5.1 Filtering by eBox consistency

```
function eBoxFiltering($V$ : $Variables$, $D$ : $I^n$, $C$ : $Constraints$) : $I^n$;
% PRE
%   $V$ a set of variables
%   $D$ a box of their corresponding domains
%   $C$ : a set of constraints over $V$
% POST
%   Return a box $D_{res}$ such that $(V,\ D_{res},\ C)\ =\ \Phi_{eBox}(V,\ D,\ C)$
begin
  queue := C;
  while queue $\neq$ $\emptyset$ do
    c := pop queue; {Suppose c is a constraint over $x_1,\ldots,x_k$}
    updatedDomVars := $\emptyset$; {Set of vars having domains updated after this iteration}
    for $x_i$ $\in$ var(c) do
      $C_x$:= C($X_1,\ldots,X_{i-1},X,X_{i+1},\ldots,X_k$);{univariate interval constraint on $X$ of c}
      $right(X_i')$ := $right$(RightNarrow($C_x$, $X_i$));
      $left(X_i')$ := $left$(LeftNarrow($C_x$, $X_i$));
      if $X_i$ $\neq$ $X_i'$ then
        $X_i$ := $X_i'$
        if $X_i'$ = $\emptyset$ then return $D$
        updatedDomVars := updatedDomVars $\bigcup$ {$x_i$}
    queue := queue $\bigcup$ {c' $\in$ C $|$ updatedDomVars $\bigcap$ var(c') $\neq$ $\emptyset$}
  return $D$
end
```

Algorithm An algorithm for test data generation with the path coverage criteria is given in Algorithm 5.2. It is based on a constraint solving algorithm for path constraint. This constraint algorithm extends existing algorithms [HMD97,Got00] in two ways. First, it handles both integer and real variables. Second, it includes a search of a test case in a resulting ϵ_box, as specified hereafter.

Specification 51 (FindSolution) Let $\mathcal{C}$ be a set of constraints, e be an ϵ_box and TS be a representative set of floating-point vectors in e. The function `FindSolution`$(\mathcal{C}, e)$ returns, if it exists, some vector $v \in TS$ such that $\forall c \in \mathcal{C}, eval(c, v)$ holds. Otherwise it returns $\emptyset$.

The `SolvePathConstraints` algorithm first applies a filtering operation with eBox consistency on the initial box. If the resulting box (D_{temp}) is empty, i.e. one of its components is empty, then there is no solution. If D_{temp} is an ϵ_box then the function `FindSolution` is called in order to find a test data in D_{temp}. Otherwise, three subproblems are derived and a solution to the initial problem is a solution to one of these subproblems.

As a path constraint is usually under-constrained, there will be several boxes containing the mathematical solutions. The constraint solving algorithm will search these boxes until it finds one where `FindSolution` returns a test case.

Although not strictly necessary, testing the middle point turns out to be practical for test data generation thanks to the under-constrained nature of path constraints.

Our constraint solving algorithm for test data generation is sound but not complete. Although all the mathematical solutions will be found, there might exist test cases (i.e. float numbers) traversing the specified path, although this float number is not within some ϵ_box containing a mathematical solution. It is known that a solver integrating integers and reals may loose its ability to prove the existence of solution in the produced boxes. This is not a limitation in the context of test data generation as a solution box is not guaranteed to contain a test case. If our constraint solving algorithm does not find test data, it is a clue that the path could be infeasible.

6 Test Data Generation: Branch and Statement Coverage

We now extend the results of the previous section to branch and statement coverage. As a branch is dual to a statement in the control flow graph, it is sufficient to concentrate on statements. All the following algorithms can easily be adapted for branch coverage. Given a statement, the different paths reaching that statement will be dynamically generated, as well as the corresponding test data. The search will be guided by a Control Dependency Graph, and the search will be pruned by our eBox consistency filter.

Algorithm 5.2 Algorithm to generate test data: path coverage

```
function TestDataGenerationPC ($P$ : $Program$, $p$ : $Path$, $D$ : $Box$) : $F^n$;
% PRE
%   $P$ The program under test
%   $p$ a path
%   $D$ The initial box
% POST
%   Returns a test case on which the path $p$ is executed
begin
  $PC$:= $\mathcal{T}(P, p)$;
  $V$ := Set of input variables of $P$
  return SolvePathConstraints($V$, $V$, $D$, $PC$)
end

function SolvePathConstraints($V, V'$ : $Variables$, $D$ : $I^n$, $C$ : $Constraints$) : $F^n$;
% PRE
%   $V$ : input variables of a program P
%   $V$, $V'$ : sets of variables with $V' \subseteq V$
%   $D$ : a box representing the domains of the variables in $V$
%   $C$ : a path constraint of a path $p$ in the program P
% POST
%   Return some vector $v \in D$ such that $v$ is a test case for path $p$
%   Otherwise it returns $\emptyset$
begin
  ($V$, $D_{temp}$, $C$) := $\Phi_{eBox}(V, D, C)$;
  if $D_{temp}$ is $\emptyset$ then return $\emptyset$;
  else
    if $D_{temp}$ is an $\epsilon\_box$ then return FindSolution($C$, $D_{temp}$);
    else
      if $V'$ is not empty then
        Choose arbitrarily a variable $x$ in $V'$;
        $m$ := $(left(X_{temp}) + right(X_{temp}))/2$;
        if $x$ is an integer variable then
          $ms$ := SolvePathConstraints($V$, $V' \setminus \{x\}$, $D_{temp}[X_{temp}/[\lfloor m \rfloor, \lfloor m \rfloor]]$, $C$);
        else
          $ms$ := SolvePathConstraints($V$, $V' \setminus \{x\}$, $D_{temp}[X_{temp}/[m, m]]$, $C$);
        if $ms \neq \emptyset$ then return $ms$
        if $x$ is an integer variable then
          $ls$ := SolvePathConstraints($V$, $V' \setminus \{x\}$, $D_{temp}[X_{temp}/[left(X_{temp}), \lfloor m \rfloor - 1]]$, $C$);
        else
          $ls$ := SolvePathConstraints($V$, $V' \setminus \{x\}$, $D_{temp}[X_{temp}/[left(X_{temp}), m]]$, $C$);
        if $ls \neq \emptyset$ then return $ls$
        if $x$ is an integer variable then
          $rs$ := SolvePathConstraints($V$, $V' \setminus \{x\}$, $D_{temp}[X_{temp}/[\lfloor m \rfloor + 1, right(X_{temp})]]$, $C$);
        else
          $rs$ := SolvePathConstraints($V$, $V' \setminus \{x\}$, $D_{temp}[X_{temp}/[m, right(X_{temp})]]$, $C$);
        if $rs \neq \emptyset$ then return $rs$ else return $\emptyset$
      else return SolvePathConstraints($V$, $V$, $D_{temp}$, $C$);
end
```

Control Dependence Graph Intuitively, a node a is linked to a node b in the control dependence graph if any execution path reaching b contains also a. In other words, reaching statement a is a necessary condition to reach statement b. Technically, control dependence is defined in terms of a CFG and the post-dominance relation among the nodes in the CFG [FOW87].

Definition 8. *A node V is* post-dominated *by a node W in G if every directed path from V to STOP (not including V) contains W.*

Definition 9. *A node Y is* control dependent *on node X iff*

1. *there exists a directed path P from X to Y with all Z in P (excluding X and Y) post-dominated by Y, and*
2. *X is not post-dominated by Y.*

Note that if Y is control dependent on X then node X must have at least 2 exits. Following one of the exits from X results in Y being executed while taking others may result in Y not being executed.

Definition 10 (Control Dependence Graph). *A* control dependence graph *(CDG) of a program P is an acyclic directed graph where the nodes are the nodes of the CFG of P, augmented with a special node ENTRY representing whatever conditions will cause the program to be executed. The edges represent the control dependencies between nodes. Edges may be labelled with conditions. An edge (X, Y) in a CDG means that Y is control dependent on X.*

Examples of CFG and CDG are given in Figure 6.2. They refer to the `nThRootBisect` program depicted in Figure 6.1. For simplicity, the SSA form is not shown here. In the CFG and CDG, the labels `T2`, `T4` and `T6` are respectively $(\mathrm{h}-\mathrm{l})^2 \geq \mathrm{e}$, `f(c)=0` and `f(l)*f(c)<0`. The labels `F2`, `F4` and `F6` are respectively ¬`T2`, ¬`T4` and ¬`T6`.

```
float nThRootBisect(float a, int n, float e)) {
PRE
    f(x) ≜  x^n - a
    a > 1, n > 1, e > 0
POST
    Let r > 0 such that r^n = a
    Return r* with (r* − r)^2  <  e

    float l, h, c;
1.  l = 1; h = a;
2.  while ((h − l)^2  ≥  e) {
3.    c = (l + h)/2;
4.    if (f(c) = 0)
5.      return c;
6.    if (f(l)*f(c) < 0)
7.      h = c;
8.    else l = c;
    }
9.  return h;
}
```

Figure 6.1: Program nThRootBisect: computing the n-th root of a number by the bissection method

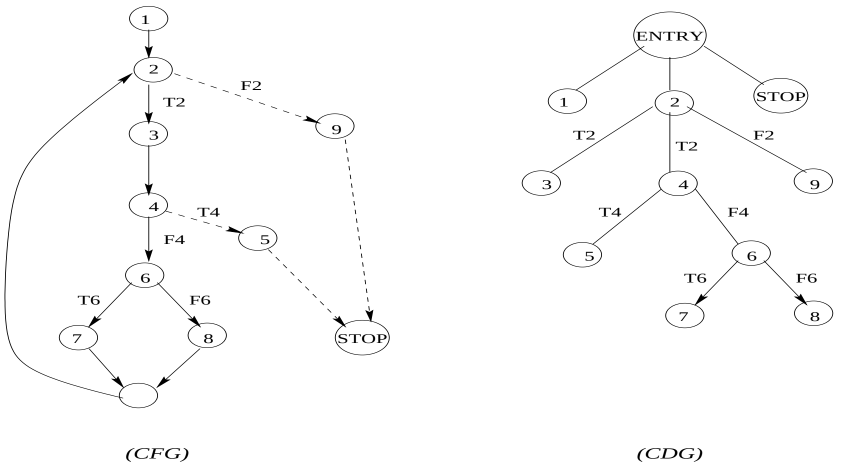

Figure 6.2: CFG and CDG of the nThRootBisect program

Definition 11 (Decision chain [Got00]). *Given a CDG G and a node n, the* decision chain *of node n in G is the sequence of nodes-labels (with label) from the entry to node n (the entry node and the node n are not in the sequence).*

For example, <2-T1, 3-F2> constitutes a decision chain for node 5 in the CDG of Figure 6.2. By definition of a CDG, the decision chain of a node n is unique (assuming programs without *goto* and *break* statements).

Definition 12 (Reachability graph [GMS00]). *The* reachability graph *for a node n in a CDG G is the smallest subgraph of G containing all the paths from the start node to node n.*

The construction of the reachability graph for a node n is straightforward.

Algorithm The generation of test data for the statemet coverage criteria is described in Algorithm 6.1. A path reaching the specified statement is constructed dynamically. When a path reaches this statement, the test data generation for the path coverage criteria is used to find a tet case. As the potential number of pathes reaching the specified statement can be large (or infinite), pruning and heuristics are used during the search. First, pruning is realized by our ebox consistency operator. A path is abandonned as soon as we detect it is an infeasible path. The search is also guided by the CDG, and more particularly by the decision chain. The algorithm always extend a path by first chosing nodes in the decision chain, as such nodes is required in the path. As a second heuristics, the exit of the loop conditions are also selected first to avoid infinite paths. This algorithm can be optimized in many ways (incremental construction of the path constraints, . . .). We however prefer to present a simple and comprehensive version. Our algorithm is not complete. It may loop or fail to find test data. Computability theory (halting problem) shows that a complete algorithm does not exists.

In the nThRootBisect program (Figures 6.1 and 6.2), let us choose node 8. The reachability graph for node 8 is depicted with solid edges. The decision chain for this node is <2-T2, 4-F4, 6-F6>. First, the path 1-2-3-4-6-8 will be constructed by the algorithm. Assuming the corresponding path constraint is inconsistent, the path 1-2-3-4-6-7-2-3-4-6-8 is next constructed. For node 5, the decision chain is <2-T2, 4-T4>. The path 1-2-3-4-5 will be first constructed, then path 1-2-3-4-6-7-2-3-4-5.

Algorithm 6.1 Algorithm to generate test data : statement coverage

```
function TestDataGenerationSC ($P$ : $Program$, $n$ : $Node$, $D$ : $Box$) : $F^n$;
% PRE
%   $P$ The program under test
%   $n$ a node
%   $D$ The initial box
% POST
%   Returns a test case on which the node $n$ is executed
begin
  $G$ := $CFG$ of $P$;
  $G1$ := $CDG$ of $G$;
  $G2$ := reachability graph for node $n$ in $G1$ ;
  $DC$ := decision chain of node $n$ in $G2$;
  $V$ := Set of input variables of $P$
  return TestGen($P$, $V$, $D$, < >,$G2$, START, $n$, $DC$); {START is the start node in $G'$}
end

function TestGen($P$ : $Program$, $V$ : $Variables$, $D$ : $Box$, $path$ : $Path$,
  $G$ : $ReachabilityGraph$, $start$, $end$ : $Node$, $DC$ : $DecisionChain$ ) : $F^n$;
% PRE
%   $C$ is $\emptyset$ or $C$ is eBox-consistent
%   $path$ a path in $P$
%   $DC$ is the decision chain of node $end$ in $G2$;
% POST
%   Returns some test case satisfying the decision chain DC
begin
   for each successor $s$ of $start$ in $G$ do
   {If $start$ in $DC$, the successors in $DC$ are assumed to be enumerated first,}
   {if $start$ is a loop condition, the exit of the loop is assumed to be selected first}
      $newPath$ = $path$ . $s$
      $PC$ = $\mathcal{T}(P, newPath)$;
      $(V, D', PC)$ := $\Phi_{eBox}(V, D, PC)$
      if $(D' \neq \emptyset)$ then
        if $(s = end)$ then
          {test data generation: path coverage}
          $result$ = SolvePathConstraint($V$, $V$, $D'$, $PC$);
          if $result \neq \emptyset$ then return $result$;
        else return TestGen($P$, $V$, $D'$, $newPath$, $G$, $s$, $end$, $DC$);
end
```

7 Implementation and Experimental Results

Prototype To validate our approach, we developed a prototype written in Java. It uses an interval arithmetic library [Hic00] for the implementation of the constraint solving algorithm.. In its present form, this prototype concentrates on the constraint solving; the program under analysis is given in its control flow graph (CFG). In the `FindSolution` function, we assume here that the underlying programming language $\mathcal{L}$ is (a purely imperative part of) Java.

Experiment 1 Our first program is a program without loop, integrating integer and float variables. This has been shown in Figure 4.1. It is taken from [GMS98], but we forced y to be an integer variable. Its SSA form were also given as well as its CFG (Figure 4.2).

A test case is searched for the path 1-3-4-6-7-9-10, represented in dashed lines. Given the initial box $(\mathrm{x}_0 \in [0, 100], \mathrm{y}_0 \in [0, 100], \mathrm{z}_0 \in [0, 100])$ our algorithm produces the test case

$$(\mathrm{x}_0 = 50, \mathrm{y}_0 = 75, \mathrm{z}_0 = 12.500000001455192)$$

Experiment 2 Our second example is a program with a loop , integrating integer and float variables (Figure 7.1). It calculates the n-th root of a number a using the Newton-Raphson method, and stops when the difference between two consecutive approximations is less than e. Its SSA form is given in Figure 7.2. We search a test case for the path 1-2-3-2-3-2-3-2-3-2-4. We thus want the body of the loop to be executed exactly four times. The corresponding path constraint is

$\mathrm{x}_{00} = a \wedge \mathrm{x}_{10} = ((\mathrm{n}-1)*\mathrm{a}+(\frac{1}{\mathrm{a}})^{\mathrm{n}-2})/\mathrm{n} \wedge (\mathrm{x}_{00}-\mathrm{x}_{10})^2 \geq \mathrm{e} \wedge \mathrm{x}_{01} = \mathrm{x}_{10} \wedge \mathrm{x}_{11} = ((\mathrm{n}-1)*\mathrm{x}_{01}+\mathrm{a}*(\frac{1}{\mathrm{x}_{01}})^{\mathrm{n}-1})/\mathrm{n} \wedge (\mathrm{x}_{01}-\mathrm{x}_{11})^2 \geq \mathrm{e} \wedge \mathrm{x}_{03} = \mathrm{x}_{11} \wedge \mathrm{x}_{13} = ((\mathrm{n}-1)*\mathrm{x}_{03}+\mathrm{a}*(\frac{1}{\mathrm{x}_{03}})^{\mathrm{n}-1})/\mathrm{n} \wedge \mathrm{x}_{03}-\mathrm{x}_{13})^2 \geq \mathrm{e} \wedge \mathrm{x}_{04} = \mathrm{x}_{13} \wedge \mathrm{x}_{14} = ((\mathrm{n}-1)*\mathrm{x}_{04}+\mathrm{a}*(\frac{1}{\mathrm{x}_{04}})^{\mathrm{n}-1})/\mathrm{n} \wedge \mathrm{x}_{04}-\mathrm{x}_{14})^2 \geq \mathrm{e} \wedge \mathrm{x}_{05} = \mathrm{x}_{14} \wedge \mathrm{x}_{15} = ((\mathrm{n}-1)*\mathrm{x}_{05}+\mathrm{a}*(\frac{1}{\mathrm{x}_{05}})^{\mathrm{n}-1})/\mathrm{n} \wedge (\mathrm{x}_{05}-\mathrm{x}_{15})^2 \geq \mathrm{e} \wedge \mathrm{x}_{02} = \mathrm{x}_{05} \wedge \mathrm{x}_{12} = \mathrm{x}_{15}$.

Given an initial box $(\mathrm{a} \in [10, 20], \mathrm{n} \in [2, 10], \mathrm{e} \in [1e-4, 1e-2])$ our algorithm produces the test case

$$\mathrm{a} = 15.0, \mathrm{n} = 2, \mathrm{e} = 0.00505$$

```
float nThRoot(float a, int n, float e) {
  $x_0$ = a
  $x_1 = ((n-1)*a + (\frac{1}{a})^{n-2})/n$
  while $((x_0 - x_1)^2 \geq e)$
    $x_0 = x_1$
    $x_1 = ((n-1)*x_0 + a*(\frac{1}{x_0})^{n-1})/n$
  return $x_1$
}
```

Figure 7.1: Program nThRoot : computing the n-th root of a number by the Newton-Raphson method

```
float nThRoot(float a, int n, float e) {
1a.  $x_{00}$ = a
1b.  $x_{10} = ((n-1)*a + (\frac{1}{a})^{n-2})/n$
2.   while $(x_{02} = \Phi(x_{00}, x_{01}),\ x_{12} = \Phi(x_{10}, x_{11})\ ;\ (x_{02} - x_{12})^2 \geq e)$
3a.    $x_{01} = x_{12}$
3b.    $x_{11} = ((n-1)*x_{01} + a*(\frac{1}{x_{01}})^{n-1})/n$
4.   return $x_{12}$
}
```

Figure 7.2: SSA form of Program nThRoot

Experiment 3 Our next example, shown in Figure 7.3 is a classical program testing the type of a triangle [PHP99]. It only contains integer variables, but has nested conditional instructions and unfeasible paths.

The path constraint generated for the path 1-3-4-5-6-8-16-17 is the following:

$i \neq 0 \wedge j \neq 0 \wedge k \neq 0 \wedge \mathrm{trityp}_2 = 0 \wedge i = j \wedge \mathrm{trityp}_3 = \mathrm{trityp}_2 + 1 \wedge \mathrm{trityp}_4 = \mathrm{trityp}_3 \wedge i \neq k \wedge \mathrm{trityp}_6 = \mathrm{trityp}_4 \wedge j \neq k \wedge \mathrm{trityp}_8 = \mathrm{trityp}_6 \wedge \mathrm{trityp}_8 = 1 \wedge i + j > k \wedge \mathrm{trityp}_{13} = 2 \wedge \mathrm{trityp}_{17} = \mathrm{trityp}_{13} \wedge \mathrm{trityp}_{18} = \mathrm{trityp}_{17}$

Given the initial box $(\mathrm{i} \in [0, 100], \mathrm{j} \in [0, 100], \mathrm{k} \in [0, 100])$, a test case (i = 50, j = 50, k = 25) is generated.

The path 1-3-4-5-6-8-18-19 is unfeasible. Its path constraint is the following:

$i \neq 0 \wedge j \neq 0 \wedge k \neq 0 \wedge \mathrm{trityp}_2 = 0 \wedge i = j \wedge \mathrm{trityp}_3 = \mathrm{trityp}_2 + 1 \wedge \mathrm{trityp}_4 = \mathrm{trityp}_3 \wedge i \neq k \wedge \mathrm{trityp}_6 = \mathrm{trityp}_4 \wedge j \neq k \wedge \mathrm{trityp}_8 = \mathrm{trityp}_6 \wedge \mathrm{trityp}_8 = 2 \wedge i + k > j \wedge \mathrm{trityp}_{14} = 2 \wedge \mathrm{trityp}_{17} = \mathrm{trityp}_{14} \wedge \mathrm{trityp}_{18} = \mathrm{trityp}_{17}$

With the initial box $(\mathrm{i} \in [0, 100], \mathrm{j} \in [0, 100], \mathrm{k} \in [0, 100])$, no test case were found.

```
int trityp(int i, int j, int k) {
1.  if ((i = 0) || (j = 0) || (k = 0))
2.    trityp = 4;
    else {
3.    trityp = 0;
4.    if (i = j)
5.      trityp = trityp + 1;
6.    if (i = k)
7.      trityp = trityp + 2;
8.    if (j = k)
9.      trityp = trityp + 3;
10.   if (trityp == 0) {
11.     if ((i + j ≤ k) || (j + k ≤ i) || (i + k ≤ j))
12.       trityp == 4;
        else
13.       trityp == 1;
      }
14.   else if (trityp > 3)
15.          trityp == 3;
16.   else if ((trityp == 1) && (i + j > k))
17.          trityp == 2;
18.   else if ((trityp == 2) && (i + k > j))
19.          trityp == 2;
20.   else if ((trityp == 3) && (j + k > i))
21.          trityp == 2;
22.   else trityp == 4;
    }
23. return trityp;
}
```

Figure 7.3: Program trityp

Experiment 4 The gcd program is shown Figure 7.4. It only has integer variables, and has a loop containing a conditional statement.

Given a path 1-2-3-1-2-4-1-2-3-5, the test case (a = 15, b = 9) is generated, with the initial box $(\mathrm{a} \in [1, 100], \mathrm{b} \in [1, 100])$.

The last two experiments show that our approach is also applicable for programs without float variables.

Experiment 5 Our last experiment illustrates the statement coverage criteria on the `nThRootBisect` program depicted in Figures 6.1 and 6.2.

When the statement (or node) 8 is selected, the program generates the path 1-2-3-4-6-7-2-3-4-6-8 and output the test case (a = 7.000000010011718, n = 2, e = 0.00505).

For statement 5, the generated path is 1-2-3-4-6-7-2-3-4-5, and the test case (a = 9.0, n = 2, e = 0.00505) is output.

```
int gcd(int a, int b) {
    while (a ≠ b) {
      if (a > b)
        a = a - b;
      else b = b - a;
    }
    return a;
}
```

Figure 7.4: Program gcd: computing the greatest common divisor of 2 integers

8 Conclusion

In this paper, we presented a novel approach for automated test data generation of imperative programs. One of the main originalities is its capacity to handle program containing *integer*, *boolean* and/or *float* variables. It handles statement, branch and path coverage criteria. Our purpose was thus to generate test data that will cause the program to traverse a specified statement, branch or path. This approach is based on consistency techniques integrating integer and float variables.

In our approach, the program under analyis is first translated into a static single assignment (SSA) form. For path coverage, a path constraint on the input variables is derived from the specified path and the program in SSA form. We showed how such a constraint can be constructed. The path constraint is then solved by an interval-based constraint solving algorithm handling integer, boolean and real variables. This algorithm used a consistency, called eBox consistency generalizing box consistency to integer and float variables. This simple consistency is sufficient for our purpose. A test input is finally extracted from the interval solutions.

For branch and statement coverage, we also proposed an algorithm for test data generation. This algorithm generates paths reaching the branch or statement. Such a generation uses consistency techniques to prune the search space, and the control dependence graph to guide the search. When such a path is generated, the algorithm for path coverage is used. A prototype has also been constructed. Experimental results showed the feasibility of our approach.

Our consistency-based approach to test data generation could be combined with existing approaches based on random or dynamic methods (e.g. [GMS99,PHP99]), especially when searching a test data exercising a specified statement of the program.

In future work, we will also consider consistency techniques allowing arrays in the program under analysis.

References

[AWZ98] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, January 1998. ACM Press.

[BM83] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 23(3):228–245, 1983.

[BM94] Marc M. Brandis and Hanspeter Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, 16(6):1684–1698, November 1994.

[BOV95] F. Benhamou, W.J. Older, and A. Vellino. Constraint logic programming on boolean, integer and real intervals. *Journal of Symbolic Computation*, 1995.

[BT95] F. Benhamou and Tourvaïne. Prolog iv: language and algorithmes. In *IVème Journées Francophones de Programmation en Logique*, pages 51–65, 1995.

[Del00] François Delobel. *Résolution de systèmes de contraintes réelles non linéaires*. PhD thesis, Université de Nice-Sophia Antipolis(UNSA), January 2000.

[DN84] J.W. Duran and S. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, July 1984.

[DO93] Richard A. DeMillo and A. Jefferson Offutt. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering and Methodology*, 2(2):109–127, April 1993.

[FK96] R. Ferguson and B. Korel. The chaning approach for software test data generation. *ACM Transactions on Software Engineering Methodology*, 5(1):63–86, 1996.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its uses in optimization. *ACMTransactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[GBR98] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. In *International Symposium on Software Testing and Analysis*, pages 53–62, 1998.

[GBR00] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. A CLP framework for computing structural test data. In *Computational Logic*, pages 399–413, 2000.

[GMS98] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. Automated test data generation using an iterative relaxation method. In *ACM SIGSOFT Sixth International Symposium on Foundations of Software Engineering(FSE-6)*, pages 231–244, Orlando, Florida, November 1998.

[GMS99] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. UNA based iterative test data generation and its evaluation. In *14th IEEE International Conference on Automated Software Engineering(ASE'99)*, pages 224–232, Cocoa Beach, Florida, October 1999.

[GMS00] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. Generating test data for branch coverage. In *15th IEEE International Conference on Automated Software Engineering (ASE'2000)*, September 2000.

[Got00] Arnaud Gotlieb. *Génération de cas de test structurel avec la programmation logique par contraintes*. PhD thesis, Université de Nice-Sophia Antipolis, January 2000.

[Hic00] T. Hickey. An interval arithmetic library, 2000. available at http://interval.sourceforge.net/interval/index.html.

[HMD97] Pascal Van Hentenryck, Laurent Michel, and Yves Deville. *Numerica. A modeling language for global optimization*. The MIT Press, Cambridge, Massachusetts, London, England, 1997.

[KC00] O. Koné and R. Castanet. Test generation for interworking systems. *Computer Communications*, 23:642–652, 2000.

[Kin76] J.C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[OJP97] A. Jefferson Offutt, Zhenyi Jin, and Jie Pan. The Dynamic Domain Reduction Procedure for Test Data Generation. *Software Practice and Experience*, 29(2):167–193, January 1997.

[PHP99] Roy P. Pargas, Mary Jean Harrold, and Robert Peck. Test-data generation using genetic algorithms. *Software Testing, Verification & Reliability*, 9(4):263–282, 1999.

[Tsa94] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1994(?).