COTTAGE: Test Data Generation based on Consistency Techniques

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

November 26, 2004

Abstract

This paper[1] presents a new approach for automated test data generation of imperative programs containing *integer*, *boolean* and/or *float* variables. A test program (with procedure calls) is represented by an Interprocedural Control Flow Graph (ICFG). The classical testing criteria (statement, branch, and path coverage), widely used in unit testing, are extended to the ICFG. Path coverage is the core of our approach. Given a specified path of the ICFG, a path constraint is derived and solved to obtain a test case. The constraint solving is carried out based on a consistency notion. For statement (and branch) coverage, paths reaching a specified node or branch are dynamically constructed. The search for suitable paths is guided by the interprocedural control dependences of the program. The search is also pruned by our consistency filter. Finally, test data are generated by the application of the proposed path coverage algorithm. The COTTAGE system, a 13,000 Java lines software, implements our approach for C programs. For each generated test data, the system also automatically generates an instrumented C program, allowing the user to verify the correctness of the test data. Experimental results, including complex numerical programs from [29], demonstrate the feasibility of the method and the efficiency of the COTTAGE system, as well as its versatility and flexibility to different classes of problems (integer and/or float variables; arrays, procedures, path coverage, statement coverage).

Keywords software testing, test data generation, path coverage, statement coverage, procedures, arrays, constraint satisfaction, consistency

1 Introduction

1.1 Test Data Generation

Software testing is an expensive and difficult task, accounting for up to 50% of the cost of software development [21]. *Test data generation* is a component of software testing, where one tries to generate test inputs covering some testing

[1] This paper is an extended version of [32, 33]

criteria (testing requirements). *Structural testing* is usually concerned with the use of a *Control flow graph* (CFG) for a program under test to guide the generation of test data. To adequately test the program at the structural level, one must consider structural elements (nodes, branches, or paths) of the CFG for coverage. For example, *statement coverage criteria* requires developing test cases to execute certain nodes of the CFG. Similarly, *branch coverage* requires test cases to traverse certain branches, and *path coverage* requires test cases to execute certain paths. Structural testing thus includes: (1) the choice of a criteria (statement, branch, or path), (2) the identification of a set of nodes, branches or paths, and (3) the generation of test data for each element of this set. The automation of the last phase (automated test data generation) is a vital challenge in software testing.

In the next, test data generation for a node will be referred to as statement coverage, for a branch as branch coverage, for a path as path coverage.

1.2 Related Work

For path coverage, one finds the following categories. *Symbolic evaluation* [5, 20] consists in replacing input variables by symbolic values, and then symbolically evaluates the statements along a path. A path constraint over these symbolic values is produced, and then solved to obtain test input. It is generally limited in handling arrays, indeterminate loops, and procedure calls. A solution for dealing with arrays in symbolic evaluation is however proposed in [6]. *Program execution based* approaches start by executing the program with an arbitrary test input(s). This input is then iteratively refined, by execution of the program, to obtain a final input(s) executing the path. The refinement is done by applying *function minimization search algorithms* [22], an *iterative relaxation method* [14], etc. These approaches exploit 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 the path. Moreover, if the path is infeasible and the associated constraints nonlinear, this approach may become difficult to apply.

For statement (or branch) coverage, various approaches can be found in the literature. They are generally classified [10] as *random*, *path-oriented*, or *goal-oriented*. We can also classify them following the underlying technique used by each approach. Path-oriented means that one needs to select a path(s) to reach the specified statement, and then generates test input for the path; while with goal-oriented, the generation of test data to execute the statement is carried out irrespectively of the path taken, i.e. the path selection is not needed.

Random test data generation [9] consists in trying test data generated randomly until the statement is executed. Many experiences [10] have shown however that it can be very inefficient to generate test data for complex programs. In *program execution based* (or *dynamic*) approaches, as discussed above, a first test data(s) is initiated with a (randomly) chosen input(s). If an undesirable execution flow is observed at some branch in the program, then a refinement process is used to find a new input(s) that will change the execution flow at this branch. The refinement is realized by applying function minimization algorithms [10] (goal-oriented), an iterative relaxation method [16] (path-oriented), *genetic algorithms* [28] (goal-oriented), *simulated annealing* [34] (goal-oriented), etc. Note that [16] is an extension of [14] for branch coverage. The results of [10] are extended in [23] to programs with procedures by considering the possible effect of statements in the called procedures on execution of the selected element. Another approach [27] incorporates ideas from symbolic evaluation and dynamic test data generation. Although dynamic approaches are powerful in handling arrays and dynamic data structures, it may require a great number of executions when the program involves many nonlinear conditions. A goal-oriented approach, based on *Constraint Logic Programming* (CLP) techniques, is given in [13]. The test data generation problem for a given statement is translated into constraints, solved by an instance of the CLP scheme. This approach offers advantages such as the handling of arrays and a restricted class of pointers. However, only integer inputs are treated. Note that a constraint solver over float numbers has recently been proposed in [25]. It is, on the other hand, limited to float inputs. In [13], procedure calls are handled, but only intraprocedural control dependences of the test program are used in the search process, even with the presence of procedure calls. Therefore, this is not precise for certain classes of programs as will be shown later. A summary of existing approaches with functionalities close to our method is given in Section 7 (Table 4).

1.3 Results and Contribution

Among the difficulties in the generation of test data is the presence in the program of arrays, procedure calls, pointers, unstructured control statements (such as goto, break), and floating-point variables. In this paper, we propose a consistency-based approach, referred to as the *consistency approach*, for test data generation of imperative programs containing *integer*, *boolean* and *float* variables, arrays, and procedure calls. Path and statement coverage are both handled. The results with statement coverage can easily extended to branch coverage.

Path coverage is the basic bloc of our approach. It is a constraint solving approach based on a consistency notion, e-box consistency, generalizing box-consistency [17] to integer, boolean, and float variables. For statement coverage, paths reaching the specified statement are dynamically constructed using consistency techniques, and the path coverage method is applied on these paths to find suitable test input. Our method for path coverage includes the following steps. (1) A path constraint is derived from a specified path. Such a constraint involves integer, boolean and float variables, as well as operations with arrays. (2) The path constraint is solved by an interval-based constraint solving algorithm, that provides interval solutions. (3) A test case is finally extracted from the interval solutions.

A prototype system, called *COTTAGE* (COnsistency Test daTA GEnerator), a 13,000 Java lines software, implements our approach for programs written in (a subset of) the C language. In the current implementation, we focus on C programs with integer and float variables, arrays, function calls, and a restricted class of one-dimensional pointers (to simulate by-reference parameters); but without dynamic data structures.

Contribution The main contribution of the paper is a new approach (based on consistency techniques) to the generation of test data for numeric programs (programs with integer, boolean and float variables) with procedure calls and arrays. This approach handles *path* and *statement* coverage criteria. Beside this general contribution, specific technical contributions of the paper include the following. (1) A new system of test data generation (COTTAGE) as mentioned above. For each generated test data, the system also automatically generates an instrumented C program, allowing the user to verify the correctness of the test data. Experimental results, including complex numerical programs from [29], demonstrate the feasibility of the method and the efficiency of the COTTAGE system, as well as its versatility and flexibility to different classes of problems (integer and/or float variables; arrays, procedures, path coverage, statement coverage). (2) Inside the system is a constraint solver suitable for test data generation (e.g. dealing with integer, boolean and float variables). (3) An extended framework on interval logic so as to handle interval constraints involving at the same time, integer, float and boolean variables, as well as the logical operators such as AND, OR, NOT.

Remark The other known method [12, 13], related to our work and also based on consistency, is limited to integer variables, and does not handle interprocedural control dependence. A constraint solver over float numbers has been proposed [25], where it is shown how such a solver could be used for test data generation. The solver is however limited to float variables, and no implementation is provided.

Compared with other approaches handling integer and float variables in the literature (e.g. [15, 28]), our approach can be seen as an alternative or as a complement. Our consistency method could be combined with dynamic approaches when searching a test data exercising a specified statement of the program.

1.4 Organization

The organization of the paper is as follows. The background is presented in the next section. An overview of our consistency approach is given in Section 3. Section 4 illustrates the generation of path constraints. Section 5 describes our test data generation algorithm for path coverage, while Section 6 proposes an algorithm for statement coverage. The COTTAGE System is then described in Section 7 and our experiments are shown in Section 8. Conclusions are finally presented in Section 9.

2 Background

2.1 Background on Test Data Generation

Transforming a Test Program into an Equivalent one The purpose of this transformation is to isolate all embedded function calls from their enclosing expressions. For each embedded function call, a new variable is added to hold its return value into the test program [1]. The transformed program is equivalent to the original one, assuming that, in an expression, all embedded function calls are evaluated. This might not be the case for non-strict operators such as the conditional AND (&&) in Java. In an expression like `x>1 && f(x)`, if `x>1` evaluates to `false`, the value of the expression is `false`, and `f(x)` is not evaluated. This restriction can easily be lifted by a more elaborated transformation, e.g. conditional AND are transformed into conditional statements.

For example, the C program (Program-1) in Figure 1 contains the function `B` with two embedded function calls. Figure 2 shows the transformed function

B without embedded function calls. In the sequel, when we refer to a program, we mean an equivalent one without embedded function calls.

```
void M(double a[10], int c) {        void B(double a[10]) {
  int i = 1;                           int i,j;
  while (i <= c) {                     scanf("%d %d", &i, &j);
    B(a);                              if (F(i) < F(j))
    i = i+1;                             C(&a[i], &a[j]);
  }                                    else C(&a[j], &a[i]);
}                                    }

void C(double *x, double *y) {
  double t;                          int F(int i) {
  if (*x > *y) {                       if (i >= 0 && i <= 9)
    t = *x;                              return i;
    *x = *y;                           else exit(1);
    *y = t;                          }
  }
}
```

Figure 1: Program-1

```
B(double a[10]) {
  int i,j,fi,fj;
  scanf("%d %d", &i, &j);
  fi = F(i);
  fj = F(j);
  if (fi < fj)
    C(&a[i], &a[j]);
  else C(&a[j], &a[i]);
}
```

Figure 2: An equivalent of Program-1's function B

Control Flow Graph The control flow of a program is usually represented by a *Control Flow Graph* (CFG) [31]. Formally, the CFG for a procedure P is a directed graph, where the nodes represent statements and the edges represent possible flow of control between nodes. The CFG contains two distinguished nodes, $Entry_P$ and $Exit_P$, representing respectively a unique entry node and a unique exit node of P. A node, representing a (conditional or loop) statement, is called a *decision node* (a point where control flow can diverge). A list of assignments without decisions is grouped in a *basic block node*. Each procedure call is represented by two nodes, a *call node* and a *return node*. An outgoing edge from a decision node is called a *branch*. Each branch of the CFG is associated with a *condition*.

Control-flow interactions among a procedure and its related called procedures are usually represented by an *Interprocedural Control Flow Graph* (ICFG) [31, 24]. Formally, the ICFG for a procedure P is a directed graph, which consists of a unique global entry node $Entry_{global}$, a unique global exit node $Exit_{global}$, and the CFGs (for P and all procedures called directly or indirectly by P). Apart from the edges of the individual CFGs, the ICFG also contains the following kinds of edges: (1) the edges ($Entry_{global}$, $Entry_P$) and ($Exit_P$, $Exit_{global}$); (2) each procedure call (represented by a call node c and a return node r) to procedure M corresponds to a *call edge* $(c, Entry_M)$ and a *return edge* $(Exit_M, r)$; (3) the edges that connect the nodes (representing a `halt` statement) to node $Exit_{global}$. Note that a `halt` statement represents an unconditional program halt such as the `exit()` system call in `C`. Each statement such as `x:=f(...)`, where `f(...)` is a function call, is represented by a pair of call and return nodes as in a procedure call. However, these nodes are now associated with `x:=f(...)`. Informally, an ICFG is constructed by connecting the individual CFGs at call sites.

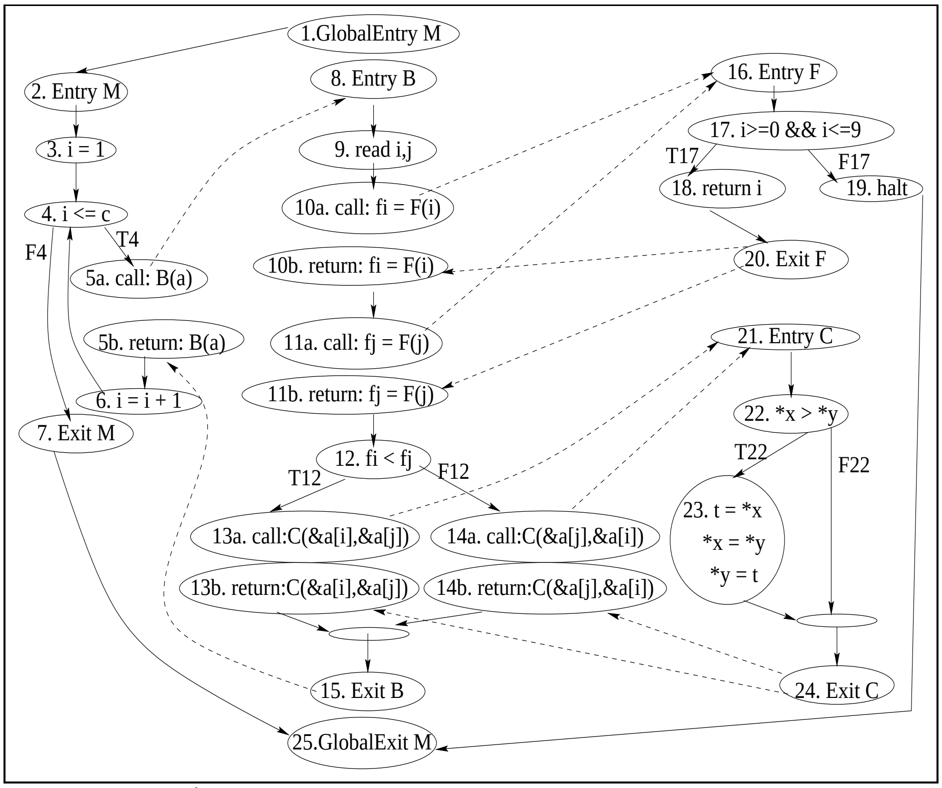

Figure 3: Interprocedural control flow graph for `M`

Figure 3 shows the ICFG for procedure `M` of Program-1 in Figure 1. The individual CFGs are connected by edges shown in dashed lines. If node i is a decision node, its true branch is labeled with a condition Ti, while its false branch is labeled with a Fi, that is the negation of Ti ($Fi = \neg Ti$). In this ICFG, the conditions $T4$, $T12$, $T17$, and $T22$ are respectively $i \leq c$, $fi < fj$, $i \geq 0 \,\&\&\, i \leq 9$, and $*x > *y$.

Path A *Path* is a sequence of nodes from the global entry node $Entry_{global}$ to a node of the ICFG. Note that a (partial) execution of a procedure P corresponds to an *execution path* in the ICFG for P. Paths, where a return edge does not match the corresponding call edge, are obviously infeasible execution paths. We thus restrict paths to feasible execution paths, where every return edge is properly matched with its corresponding call edge. Note that a path can be an *unbalanced-left path* [24], representing an execution in which not all of the procedure calls have been completed, i.e. there are more call edges than return ones in the path.

2.2 Background on Consistency

Path Constraint A *basic constraint* is a simple relational expression of the form 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 combination of basic constraints using the following logical operators $\{NOT, AND, OR\}$. We assume that the logical operators of the programming language of the program under analysis correspond to those constraints. Otherwise, the constraints can easily be extended. A path of the ICFG can be represented by a list of constraints with one constraint for each condition on the path. This list of constraints is called a *path constraint* where the constraints of the list are connected by the logical AND.

CSP, Consistency, and Constraint Solving Many important problems in areas like artificial intelligence and operations research can be viewed as *Constraint Satisfaction Problems* (CSP). A CSP $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ is defined by a finite set of variables $\mathcal{V}$ taking values from finite or continuous domains $\mathcal{D}$ 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 [35], and have been used extensively in many constraint softwares such as Numerica [17], Prolog IV [3], CLP(BNR) [2], etc.

Notations $\mathcal{R}$ denotes the set of real numbers (reals); $\mathcal{F}$ the set of floating-point numbers (float numbers) represented on a computer; $\mathcal{B}ool$ the set $\{false, true\}$. $\mathcal{F}$ is thus a finite subset of $\mathcal{R}$. The elements of $\mathcal{F}$ are called $\mathcal{F}$-numbers. The set of intervals is denoted by $\mathcal{I}$; the set of boolean intervals by $\mathcal{BI}$, where $\mathcal{BI} = \{[0, 0], [0, 1], [1, 1]\}$ (0 and 1 represent respectively $false$ and *true*). $\mathcal{BI}$ is thus a subset of $\mathcal{I}$. Capital letters denote intervals. If a is a real ($\mathcal{F}$-number or not), a^+ denotes the smallest $\mathcal{F}$-number strictly greater than a, and a^- the largest $\mathcal{F}$-number strictly smaller than a.

If x is a real, $\lfloor x \rfloor$ denotes the largest integer that is not larger than x and $\lceil x \rceil$ the smallest integer that is not smaller than x. The lower and upper bounds of an interval X are $\mathcal{F}$-numbers, and denoted respectively by *left*(X) and *right*(X). Boldface letters denote vectors of objects. The domain of a simple variable x is denoted by *dom*(x). If a is an array variable, *dom*(a) denotes the domain for its array elements and $length(a)$ its length (i.e. the number of elements). A *canonical interval* is an interval of the form $[a, a]$ or $[a, a^+]$, where a is a $\mathcal{F}$-number. An interval X is an ϵ*_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 ϵ*_interval* [17].

Test cases An (integer, boolean or float) *input variable* is either an input parameter or a variable in an input statement of program P. The domain of a boolean variable is an element of BI. The domain of an integer variable is an interval, representing a set of consecutive integers. The domain of a float variable is an interval of $\mathcal{F}$-numbers. Let $x_1, \ldots, x_n$ be n input variables of P, and D_k be 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 $i_k \in D_k \quad (1 \leq k \leq n)$.

The execution of the program (on the specified path) uses operators defined on $\mathcal{F}$-numbers, integers, and booleans. We assume here that the test program is written in some fixed imperative language $\mathcal{L}$.

Definition 1 (eval). Let c be a constraint, and $\mathbf{v}$ be a test input. 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. The test input $\mathbf{v}$ is said to be a float $\mathcal{L}$ solution.

Definition 2 (Path Constraint). A constraint c is said to be a *path constraint* for a 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 (of an ICFG), a test input $\mathbf{v}$ is a *test case* for p if $eval(c, \mathbf{v})$ holds, where c is a path constraint for p.

Definition 4. Given a node n (of an ICFG), a test input $\mathbf{v}$ is a *test case* for n if there exists a path p traversing n such that $\mathbf{v}$ is a test case for p.

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 by either executing the program under analysis, or by simulating such an execution (when the real environment is not available).

It is important to distinguish the real (or mathematical) solutions of a path constraint from its test cases (float $\mathcal{L}$ solutions). First, a mathematical solution may not be a float number. Second, a float (mathematical) solution $\mathbf{v}$ of a path constraint may not traverse the specified path, i.e. $c(\mathbf{v}) \not\Rightarrow eval(c, \mathbf{v})$. For example, the constraint, $c(x) \quad \triangleq \quad x = \frac{x}{3} + \frac{x}{3} + \frac{x}{3}$, is mathematically true for all $\mathcal{F}$-number in $\mathcal{F}$. However $eval(c, 1)$ may evaluate to false in some programming languages. Likewise, constraints may have float $\mathcal{L}$ solutions, while having no mathematical solution, i.e. $eval(c, \mathbf{v}) \not\Rightarrow c(\mathbf{v})$. [25] illustrates that the constraint, $16.0 + x = 16.0 \ \wedge \ x > 0$, actually possesses many float $\mathcal{L}$ solutions.

Classical Interval Programming Computation with the reals is actually difficult, since only a finite subset of reals can be represented on a computer. This means that the computer can only work with $\mathcal{F}$-numbers, and all real operations are actually operations on $\mathcal{F}$-numbers, which are commonly known as not sound. Because the result of an operation may not be computed exactly (due to round-off errors with sub-operations), or may not be representable in $\mathcal{F}$. One must approximate a real r by r^+ (upward rounding), or by r^- (downward rounding). To solve continuous constraints (over the reals) with traditional numerical methods, one thus obtains $\mathcal{F}$-numbers that are approximations of the mathematical solutions.

Interval methods solve the constraints by a different approach, which returns small intervals enclosing the mathematical solutions. They automatically bound numerical errors, and so ensure the reliability of the results. The basic idea consists in associating with each variable an interval representing its domain. The original problem is then pruned (by some consistency techniques) before divided into sub-problems (by splitting the interval associated with a variable), until all solutions are obtained. Such consistency techniques (on intervals) are designed to reduce the size of the intervals without removing solutions of the constraints [17].

We give here some important definitions of *Interval Analysis* [26], borrowed directly from, or based on, those in [17]. Since the goal is to work with intervals, all objects of the real space–such as reals, real sets, real functions, and real relations– should have an *interval extension* in the interval space. Note that interval extensions are not unique.

Definition 5 (Interval). An interval $I = [a, b]$, with $a, b \in \mathcal{F}$, denotes the set $\{x \in \mathcal{R} \mid a \leq x \leq b\}$. Also, given an interval I, $left(I)$ and $right(I)$ denote respectively a and b. The set of intervals is denoted by $\mathcal{I}$.

Definition 6 (Interval Extension). Let S be a subset of $\mathcal{R}$. The interval extension of S, denoted by $\Box S$, is the smallest interval I such that $S \subseteq I$. When $S = \{r\}$, we denote its interval extension by $\Box r$, and its value is the interval $[r, r]$ if r is an $\mathcal{F}$-number, and $[r^-, r^+]$ otherwise.
An *interval function* $F : \mathcal{I}^n \to \mathcal{I}$ is an interval extension of $f : \mathcal{R}^n \to \mathcal{R}$ if $\forall \mathbf{I} \in \mathcal{I}^n : f(\mathbf{I}) \subseteq F(\mathbf{I})$, where $f(\mathbf{I}) = \{f(\mathbf{r}) \mid \mathbf{r} \in \mathbf{I}\}$.

Definition 7. An *interval relation* $C : \mathcal{I}^n \to \mathcal{B}ool$ is an interval extension of the relation $c : \mathcal{R}^n \to \mathcal{B}ool$ if
$\forall \mathbf{I} \in \mathcal{I}^n \ : \ (\exists \mathbf{r} \in \mathbf{I} : c(\mathbf{r})) \Rightarrow C(\mathbf{I})$

The objective of an interval extension is to preserve the mathematical solutions. Given a function f, the *optimal interval extension* is the interval function returning $\Box(f(\mathbf{I}))$. Many usual functions possess an optimal interval extension. For example, the interval function $\oplus$, $[a_1, b_1] \oplus [a_2, b_2] = [(a_1 + a_2)^-, (b_1 + b_2)^+]$, is an optimal interval extension of the addition of two reals. Note that $a_1 + a_2$ or $b_1 + b_2$ may not be representable in $\mathcal{F}$. [7] reported however that there exist functions, due to practical reasons, we cannot calculate their optimal interval extension, namely the function $f(x) = x * sin(x)$.

Given an arbitrary function, one can obtain several (non-optimal) interval extensions [7], based on interval extensions for its primitive operations. The *natural interval extension* is however used in our work so as to also conserve float solutions, as illustrated hereafter.

Definition 8 (Natural Interval Extension). Given a function f, the *natural interval extension* of f is obtained by replacing in the definition of f, each constant k by its approximation ($\Box k$), each real variable x by the interval variable X, each real operation g by its interval extension G.

Proposition 1 ([26]). If $F : \mathcal{I}^n \to \mathcal{I}$ is the *natural interval extension* of $f : \mathcal{R}^n \to \mathcal{R}$, then F is an interval extension of f, i.e. $\forall \mathbf{I} \in \mathcal{I}^n : f(\mathbf{I}) \subseteq F(\mathbf{I})$.

By Proposition 1, $F(\mathbf{I})$ contains thus at least all real solutions of $f(\mathbf{I})$.

For instance, the natural interval extension of, $f(x) = x^{2.3} - 2x + sin(x)$, is the interval function, $F(X) = (X \odot \Box 2.3) \ominus \Box 2 \otimes X \oplus SIN(X)$, where $\ominus$, $\otimes$, $\odot$, and SIN are the interval extensions of subtraction, multiplication, exponentiation, and the trigonometric sin function.

2.3 Extended Framework on Interval Logic

We here extend the classical definition of interval extension, and build a new interval logic framework to handle interval constraints involving at the same time, integer, float and boolean variables, as well as the logical operators such as AND, OR, NOT. Such a framework is needed for the following reasons. First, Definition 7 means that C is a mapping from I^n to the set $\{false, true\}$. Assuming C is an interval extension of relation c, let us denote $c(\mathbf{I}) = \{c(\mathbf{r}) \mid \mathbf{r} \in \mathbf{I}\}$ with $\mathbf{I} \in \mathcal{I}^n$. If $(\exists \mathbf{r} \in \mathbf{I})\ c(\mathbf{r})$, then we have $\{C(\mathbf{I})\} \subseteq c(\mathbf{I})$. This leads to the following consequence: $[2,4] < [1,3]$ evaluates to *true*, from which one can deduce that $not([2,4] < [1,3])$ evaluates to $false$. Paradoxically, the negation can evaluate to *true* if one treats $not([2,4] < [1,3])$ as $[2,4] \geq [1,3]$. Second, we need a framework in which we can define interval extensions for constraints involving boolean variables as well as the logical operators, such as $c(b,x,y) \triangleq not(b)\ and\ (x > 1)\ or\ (y < 2)$, where b is a boolean variable, while x, y are (integer or float) variables. We must then be able to evaluate $C(b : [0,1], x : [2,3], y : [3,5])$, for instance.

Definition 9 (Interval Extension of a Relation (Constraint)). An interval relation $C : \mathcal{I}^n \to \mathcal{BI}$ is an interval extension of the relation $c : \mathcal{R}^n \to \mathcal{B}ool$ if $\forall \mathbf{I} \in \mathcal{I}^n : c(\mathbf{I}) \subseteq C(\mathbf{I})$, where $c(\mathbf{I}) = \{c(\mathbf{r}) \mid \mathbf{r} \in \mathbf{I}\}$, and the convention that $\{false\} = [0,0]$, $\{true\} = [1,1]$, and $\{false, true\} = [0,1]$.

Interval extensions for the relational operators $\{<, \leq, >, \geq, =, \neq\}$ are developed, based on the following definition.

Definition 10. Given a relation $c : \mathcal{R}^n \to \mathcal{B}ool$, the corresponding interval relation $C : \mathcal{I}^n \to \mathcal{BI}$ is constructed such that for all $\mathbf{I} \in I^n$
if $\nexists \mathbf{x} \in \mathbf{I} : c(\mathbf{x})$ then $C(\mathbf{I}) = [0,0]$
if $\exists \mathbf{x} \in \mathbf{I} : c(\mathbf{x}) \bigwedge \exists \mathbf{y} \in \mathbf{I} : \neg c(\mathbf{y})$ then $C(\mathbf{I}) = [0,1]$
if $\forall \mathbf{x} \in \mathbf{I} : c(\mathbf{x})$ then $C(\mathbf{I}) = [1,1]$

Proposition 2. The interval relation C, as defined in Definition 10, is an interval extension of the relation c.

For instance, an interval extension of the relational operator $\leq$ is the following: $[a_1, a_2] \leq [b_1, b_2]$ is $[0,0]$ if $a_1 > b_2$, $[1,1]$ if $a_2 \leq b_1$, and $[0,1]$ otherwise. We now define interval extensions for the logical operators *not*, *and*, and *or*.

Definition 11. Let a_1, b_1, a_2, and b_2 be values taken in $\{0,1\}$ such that $a_1 \leq b_1$ and $a_2 \leq b_2$, then the interval logical operators NOT, AND, and OR are defined as follows

- $NOT([a_1, b_1]) = [1 - b_1, 1 - a_1]$,
- $[a_1, b_1]\ AND\ [a_2, b_2] = [min(a_1, a_2), min(b_1, b_2)]$,
- $[a_1, b_1]\ OR\ [a_2, b_2] = [max(a_1, a_2), max(b_1, b_2)]$.

Proposition 3. The interval logical operators NOT, AND, and OR, as defined in Definition 11, are respectively interval extensions of the logical operators *not*, *and*, and *or*.

An *interval solution* of a set of constraints is a box containing solutions of the different constraints. It is defined as follows.

Definition 12. Let $S = \{c_1, \dots, c_m\}$ be a set of constraints. A box $\mathbf{X} \in \mathcal{I}^n$ is an interval solution of S if $right(C_i(\mathbf{X})) = 1$, for all i $(1 \le i \le m)$, where the C_i are respectively the natural interval extension of the c_i.

For simplicity, $C(\mathbf{X})$ will denote $right(C(\mathbf{X})) = 1$ (i.e. $C(\mathbf{X})$ is $[0, 1]$ or $[1, 1]$) and $\neg C(\mathbf{X})$ for $right(C(\mathbf{X})) = 0$ (i.e. $C(\mathbf{X})$ is $[0, 0]$).

3 Overview of the consistency approach

We are now able to precisely state our test data generation problem.

Problem statement *Given a node n, a branch b or a path p of the ICFG associated with a test procedure P (possibly with procedure calls), generate a test input i such that P when executed on i will cause n, b or p to be traversed.*

This section describes the consistency approach for test data generation with path and statement coverage. It is a constraint solving approach based on a consistency notion, e-box consistency, generalizing box-consistency [17] to integer, boolean, and float variables. Path coverage is the core of our approach. It includes the following steps.

1. A path constraint is derived from the specified path of the ICFG. Such a constraint involves integer, boolean and float variables, as well as operations with arrays.
2. The path constraint is solved by an interval-based constraint solving algorithm. The idea of such a solving algorithm is as follows.
   - An initial box is provided.
   - Consistency techniques are used to prune the box.
   - The box is splitted into some parts, which are then explored recursively until obtaining epsilon boxes —very small boxes— containing float solutions of the path constraint. These epsilon boxes are called interval solutions.
3. A test case is finally extracted from the interval solutions.

For statement coverage, paths reaching the specified statement are dynamically constructed. The search for such paths is guided by the interprocedural control dependences of the program, as well as pruned by our e-box consistency filter to avoid exploring infeasible paths. Our algorithm for path coverage is then applied on these paths to generate test data. It should be noted that as a branch is dual to a statement in the control flow graph, all the following results with statement coverage can easily be extended to branch coverage.

It is important to precise the specificities of solving a path constraint compared to classical interval-based constraint solving. First, a path constraint is usually under-constrained; there usually exist many test inputs traversing the specified path (except for an infeasible path) while we are interested by finding one of them. Existing constraint systems, such as Numerica [17], are not always appropriate for under-constrained systems as they try to generate all the solutions. Second, existing solvers–Numerica, Prolog IV [3], and CLP(BNR) [2]–will produce (small) intervals containing the mathematical solutions of the path constraint. A mathematical solution can be a real which is not a float number. Moreover, even if a mathematical solution is a float number, 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 mathematically sound. Third, the goal of existing consistency techniques is to preserve all mathematical solutions in pruning the search space, and therefore may not ensure preserving all float solutions (solutions with the programming language operators) [25]. In contrast, the goal of our consistency techniques is to preserve float solutions. Our constraint solver in turn returns float solutions as test cases, and therefore ensure traversing the path. These differences make that existing constraint solving approaches cannot be used solely to generate test data for programs with integer, boolean, and float variables. Finally, it should be noticed that any constraints solving system may produce an interval without mathematical solution.

4 Generation of Path Constraints

4.1 Algorithm

Given a path of an ICFG, we propose an algorithm (Algorithm 1) to construct a path constraint. Indexed variables are used to hold the definitions of the original variables in the path.(Assignments to a variable are referred to as its *definitions*) For example, for variable x, its first definition in the path is assigned to x_0, its second to x_1, and so on. All uses of this variable are renamed accordingly and refer to its last definition. Since indexed variables have a unique definition, we will refer to them as *value instances* of the original variables.

Algorithm 1 Path constraint generation for a path in the ICFG

```
function PathConstraintGeneration(P:Procedure,G:ICFG,p:Path) : CSP;
PRE G is the ICFG for test procedure P
    p is a path p_1,...,p_n in G
POST return a path constraint for path p
declare
  PC : path constraint for path p
begin
  PC := ∅; {PC is initially empty}
  for each i from 1 to n do
    PC := PC ∧ ConstraintsForNode(p_i);
    if (p_i is a decision node) and (i < n) then
      PC := PC ∧ ConstraintsForBranch(< p_i, p_{i+1} >);
  return PC;
end
```

The algorithm `PathConstraintGeneration` (Algorithm 1) takes as input a path in the ICFG. It makes a traversal of the path to generate constraints for its nodes and branches. The generated path constraint is the conjunction of all these constraints.

`ConstraintsForNode` and `ConstraintsForBranch` respectively generate constraints for a node and a branch of the ICFG. They will be described by the following definition.

Definition 13. Let p be a path of the ICFG, p_i be a node of p, and $< p_i, p_{i+1} >$ be a branch of p. Then $\mathcal{PC}(p, p_i)$ and $\mathcal{PC}(p, < p_i, p_{i+1} >)$ respectively denote constraints generated for p_i and $< p_i, p_{i+1} >$.

Let $\mathcal{PC}(p)$ denote the path constraint generated for the path p, then from Algorithm 1 and Definition 13, we have

$$\mathcal{PC}(p) = \bigwedge_{p_i\ :\ a\ node\ of\ p} \mathcal{PC}(p, p_i) \quad \bigwedge_{<p_i, p_{i+1}>\ :\ a\ branch\ of\ p} \mathcal{PC}(p, < p_i, p_{i+1} >)$$

Depending on the type of p_i (which can be a global entry, an assignment, an input statement, a call node, etc.), $\mathcal{PC}(p, p_i)$ and $\mathcal{PC}(p, < p_i, p_{i+1} >)$ are constructed accordingly as follows.

Global Entry Node p_i is the global entry node of the ICFG associated with a test procedure P. Suppose that P has the following parameters: x (a simple variable), a (an array variable). Then, $\mathcal{PC}(p, p_i)$ is

$$\mathrm{x}_0 \in dom(x) \wedge \bigwedge_{0 \leq i < length(a)} \mathrm{a}_0[i] \in dom(a)$$

where $dom(x)$ denotes the domain for x, and $dom(a)$ the domain for all array elements of a. These constraints thus aim to define input variables from the formal parameters of procedure P.

Note that the initial domain (interval) may depend on the programming language $\mathcal{L}$, but can also be fixed by the user. We focus our presentation to one-dimensional arrays, but the approach itself can be easily generalized to multi-dimensional arrays. We also suppose that the size of array variable a is specified. If this is not the case, we note only that a_0 is an input variable. And no $\mathrm{a}_0[i]$ are created at this node, since $length(a)$ is unknown. Later, when we deal with an $\mathrm{a}_0[i]$ (i is a number), and if no input variable representing $\mathrm{a}_0[i]$ exists, then an input variable $\mathrm{a}_0[i]$ is created.

Definition 14. Let exp be an expression. Then $\overline{exp}$ denotes a version of exp in which each variable is substituted by its last value instance.

Assignment

- If p_i is an assignment to a simple variable, `x := exp`, then $\mathcal{PC}(p, p_i)$ is

$$x_k = \overline{exp}$$

  where k is the smallest integer not yet used for identifier x. Note that this sort of equality constraints, associated with assignments, will be denoted as $x_k := \overline{exp}$.

- If p_i is an assignment to an array element, `a[j]:=exp`, then $\mathcal{PC}(p, p_i)$ is

$$\mathsf{na4}(a_{k'}, a_k, \overline{j}, \overline{exp})$$

  where a_k is the last value instance of a; k' is the smallest integer not yet used for identifier a. The constraint $\mathsf{na4}$ (na represents New Array) is defined hereafter.

Definition 15. The constraint $\mathsf{na4}(b, a, j, v)$ states that b is an array which is of the same size as a and has the same component values, except for v as the value of its j-th component. It can be defined more formally as follows.

$$\mathsf{na4}(b, a, j, v) \triangleq b[j] = v \bigwedge_{i \neq j} b[i] = a[i]$$

Definition 16. By convention, when all the elements of array a are *null* (non-initialized), we will denote this as $a = null$.

For example, a code such as, `int a[5]; a[0] = 8; a[1] = 7;`, generates the following constraint, $\mathsf{na4}(\mathrm{a}_0, null, 0, 8) \wedge \mathsf{na4}(\mathrm{a}_1, \mathrm{a}_0, 1, 7)$.

Input Statement

- If p_i is an input statement to a simple variable, `read x` (in C, this is realized by `scanf`), then $\mathcal{PC}(p, p_i)$ is

$$x_k \in dom(x)$$

  where k is the smallest integer not yet used for identifier x. This constraint defines x_k as an input variable.

- If p_i is an input statement to an array element, `read a[j]`, then $\mathcal{PC}(p, p_i)$ is

$$\mathsf{na3}(a_{k'}, a_k, \overline{j})$$

  where a_k is the last value instance of a; k' is the smallest integer not yet used for identifier a. The constraint $\mathsf{na3}$ is defined as follows.

Definition 17. The constraint $\mathsf{na3}(b, a, j)$ is formally defined as

$$\mathsf{na3}(b, a, j) \triangleq b[j] \in dom(b) \bigwedge_{i \neq j} b[i] = a[i]$$

The goal of this constraint is to define $b[j]$ as an input variable.

Decision Node If p_i is a decision node, then $\mathcal{PC}(p, p_i)$ is empty.
However $\mathcal{PC}(p, < p_i, p_{i+1} >)$ is $\overline{c}$, where c is the condition associated with the branch $< p_i, p_{i+1} >$.

Procedure Call If p_i is a call node to a procedure P (the control is going to pass to P), then for each pair (x', x) —where x' is an actual parameter of the call, and x is the corresponding formal parameter of P (that is a by-value or by-reference parameter)— an assignment $x := x'$ is generated. These assignments are converted into constraints, that are then affected to $\mathcal{PC}(p, p_i)$.

If p_i is the return node of a call to procedure P (the control just quits P), an assignment $x' := x$ is generated if x is a by-reference parameter. Moreover, if the return node is associated with $z := P(\dots)$, an assignment is also generated. All these assignments are converted into constraints, that are then affected to $\mathcal{PC}(p, p_i)$.

This way of handling parameters is commonly known as the *call by value-result* mode of parameter passing, where the parameters of the procedure are not directly bound to the variable's address. Rather, they have their own space within their scope, and the new values of the parameters are copied back into the caller's variables only when the procedure is terminated.

To illustrate the difference between the call by reference and the call by value-result modes, consider the following C code [8]:

```
void a(int *x, int *y) {
  *x = 1;
  *y = 2;
}

int t;
a(&t,&t);
```

Then with the call by value-result mode, the value of t after the call depends on the order of parameter copies when the call is finished; while with the call by reference mode, the value of t will always be 2. This problem is due to aliasing (i.e. if x and y refer to the same variable or address, then x and y are *aliased*). Since in this work, we rather focus our attention to the feasibility of applying our consistency approach for test data generation, the aliasing problem is left for future work.

4.2 Example

Example 1. We illustrate the operation of the algorithm on the path 1-2-3-4-5a-8-9-10a-16-17-18-20-10b-11a-16-17-18-20-11b-12-13a-21-22-23-24-13b-15-5b-6-4-7-25 (in Figure 3). Constraints are generated for the nodes as follows.
Node 1: $\bigwedge_{0 \leq i < 10} \mathrm{a}_0[i] \in dom(a) \wedge \mathrm{c}_0 \in dom(c)$;

> a and c are the parameters of procedure M (Figure 1). The constraint defines thus input variables.

Node 2: no constraints are generated;
Node 3: $\mathrm{i}_0 := 1$;
Node 4-$T$4: $\mathrm{i}_0 \leq \mathrm{c}_0$;
Node 5a: $\mathrm{a}_1 := \mathrm{a}_0$;
Node 8: no constraints;
Node 9: $\mathrm{i}_1 \in dom(i) \wedge \mathrm{j}_0 \in dom(j)$;
Node 10a: $\mathrm{i}_2 := \mathrm{i}_1$;
Nodes 16, 17-$T$17, 18, 20: $\mathrm{i}_2 \geq 0 \wedge \mathrm{i}_2 \leq 9$;
Node 10b: $\mathrm{fi}_0 := \mathrm{i}_2$;
Node 11a: $\mathrm{i}_3 := \mathrm{j}_0$;
Nodes 16, 17-$T$17, 18, 20: $\mathrm{i}_3 \geq 0 \wedge \mathrm{i}_3 \leq 9$;
Node 11b: $\mathrm{fj}_0 := \mathrm{i}_3$; Node 12-$T$12: $\mathrm{fi}_0 < \mathrm{fj}_0$;
Node 13a: $\mathrm{x}_0 := \mathrm{a}_1[\mathrm{i}_1] \wedge \mathrm{y}_0 := \mathrm{a}_1[\mathrm{j}_0]$;
Nodes 21, 22-$T$22, 23, 24: $\mathrm{x}_0 > \mathrm{y}_0 \wedge \mathrm{t}_0 := \mathrm{x}_0 \wedge \mathrm{x}_1 := \mathrm{y}_0 \wedge \mathrm{y}_1 := \mathrm{t}_0$;
Node 13b: $\mathsf{na4}(\mathrm{a}_2, \mathrm{a}_1, \mathrm{i}_1, \mathrm{x}_1) \wedge \mathsf{na4}(\mathrm{a}_3, \mathrm{a}_2, \mathrm{j}_0, \mathrm{y}_1)$;
Nodes 15, 5b: $\mathrm{a}_4 := \mathrm{a}_3$;
Nodes 6, 4-$F$4, 7, 25: $\mathrm{i}_4 := \mathrm{i}_0 + 1 \wedge \neg(\mathrm{i}_4 \leq \mathrm{c}_0)$.

A path constraint is composed of: (1) constraints defining input variables: simple input variable and input array element (na3 constraints), (2) assignment constraints: equality constraints with ":=" notation for simple variables, and na4 constraints for array elements, (3) branch constraints (constraints for the branches of the path). However, only the branch constraints (3) represent the conditions which must be satisfied so that the path is traversed. The other types of constraints (1) and (2), as will be shown later, are used in the simplification of the branch constraints in terms of input variables. The solving of the path constraint is the solving of its branch constraints. In the CSP associated with a path constraint, only the input variables will have a domain. There is no need to define a domain for the other variables as they are defined in terms of input variables or constraints. If it is not the case, the program is referring to non-initialized variables, and is thus incorrect.

4.3 Analysis

Proposition 4. The constraint generated by Algorithm 1 is a path constraint.

Proof. The path can be seen as a program (called *path program*) if we replace every condition c_i of the path by an assignment $b_i := c_i$, where b_i is a boolean. A test input $\mathbf{v}$ will traverse the path if after executing the path program, all the b_i will become true. To prove that the constraint generated by Algorithm 1 —as denoted above by $\mathcal{PC}(p)$— is a path constraint, we must show that $\bigwedge b_i$ is equivalent to $\mathcal{PC}(p)$.

As the above abstraction of the algorithm (namely, the idea of using indexed variables) follows the same principle as the Static Single Assignment (*SSA*) form [4] (which is an equivalent representation of a program), we can conclude that the generated constraint $\mathcal{PC}(p)$ is actually equivalent to the path program. In other words, $\mathcal{PC}(p)$ is a path constraint. This means that if a test input $\mathbf{v}$ traverses the path p, then the constraint $\mathcal{PC}(p)$ (evaluated by using the operators of the programming language $\mathcal{L}$) will be satisfied by that input. □

Note that in an SSA form, there is only one assignment to each variable in the entire program, and each use of a variable refers to only one assignment. Thanks to this form, one can reason easily about variables because if two variables have the same name, then they contain the same value wherever they occur in the program. Here is an example of a simple sequence of assignments and its corresponding SSA form :

Original form : `x = 0; y = x+1; x = x+y; y = x+y;`

SSA form : $\mathtt{x_1 = 0;\ y_1 = x_1+1;\ x_2 = x_1+y_1;\ y_2 = x_2+y_1;}$

Our constraints dealing with arrays such as na3 and na4 constraints, are also inspired from SSA form. Indeed, SSA form provides a special expression, among others, to handle arrays, $update(a, j, w)$ which evaluates to an array that has the same size and the same elements as a, except for the j-th element where the value is w.

In another work [13], also based on SSA form, a definition statement $a_1 = update(a_0, j, w)$ is translated into

$$\mathsf{element}(J, A_1, W) \bigwedge_{I \neq J} (\mathsf{element}(I, A_0, V) \wedge \mathsf{element}(I, A_1, V))$$

where the constraint $\mathsf{element}(I, L, V)$ expresses that V is the I-th element in the list L. Note that in [13], the initial program is first transformed into an SSA form, and constraints reaching a node (statement) are then constructed from this form; while in our work, given a specified path, we rather make a traversal of the path to construct directly an SSA-form-like path constraint.

5 Test Data Generation for Path Coverage

This section describes a constraint solving algorithm for test data generation under the path coverage criteria. The ideas underlying the conservation of float solutions in our filtering algorithms will also be highlighted. We first define a consistency notion, called *e-box consistency*, that is the core of our solver for the test data generation problem.

5.1 Consistency

We introduced e-box consistency in [32] as an extension of the classical box-consistency [17] to integer, boolean, and float variables. The objective is to reduce the domains of the variables (i.e. their intervals) without removing solutions.

Definition 18 (e-box consistency). Let $P = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ be a CSP where $\mathcal{V} = (x_1, \ldots, x_n)$, a set of (float and integer) variables; $\mathcal{D} = (X_1, \ldots, X_n)$ 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 *e-box consistent* in $\mathcal{D}$ if for all x_i $(1 \leq i \leq k)$

- if x_i is a float 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 the natural interval extension of constraint c.

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

Given the initial domains of the variables (the initial box), the purpose of filtering is to obtain the smallest box, satisfying the e-box consistency, and without removing any solutions from the initial box. In constraint programming, one finds a lot of sophisticated consistencies dealing with real solutions, such as used in Prolog IV, CLP(BNR), or Numerica. A consistency dealing with float solutions was also proposed in [25], where the definition is simpler than the e-box consistency as it concentrates only on float variables.

Definition 19 (Filtering by e-box consistency). Filtering by e-box consistency of a CSP $P = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ is a CSP $P' = (\mathcal{V}, \mathcal{D}', \mathcal{C})$ such that (1) $\mathcal{D}' \subseteq \mathcal{D}$, (2) P and P' have the same float solutions, and (3) P' is e-box consistent in $\mathcal{D}'$.

Our filtering algorithm is based on the property that if $C(\mathbf{X})$ does not hold, i.e. $right(C(\mathbf{X})) = 0$, then no solution of c lies in box $\mathbf{X}$, that can then be pruned. We denote by $\Phi_{e-box}(P)$, the filtering by e-box consistency of P. Note that the filtering by e-box consistency of a CSP, by its definition, always exists and is unique. An implementation of $\Phi_{e-box}(CSP)$, called `PhiEBox`, is presented in Algorithm 2, that is an extension of a filtering algorithm in [17]. Technically, our `LeftNarrow` algorithm is simpler than a standard one [7] for the classical box-consistency [17], but safe for float solutions. Our `LeftNarrow` consists in applying recursively a domain-splitting on the initial interval to obtain the left-most zero canonical interval, while the standard `LeftNarrow` recursively applies two operations, an iterator of Newton and a domain-splitting on the initial interval. Because of the use of the Taylor interval extension, and of the fact that the iterator of Newton aims to prune parts —which do not have mathematical solutions— to make the algorithm converge more quickly, it may not be safe for float solutions as shown in [25].

Algorithm 2 Φ_{e-box}

```
function PhiEBox(V : Variables, D : Box, C : Constraints) : CSP;
PRE
  V a set of variables
  D a box of their corresponding domains
  C a set of constraints over V
POST
  Return a CSP (V, D', C) such that (V, D', C) = Φ_{e-box}(V, D, C)
begin
queue := C;
while queue ≠ ∅ do
   c := dequeue(queue); {Suppose c is a constraint over x_1, ... , x_k}
   updatedDomVars := ∅; {A set of variables with domain updated}
   for x_i ∈ variables(c) do
     C_X := C(X_1, ..., X_{i-1}, X, X_{i+1}, ..., X_k);{univariate interval constraint}
     left(X'_i) := left(LeftNarrow(C_X, X_i));
     right(X'_i) := right(RightNarrow(C_X, X_i));
     if X_i ≠ X'_i then
       X_i := X'_i;
       if X_i = ∅ then return (V, ∅, C);
       updatedDomVars := updatedDomVars ⋃ {x_i};
   endfor
   queue := queue ⋃ {c' ∈ C | updatedDomVars ⋂ variables(c') ≠ ∅};
  endwhile
return (V, D, C);
end
```

5.2 Conservation of Float Solutions

Since interval libraries are traditionally constructed to preserve mathematical solutions, it may be that float solutions will not be preserved when using such libraries. As introduced in Section 2, our aim is to obtain float solutions. We describe here the core results related to the conservation of float solutions in our filtering algorithms. We first give some necessary definitions.

As discussed earlier, $\mathcal{F}$-numbers are a finite and discrete version of the real numbers on a computer. And all real operations are thus replaced by operations over $\mathcal{F}$-numbers. As the result of an operation over $\mathcal{F}$-numbers may not be an $\mathcal{F}$-number, rounding is necessary to close the operations over $\mathcal{F}$. The IEEE 754 standard for floating-point arithmetic proposes the following four rounding modes:

- $+\infty$: which maps x to the smallest $\mathcal{F}$-number x_k such that $x \leq x_k$.
- $-\infty$: which maps x to the greatest $\mathcal{F}$-number x_k such that $x_k \leq x$.
- 0 : which is equivalent to the rounding mode $+\infty$ if $x < 0$ and to $-\infty$ if $x \geq 0$.
- *near* : which maps x to the nearest $\mathcal{F}$-number.

An overview of the IEEE 754 standard can be found in [25].

Definition 20. Let $f : \mathcal{R}^n \rightarrow \mathcal{R}$ be a real expression. Then we denote by f_r, a corresponding float expression of f, where r represents one of the following rounding modes $\{+\infty, -\infty, 0, near\}$

The following proposition, given in [25] with a sketch of proof, is the basis of the conservation of float solutions.

Proposition 5. Assuming every basic operation has an optimal interval extension, if $F : \mathcal{I}^n \rightarrow \mathcal{I}$ is the *natural interval extension* of a real expression $f : \mathcal{R}^n \rightarrow \mathcal{R}$, then for all rounding mode $r \in \{+\infty, -\infty, 0, near\}$ and $\forall \mathbf{I} \in \mathcal{I}^n$, we have $f_r(\mathbf{I}) \subseteq F(\mathbf{I})$,
where $f_r(\mathbf{I}) = \{f_r(\mathbf{v}) \mid \mathbf{v} \in \mathbf{I} \text{ and } \mathbf{v} \in \mathcal{F}^n\}$.
Note that for all $r \in \{+\infty, -\infty, 0, near\}$, $f_{-\infty}(\mathbf{v}) \leq f_r(\mathbf{v}) \leq f_{+\infty}(\mathbf{v})$.

This proposition states that natural interval extensions conserve float solutions, when all the basic operations have an optimal interval extension. Conservation of float solutions here is reflected by the fact that for whatever rounding mode r being used, interval evaluation on interval $\mathbf{I}$, $F(\mathbf{I})$, contains at least all float solutions of $f_r(\mathbf{I})$.

Proposition 5 requires the interval extension of the basic operations to be optimal, what may not be the case for some basic operations (e.g. sin, ln, etc.). The following property extends a property in [19, 25] when the interval extensions of some basic operations are not optimal.

$$\forall \mathbf{I} \in \mathcal{I}^n, [min(f_{-\infty}(\mathbf{I})), max(f_{+\infty}(\mathbf{I}))] \subseteq F(\mathbf{I}) \tag{1}$$

If Property (1) is satisfied by the non-optimal basic operations, then the interval extension F conserves float solutions, because $f_r(\mathbf{I}) \subseteq [min(f_{-\infty}(\mathbf{I})), max(f_{+\infty}(\mathbf{I}))]$, whatever rounding mode r being used.

In practice, it is sometimes difficult to have Property (1) for basic operations when only the rounding mode $near$ is used such as in Java. However, if Properties (2) and (3) below are satisfied, then one can show that float solutions are also preserved. Because for any $\mathbf{I} \in \mathcal{I}^n$,
$[min(f_{-\infty}(\mathbf{I})), max(f_{+\infty}(\mathbf{I}))] \subseteq [min(f^-_{near}(\mathbf{I})), max(f^+_{near}(\mathbf{I}))] \subseteq F(\mathbf{I})$.

$$\forall \mathbf{I} \in \mathcal{I}^n, [min(f^-_{near}(\mathbf{I})), max(f^+_{near}(\mathbf{I}))] \subseteq F(\mathbf{I}) \tag{2}$$

where $f^-_{near}(\mathbf{I}) = \{(f_{near}(\mathbf{v}))^- \mid \mathbf{v} \in \mathbf{I} \text{ and } \mathbf{v} \in \mathcal{F}^n\}$ and $f^+_{near}(\mathbf{I}) = \{(f_{near}(\mathbf{v}))^+ \mid \mathbf{v} \in \mathbf{I} \text{ and } \mathbf{v} \in \mathcal{F}^n\}$.

$$\forall \mathbf{v} \in \mathcal{F}^n, (f_{near}(\mathbf{v}))^- \leq f_{-\infty}(\mathbf{v}) \leq f_{near}(\mathbf{v}) \leq f_{+\infty}(\mathbf{v}) \leq (f_{near}(\mathbf{v}))^+ \tag{3}$$

5.3 Algorithm

Our algorithm for path coverage (given in Algorithm 3) includes the following steps. (1) A path constraint is derived from the specified path of the ICFG (by Function `PathConstraintGeneration` given in the previous section). Such a constraint involves integer, boolean and float variables, as well as operations with arrays. Note that the branch constraints BC mean constraints generated for the branches of the path. They represent the conditions which must be

Algorithm 3 Generation of test data for path coverage

```
function TestDataGenPC(P:Procedure,G:ICFG,p:Path):F^n;
PRE G the ICFG for test procedure P
    p a path in G
POST a test case on which the path p is executed
begin
  PC:= PathConstraintGeneration(P,G,p);
  BC := the branch constraints of PC;
  OC := PC \ BC;
  V := the currently identified input variables in BC;
  D := the domains of the variables in V;
  return SolvePathConstraints(V, V, D, BC, OC);
end

function SolvePathConstraints(V,V':Variables,D:Box,
                              BC:BranchConstraints,OC:OtherConstraints):F^n;
PRE V the currently identified input variables in BC
    V' a subset of V (V' ⊆ V)
    D a box representing the domains of the variables in V
POST Return some vector v ∈ D such that v is a test case for path p
     Otherwise it returns ∅
begin
  (V_t, D_t, BC_t, OC_t) := Filtering(V, D, BC, OC);
  if D_t is ∅ then return ∅;
  else
    if D_t is an ε_box then return FindSolution(BC_t,D_t);
    else
      if V' is not empty then
        Choose arbitrarily a variable x in V';
        m := (left(X_t) + right(X_t))/2;
        if x is an integer variable then
          ms := SolvePathConstraints(V_t, V' \ {x}, D_t[X_t/[⌊m⌋,⌊m⌋]], BC_t, OC_t);
        else ms := SolvePathConstraints(V_t, V' \ {x}, D_t[X_t/[m,m]], BC_t, OC_t);
        if ms ≠ ∅ then return ms;
        if x is an integer variable then
          ls := SolvePathConstraints(V_t, V' \ {x}, D_t[X_t/[left(X_t),⌊m⌋ - 1]], BC_t, OC_t);
        else ls := SolvePathConstraints(V_t, V' \ {x}, D_t[X_t/[left(X_t),m^-]], BC_t, OC_t);
        if ls ≠ ∅ then return ls;
        if x is an integer variable then
          rs := SolvePathConstraints(V_t, V' \ {x}, D_t[X_t/[⌊m⌋ + 1,right(X_t)]], BC_t, OC_t);
        else rs := SolvePathConstraints(V_t, V' \ {x}, D_t[X_t/[m^+,right(X_t)]], BC_t, OC_t);
        if rs ≠ ∅ then return rs else return ∅;
      else return SolvePathConstraints(V_t,V_t,D_t,BC_t,OC_t);
end
```

satisfied so that the path is traversed. The other types of constraints (OC) are used in the simplification of the branch constraints in terms of input variables. (2) The path constraint is solved by an interval-based constraint solving algorithm (Function `SolvePathConstraints`). Function `Filtering` (given below in Algorithm 4) prunes first the path constraint before it is explored further by a domain-splitting. (3) Given a path constraint, the output of the constraint solver is either a (set of) interval solutions of size epsilon (ϵ_box), or that the path constraint has no interval solution. When the path constraint has no interval solution, the path is actually infeasible. When an ϵ_box is returned, a test case is extracted by function `FindSolution`, as specified hereafter.

Specification 1 (FindSolution). Let $\mathcal{C}$ be a path constraint, 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 $\mathbf{v} \in TS$ such that $eval(\mathcal{C}, \mathbf{v})$ holds. Otherwise it returns $\emptyset$.

It is interesting to highlight the main difference between the following two cases, the path constraint with and without arrays. Given a path constraint without arrays, its branch constraints are simplified once for all, in terms of input variables by recursively replacing non-input variables by their definitions in some assignment constraints of the path constraint. These simplified branch constraints together with an initial box (representing the domains of the input variables) are then solved to develop test cases executing the path. However when a path constraint involves arrays, it is not always possible to simplify all of its branch constraints in terms of input variables with the initial box. For example, suppose $a[i]$ (i is an expression involving input variables) is an array reference occurring in a branch constraint, then it is generally impossible to determine which array element $a[i]$ is. Therefore, the branch constraints will be simplified incrementally along with their resolution. The simplification is thus integrated in the filtering (function `Filtering` in Algorithm 4), which in turn is integrated in the path constraint solving (function `SolvePathConstraints`) as illustrated above. Note also that the number of (currently identified) input variables can change over the solving process. Indeed, input variables are defined by a constraint $x \in dom(x)$ (defining input variable x) or $\mathsf{na3}(b, a, j)$ (defining input variable $b[j]$). If j is not a number, $b[j]$ can only be added to the set of input variables when j can be simplified into a number. The function `Filtering` realizes the filtering on the path constraint. The path constraint is represented by the branch constraints and the other constraints. As explained in Section 4, the pruning is only performed on the branch constraints. The function `Simplify` (Algorithm 5) simplifies the branch constraints by extracting information from the other constraints. The number of known input variables may increase after a simplification.

In Algorithm `Filtering`, the branch constraints are first simplified (line 1). The pruning of the branch constraints involving only input variables is performed in line 3. When the resulting box (D_t') is empty, the CSP is inconsistent. If there are branch constraints not involving input variables, these are simplified using the reduced domains. This is performed until $C_1 = \emptyset$ (nothing to prune), or no pruning is achieved ($D_t' = D_t$), or all branch constraints only involve input variables ($C_2 = \emptyset$). Finally the function returns a new CSP (line 5), satisfying (1) $Store$ (all branch constraints involving only input variables) is e-box consistent in box D_t, and (2) C_2 (the other branch constraints involving non-input variables) cannot be simplified further with box D_t.

Algorithm 4 Filtering of path constraints

```
function Filtering(V:Variables,D:Box,BC:BranchConstraints,OC:OtherConstraints):CSP;
PRE
  (V*,D,BC ∧ OC) is a CSP where V* = vars(BC ∧ OC)
  V the set of input variables currently identified in branch constraints BC (V ⊆ V*)
  D a box representing the domains of the variables in V
POST
  Return a CSP (V,∅,BC ∧ OC) if BC is detected as inconsistent.
  Otherwise return (V', D', BC' ∧ OC')
  with • V ⊆ V' ⊆ V*
       • (V*, D', BC' ∧ OC') CSP equivalent to (V*,D,BC ∧ OC)
       • BC' = BC'_1 ∧ BC'_2
       • BC'_1 e-box consistent
begin
1:(V_t, D_t, BC_t, OC_t) := Simplify(V, D, BC, OC);
  C_1 := branch constraints (involving only input variables) of BC_t;
  C_2 := BC_t \ C_1;
  Store := C_1;
2:while C_1 ≠ ∅ do
3:  (V_t, D'_t, Store) := PhiEBox(V_t, D_t, Store);
    if D'_t = ∅ then return (V,∅,BC,OC);
    if D'_t = D_t then break;
    D_t := D'_t;
    if C_2 = ∅ then break;
4:  (V'_t, D'_t, C'_2, OC'_t) := Simplify(V_t, D_t, C_2, OC_t);
    C_1 := branch constraints (involving only input variables) of C'_2;
    C_2 := C'_2 \ C_1;
    Store := Store ∧ C_1;
    V_t := V'_t;  D_t := D'_t; OC_t := OC'_t;
  endwhile
5:return (V_t, D_t, Store ∧ C_2, OC_t);
end
```

We now analyze in detail the function `Simplify`. The function `Simplify` returns an equivalent but simplified CSP. The objective is to simplify the branch constraints BC in terms of the input variables in V with the box D. If BC involves only input variables (line 1), the function returns the input CSP without modifications. Otherwise, it enters in the main loop until no more simplification can be done. The following simplifications are performed. In line 2, every non-input simple variable x is replaced by its definition. A variable is simple if it is neither an array variable nor an array element. Note that there must exists an assignment constraint, $x := def(x)$, for non-input variable x in OC. The replacement of a simple variable by its definition is only carried out in a "symbolic" manner. We illustrate our idea by an example. Assuming x appears in a data structure DS_1, and $def(x)$ is represented by a structure DS_2, then a link is established between DS_1 and DS_2. Therefore, we are not dealing with important-sized expressions as resulted from an actual simplification. This simplification of simple variables is performed only once in the first call of the 5 function. The simplification of array variables is however more complex and must be done incrementally during the solving process.

Lines 4 and 5 simplify the constraints na3 and na4 in OC. Line 6 simplifies reference to array element $b[i]$, where the index is known. Finally, in line 7, every reference to such array element $b[i]$ is propagated in some constraint of OC, which can be one of the following constraints: $b := a$ (this kind of constraints is generated only during parameter passing of array variables), in line 7a; na3 (line 7b); and na4 (line 7c).(See Definition 16 for the definition of *null* arrays.) An inconsistency can be detected when an array element is used in an expression without being initialized.

Algorithm 5 Simplification of path constraints

function Simplify(V:$Variables$, D:Box, BC:$BranchConstraints$, OC:$OtherConstraints$):CSP;
PRE
  $(V^*, D, BC \wedge OC)$ is a CSP with $V^* = vars(BC \wedge OC)$
  V the set of input variables currently identified in branch constraints BC $(V \subseteq V^*)$
  D a box representing the domains of the variables in V
POST
  Return a CSP $(V, \emptyset, BC \wedge OC)$ if BC is detected as inconsistent.
  Otherwise, return $(V', D', BC' \wedge OC')$
  with • $V \subseteq V' \subseteq V^*$
    • $(V^*, D', BC' \wedge OC')$ CSP equivalent to $(V^*, D, BC \wedge OC)$
begin
1:if BC involves only input variables then return (V, D, BC, OC);
  else
2: while $\exists$ a simple and non-input variable x in $BC \wedge OC$ do
      $BC := BC[x/def(x)]$; {There must exists a constraint, $x := def(x)$, in OC}
      $OC := OC[x/def(x)]$; {simplification for variable x once for all}
    $simplify$:= $true$;
3: while $simplify$ do
      $simplify$:= $false$;
4:    foreach constraint na3(b, a, j) in OC with $b[j]$ not in V such that
      j involves only input variables with their domains being point intervals do
        $jval$:= value of j;
        OC[na3(b, a, j)/na3(b, a, $jval$)]; $BC := BC[b[j]/b[jval]]$;
        $V := V \cup \{b[jval]\}$;
        $simplify$:= $true$;
5:    foreach na4(b, a, j, v) in OC such that
      j involves only input variables with their domains being point intervals do
        j is simplified into a number $jval$;
        OC[na4(b, a, j, v)/na4(b, a, $jval$, v)];
        $simplify$:= $true$;
6:    foreach $b[i]$ in BC such that
      i involves only input variables with their domains being point intervals do
        i is simplified into a number $ival$;
        $BC[b[i]/b[ival]]$;
        $simplify$:= $true$;
7:    foreach $b[i]$ in BC such that i is a number and $b[i]$ is not an input variable do
7a: case $\exists$ $(b := a)$ in OC : $BC[b[i]/a[i]]$; $simplify$:= $true$;
7b: case $\exists$ na3(b, a, j) in OC | j is a number :
          if $a \neq null$ then $BC[b[i]/a[i]]$; $simplify$:= $true$; else return $(V, \emptyset, BC, OC)$;
7c: case $\exists$ na4(b, a, j, v) in OC | j is a number :
          if $i = j$ then $BC[b[i]/v]$; $simplify$:= $true$;
          else if $a \neq null$ then $BC[b[i]/a[i]]$; $simplify$:= $true$;
          else return $(V, \emptyset, BC, OC)$;
        endcase
    endwhile
8: return (V, D, BC, OC);
  endif
end

Example

Example 2. As an example for path coverage, with the path given in Example 1 (Section 4) and an initial box ($a_0 : [5, 20], c_0 : [1, 10], i_1 : [-5, 20], j_0 : [-5, 20]$), we obtained the test case: $a_0 = (12.5, 12.5, 12.5, 12.5, 12.5, 12.5, 12.5, 8.75, 12.5, 12.5)$, $c_0 = 1$, $i_1 = 4$, $j_0 = 7$. Note that a_0 (array variable), c_0, i_1, and j_0 are the input variables generated during the path constraint generation.

Analysis Our constraint solving algorithm for path coverage (Algorithm 3) is sound but not complete, because the `FindSolution` function is incomplete. If it does not find test data, it could be that the path is infeasible. Indeed, given an epsilon interval solution, we take only some points in it to check if they are test cases. Of course, we can make a complete labeling in interval solutions, and hence having a complete solver, but then the complexity may become too expensive. However, since path constraints are usually under-constrained (there are many test cases traversing the path if it is feasible), and the epsilon can be chosen very small (usually *1e-16* in our experiments), even a middle point in the interval solution turns out to be sufficient as will be illustrated by our experiments.

The constraint solving problem handled in this work, as well as many other constraint solving problems, is $\mathcal{NP}$-complete, because the SAT problem in computability theory can be reduced to such problems. This means that backtracking search is, in general, an important technique in solving them. As a consequence, our constraint solving algorithm can be considered as belonged to the class of algorithms based on backtracking search with propagation (in Constraint Programming). The propagation is realized here by our e-box consistency filter.

6 Test Data Generation for Statement Coverage

This section proposes an algorithm of test data generation for statement coverage (searching for test data traversing a node of the ICFG). As a branch is dual to a statement in the control flow graph, all the following algorithms can easily be adapted for branch coverage. Given a node, different paths reaching the node will be dynamically generated. The search will be guided by a Control Dependence Graph, as well as pruned by our e-box consistency filter. First, two different control dependences for programs with procedure calls are introduced: the intraprocedural and the interprocedural control dependences. We will show that the interprocedural control dependence is better for our purpose.

6.1 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 [11].

Definition 21. A node V is *post-dominated* by a node W in a CFG G, if every directed path from V to $Exit_G$ (not including V) contains W. A node Y is *control dependent* on node X iff (1) there exists a directed path P from X to

Table 1: Intraprocedural control dependences of Program 1

Nodes	Control Dependent On
3,4,7	(2, true)
4,5a,5b,6	(4, T4)
9,10a,10b,11a,11b,12,15	(8, true)
13a,13b	(12, T12)
14a,14b	(12, F12)
17,20	(16, true)
18	(17, T17)
19	(17, F17)
22,26	(21, true)
23,24,25	(22, T22)

Table 2: Interprocedural control dependences of Program 1

Nodes	Control Dependent On
3,4	(2, true)
5a,8,9,10a,16,17	(4, T4)
7	(4, F4)
13a,13b	(12, T12)
14a,14b	(12, F12)
4,5b,6,10b,11a,11b,12	(17, T17)
15,16,17,18,20,21,22	(17, T17)
19	(17, F17)
23,24,25	(22, T22)

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 node Y is control dependent on node X then node X must have two branches. Following one of the branches results in Y being executed while taking the other results in Y not being executed.

Intraprocedural control dependence analysis is carried out independently on individual procedures, calculating thus control dependences that exist within them. Concretely, given the CFG for each procedure, intraprocedural control dependences for the procedure are obtained by applying an existing algorithm for control dependence computation [11] to the CFG. Table 1 illustrates the intraprocedural control dependences for all procedures of Program 1. Note that (1) the CFGs for those procedures are extracted from the ICFG (in Figure 3) by ignoring, for each call site, its pair of call and return edges, and connecting directly its call node with its return node; (2) we view the entry node of the CFG associated with a procedure as a predicate node representing the conditions that cause the procedure to be executed, and therefore nodes in the CFG that are not control dependent on any condition nodes are control dependent on the entry node. In the table, for example, node 3 is control dependent on node *Entry M* (node 2) with condition *true*, and node 5a on node 4 with condition $T4$.

Interprocedural control dependence analysis accounts for interactions between individual procedures. Those interactions are reflected by call and return edges, connecting the individual CFGs, in the ICFG. Interprocedural control dependence can be computed for the nodes of the ICFG by an existing technique [31]. Table 2 illustrates the interprocedural control dependences for Program 1. A comparison between these dependences and those computed intraprocedurally (in Table 1) shows several differences. (1) There are intraprocedural dependences which are ignored in the interprocedural context, e.g. node 9 is intraprocedurally control dependent on node $EntryB$ (node 8) while this dependence is not interprocedurally necessary. (2) There are interprocedural dependences between nodes in different procedures while these dependences cannot be computed intraprocedurally, e.g. node 6 is interprocedurally control dependent on node 17. Note that the presence of embedded `halt` statements in called procedures are not the only cause of such dependences [31]. (3) There are interprocedural dependences between nodes in the same procedures, yet these dependences are not intraprocedurally established, e.g. node 7 is interprocedurally dependent on node 4 while this is not intraprocedurally detected. All these differences show that intraprocedural control dependences can be imprecise to guide the search of test data for programs with procedure calls. We hence choose to use an interprocedural control dependence graph for this purpose.

Definition 22 (ICDG). An *interprocedural control dependence graph* (ICDG) for a procedure P is a directed graph where the nodes are the nodes of the ICFG associated with P. The edges represent the interprocedural control dependences between nodes. Edges are labeled with conditions. An edge (X, Y) in a ICDG means that Y is interprocedurally control dependent on X. There will be however no edge for a node that is interprocedurally control dependent on itself.

Figure 4 depicts the ICDG for Program 1, which is actually a graphical representation of Table 2. Note that, for simplicity, additional nodes are introduced in the ICDG to group all nodes with the same control conditions together, e.g. nodes 5a,8,9, ... (interprocedurally control dependent on node 4 with condition $T4$) are grouped together under an additional node.

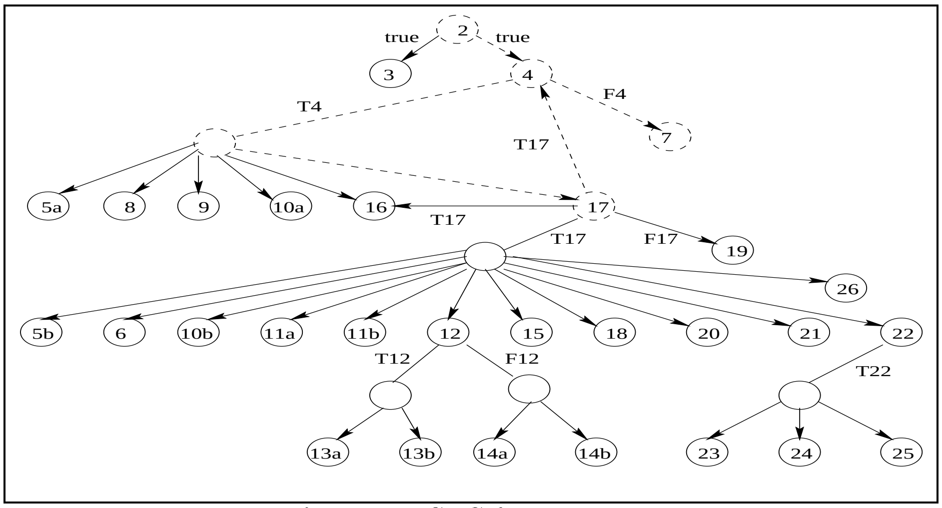

Figure 4: ICDG for Program 1

Definition 23 (Reachability graph). The *reachability graph* for a node n in a directed graph G (with a unique start node) is the smallest subgraph of G, containing all the paths from the start node to node n.

Definition 24 (Decision graph). The *decision graph* for a node n in an ICDG G is the reachability graph for n in G.

The construction of the reachability graph and the decision graph for a node is straightforward. For example, the decision graph for node 7 is depicted in dashed lines in Figure 4. Given the decision graph for a node, a path from the root of the graph to the node contains a set of constraints that must be satisfied by a class of inputs causing the node to be executed. For example, the path 2-4-7 in the decision graph for node 7 corresponds to inputs executing node 7 with no passage in the loop (associated with condition node 4), while the path 2-4-17-4-7 corresponds to inputs executing node 7 with one passage in the loop. Therefore, the decision graph for a node captures all the possible constraints to satisfy to reach the node.

6.2 Algorithm

Algorithm 6 Statement coverage

```
function TestDataGenerationSC(P:Procedure, G:ICFG, N:Node) : F^n;
PRE G The ICFG for test procedure P
    N a node in G
POST a test case traversing node N
begin
  G1 := reachability graph for node N in G;
  G2 := ICDG for G;
  DG := decision graph for node N in G2;
  return TestGen(P, G1, <START>, START, N, DG);
  {START is the start node in G1}
end

function TestGen(P:Procedure, G:ReachabilityGraph,
                 path:Path, start:Node, end:Node, DG:DecisionGraph) : F^n;
PRE path a path in G
   DG the decision graph for node end
POST a test case traversing node end
begin
  for each successor s of start in G do
  {If start in DG, the successors in DG are selected first,}
  {if start is a loop, the exit of the loop is selected first}
    newPath = path . s ;
    PC := PathConstraintGeneration(P,G,newPath);
    BC := the branch constraints of PC;
    OC := PC \ BC;
    V := the input variables currently identified in BC;
    D := the domains of the variables in V;
    (V', D', BC', OC') := Filtering(V, D, BC, OC);
    if (D' ≠ ∅) then
      if (s = end) then
        {test data generation for path coverage}
        result = SolvePathConstraint(V',V',D',BC',OC');
        if result ≠ ∅ then return result;
      else return TestGen(P,G,newPath,s,end,DG);
  endfor
  return ∅;
end
```

The generation of test data for statement coverage is described in Algorithm 6 (`TestDataGenerationSC`). Paths reaching the input node (node N) are dynamically constructed. When a path reaches this node, test data generation for path coverage is used to find a test case. Note that the search for such paths is carried out on the reachability graph for the input node in the ICFG. As the potential number of paths reaching the node can be large (or infinite), heuristics and pruning are used during the search. First, the search is guided by the ICDG, and more particularly by the decision graph. The algorithm always extends a path by first choosing nodes in the decision graph, as such nodes are required in the path. Second, the exit of the loop is also selected first to avoid infinite paths. Third, the search is pruned by our e-box filtering operator (function `Filtering` in Algorithm 4). A path is abandoned as soon as we detect that it is an infeasible path. This algorithm can be optimized in many ways (incremental construction of path constraints, ...). We however prefer to present a simple and comprehensive version.

As an example, consider node 7 in Figure 3, as well as its decision graph shown in dashed lines in Figure 4. First, the path 1-2-3-4-7 will be constructed by the algorithm. Assuming that the corresponding path constraint is inconsistent, a path 1-2-3-4-5a-8- ... (entering in the main loop) is next constructed.

6.3 Analysis

Our algorithm of test data generation for statement coverage is sound but not complete. It may loop or fail to find test data. This follows from the fact that determining whether a node of the control flow graph is executable, is undecidable in the general case (reduced to the halting problem in computability theory) [36]. Also, it was reported in [12] that there exist loops, for which the termination with some data is unknown up till now. So it is impossible to determine if an instruction placed after a loop is executable in the general case.

7 The COTTAGE System

A system written in Java, called COTTAGE, was developed for test data generation of programs written in (a subset of) C. The system is an extension of our previous prototype [33]. It uses an interval arithmetic library [18], based on [19] for the implementation of the constraint solving algorithm, as well as algorithms from [31] to construct the interprocedural control dependences of the test program. Without these libraries, the COTTAGE system is about 13,000 Java lines. The implementation is designed, however, to be independent from the programming language used by the program under test. This means that the source code, written in some imperative language $\mathcal{L}$, is first translated into an internal representation, that are common for all languages, such as C, Pascal, etc. We first describe the architecture of the COTTAGE system.

Implementation A dataflow diagram of the COTTAGE system is given in Figure 5. A single source program (possibly containing multiple procedures) and initial data, such as the domains for the input variables and the name of a test procedure in the source program—are input to the `Parser/Analyzer` component.

The `TestGenerator` component inputs an ICFG and its corresponding ICDG (Interprocedural Control Dependence Graph) from the `Parser/Analyzer` component. It tries to generate test cases for all nodes of the ICFG. This component implements thus our algorithm of test data generation for statement

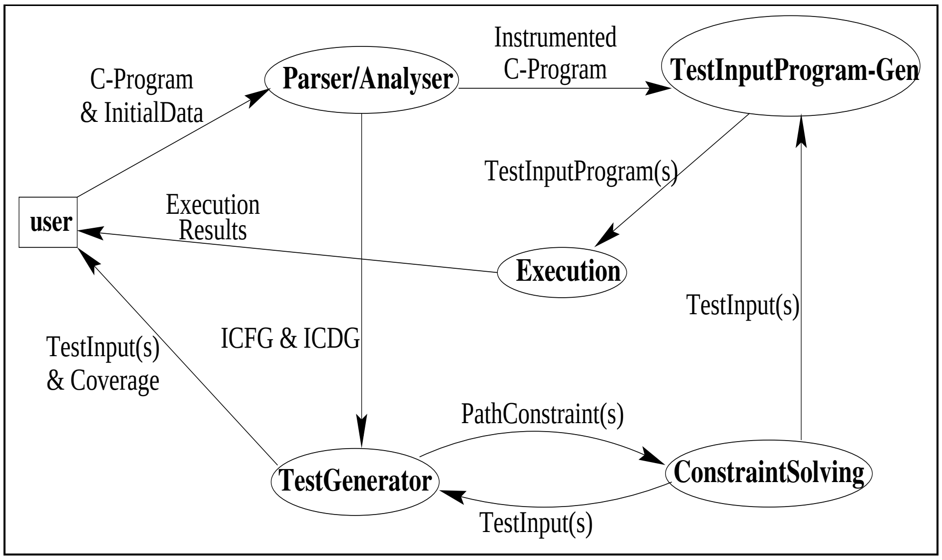

Figure 5: Dataflow diagram of the COTTAGE system

coverage. For each node of the ICFG, a path(s) reaching the node is dynamically constructed. A path constraint is then generated and passed to the `ConstraintSolving` component. The `ConstraintSolving` component generates a test input following Algorithm 5, and passes it to the `TestInputProgram-Gen` component. Since our implementation (more specifically our constraint solving algorithm) is written in Java, the generated test input is thus a float solution, in Java, of the path constraint. Since the program under test is in C, we must also verify that the test input is a float solution, in C, of the path constraint. In other words, we must verify that the test input actually traverses the path on execution of the C program under test. This can be verified by running a corresponding instrumented C program with the test input. The `TestInputProgram-Gen` component therefore receives a test input and an instrumented C-program. It then generates a C program (referred to as a *TestInputProgram*) for running the instrumented C-program on the test input.

The `Execution` component first compiles, and then runs the TestInputProgram. Currently, the user carries out this component, and verifies the execution results for the *actual coverage* of the test inputs generated. However, the component should ideally be automated. And the execution results should be sent back to the `TestGenerator` component. Therefore, a test input not traversing the node will be rejected, and other test inputs will be generated for the node until obtaining a test input actually executing the node (a test case for the node).

Finally, the `TestGenerator` component reports all test inputs found, as well as the *predicted* statement coverage for all the nodes of the ICFG. The predicted coverage means the coverage, calculated without connection with the C language.

FindSolution function (Specification 1) In our implementation, given an epsilon interval-solution, we simply select its middle point to check if it satisfies the path constraint. If so, it is actually a float solution (in Java), and it is returned as a predicted test case for the path. Experiments will show that this simple and efficient implementation turns out to be sufficient. Of course, a general *labeling* strategy, such as described in [25], can also be applied to the epsilon interval solution. The labeling is based on a uniform exploration of the domain. It is parameterised by the number of levels of exploration (labeling level, for short). Figure 6 illustrates this enumeration process on one variable. The numbers correspond to the levels. Using this labeling strategy, on our test

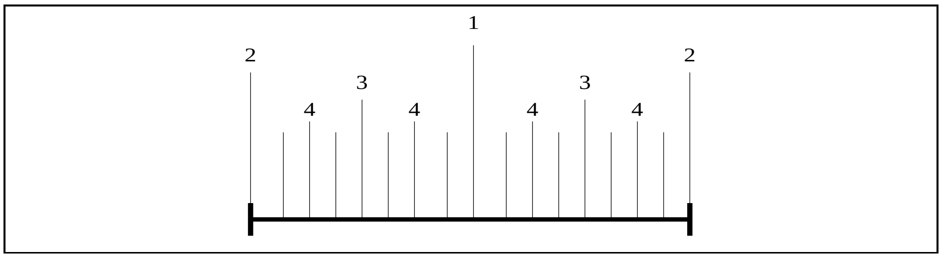

Figure 6: Labeling level

cases, we only observed little change (in time) to find a test case, compared with the default labeling level (one) where only a middle point is chosen.

Conservation of float solutions Function calls to built-in functions such as *exp* (Euler number e raised to the power of a number), *log* (the natural logarithm of a number), *sin*, etc, are treated as basic operators, i.e. these function calls are not developed in the ICFG. The interval extensions of these functions were already available in [18] or constructed in our Java implementation. It is not clear whether the Java implementation in [18] of transcendental functions satisfies the property for conserving float solutions as stated in Subsection 5.2. We however checked this property experimentally. Note that to make our approach work, interval extensions for built-in functions, relations, and operators of the C programming language have been implemented in Java. There is thus a possibility that a float solution found by the constraint solving system is discarded as a solution by the Execution part of the system (executing the instrumented C program).

Parameters A parameter of the COTTAGE system is the size of *epsilon*. The smaller the epsilon is, the more time is required to find an interval solution, but the resulting epsilon interval is more precise, and the `FindSolution` function has more chances to find a solution. Another parameter is a time limit (*timeout*) for solving a path constraint. This allows the system to escape complex (and usually unsound) path constraints. Finally, the *labeling-level* as discussed above is also a parameter of the system.

The subset of C The COTTAGE system is able to generate test data for programs written in a subset of C. The following features are not yet supported by our system: (1) non-numeric types such as string, enum, struct, union, general pointers, ... ; (2) type-qualifiers: `const` and `volatile`; (3) storage-class-specifiers: `auto`, `register`, `static`, `extern` (since we handle only a single source file), `typedef`; (4) labeled-statements such as `case`, `default`, and `switch`; (5) jump-statement `goto` (note that `continue` and `break` statements are both handled); (6) operators: %, <<, >>, &, ∧, |, ?, `sizeof`; (7) control-lines, except `#define` for numeric constants and `#include` that are both handled.

Type analysis Type analysis is important in our system to ensure the precision of interval evaluations. Let us take an example to illustrate this. Suppose an expression $(x + y) * 2.0$, where x and y are integer variables. Then the type of the whole expression is float, while sub-expression $x + y$ is of type integer. Suppose also that x is associated with the interval $[2, 4]$, and y with $[5, 6]$. Since we work with an interval library where the bounds of an interval are floats, $[2, 4]$ and $[5, 6]$ are in fact represented respectively by $[2.0, 4.0]$ and $[5.0, 6.0]$, and the bounds of their intervals are always rounded to integer values.

Soundness and Completeness of COTTAGE Since the system verifies that the generated test inputs actually traverse the corresponding paths on execution of the C program under test, soundness of COTTAGE is ensured. However, three incompleteness cases may arise. (1) The constraint solver provides solution boxes covering all mathematical solutions and float Java solutions of a path constraint. But a float C solution could not be covered by the provided boxes. (2) An (efficient) implementation of `FindSolution` could fail to find a float C solution in an interval. (3) The system could fail to find a solution for a path constraint because of the time limit for solving a path constraint.

Case (1) is a limitation of our approach as we perform the search for a solution in a programming language independent from the program under test. This however provides more flexibility, and allows our system to handle testing with different programming languages. In practice, as illustrated by our experiments, the resulting theoretical limitation has little effect on the system. First, most solvable path constraints have many possible test cases. The objective is to find one test case per node (or branch), not to find all of them. Second, the implementation of the constraint solver is designed to limit this problem. For instance, the basic interval operations preserve the float solutions in Java.

Cases (2) and (3) are necessary limitations to ensure the efficiency of our system. However, as shown by the experiments, the choice of the epsilon value reduce this problem while increasing the overall efficiency of the system.

8 Experiments

We performed our experiments on a 900MHz UltraSparcIII+ machine, with the following programs. `NthRootBisect` [32] calculates the n-th root of a number using the Newton-Raphson method. This program uses integer and float variables, but no arrays nor procedures. `Sample` is the “sample” program with arrays, described in [10]. `Tritype` is a classical program, testing the type of a triangle [28]. It only contains integer variables, but has nested conditional instructions and infeasible paths. `Proc` is the program `program-1` in Figure 1, with nested procedure calls. `BSearch` [10, 13] is a binary search program involving arrays. The `CMichel` program is a small example from [25] with the instruction `if (16.0+x==16 && x>0) return 1; else return 0;`. Although the test is always mathematically false, there exist float values satisfying this test in C. The other examples are real scientific programs taken from [29], and involving math library functions (e.g. *log, exp, pow, sqrt*). The `gaujac` program calculates the Gauss-Jacobi integration formula. This program involves complex (nonlinear) expressions, 3 nested loops, arrays, and procedure calls. The `expint` program [29], which has also been experimented in [16], calculates exponential integrals, and involves nonlinear expressions and nested loops. The `gamdev` program [29] (also experimented in [16]) generates random numbers, and involves nonlinear expressions, nested loops, arrays, and procedure calls. The `bessi` program [29] calculates the modified Bessel functions, and involves procedure

Table 3: Programs and Experimentation results

Programs	Int.	Float	Array	Proc.	Timeout (sec.)	Epsilon	Nodes	Average (sec.)	Max (sec.)	Total (sec.)	Predicted Cover.	Actual Cover.
NthRootBisect	yes	yes	no	no	15	1e-16	10	0.04	0.2	0.4	100%	100%
Sample	yes	no	yes	no	15	1e-16	15	0.02	0.2	0.3	100%	100%
Tritype	yes	no	no	no	15	1e-16	24	0.01	0.1	0.3	100%	100%
Proc	yes	yes	yes	yes	15	1e-16	27	0.03	0.6	0.7	100%	100%
BSearch	yes	yes	yes	no	15	1e-16	10	0.02	0.1	0.2	100%	100%
CMichel	no	yes	no	no	15	1e-16	4	0.02	0.08	0.09	100%	100%
gaujac	yes	yes	yes	yes	15	1e-16	40	3.4	135.9	136.2	100%	100%
expint	yes	yes	no	no	15	1e-16	35	0.47	15.1	16.6	100%	100%
gamdev	yes	yes	yes	yes	15	1e-16	46	0.01	0.3	0.5	100%	100%
bessi	yes	yes	no	yes	15	1e-16	27	0.06	1.6	1.62	100%	100%
ei	yes	yes	no	no	15	1e-16	22	0.14	1.3	3	95%	95%
ei	yes	yes	no	no	15	1e-32	22	0.13	1.3	2.8	100%	100%
ei-dead	yes	yes	no	no	15	1e-32	27	0.14	1.9	3.8	92%	92%

calls. The `ei` program [29] also calculates exponential integrals, and involves a very small constant (*1e-30*). Finally, `ei-dead` is the `ei` program extended with two unreachable statements, in and after the main loop. The C code of these programs is given in the appendix. Table 3 summarizes these programs and the experimental results. Note that the value of the parameter *labeling-level*, used in these experiments, is one by default.

Our test generation procedure consists in trying to generate a test case for each node of the ICFG, and then reporting the *predicted statement coverage* (the percentage of nodes for which the constraint solving algorithm found a float Java solution) and the *actual coverage* (the percentage of nodes for which the float Java solution is a test case of the C program). Note that when a test data is generated for a node, we also obtain a path traversing the node. All other nodes involved in the path are then marked as covered by the same test data. For each program, Table 3 lists the values of the parameters epsilon (size of the interval solutions) and timeout (timeout for solving a path constraint), the number of nodes of the corresponding ICFG (Nodes), the average time in seconds spent on a node (Average), the maximum time in seconds spent on a node (Max), the total time in seconds to generate test cases for all the nodes (Total), the predicted statement coverage (Predicted Coverage), and the actual statement coverage (Actual Coverage).

Efficiency Even with the complex scientific programs, `gaujac`, `expint`, `gamdev`, `bessi` and `ei`, the time performance indicates that our method is practical. It is difficult to provide a time complexity analysis as the general problem of solving a set of constraints is NP-hard. Efficiency should therefore be measured on specific classes of problems. Moreover, choosing a "good" path reaching a node, where one quickly gets a test case, is another problem. Indeed, at a decision node, where its two successors have the same priority to be chosen during the path construction, the choice of the next successor has a great influence on the time complexity. Taking the `gaujac` program as an example, its related results reported in Table 3 correspond to the default behavior of our test data generator. However, when we change the branch taken by default at a node, the time needed to cover all the nodes is only 3.6 seconds! The speed-up here is thus around 38.7. Note that our implementation allows us to observe all paths reaching a node during the path construction, as well as to specify at specific node, the strategy for the path generation.

Coverage and completeness For programs without dead code, the COTTAGE system is able to achieve 100% coverage on all the experimented programs, even the complex scientific ones. On all our experiments, the actual coverage is also the predicted coverage. This illustrates that the completeness issue raised in the previous section is, in practice, not really problematic. The `CMichel` example illustrates that the constraint solver is also able to find non mathematical solutions (here it found the value *x=1.3322...E-15* for the test). The coverage of the `ei-dead` example is only 92% because 2 nodes are unreachable; they have been detected by the system.

The achieved coverage justifies our approach to use an implemention language (Java) different from the language of the tested programs (C). This generic approach allows us to use the COTTAGE system for testing programs in other programming languages.

Procedures Our interprocedural control dependence analysis, as described in [33], enables the system to handle procedures with greater precision than the classical intraprocedural analysis. In the `proc` example, the *halt* statement in a nested procedure is handled without any problem.

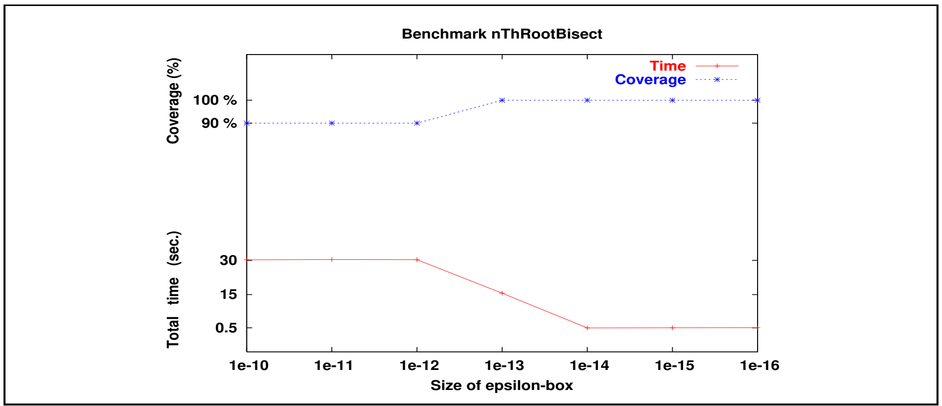

Figure 7: Experiments with `nThRootBisect`

Choosing the parameters For some programs, the default value of the parameters (epsilon and timeout) has to be adapted to achieve completeness. The epsilon value may influence not only the coverage, but also the execution time, as illustrated in Figure 7 on the `nThRootBisect` example. With small epsilon values, our implementation of `FindSolution` with the default *labeling-level* (choosing the middle point) has more chance to find a float Java solution. With larger epsilon values, the system could generate many interval solutions before `FindSolution` finds a float Java solution, increasing thus the execution time, or reducing the coverage. More sophisticated implementation of `FindSolution` could be designed, but this is not a central point given the efficiency of the constraint solver for small epsilon values.

The `ei` program, involving a *1e-30* constant illustrates the necessity to choose the epsilon value according the constants used in the program. The default *1e-16* epsilon value achieves only 95% coverage while a *1e-32* value achieves 100% coverage. The execution time is even better (pruning is slightly more efficient).

For some programs such as `expint`, increasing the timeout parameter can affect the execution time. A too small timeout could also reduce the coverage.

Complex paths The `expint` and `ei` programs raise an error if no solution is found after `MAXIT` iterations. Our system is of course not able to find a test case achieving this error statement with the original value of `MAXIT` (100) as

Table 4: A summary of test data generators

Methods	Ref.	Int.	Float	Arrays	Proc.	Path Coverage	Statement Coverage
Consistency	*this*	*yes*	*yes*	*yes*	*yes*	*yes*	*yes*
Testgen	[23]	yes	yes	yes	yes	yes	yes
Relaxation	[16]	yes	yes	yes	yes	yes	yes
InKa	[13]	yes	no	yes	yes	no	yes
Genetic	[28]	yes	yes	yes	no	no	yes
Symbolic	[5]	yes	yes	yes[1]	yes	yes	no

[1] Array references depending on input variables are not handled

the number and the complexity of the path constraints is too high. In the experiments reported in Table 3, the value of `MAXIT` is 10 for `expint`, and 17 for `ei`.

Initial domain values The choice of the initial domains of the input variables may influence the coverage and the efficiency of the system. A too small initial domain could not covered some of the nodes. A large domain may increase the computation time. In our experiments, we choose large initial domains to favor coverage. In the `expint` program for instance, the initial domains are $n = [1, 30]$ and $x = [-1000, 1000]$. For `ei`, we set $x = [-1, 10000]$ because $x < 0$ is an error case.

Comparison with existing methods Table 4 summarizes the existing methods with functionalities close to our method (first line in the table). Two other methods offer the same functionalities, [23] and [16]. As in these methods, our prototype is able to achieve 100% coverage on the examples, but our set of examples contains more complex programs. It is difficult to compare the efficiency of the different methods because efficiency information is sometimes partial or missing. When this information is available, the measures can be incomparable (number of iterations versus execution time versus theoretical complexity). When it is comparable, one should consider the differences in the underlying hardware.

In [16], an execution time of 98 and 42 seconds (Windows NT, 400MHz Pentium II) is reported to find test data for two branches of the benchmark `expint`, and an execution time of 117.4 seconds to find test data for one branch of the benchmark `gamdev`. For the `expint` program, our system generates 10 path constraints to achieve a full coverage in 16.6 seconds; and for the `gamdev` program, our system generates 6 path constraints to achieve a full coverage in 0.5 seconds.

The experimented scientific programs taken from [29] are so far the most complex (in terms of the complexity of expressions) of our examples. In the literature, we rarely find such complex examples used by other methods. More importantly, our system strengthens the possibility of applying constraint programming for test data generation as stated in [13].

9 Conclusion

In this paper, we first presented our consistency approach for test data generation of imperative programs with float, integer and boolean variables, as well as procedure calls and arrays. Test programs (with procedure calls) are represented by an interprocedural control flow graph (ICFG). The testing criteria (path, statement, and branch coverage) are then defined in terms of the ICFG. Our purpose was thus to generate test data that will cause the program to traverse a specified path, node, or branch of the ICFG. For path coverage, the search for test data is reduced to the solving of path constraints. Such a solving is based on consistency techniques, aiming at reducing the domains of the variables. The main originality of our method is a constraint solver, dealing with float, integer, and boolean variables for test data generation. For statement coverage, suitable paths reaching the specified node are dynamically constructed. The search is guided by the interprocedural control dependence graph, as well as pruned by our e-box consistency filter. When such a path is found, our algorithm for path coverage is then applied.

We then presented our COTTAGE system, a 13,000 Java lines software, implementing our method, for test data generation of C programs. Various experiments, including complex programs (in terms of nonlinear expressions) taken from the numerical computing book [29], have been reported. These experiments showed the coverage of our system, as well as its versatility and flexibility to different classes of problems (integer and/or float variables; arrays, procedures, path coverage, statement coverage). They demonstrate the feasibility of the method, its efficiency and its potential to handle complex programs.

Our constraint solver could be combined with existing approaches based on dynamic methods (e.g. [16, 28]), especially when searching a test data exercising a specified statement of the program. Future work includes the development of different strategies for the `FindSolution` function such as using `local search` in epsilon interval solutions. Extension of the considered subset of C, such as pointers, will be investigated. The possibility of error detection will also be considered, by adding new kinds of constraints modeling error conditions such as in [30].

References

[1] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1986.

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

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

[4] 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.

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

[6] Alberto Coen-Porisini and Flavio De Paoli. Array representation in symbolic execution. *Computer Languages*, 18(3):197–216, 1993.

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

[8] John Doppke and Artur Klauser. Pl concepts: Parameter passing. Available at http://www.cs.colorado.edu/~humphrie/pl/param.html.

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

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

[11] 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.

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

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

[14] 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.

[15] 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.

[16] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. Generating test data for branch coverage. In *15th IEEE International Conference on Automated Software Engineering(ASE00)*, September 2000.

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

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

[19] Timothy J. Hickey, Qun Ju, and Maarten H. van Emden. Interval arithmetic: From principles to implementation. *Journal of the ACM*, 48(5):1038–1068, 2001.

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

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

[22] Bogdan Korel. Automated software test data generation. *IEEE Trans. Software Eng.*, 16(8):870–879, 1990.

[23] Bogdan Korel. Automated test data generation for programs with procedures. In Steven J. Ziel, editor, *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*, pages 209–215, 1996.

[24] David Melski and Thomas W. Reps. Interprocedural path profiling. In *Computational Complexity*, pages 47–62, 1999.

[25] C. Michel, M. Rueher, and Y. Lebbah. Solving constraint over floating-point numbers. In *Seventh International Conference on Principles and Practice of Constraint.* Springer Verlag, LNCS, 2001.

[26] R.E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, NJ, 1966.

[27] 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.

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

[29] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C. The Art of Scientific Computing. Second Edition.* Cambridge University Press, 1992.

[30] Debra J. Richardson and Margaret C. Thompson. An analysis of test data selection criteria using the RELAY model of fault detection. *IEEE Transactions on Software Engineering*, 19(6):533–553, June 1993.

[31] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. Interprocedural control dependence. *Software Engineering and Methodology*, 10(2):209–254, 2001.

[32] Nguyen Tran Sy and Yves Deville. Automatic test data generation for programs with integer and float variables. In *16th IEEE International Conference on Automated Software Engineering(ASE01)*, November 2001.

[33] Nguyen Tran Sy and Yves Deville. Consistency techniques for interprocedural test data generation. In *Proceedings of the ESEC/FSE'03*, pages 108–117, 2003.

[34] Nigel Tracey, John Clark, Keith Mander, and John A. McDermid. An automated framework for structural test-data generation. In *ASE'98*, pages 285–288, 1998.

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

[36] Elaine J. Weyuker. Translatability and decidability questions for restricted classes of program schemas. *SIAM J. Comput.*, 8(4):587–598, 1979.

APPENDIX

A Benchmarks

A.1 NthRootBisect.c

```
#include <math.h>

double nThRootBisect(double a, int n, double e) {
  double l, h, c;

  l = 1; h = a;
  while ((h - l)*(h - l) >= e) {
    c = (l + h)/2;
    if (pow(c, n) - a == 0) return c;
    if ((pow(l, n) - a)*(pow(c, n) - a) < 0) h = c;
    else l = c;
  }
  return h;
}
```

A.2 Sample.c

```
int sample(int a[10], int b[10], int target) {
  int i, fa, fb;

  i = 0;
  fa = 0;
  fb = 0;
  while (i <= 9) {
    if (a[i] == target) fa = 1;
    ++i;
  }
  if (fa == 1) {
    i = 0;
    fb = 1;
    while (i <= 9) {
      if (b[i] != target) fb = 0;
      ++i;
    }
  }
  if (fb == 1) return 0;
  else return 1;
}
```

A.3 Tritype.c

```
int tritype(int i, int j, int k) {
  int trityp;

  if ((i == 0) || (j == 0) || (k == 0)) trityp = 4;
```

```
  else {
    trityp = 0;
    if (i == j) trityp = trityp+1;
    if (i == k) trityp = trityp+2;
    if (j == k) trityp = trityp+3;
    if (trityp == 0) {
      if ((i+j <= k)||(j+k <= i)||(i+k <= j)) trityp = 4;
      else trityp = 1;
    }
    else if (trityp > 3) trityp = 3;
    else if ((trityp == 1) && (i+j > k)) trityp = 2;
    else if ((trityp == 2) && (i+k > j)) trityp = 2;
    else if ((trityp == 3) && (j+k > i)) trityp = 2;
    else trityp = 4;
  }
  return trityp;
}
```

A.4 Proc.c

```
#include <stdio.h>

void b(double a[10]);
void c(double *x, double *y);
int f(int i);

void proc(double a[10], int c) {
  int i;

  i = 1;
  while (i <= c) {
    b(a);
    ++i;
  }
}

void b(double a[10]) {
  int i, j, fi, fj;

  printf("i j ? "); scanf("%d %d", &i, &j);
  fi = f(i);
  fj = f(j);
  if (fi < fj) c(&a[i], &a[j]);
  else c(&a[j], &a[i]);
}

void c(double *x, double *y) {
  double t;

  if (*x > *y) {
    t = *x;
```

```
    *x = *y;
    *y = t;
  }
}

int f(int i) {
  if (i >= 0 && i <= 9) return i;
  else exit(1);
}
```

A.5 BSearch.c

```
int bsearch(double a[10], double elem) {
  int low = 0;
  int high = 9;
  int mid;
  while (high >= low) {
    mid = (low + high)/2;
    if (elem == a[mid]) return 1;
    if (elem > a[mid]) low = mid + 1;
    else high = mid - 1;
  }
  return 0;
}
```

A.6 CMichel.c

```
int cMichel(double x) {
  if (16.0+x == 16.0 && x > 0) return 1;
  else return 0;
}
```

A.7 gaujac.c

```
#include <math.h>
#define EPS 3.0e-14
#define MAXIT 10
#define N 6

double gammln(double xx);

void gaujac(double x[N], double w[N], double alf, double bet) {
  int i,its,j;
  double alfbet,an,bn,r1,r2,r3;
  double a,b,c,p1,p2,p3,pp,temp,z,z1,gl1,gl2,gl3,gl4;

  for (i=1;i<=N;++i) {
    if (i == 1) {
      an=alf/N;
      bn=bet/N;
      r1=(1.0+alf)*(2.78/(4.0+N*N)+0.768*an/N);
```

```
    r2=1.0+1.48*an+0.96*bn+0.452*an*an+0.83*an*bn;
    z=1.0-r1/r2;
  } else if (i == 2) {
    r1=(4.1+alf)/((1.0+alf)*(1.0+0.156*alf));
    r2=1.0+0.06*(N-8.0)*(1.0+0.12*alf)/N;
    r3=1.0+0.012*bet*(1.0+0.25*fabs(alf))/N;
    z -= (1.0-z)*r1*r2*r3;
  } else if (i == 3) {
    r1=(1.67+0.28*alf)/(1.0+0.37*alf);
    r2=1.0+0.22*(N-8.0)/N;
    r3=1.0+8.0*bet/((6.28+bet)*N*N);
    z -= (x[0]-z)*r1*r2*r3;
  } else if (i == N-1) {
    r1=(1.0+0.235*bet)/(0.766+0.119*bet);
    r2=1.0/(1.0+0.639*(N-4.0)/(1.0+0.71*(N-4.0)));
    r3=1.0/(1.0+20.0*alf/((7.5+alf)*N*N));
    z += (z-x[N-4])*r1*r2*r3;
  } else if (i == N) {
    r1=(1.0+0.37*bet)/(1.67+0.28*bet);
    r2=1.0/(1.0+0.22*(N-8.0)/N);
    r3=1.0/(1.0+8.0*alf/((6.28+alf)*N*N));
    z += (z-x[N-3])*r1*r2*r3;
  } else {
    z=3.0*x[i-2]-3.0*x[i-3]+x[i-4];
  }

  alfbet=alf+bet;
  for (its=1;its<=MAXIT;++its) {
    temp=2.0+alfbet;
    p1=(alf-bet+temp*z)/2.0;
    p2=1.0;
    for (j=2;j<=N;++j) {
      p3=p2;
      p2=p1;
      temp=2*j+alfbet;
      a=2*j*(j+alfbet)*(temp-2.0);
      b=(temp-1.0)*(alf*alf-bet*bet+temp*(temp-2.0)*z);
      c=2.0*(j-1+alf)*(j-1+bet)*temp;
      p1=(b*p2-c*p3)/a;
    }
    pp=(N*(alf-bet-temp*z)*p1+2.0*(N+alf)*(N+bet)*p2)/(temp*(1.0-z*z));
    z1=z;
    z=z1-p1/pp;
    if (fabs(z-z1) <= EPS) break;
  }
  if (its > MAXIT) {
    printf("too many iterations in gaujac\n");
    exit(1);
  }
  x[i-1]=z;
  gl1 = gammln(alf+N);
```

```
    gl2 = gammln(bet+N);
    gl3 = gammln(N+1.0);
    gl4 = gammln(N+alfbet+1.0);
    w[i-1]=exp(gl1+gl2-gl3-gl4)*temp*pow(2.0,alfbet)/(pp*p2);
  }
}

double gammln(double xx) {
  double x,y,tmp,ser;
  static double cof[6]={76.18009172947146,-86.50532032941677,
                        24.01409824083091,-1.231739572450155,
                        0.1208650973866179e-2,-0.5395239384953e-5};
  int j;

  y=x=xx;
  tmp=x+5.5;
  tmp -= (x+0.5)*log(tmp);
  ser=1.000000000190015;
  for (j=0;j<=5;++j) ser += cof[j]/++y;
  return -tmp+log(2.5066282746310005*ser/x);
}
```

A.8 expint.c

```
#include <math.h>
#define MAXIT 10
#define EULER 0.5772156649
#define FPMIN 1.0e-30
#define EPS 1.0e-7

double expint(int n, double x) {
  int i,ii,nm1;
  double a,b,c,d,del,fact,h,psi,ans;

  nm1=n-1;
  if (n < 0 || x < 0.0 || (x==0.0 && (n==0 || n==1)))
    exit(1);
  else {
    if (n == 0) ans=exp(-x)/x;
    else {
      if (x == 0.0) ans=1.0/nm1;
      else {
        if (x > 1.0) {
          b=x+n;
          c=1.0/FPMIN;
          d=1.0/b;
          h=d;
          for (i=1;i<=MAXIT;++i) {
            a = -i*(nm1+i);
            b += 2.0;
            d=1.0/(a*d+b);
```

```
            c=b+a/c;
            del=c*d;
            h *= del;
            if (fabs(del-1.0) < EPS) {
              ans=h*exp(-x);
              return ans;
            }
          }
          exit(1);
        } else {
          if (nm1 != 0) ans = 1.0/nm1; else ans = -log(x)-EULER;
          fact=1.0;
          for (i=1;i<=MAXIT;++i) {
            fact *= -x/i;
            if (i != nm1) del = -fact/(i-nm1);
            else {
              psi = -EULER;
              for (ii=1;ii<=nm1;++ii) psi += 1.0/ii;
              del=fact*(-log(x)+psi);
            }
            ans += del;
            if (fabs(del) < fabs(ans)*EPS) return ans;
          }
          exit(1);
        }
      }
    }
  }
  return ans;
}
```

A.9 gamdev.c

```
#include <math.h>

#define IA 16807
#define IM 2147483647
#define AM 4.656612875245797e-10
#define IQ 127773
#define IR 2836
#define NTAB 32
#define NDIV 67108864
#define EPS 1.2e-7
#define RNMX 0.99999988

long iy=0;
long iv[NTAB];
double ran1(long *idum);

double gamdev(int ia, long *idum) {
  int j;
```

```
  double am,e,s,v1,v2,x,y;
  double r1,r2,r3;

  if (ia < 1) exit(1);
  if (ia < 6) {
    x=1.0;
    for (j=1;j<=ia;++j) {
      r1 = ran1(idum);
      x *= r1;
    }
    x = -log(x);
  } else {
    do {
      do {
        do {
          v1=ran1(idum);
          r2 = ran1(idum);
          v2=2.0*r2-1.0;
        } while (v1*v1+v2*v2 > 1.0);
        y=v2/v1;
        am=ia-1;
        s=sqrt(2.0*am+1.0);
        x=s*y+am;
      } while (x <= 0.0);
      e=(1.0+y*y)*exp(am*log(x/am)-s*y);
      r3 = ran1(idum);
    } while (r3 > e);
  }
  return x;
}

double ran1(long *idum) {
  int j;
  long k;
  double temp;

  if (*idum <= 0 || iy == 0) {
    if (-(*idum) < 1) *idum=1;
    else *idum = -(*idum);
    for (j=NTAB+7;j>=0;--j) {
      k=(*idum)/IQ;
      *idum=IA*(*idum-k*IQ)-IR*k;
      if (*idum < 0) *idum += IM;
      if (j < NTAB) iv[j] = *idum;
    }
    iy=iv[0];
  }
  k=(*idum)/IQ;
  *idum=IA*(*idum-k*IQ)-IR*k;
  if (*idum < 0) *idum += IM;
  j=iy/NDIV;
```

```
  iy=iv[j];
  iv[j] = *idum;
  temp=AM*iy;
  if (temp > RNMX) return RNMX;
  else return temp;
}
```

A.10 bessi.c

```
#include <math.h>
#define ACC 40.0
#define BIGNO 1.0e10
#define BIGNI 1.0e-10

double bessi0(double x);

double bessi(int n, double x) {
  int j;
  double bi,bim,bip,tox,ans,bsi0x;

  if (n < 2) exit(1);
  if (x == 0.0)
    return 0.0;
  else {
    tox=2.0/fabs(x);
    bip=ans=0.0;
    bi=1.0;
    for (j=2*(n+(int)sqrt(ACC*n));j>0;--j) {
      bim=bip+j*tox*bi;
      bip=bi;
      bi=bim;
      if (fabs(bi) > BIGNO) {
        ans *= BIGNI;
        bi *= BIGNI;
        bip *= BIGNI;
      }
      if (j == n) ans=bip;
    }
    bsi0x = bessi0(x);
    ans *= bsi0x/bi;
    if (x < 0.0 && n != ((int)n/2)*2) return -ans;
    else return ans;
  }
}

double bessi0(double x) {
  double ax,ans,y;

  ax=fabs(x);
  if (ax < 3.75) {
    y=x/3.75;
```

```
    y*=y;
    ans=1.0+y*(3.5156229+y*(3.0899424+y*(1.2067492
        +y*(0.2659732+y*(0.360768e-1+y*0.45813e-2)))));
  } else {
    y=3.75/ax;
    ans=(exp(ax)/sqrt(ax))*(0.39894228+y*(0.1328592e-1
        +y*(0.225319e-2+y*(-0.157565e-2
        +y*(0.916281e-2+y*(-0.2057706e-1+y*(0.2635537e-1+y*(-0.1647633e-1
        +y*0.392377e-2))))))));
  }
  return ans;
}
```

A.11 ei.c

```
#include <math.h>
#include <stdio.h>

#define EULER 0.57721566
#define MAXIT 17
#define FPMIN 1.0e-30
#define EPS 6.0e-8

double ei(double x) {
  int k;
  double fact,prev,sum,term;

  if (x <= 0.0) exit(1);
  if (x < FPMIN) return log(x)+EULER;
  if (x <= -log(EPS)) {
    sum=0.0;
    fact=1.0;
    for (k=1;k<=MAXIT;++k) {
      fact *= x/k;
      term=fact/k;
      sum += term;
      if (term < EPS*sum) break;
    }
    if (k > MAXIT) exit(1);
    return sum+log(x)+EULER;
  } else {
    sum=0.0;
    term=1.0;
    for (k=1;k<=MAXIT;++k) {
      prev=term;
      term *= k/x;
      if (term < EPS) break;
      if (term < prev) sum += term;
      else {
        sum -= prev;
        break;
```

```
      }
    }
    return exp(x)*(1.0+sum)/x;
  }
}
```

A.12 ei-dead.c

```
#include <math.h>
#include <stdio.h>
#define EULER 0.57721566
#define MAXIT 17
#define FPMIN 1.0e-30
#define EPS 6.0e-8

double eiDead(double x) {
  int k;
  double fact,prev,sum,term;

  if (x <= 0.0) exit(1);
  if (x < FPMIN) return log(x)+EULER;
  if (x <= -log(EPS)) {
    sum=0.0;
    fact=1.0;
    for (k=1;k<=MAXIT;++k) {
      fact *= x/k;
      if (k > MAXIT) return 0;
      term=fact/k;
      sum += term;
      if (term < EPS*sum) break;
    }
    if (k > MAXIT) exit(1);
    if (term >= EPS*sum) return 0;
    return sum+log(x)+EULER;
  } else {
    sum=0.0;
    term=1.0;
    for (k=1;k<=MAXIT;++k) {
      prev=term;
      term *= k/x;
      if (term < EPS) break;
      if (term < prev) sum += term;
      else {
        sum -= prev;
        break;
      }
    }
    return exp(x)*(1.0+sum)/x;
  }
}
```