Consistency Techniques for Interprocedural Test Data Generation

Nguyen Tran Sy
tsn@info.ucl.ac.be

Yves Deville
yde@info.ucl.ac.be

Computing Science and Engineering Department
Université catholique de Louvain
Place Saint-Barbe 2
B-1348 Louvain-la-Neuve, Belgium

ABSTRACT

This paper presents a novel approach for automated test data generation of imperative programs containing *integer*, *boolean* and/or *float* variables. It extends our previous work to programs with procedure calls and arrays. 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. For path coverage, the specified path is transformed into a *path constraint*. Our previous consistency techniques, the core idea behind the solving of path constraints, have been extended to handle procedural calls and operations with arrays. For statement (and branch) coverage, paths reaching the 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 a new specialized consistency filter. Finally, test data are generated by the application of the proposed path coverage algorithm. A prototype has been implemented. Experiments show the feasibility of the approach.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Experimentation, Algorithms, Measurement, Performance

Keywords

software testing, test data generation, procedures, arrays, constraint satisfaction, consistency

1. INTRODUCTION

Structural testing techniques are usually concerned with the use of the control-flow a program to guide the generation of test data. The control-flow, in turn, is represented by a Control Flow Graph (CFG). To adequately test the program at the structural level, we must consider structural elements (nodes, branches, or paths) of the CFG for coverage. For example, *statement coverage* 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) choice of a criterion (statement, branch or path), (2) identification of a set of nodes, branches or paths, and (3) generation of test data for each element of this set. The automation of the last phase is a vital challenge in software testing.

Classical testing approaches can be classified into the following categories. *Random* test data generation [2] consists in trying test data generated randomly until an element is executed. Many experiences have shown however that it can be very inefficient to generate test data for statement coverage, for example. *Symbolic evaluation* [10, 1] consists in replacing input variables by symbolic values, and then symbolically evaluates the statements along a path. It is however limited in handling arrays[1] and procedure calls. *Program execution based* (or *dynamic*) approaches start by executing the program with an arbitrary test input. This input is then iteratively refined, by execution of the program, to obtain a final input, executing a path [6], a branch [7], or a statement [3]. The results in [3] are extended in [11] to programs with procedures by considering the possible effect of statements in the called procedures on execution of the selected element. Although dynamic approaches are powerful in handling arrays and dynamic data structures, it may require a great number of executions of the program. Another approach [15] uses *genetic algorithms* to guide the search process; it is however restricted to programs without procedure calls. An approach, based on *Constraint Logic Programming* (CLP) techniques, has been proposed in [5], where statement coverage is handled. 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,(although a constraint solver over float numbers has recently been proposed in [13]). Procedure calls are handled, but only the pass-by-value mechanism for

[1] Consider, for instance, an array element $A[i]$. Since i may depend on input variables, it is then impossible to determine which array element is referenced.

ESEC/FSE'03, September 1–5, 2003, Helsinki, Finland.

passing parameters is considered. Only intraprocedural control dependences of the test program are used in the search process, even when it contains procedure calls. Therefore, this is not precise for certain classes of programs as will be shown later. *Interval Arithmetic* [14] has been exploited for test data generation in [18] and in our previous work [20]. The branch-and-bound technique proposed in [18] is limited to programs without loops and does not handle arrays, nor procedure calls. A summary of existing approches with functionalities close to our method is given in the Experimentation Section (Table 5).

In the following problem statement, an Interprocedural Control Flow Graph (ICFG) is a classical representation of programs. A precise definition will be given in Section 2.

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.*

We propose a novel consistency-based approach for interprocedural test data generation. Statement, branch and path coverage criteria are all handled. Path coverage is the core of our approach. It includes the following steps. (1) A path constraint is derived from a specified path of the ICFG. Such a constraint can involve operations with arrays. (2) The path constraint is solved by a new specialized interval-arithmetic-based constraint solver extended to handle constraints involving arrays. (3) A test case is extracted from the interval solutions.

For statement (and branch) coverage, paths reaching the specified node or branch are dynamically constructed. Our algorithm for path coverage is then applied on these paths to generate test data.

Contribution. The main contribution of the paper is a novel approach (based on consistency techniques), which is an extension of our previous paper [20] to generate test data for numeric programs (programs with integer, boolean and float variables) with procedure calls and arrays. This approach handles *branch*, *statement* and *path* coverage criteria. Specific technical contributions of the paper include the following. (1) A new method to obtain a path constraint directly from a path's traversal. (2) Two mechanisms for passing parameters (pass-by-value and pass-by-reference) in procedure calls are handled. (3) An improvement on our previous consistency techniques to tackle specific constraints involving arrays. (4) The proposed interval constraints solver integrates integers, reals, and booleans, as well as the logical operators AND, OR, NOT. (5) For statement and branch coverage, interprocedural control dependences are used during the search process, when the test procedure contains procedure calls.

Organization. The organization of the paper is as follows. The background is presented in the next section. Section 3 illustrates the generation of path constraints. Section 4 describes a test data generation algorithm for path coverage, while Section 5 proposes an algorithm for statement coverage. Experiments with our prototype are shown in Section 6. Conclusions are finally presented in Section 7.

2. BACKGROUND

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. 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.

In the program given in Figure 1, the parameters declared with the `var` keyword are pass-by-reference parameters while the other parameters are pass-by-value parameters. Figure 2 shows the transformed procedure `B` without embedded function calls. In the sequel, when we refer to a program, we mean an equivalent one without embedded function calls.

Interprocedural Control Flow Graph. A *control flow graph* (CFG) for procedure P is a directed graph, where the nodes represent statements and predicates (conditional and loop statements) of P, except that each procedure call and function calls are represented by two nodes, a *call node* and a *return node*; the edges represent possible flow of control between nodes. The CFG also 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 predicate statement, is called a *predicate node*. An outgoing edge from a predicate node is called a *branch*. Each branch of the CFG is associated with a *condition*.

An *interprocedural control flow graph* (ICFG) [19, 12] for 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(...)` (`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(...)`.

```
M(var real a[10], int c)        B(var real a[10])
begin M                         begin B
  int i = 1;                      int i,j;
  while i <= c do                 read i,j;
    B(a);                         if F(i) < F(j) then
    i = i + 1;                      C(a[i], a[j]);
  endwhile                        else C(a[j], a[i]);
end M                             endif
                                end B
C(var real x, var real y)
begin C                         int F(int i)
  real t;                       begin F
  if x > y then                   if i >= 0 & i <= 9 then
    t = x;                          return i;
    x = y;                        else halt;
    y = t;                        endif
  endif                         end F
end C
```

Figure 1: Program 1

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 predicate 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 ($Fi = Ti$).

```
B(var real a[10])
begin B
  int i,j,fi,fj;
  read i,j;
  fi = F(i); fj = F(j);
  if (fi < fj) then
    C(a[i], a[j]);
  else C(a[j], a[i]);
  endif
end B
```

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

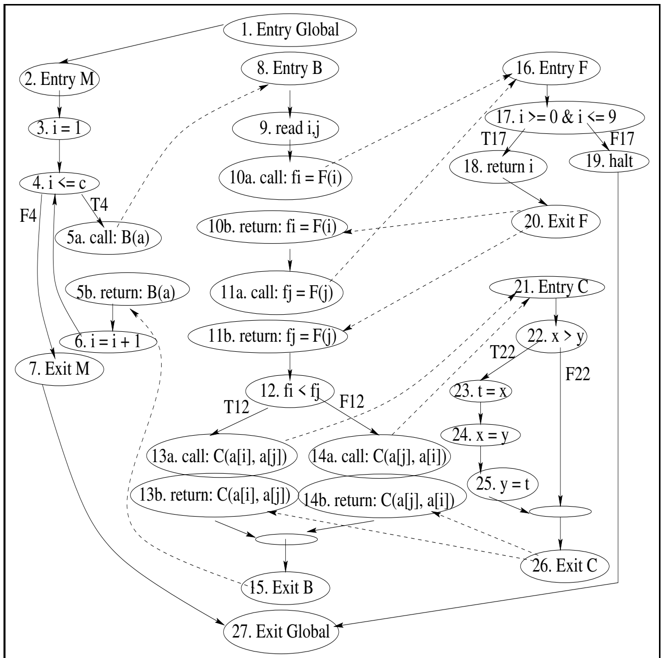

Figure 3: Interprocedural control flow graph for `M`

In this ICFG, the conditions $T4$, $T12$, $T17$ and $T22$ are respectively **`i<=c`**, **`fi<fj`**, **`i>=0 & i<=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* [12], 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.

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.

CSP and Constraint Solving. A *constraint satisfaction problem* (CSP) $P = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ 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.

Consistency techniques are algorithms that reduce the search space by removing, from the domains, values that cannot appear in a solution. Consistency algorithms play an important role in the resolution of CSP [21].

Interval programming methods have been designed to solve (continuous) constraints over the real numbers. The basic idea is to associate with each variable an interval representing its domain. Consistency techniques (on continuous domains) thus aim at reducing the size of the intervals without removing solutions of the constraints. Such consistency techniques are usually coupled in methods for solving such constraints [8].

Notations. The set of floating-point numbers (F-numbers) is denoted by F. The set of intervals is denoted by I. The set of boolean intervals is denoted by BI, where $BI = \{[0, 0], [0, 1], [1, 1]\}$ (0 and 1 respectively represent $false$ and $true$). BI is thus a subset of I. Capital letters denote intervals. Constraints involve reals, integers, and booleans. If a is a F-number, a^+ denotes the smallest F-number strictly greater than a and a^- the largest F-number strictly smaller than a. If x is a real number, $\lfloor x \rfloor$ denotes the largest integer that is not larger than x and $\lceil x \rceil$ the smallest integer that is not smaller than x. The lower and upper bounds of an interval X are 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 $length(a)$ is 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 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* [8].

Test cases. An (integer, boolean or float) *input variable* is either an input parameter or a variable in an input statement of 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 float 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 a specified path) uses operators defined on F-numbers, integers and booleans. We assume here that the test program is written in some fixed imperative language $\mathcal{L}$.

Definition 1. *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.*
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.*
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.*

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 cases exist, 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 real (or mathematical) solutions from float solutions of a path constraint. As introduced in [20], a real 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, $c(x) \triangleq x = \frac{x}{3} + \frac{x}{3} + \frac{x}{3}$, is mathematically true for all floating-point number in F. However $eval(c, 1)$ may evaluate to false in some programming languages. Likewise, constraints may have float solutions, while having no real ones, i.e. $eval(c, \mathbf{v}) \not\Rightarrow c(\mathbf{v})$. This was illustrated in [13] with the constraint, $16.0 + x = 16.0 \ \wedge \ x > 0$.

Framework on Interval Logic. We extend interval logic framework to handle interval constraints involving, at the same time, integers, reals, and booleans, as well as the logical operators such as AND, OR, NOT. Arithmetic operators are first classically extended for intervals [8]. In classical interval programming, an interval extension of a constraint $c(x)$ is an interval constraint $C(X)$, such that for all interval X, $(\exists x \in X : c(x)) \Rightarrow C(X)$ [8]. This means that C is a mapping from I to the set $\{false, true\}$. A disadvantage of this definition can be illustrated by the following example. $[2, 4] < [1, 3]$ evaluates to $true$, from which we 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 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.

Interval-extension constraints have thus to be extended to logical operators.

DEFINITION 2. *An interval extension of a n-ary constraint $c(x_1, \ldots, x_n)$ is a mapping $C : I^n \rightarrow BI$ such that for all $\mathbf{X} \in I^n$*

if $\nexists \mathbf{x} \in \mathbf{X} : c(\mathbf{x})$ *then* $C(\mathbf{X}) = [0, 0]$
if $\exists \mathbf{x} \in \mathbf{X} : c(\mathbf{x}) \bigwedge \exists \mathbf{y} \in \mathbf{X} : \neg c(\mathbf{y})$ *then* $C(\mathbf{X}) = [0, 1]$
if $\forall \mathbf{x} \in \mathbf{X} : c(\mathbf{x})$ *then* $C(\mathbf{X}) = [1, 1]$

For instance, an interval extension of the $\leq$ relational operator 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.

DEFINITION 3. *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*
$[a_1, b_1] \ AND \ [a_2, b_2] = [min(a_1, a_2), min(b_1, b_2)]$,
$[a_1, b_1] \ OR \ [a_2, b_2] = [max(a_1, a_2), max(b_1, b_2)]$,
$NOT([a_1, b_1]) = [1 - b_1, 1 - a_1]$.

An interval solution of a set of constraints is then defined as follows.

DEFINITION 4. *Let $S = \{c_1, \ldots, c_m\}$ be a set of constraints. A box $\mathbf{X} \in I^n$ is an interval solution of S if $right(C_i(\mathbf{X})) = 1$, i.e. $\exists \mathbf{x} \in \mathbf{X} : C_i(\mathbf{X})$, for all i: $1 \leq i \leq m$, where the C_i are respectively an 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 $C(\mathbf{X})$ will denote $right(C(\mathbf{X})) = 0$ (i.e. $C(\mathbf{X})$ is $[0, 0]$).

3. GENERATION OF PATH CONSTRAINTS

Given a path of an ICFG, we propose an algorithm to construct a path constraint. Indexed variables are used to hold the definitions of the original variables in the path. 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.

The algorithm (named `PathConstraintGeneration`) takes as input a path in the ICFG for the test procedure, and outputs a path constraint. For lack of space, the algorithm, which is an extension of an algorithm presented in [20], will not be presented here. Some operations with arrays are transformed into the following complex constraints: $\mathsf{na3}$ and $\mathsf{na4}$. (1) A 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. By convention, when all the elements of array a are *null* (non-initialized), we will denote this as $a = null$. The constraint $\mathsf{na4}(b, a, j, v)$ can be defined more formally as follows:

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

An assignment to an array element, `a[j]:=exp`, is then transformed into the constraint, $\mathsf{na4}(a_{k+1}, a_k, \overline{j}, \overline{exp})$, where a_k is the last value instance of a, $\overline{j}$ and $\overline{exp}$ are respectively a version of j and exp, in which each variable is substituted by its last value instance. (2) The constraint $\mathsf{na3}(b, a, j)$ defines $b[j]$ as an input variable.It is defined as follows:

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

An input statement to an array element, `read a[j]`, is transformed into the constraint: $\mathsf{na3}(a_{k+1}, a_k, \overline{j})$, where a_k is the last value instance of a, $\overline{j}$ is a version of j in which each variable is substituted by its last value instance.
Parameters in procedure calls are handled as follows. Each actual parameter x' of a call to procedure P, together with the corresponding formal parameter x of P (that is either a pass-by-value or pass-by-reference parameter), is translated into an assignment $x := x'$ when the control is passed to P. When the control quits P, an assignment $x' := x$ is generated only if x is a pass-by-reference parameter.

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-21-22-23-24-25-26-13b-15-5b-6-4-7-27 (in Figure 3). The algorithm involves two main steps. Step 1 consists in defining input variables from the formal parameters of the test procedure. Step 2 makes a traversal of the path to generate constraints for its nodes and branches. The path constraint generated is the conjunction of all constraints obtained from Step 1 and Step 2.

Step 1: The constraint, $\bigwedge_{0 \leq i \leq 9} \mathrm{a}_0[i] \in dom(\mathrm{a}_0) \wedge \mathrm{c}_0 \in dom(\mathrm{c}_0)$, is generated, defining the input variables. Note that (1) a and c are parameters of procedure `M` (Figure 1); (2) since a_0 is a value instance of a, it has the same properties as a wrt its length and the domain for its elements ($dom(a_0)$ is $dom(a)$); (3) similarly, $dom(c_0)$ is $dom(c)$.

Step 2: for nodes 1,2: no constraints are generated;
node 3: $\mathrm{i}_0 := 1$; node 4-$T4$: $\mathrm{i}_0 \le \mathrm{c}_0$; node 5a: $\mathrm{a}_1 := \mathrm{a}_0$;
nodes 8,9: $\mathrm{i}_1 \in dom(\mathrm{i}_1) \wedge \mathrm{j}_0 \in dom(\mathrm{j}_0)$;
node 10a: $\mathrm{i}_2 := \mathrm{i}_1$;
nodes 16,17-$T17$,18,20: $\mathrm{i}_2 \ge 0 \wedge \mathrm{i}_2 \le 9$;
node 10b: $\mathrm{fi}_0 := \mathrm{i}_2$; node 11a: $\mathrm{i}_3 := \mathrm{j}_0$;
nodes 16,17-$T17$,18,20: $\mathrm{i}_3 \ge 0 \wedge \mathrm{i}_3 \le 9$;
node 11b: $\mathrm{fj}_0 := \mathrm{i}_3$; node 12-$T12$: $\mathrm{fi}_0 < \mathrm{fj}_0$;
node 13a: $\mathrm{x}_0 := \mathrm{a}_1[\mathrm{i}_1] \wedge \mathrm{y}_0 := \mathrm{a}_1[\mathrm{j}_0]$;
nodes 21,22-$T22$,23,24,25,26: $\mathrm{x}_0 > \mathrm{y}_0 \wedge \mathrm{t}_0 := \mathrm{x}_0 \wedge \mathrm{x}_1 := \mathrm{y}_0 \wedge \mathrm{y}_1 := \mathrm{t}_0$;
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-$F4$,7,27: $\mathrm{i}_4 := \mathrm{i}_0 + 1 \wedge (\mathrm{i}_4 \le \mathrm{c}_0)$.

A path constraint is composed of: (1) constraints defining input variables: simple input variable and array element (na3 constraint), (2) assignment constraints: equality constraints with ":=" notation for simple variable and na4 constraints for array elements, (3) branch constraints (constraints for the branches of the path). However, only the branch constraints represent the conditions which must be satisfied so that the path is traversed. The other types of constraints, as will be shown in the next section, 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. TEST DATA GENERATION: PATH COVERAGE

A test data generation algorithm for path coverage criterion is presented. The algorithm makes use of a consistency technique based on a consistency notion (*eBox consistency*).

Consistency.

The eBox consistency, introduced in [20], is an extension of the classical Box consistency [8] to handle both real, integer and boolean variables.

DEFINITION 5 (EBOX CONSISTENCY). *Let $P = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ be a CSP where $\mathcal{V} = (x_1, \ldots, x_n)$, a set of (real and integer) variables; $\mathcal{D} = (X_1, \ldots, X_n)$ with $X_i = [l_i, r_i]$ the domain of x_i $(1 \le i \le n)$; $\mathcal{C} = (c_1, \ldots, c_m)$, a set of constraints defined on $x_1, \ldots, x_n$ and $c \in \mathcal{C}$ be a k-ary constraint on the variables $(x_1, \ldots, x_k)$. The constraint c is* eBox-consistent *in $\mathcal{D}$ if for all x_i $(1 \le i \le k)$*

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

where C is an interval extension of constraint c.

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

DEFINITION 6 (FILTERING BY EBOX CONSISTENCY). *Filtering by eBox consistency of a CSP $P = (V, D, \mathcal{C})$ is a CSP $P' = (V, D', \mathcal{C})$ such that (1) $D' \subseteq D$, (2) P and P' have the same solutions, and (3) P' is eBox-consistent in D.*

The objective of filtering is to reduce as much as possible the domains of the variables (i.e. their interval) without removing solutions from the initial domains. Filtering algorithms are based on the property that if $C(I_1, \ldots, I_n)$ does not hold (i.e. $right(C(X)) = 0$), then no solution of c lies in $I_1 \ldots I_n$, that can then be pruned. We denote $\Phi_{eBox}(P)$, the filtering by eBox consistency of P. Note that the filtering by eBox consistency of a CSP, by its definition, always exists and is unique. An implementation of $\Phi_{eBox}(CSP)$ will not be presented here. It can however easily be constructed as an adaptation of a filtering algorithm in [8]. A fundamental difference should be noted: the filtering algorithm in [8] aims to obtain a box containing all real solutions, while our algorithm aims to obtain a box containing all float solutions. Technically, our algorithm is simpler, consisting in applying recursively a domain-splitting on the initial box to prune parts which do not have float solutions. Assuming that (1) the evaluation order of the operators of the interval constraint $C(I_1, \ldots, I_n)$ is the same as the evaluation order of the constraint $c(x_1, \ldots, x_n)$ in the programming language of the test program, and (2) the basic interval operations are conservative on the floats (which is actually the case), our filtering algorithm is then conservative on the floats.

Algorithm.

Our algorithm is given in Algorithm 1. Function `PathConstraintGeneration` is the new method for path constraint generation (presented in the previous section). Function `EBoxFiltering` (in Algorithm 2) is a new filtering technique for path constraints involving arrays. Note that the search for a test case in a resulting *ϵ_box* is accomplished by function `FindSolution`, as specified hereafter.

SPECIFICATION 1 (FINDSOLUTION). Let $\mathcal{C}$ be a set of constraints, e be an *ϵ_box* and TS be a representative set of floating-point vectors in e. The function `FindSolution`$(\mathcal{C}, e)$ returns, if it exists, some vector $\mathbf{v} \in TS$ such that $\forall c \in \mathcal{C}, eval(c, \mathbf{v})$ holds. Otherwise it returns $\emptyset$.

It is interesting to highlight the main differences between the algorithm and our previous algorithm [20] dealing with path constraints 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 `EBoxFiltering`), which, in turn, is integrated in the path constraint solving (function `SolvePathConstraints`). 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 input variables

Algorithm 1 Generation of test data: 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 := set of 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 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) := EBoxFiltering(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
```

set when j can be simplified into a number. The function `EBoxFiltering` (Algorithm 2) realizes the filtering on the path constraint. The path constraint is represented by the branch constraints and the other constraints. As explained in Section 3, the pruning is only performed on the branch constraints. The function `Simplify` (Algorithm 3) simplifies the branch constraints by extracting information from the other constraints. The number of known input variables may increase after a simplification.

In Algorithm `EBoxFiltering`, 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 eBox-consistent in box D_t, (2) C_2 (the other branch constraints involving non-input variables) cannot be simplified further with box D_t.

We conclude this section by analyzing 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. Note that there must exists an assignment constraint, $x := def(x)$, for non-input variable x in OC; a variable is simple if it is neither an array variable nor an array element. As such a simplification is done only once during the solving of the path constraint, for efficiency purpose, instructions in line 2 can be transfered from Algorithm 3 to Algorithm 1. Following the simplification of non-input simple variables, the next steps have the purpose of simplifying constraints involving arrays. 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 the other constraints. An inconsistency can be detected when an array element is used in an expression without being initialized.

5. TEST DATA GENERATION: STATEMENT COVERAGE

As presented in Section 1, test data generation for statement (and branch) coverage consists in searching for test data traversing certain nodes (branches) of the ICFG associated with the test procedure. It is sufficient to concentrate on statement coverage. All the following algorithms can easily be adapted for branch coverage. The search is guided by a control dependence graph. Two different control dependences for programs with procedure calls are introduced: the intraprocedural and the interprocedural control dependences. We will show that interprocedural control dependence is better for our purpose.

Control Dependence Graph. Control dependence captures the effects of predicate statements on the program's behavior. Technically, control dependence is defined in terms of a CFG and the post-dominance relation among the nodes in the CFG [4].

DEFINITION 7. *A node V is* post-dominated *by a node W in G if every directed path from V to STOP (not including V) contains W. A node Y is* control dependent *on node X iff (1) there exists a directed path P from X to Y with all Z in P (excluding X and Y) post-dominated by Y, and (2) X is not post-dominated by Y.*

Note that if Y is control dependent on X then node X must have at least two exits. Following one of the exits from X results in Y being executed while taking others may result 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 [4] 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

Algorithm 2 Filtering of path constraints

function EBoxFiltering(V:$Variables$,D:Box,BC:$BranchConstraints$,OC:$OtherConstraints$):CSP;
PRE $(V^*, D, BC \wedge OC)$ is a CSP
  V 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 an equivalent CSP $(V', D', BC' \wedge OC')$ with $V \subseteq V' \subseteq V^*$ and BC'_1 is eBox-consistent,
  where BC' = $BC'_1 \wedge BC'_2$ (BC'_1 contains the branch constraints involving only input variables)
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 := $BC_t \setminus C_1$;
  $Store$:= C_1;
2:while $C_1 \neq \emptyset$ do
3: $(V_t,\ D'_t,\ Store)$:= $\Phi_{eBox}(V_t,\ D_t,\ Store)$;
    if $D'_t = \emptyset$ then return $(V, \emptyset, BC, OC)$;
    if $D'_t = D_t$ then break;
    D_t := D'_t;
    if $C_2 = \emptyset$ 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 := $C'_2 \setminus C_1$;
    $Store$:= $Store \wedge C_1$;
    V_t := V'_t;
    D_t := D'_t;
    OC_t := OC'_t;
  endwhile
5:return $(V_t,\ D_t,\ Store \wedge C_2,\ OC_t)$;
end

Algorithm 3 Simplification of path constraints

function Simplify(V : $Variables$, D : Box,BC:$BranchConstraints$,OC:$OtherConstraints$) : CSP;
PRE $(V^*, D, BC \wedge OC)$ is a CSP
  V 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 an equivalent CSP $(V', D', BC' \wedge OC')$ with BC' is a simplified version of BC.
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 an assignment constraint, $x := def(x)$, in OC}
      $OC := OC[x/def(x)]$;
      $OC := OC \setminus \{x := def(x)\}$; {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 | 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 | 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 | i is a number and $b[i]$ is not an input variable do
        case $\exists$ $(b := a)$ in OC : $BC[b[i]/a[i]]$; $simplify$:= $true$;
        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 $(V, \emptyset, BC, OC)$;
        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

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)

(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 predicate 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 [19]. 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 [19]. (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 8 (Interprocedural CDG). *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.*

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.

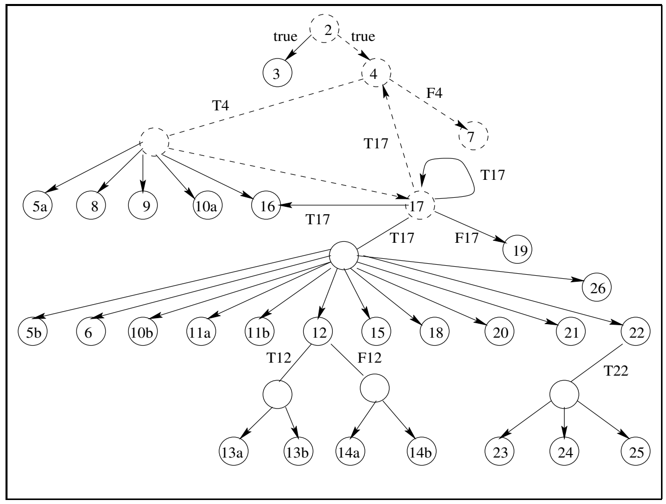

Figure 4: ICDG for Program 1

Definition 9. *The* decision graph *for a node n in a ICDG G is the smallest subgraph of G, containing all the paths from the start node to node n, and without edges connecting a node to itself.*

The construction of the decision graph for a node n 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 with predicate 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.

Algorithm. In [20], we proposed an algorithm for the generation of test data for statement coverage, based on control dependence graph, but limited to programs without procedure calls. This algorithm can easily be extended for programs with procedure calls by using interprocedural control dependence graph and decision graphs. Note that the search is pruned by using the filtering algorithm developed for path constraints.

Table 3: Program under analysis

Programs	Int.	Float	Bool.	Arrays	Proc.
Program-1	yes	yes	no	yes	yes
NthRootBisect	yes	yes	no	no	no
Sample-1	yes	no	yes	yes	no
Sample-2	yes	no	yes	yes	yes
BSearch	yes	yes	no	yes	no
Gaujac	yes	yes	no	yes	yes

6. EXPERIMENTAL RESULTS

Implementation. To determine the effectiveness of our approach, a prototype written in Java, which is an extension of our previous prototype [20], has been developed. It uses an interval arithmetic library [9] for the implementation of the constraint solving algorithm, and algorithms from [19] to construct ICDG. The prototype is independent from the programming language used by the program under analysis. This means that the source code, written in some imperative language $\mathcal{L}$, is first translated into an internal representation and ICFG, which are common for all languages, such as C, Pascal, etc. Currently, the prototype uses the internal representation and ICFG as its input. In the `FindSolution` function, given an epsilon interval solution, we simply select its middle point to check if it satisfies the path constraint. And if so, it will be a test case for the path. Of course, more sophisticated *labeling* strategies, such as described in [13], can also be applied to the epsilon interval solution. However, when the epsilon is set to a very small number, such as *1e-16* in our prototype, the middle point turns out to be sufficient as will be shown in experiments. Note that the smaller the epsilon is, the more time to find an interval solution is required.

Without the used libraries, our prototype has 80 classes and a total length of 6000 lines. Function calls to built-in functions such as *exp* (Euler's 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 [9] or constructed in our prototype.

Experiments. We performed our experiments on a 900MHz UltraSparcIII+ machine, with the following programs.
`Program-1` is the program of this paper depicted in Figure 1. `NthRootBisect` [20] 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-1` is the "sample" program with arrays, described in [3]. `Sample-2` is the "sample" program with procedure calls and arrays, proposed in [11]. This program is an equivalent version of `Sample-1`, but with procedure calls. `BSearch` [3, 5] is a binary search program involving arrays. Finally, we tested the `gaujac` program in [16], which is a scientific program calculating the Gauss-Jacobi integration formula. This program involves complex (non linear) expressions, 3 nested loops, arrays, and procedure calls. A similar, but simpler program has been experimented in [7]. Table 3 summarizes these programs.

As an example for path coverage, with the path given in Section 3 and an initial box $(a_0 : [5, 20], c_0 : [1, 10], i_1 : [-5, 20], j_0 : [-5, 20])$, we obtained, in 0.165 seconds, 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, j_0 are input variables generated during the path constraint generation.

Table 4: Experimentation results

Programs	Nodes	Average (sec.)	Max (sec.)	Tot. (sec.)	Cover.
Program-1	29	0.003	0.056	0.11	100%
NthRootBisect	11	0.037	0.312	0.41	100%
Sample-1	18	0.036	0.328	0.66	100%
Sample-2	26	0.029	0.338	0.74	100%
BSearch	12	0.109	0.854	1.32	100%
Gaujac	67	4.589	152.6	307	100%

For statement coverage, our test generation procedure consists in trying to generate a test case for each node of the ICFG, and then reporting the achieved statement coverage (the percentage of nodes for which a test case has been found). The results of the experiments are summarized in Table 4. For each program, the table lists the number of nodes of its 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 (Tot.), and the achieved statement coverage (Cover.). Except for the complex `gaujac` program, the method is very efficient. 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.

Table 5 summarizes the existing methods with functionalities close to our method (first line in the table). Two other methods offer the same functionalities, [11] and [7]. 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 uncomparable (number of iterations versus execution time versus theoretical complexity). When it is comparable, one should consider the differences in the underlying hardware.

In [7], an execution time of 98 and 42 seconds (Windows NT, 400MHz Pentium II) is reported to find a test data for two branches of an exponential integral programs (program with float variables and non linear tests). In the `BSearch` example in [5], it is reported that Inka did not spend more than 10 seconds on each node (300 MHz Sun Ultra Sparc5), while our maximum becomes 2.75 seconds on such a machine. The speedup here is thus around 3.6. In [5], it is also reported that Inka is about 10 times faster than TestGen (on comparable computers), hence we obtain a speedup around 36 between our prototype and TestGen. The `Gaujac` program is by far the most complex of our examples. In the literature, we did not find such a complex example (in terms of the complexity of expressions) used by another methods.

These experiments, their analysis and their comparison with exisiting methods show the versatility and flexibility of the approach to different classes of problems (integer and/or float variables; arrays, procedures, path coverage, statement coverage). They also demonstrate the feasibility of the method, its efficient and its potential to handle complex programs.

7. CONCLUSION

In this paper, we presented a novel approach for interprocedural test data generation of imperative programs. It ex-

Table 5: A summary of different test data generators

Methods	Reference	Integer	Float	Arrays	Procedure	Path Coverage	Statement/Branch Coverage
Consistency	*this*	*yes*	*yes*	*yes*	*yes*	*yes*	*yes*
Testgen	[11]	yes	yes	yes	yes	yes	yes
Relaxation	[7]	yes	yes	yes	yes	yes	yes
InKa	[5]	yes	no	yes	partial[1]	no	yes
Genetic	[15]	yes	yes	yes	no	no	yes
Symbolic	[1]	yes	yes	partial[2]	yes	yes	no

[1] The pass-by-reference mechanism for passing parameters is not handled
[2] Array references depending on input variables are not handled

tended our previous work to numeric programs (containing integer and float variables) with 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. 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. For statement coverage, the search for suitable paths is guided by the interprocedural control dependences of the programs. The underlying algorithms have been described. The developed prototype illustrated the versatility and the efficiency of the method, as well as its potential to handle complex programs.

Different areas will be investigated in future work. Front-ends for specific languages, translating the program under test into ICFG, will be considered. Different strategies for the `FindSolution` functions will also be developped. The introduction of pointers will also be investigated. The possibility of error detection will also be considered, by adding new kinds of constraints modeling error conditions such as in [17].

8. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers, Michel Rueher, and Baudouin Le Charlier for their helpful comments and suggestions.

9. REFERENCES

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

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

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

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

[5] A. Gotlieb, B. Botella, and M. Rueher. A CLP framework for computing structural test data. In *Computational Logic*, pages 399–413, 2000.

[6] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. In *ACM SIGSOFT Sixth International Symposium on Foundations of Software Engineering(FSE-6)*, Nov. 1998.

[7] N. Gupta, A. P. Mathur, and M. L. Soffa. Generating test data for branch coverage. In *15th IEEE International Conference on Automated Software Engineering(ASE00)*, September 2000.

[8] P. V. Hentenryck, L. Michel, and Y. Deville. *Numerica. A modeling language for global optimization*. The MIT Press, Cambridge, Massachusetts, London, 1997.

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

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

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

[12] D. Melski and T. W. Reps. Interprocedural path profiling. In *Computational Complexity*, 1999.

[13] 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.

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

[15] R. P. Pargas, M. J. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

[16] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C. The Art of Scientific Computing. Second Edition*. Cambridge University Press, 1992.

[17] D. J. Richardson and M. 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.

[18] G. Schumacher and A. Bantle. Automatic test case generation using interval arithmetic. In *Proceedings of the SCAN2000/INTERVAL2000*, Germany, 2000.

[19] S. Sinha, M. J. Harrold, and G. Rothermel. Interprocedural control dependence. *Software Engineering and Methodology*, 10(2):209–254, 2001.

[20] N. T. Sy and Y. Deville. Automatic test data generation for programs with integer and float variables. In *16th IEEE International Conference on Automated Software Engineering(ASE01)*, 2001.

[21] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.