Symbolic execution of floating-point computations

Bernard Botella, Arnaud Gotlieb, Claude Michel

Symbolic execution of floating-point computations [*]

Bernard Botella [a], Arnaud Gotlieb [b,*], Claude Michel [c]

[a] *THALES Airborne Systems 2 av. Gay-Lussac 78851 Elancourt Cedex, FRANCE*

[b] *IRISA / INRIA Campus Beaulieu 35042 Rennes cedex, FRANCE*

[c] *I3S-CNRS 930, route des Colles, BP 154, 06903 Sophia Antipolis cedex,*
FRANCE

Abstract

Symbolic execution is a classical program testing technique which evaluates a selected control flow path with symbolic input data. A constraint solver can be used to enforce the satisfiability of the extracted path conditions as well as to derive test data. Whenever path conditions contain floating-point computations, a common strategy consists of using a constraint solver over the rationals or the reals. Unfortunately, even in a fully IEEE-754 compliant environment, this leads not only to approximations but also can compromise correctness: a path can be labelled as infeasible although there exists floating-point input data that satisfy it. In this paper, we address the peculiarities of the symbolic execution of program with floating-point numbers. Issues in the symbolic execution of this kind of programs are carefully examined and a constraint solver is described that supports constraints over floating-point numbers. Preliminary experimental results demonstrate the value of our approach.

Key words: Symbolic execution, Floating-point computations, Automatic test

1 Introduction

Structural testing is usually required to find a test set that activates control flow paths that cover a selected testing criterion (e.g. all_statements, all_branches, ...). Introduced by King in the context of Software Testing [1], symbolic execution consists in statically evaluating statements of a program to find a test datum that activates a given control flow path. Input variables are replaced by symbolic input data and each statement of the path is evaluated by replacing internal references with an expression over the symbolic input data. Symbolic execution computes so-called path conditions that are constraints on the symbolic input data that characterize the selected path. Solving the path conditions permits input data to be obtained that activate the path. As only input values are generated, such an approach relies on the availability of an oracle. An oracle is just a procedure that checks the computed outcomes and produces a testing verdict. Symbolic execution can be used to address the path feasibility problem [2,3]. When the constraint set equivalent to the path conditions is unsatisfiable, then the selected path is shown to be infeasible. Note, however, that finding all the infeasible paths of a program is a classical undecidable problem [4]. Symbolic execution has been used in numerous applications, such as automatic structural test data generation [5,6,7,8,9,10,11], mutation-based testing [12], program specialization

* Corresponding author.
    Email address: `Arnaud.Gotlieb@irisa.fr` (Arnaud Gotlieb).

[13], parallelizing compilers [14], program and property proving [15,16], just to name a few.

Issues in floating-point computations. It is well known that reasoning over the reals or the rationals leads to some inconsistencies when the results are directly mapped over to the floating-point numbers [17]. In such a case, even in an environment which complies to the IEEE-754 standard for binary floating-point arithmetic [18], the symbolic execution of a program path which involves floating-point variables can produce not only inexact results but also incorrect ones. For example, consider the C program given in Fig.1 and the symbolic execution of path $1\rightarrow2\rightarrow3\rightarrow4$. The associated path conditions can be written as $\{x > 0.0, x + 1.0e12 = 1.0e12\}$. It is trivial to verify that these constraints do not have any solution over the reals or the rationals and a solver over the reals like the IC library of the Eclipse Prolog system [19] will immediately detect this. However, any IEEE-754 single-format floating-point numbers of the closed interval $[1.401298464324817e - 45, 32767.9990234]$ is a solution of these path conditions. Hence, a symbolic execution tool working over the reals or the rationals will declare this path as being infeasible although this is clearly incorrect. Conversely, consider the path conditions $\{x < 10000.0, x + 1.0e12 > 1.0e12\}$ which could easily be extracted by the symbolic execution of path $1\rightarrow2\rightarrow3\rightarrow4$ of program foo2 of Fig.2. All the reals of the open interval $(0, 10000)$ are solutions of these path conditions. However, there is no single floating-point value able to activate the path $1\rightarrow2\rightarrow3\rightarrow4$. Indeed, for any single floating-point number x_f in $(0, 10000)$, we have $x_f + 1.0e12 = 1.0e12$ [1]. Hence the path $1\rightarrow2\rightarrow3\rightarrow4$ is actually infeasible although a symbolic execution tool over the reals or the rationals would

---

[1] This behaviour is called the addition *absorption*.

```
float foo1(float x) {
    float y = 1.0e12, z ;
1.  if (x > 0.0)
2.      z = x + y ;
3.  if (z == y)
4.      . . .
```

```
float foo2(float x) {
    float y = 1.0e12, z ;
1.  if (x < 10000.0)
2.      z = x + y ;
3.  if (z > y)
4.      . . .
```

Figure 1. Program foo1   Figure 2. Program foo2

have declared it as feasible.

This kind of behaviour can be obtained with any of the available solvers over the reals or the rationals. These solvers use a Linear Programming algorithm as in the clpr or in the clpq framework, or interval propagation with floating-point numbers to bound the reals such as in Ilog Solver, Eclipse IC [19], RealPaver [20] or Interlog [21,22]. The key issue here is that these solvers obey to mathematical rules which do not hold for floating-point arithmetic. As a matter of fact, floating-point arithmetic is quite poor. For example, with floating-point numbers, $x + (y + z)$ is not in general equal to $(x + y) + z$. Moreover, interval propagation based solvers assume that if $z = x + y$ then $x = z - y$. Unfortunately, due to rounding operations, this does not hold for floating-point arithmetic.

Such problems might be seen as unavoidable. By contrast, this paper introduces the techniques required to correctly handle these kinds of issues. Our approach is based on the following two steps:

• In a first step, complex expressions over the floating-point numbers are translated into equivalent relations which capture all the semantics of the floating-point operations; these relations are binary or ternary constraints

4

over the floating-point numbers.

- In a second step, a solver dedicated to floating-point numbers is used to solve the resulting constraints; this solver handles these constraints according to the semantics of floating-point arithmetic.

For example, consider again the path conditions extracted from Fig.1 and assume that the initial domain of variable x is $[-INF, +INF]$. The first constraint $x > 0.0$ reduces the interval of x to $[1.401298464324817e - 45, +INF]$, the lower bound of which is the smallest non-zero positive number that can be represented in IEEE single-format floating-point arithmetic. Then, the second constraint $x + 1.0e12 = 1.0e12$ reduces [2] the domain of x to $[1.401298464324817e - 45, 32767.9990234]$. In this example, all the values of the resulting interval are solutions of the path conditions. Hence, it suffices to take any of the single floating-point of this interval to find a test datum that activates path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ of the foo1 program. However, this is not generally the case and one must resort to enumeration to find a solution.

Contributions of the paper. This paper introduces new techniques to symbolically execute programs which involve floating-point computations. The paper extends the theoretical work of Michel [23] on the design of exact projection functions of constraints over the floating-point numbers. Practical details on how to build correct and efficient projection functions over floating-point intervals are given. The paper covers not only arithmetic operators but also comparison and format-conversion operators. FPSE, a symbolic execution tool for ANSI C floating-point computations, has been developed to validate the proposed approach. This paper describes its design and implementation and

---

[2] In IEEE-754 single-format, the constant $1.0e12$ is interpreted as 999999995904.

reports some initial experimental results. Note, however, that the paper does not address the general problem of testing floating-point computations. In particular, it does not study the difficult problem of obtaining a correct (but not necessarily exact) oracle in the presence of floating-point computations.

Contents. Section 2 briefly recalls the main principles of symbolic execution and reviews how several symbolic execution tools handle the problem of floating-point computations. Section 3 explains the essence of the IEEE-754 standard for binary floating-point arithmetic and indicates the limitations of the proposed approach. Section 4 presents the design of efficient projection functions over floating-point variables. Section 5 explains how to deal with symbolic values such as infinities. Section 6 describes FPSE and reports some experimental results. Finally, the last section describes directions for further work.

2 Related work

Only a few studies deal with floating-point computations in the Software Testing community. According to our knowledge, the only directly related work is that of Miller and Spooner [24]. Thirty years ago, they studied how to generate automatically floating-point test data for imperative programs. Their work opened the door for execution-based test data generation methods which does not suffer of the above mentioned problems. However, their approach makes only use of program executions and do not rely on symbolic reasoning. Thus, it cannot be used to study path feasibility.

At a time when no standard for floating-point arithmetic was available, sym-

bolic execution was pioneered by King [1], Clarke [5], Howden [6] and others
[7,8,9] in several systems. SELECT [7] and DAVE [5] exploited Linear Pro-
gramming algorithms to solve linear path conditions over the reals. CASEGEN
[8] utilized ad-hoc procedures based on try-by-value methods to solve non-
linear equations and used inequalities over the reals and to find test data that
activated a selected path in the control flow graph. Although these systems
were using floating-point operations in their computations, they solved path
conditions over the reals. Thus, they did not conform to the floating-point
computations of the program under test.

SMOTL [9] and more recently GODZILLA [12] took advantage of domain re-
duction techniques to prune the search space of integers inequalities. Gotlieb
et al. [25] applied Constraint Logic Programming over finite domains to solve
constraints extracted from imperative programs in the tool INKA [26]. The
proposed framework dealt only with constraints over integers (possibly non-
linear) to automatically generate test data. SMOTL, GODZILLA and INKA
did not address the problem of floating-point computations in symbolic exe-
cution but they did use domain and interval propagation techniques to solve
constraint systems. The method used in the current paper to solve path cond-
itions over floating-point variables is closely related to these techniques.

More recently, Meudec followed a similar path in [11] and proposed solving
path conditions over floating-point variables by means of a constraint solver
over the rationals in the ATGen symbolic execution tool. The `clpq` library
[27] of the Constraint Logic Programming system ECLIPSE was used to solve
linear constraints over rationals computed with an arbitrary precision using an
extended version of the simplex algorithm. Although this approach appears to
be of particular interest in practice, it fails to handle correctly floating-point

7

computations.

Hence, the problem of floating-point computations in symbolic execution have not been seriously addressed in the past. Although several works deal with floating-point computations, none of them provide a correct handling of floating-point computations. Indeed, floating-point computation can be correctly handled neither with constraint solvers over the reals nor with constraint solvers over the rationals. Dealing with floating-point computations requires the development of a new constraint solver dedicated to floating-point numbers.

3   Preliminaries

This section introduces the arithmetical model specified by the IEEE-754 standard for binary floating-point arithmetic [18] and explains the limitations and notations of the proposed approach.

3.1   IEEE-754

IEEE-754 specifies two basic binary floating-point formats (single and double) and two extended formats. Each floating-point number is a triple (s, e, f) of bit patterns where s is the sign bit, e the biased exponent, and f the significand. The single format occupies 32 bits (1 bit for the sign, 8 for the exponent and 23 for the significant) while the double occupies 64 bits (1 bit for the sign, 11 for the exponent and 52 for the significant). The standard does not give a strict specification of the extended formats, but it does prescribe some minimal requirements over their sizes. For example, a double extended must occupy at least 79 bits. Each format defines several classes of numbers: nor-

malized numbers, denormalized numbers, signed zeros, infinities and NaNs (which stands for Not-a-Number). For the single format, normalized numbers corresponds to an exponent value $0 < e < 255$ and a value given by the formula: $(-1)^s \ 1.f \ 2^{e-127}$. Denormalized numbers correspond to an exponent $e = 0$ and a value given by $(-1)^s \ 0.f \ 2^{-126}$ where $f \neq 0$. Note that the significand possesses a hidden bit which is 1 for normalized numbers and 0 for denormalized. Note also that the bias is equal to 127 for the single format [3] and the exponent is -126 for denormalized numbers. There are two infinities (noted $+INF$, $-INF$ with $e = 255, f = 0$) and two signed zeros (noted $+0.0$, -0.0 with $e = 0, f = 0$) that allow certain algebraic properties to be maintained [17]. NaNs ($e = 255, f \neq 0$) are used to represent the results of invalid computations such as a division or a subtraction of two infinities. They allow the program execution to continue without being halted by an exception. IEEE-754 indicates four types of rounding directions: toward the nearest representable value, with "even" values preferred whenever there are two nearest representable values (to-the-nearest), toward negative infinity (down), toward positive infinity (up) and toward zero (chop). The most important requirement of IEEE-754 arithmetic is the accuracy of floating-point computations: each of the following operations, add, subtract, multiply, divide, square root, remainder, conversions and comparisons, must deliver to its destination the exact result if possible or the floating-point number that requires the least modification of the exact result w.r.t. the prescribed rounding mode and the result

---

[3] The actual value of the exponent is $E - bias$, where E is the exponent value in the floating-point number representation. Thus, with single format floating-point numbers, the maximum value of the exponent is 127 and the minimum value is -126.

format destination. It is said that these operations are correctly rounded [4]. For example, the single-format result of $999999995904 + 10000$ is [5] 999999995904 which is the single-format floating-point number nearest to the exact result over the reals. This example shows that the accuracy requirement of IEEE-754 does not prevent surprising results from arising (the second operand is absorbed by the addition operator).

3.2 Limitations and notations

In the sequel, we assume an IEEE-754 compliant floating-point unit. The types of floating-point numbers manipulated by the program are limited to the single and the double-format. The proposed framework currently handles only the to-the-nearest rounding direction, which is the default rounding mode in most programming languages. A decimal constant (such as $1.0e12$) denotes a floating-point value, and thus, has to be understood as the nearest floating-point number according to the default rounding mode (*i.e.* as 999999995904 with a to-the-nearest rounding mode). Zeros and infinities are handled but NaNs are not. Thus any floating-point unknown is assumed to take only a numerical or infinity value. Henceforth x^+ (resp. x^-) denotes the smallest (resp. greatest) floating-point number greater (resp. smaller) than x, with respect to its format. Moreover, $mid(a, b)$ denotes the floating-point number at the middle [6] of a and b. Finally, let $\oplus, \ominus, \otimes, \oslash$ denote floating-point operations

---

[4] IEEE-754 says equivalently "exactly rounded".

[5] These two decimals can be exactly represented by single binary floating-point numbers.

[6] which is a floating-point number of a wider format than the one of its two operands.

(*i.e.* the format dependent result of a to-the-nearest rounding of the exact result) whereas $+, -, *, /$ denote the same operations over the reals. This paper addresses only the problem of dealing with floating-point variables in symbolic execution; other issues such as dealing with loops, arrays and pointers in symbolic execution are out of the scope of this paper. These problems are more detailed in [28,29,10,11,30]. Finally, the combination of integers and floating-point expressions into a symbolic execution framework are not detailed here. Hence, programs are limited to floating-point data types.

4 Symbolic execution

Symbolic execution has been formally described by Clarke and Richardson in [28]. This technique is based on the selection of a single path of the control flow graph and the computation of symbolic states. When one has to deal with floating-point computations, special attention must be paid to the way expressions are evaluated, as described in this section.

4.1 Control flow graph and paths

The control flow graph of a program P is a connected oriented graph composed of a set of vertices, a set of edges and two distinguished nodes, e the unique entry node, and s the unique exit node. Each node represents a basic block and each edge represents a possible branching between two basic blocks. A path of P is a finite sequence of edge-connected nodes of the control flow graph which starts on e. $Var(P)$ denotes the set of variables of P.

11

4.2 Symbolic states and expressions

Symbolic execution works by computing symbolic states for a given path. A *symbolic state* for path $e{\rightarrow}n_1{\rightarrow}\ldots{\rightarrow}n_k$ in P is a triple

$(e{\rightarrow}n_1{\rightarrow}\ldots{\rightarrow}n_k, \{(v, \phi_v)\}_{v\in Var(P)}, c_1\wedge\ldots\wedge c_n)$ where ϕ_v is a symbolic expression associated to the variable v and $c_1 \wedge \ldots \wedge c_n$ is a conjunction of symbolic expressions, called path conditions. A *symbolic expression* is either a symbolic value (possibly **undef**) or a well parenthesized expression composed over symbolic values. In fact, when computing a new symbolic expression, each internal variable reference is replaced by its previous symbolic expression. For example, the symbolic state of path $1{\rightarrow}2{\rightarrow}3{\rightarrow}4$ in the program of Fig. 1 can be obtained by the following sequence of symbolic states :

$(1{\rightarrow}2, \{(x, X), (y, 1.0e12), (z, \textbf{undef})\}, X > 0.0)$

$(1{\rightarrow}2{\rightarrow}3, \{(x, X), (y, 1.0e12), (z, X \oplus 1.0e12)\}, X > 0.0)$

$(1{\rightarrow}2{\rightarrow}3{\rightarrow}4, \{(x, X), (y, 1.0e12), (z, X{\oplus}1.0e12)\}, X > 0.0{\wedge}X{\oplus}1.0e12{==}1.0e12)$

where X is the symbolic value of the input variable x.

Usually, symbolic expressions and path conditions hold only over symbolic input values. However, when floating-point computations are involved in the path, other symbolic values can appear in the symbolic expressions, as described below.

4.3 Forward/backward analysis

Symbolic states are computed by induction on their path by a forward or a backward analysis [28]. Each statement of each node of the path is symbolically evaluated using an evaluation function which computes the symbolic

states. Forward analysis follows the statements of the selected path in the same direction as that of actual program execution, whereas backward analysis uses the reverse direction. Backward analysis is usually preferred when one only wants to compute the path conditions.

4.4 Normalization

In the presence of floating-point computations, special attention must be paid to conform to the actual execution of program. It is necessary to take into account the evaluation order and the precedence of expression operators as specified by the language[7]. The idea is to exploit the expression's shape of the abstract syntax tree built by the compiler of the program without any rearrangement nor any simplification due to optimizations[8]. When symbolic expressions are directly extracted from the abstract syntax tree then, not only the operator precedence is respected but also is the order in which operands are evaluated. This is not always the case when symbolic expressions are extracted from source code by an analyzer. Preserving the order of evaluation in the analyzer is essential with floating-point computations as simple algebraic properties such as associativity or distributivity are lost. An approach called normalization is proposed here. It decomposes expressions and takes into account the above requirements. Normalization makes symbolic expressions over the floating-point numbers independent from the compiling envi-

[7] Some languages are quite permissive and give to the compiler some freedom in the interpretation of floating-point expressions. In such a case, we have to observe the actual behaviour of the compiler.

[8] Compiler optimisation flags are not allowed here particularly when they rearrange instructions.

13

ronment.

Any of the symbolic expressions is decomposed in a sequence of assignments where fresh temporary variables[9] are introduced bearing in mind that the order of evaluation must be preserved. For example, let $E = v_1 \otimes v_2 \otimes v_3 \oplus v_4$ then the resulting decomposition is $E = t_1 \oplus v_4 \wedge t_1 = t_2 \otimes v_3 \wedge t_2 = v_1 \otimes v_2$ because $\otimes$ has a higher priority than $\oplus$ and operands are evaluated from left to right. This decomposition requires that intermediate results of an operation conform to the type of storage of its operands[10]. In the previous example, if v_1 and v_2 are of single-format, then the temporary variable t_2 must also be single-format. As a result, path conditions are only composed of binary or ternary symbolic expressions that have a single operator over a known floating-point format. This form is called the normalized form of a symbolic expression.

5   Solving path conditions over the floating-point numbers

In this section, the floating-point variables are supposed to take a numerical value. We assume here that the computations do not overflow or raise exceptions. These behaviours are handled by means of infinites and NaNs and will be considered in the next section.

Path conditions are composed of normalized symbolic expressions over floating-

---

[9] The introduction of temporary variables does not change the semantic of floating-point computations as long as it maps the behaviour of the compiler and of the floating-point unit.

[10] This property is not a requirement of IEEE-754 and consequently it is not always true. For example, on Intel's architectures extended formats are used by default to store intermediate results

point input and temporaries variables. Each of these variables takes its numerical values within a finite interval of possible floating-point values w.r.t. its format. Intervals are represented by a couple of bounds that can possibly be provided by the user. By default, any numerical single-format floating-point values belongs to $[-3.40282347e38, \ 3.40282347e38]$ and any double-format values belongs to $[-1.7976931348623158e308, \ 1.7976931348623158e308]$.

5.1 Interval propagation

The solving process is based on interval propagation [31,32], which is a classical technique used to compute the set of solutions of non-linear constraints over the reals. The technique takes advantage of interval arithmetic [33] and relational arithmetic [34] to reduce the domains of the variables. If $I_x = [a, b]$ and $I_y = [c, d]$ then interval arithmetic says that $I_{x+y} = [a+c, b+d]$ contains all possible values for the expression $x + y$ when $x \in I_x$ and $y \in I_y$. In the same way, $I_{x-y} = [a-d, b-c]$, $I_{x*y} = [min(a*c, a*d, b*c, b*d), max(a*c, a*d, b*c, b*d)]$, $I_{exp(x)} = [exp(a), exp(b)]$, etc. Relational arithmetic allows decomposing the constraints in projection functions over intervals. For example, the constraint $z = x + y$ is decomposed into three projection functions:

$$I_z \leftarrow I_{x+y} \cap I_z, \ \ I_x \leftarrow I_{z-y} \cap I_x, \ \ I_y \leftarrow I_{z-x} \cap I_y$$

A constraint propagation algorithm uses these projection functions to compute a conservative approximation of the solutions of the constraint system. The following example of a constraint system over the reals illustrates this technique.

Example 1 *Let $x \in (-\infty, +\infty), y \in (-\infty, +\infty)$ be two real unknowns in the constraint system $y = log(x), x + y = 0$. After a decomposition of the constraints into*

projection functions, the following successive approximations of x and y are obtained by interval propagation :

$$x \in (-\infty, +\infty) \quad x \in [0, +\infty) \quad x \in [0, 1] \quad x \in [0.56, 1] \quad x \in [0.56, 0.57] \quad \ldots$$

$$y \in (-\infty, +\infty) \quad y \in (-\infty, 0] \quad y \in [-1, 0] \quad y \in [-1, -0.56] \quad y \in [-0.57, -0.56] \quad \ldots$$

Interval propagation has been applied in several systems [35,32] and two authors of the present paper contributed to the development of one of them, namely Interlog [21,22]. The work presented here mainly consists in adapting a real-based interval propagation system to floating-point numbers. It essentially requires modifying projection functions to handle conservatively the domains of floating-point variables. In the next subsections, interval propagation of floating-point intervals and projection functions for floating-point constraints are described.

5.2   *Propagation over floating-point intervals*

During interval propagation, projection functions are incrementally introduced into a *propagation queue*. An iterative algorithm manages each function one by one into this queue by filtering the domains of floating-point variables of their inconsistent values. Filtering algorithms consider only the bounds of the domains to eliminate inconsistent values. When the domain of a variable has been narrowed then the algorithm reintroduces in the queue all the projection functions in which this variable appears in order to propagate this information. The algorithm iterates until the queue becomes empty, which corresponds to a state where no more pruning can be performed (a fixpoint).

When selected in the propagation queue, each function is added into a *constraint–store*. The constraint–store is contradictory when the domain of at least one

variable becomes empty during the propagation. In this case, the set of constraints (path conditions) is known to be unsatisfiable and the corresponding path is shown to be infeasible. The interval propagation process reaches a fixpoint because only a finite number of floating-point values can be removed from the domains. This fixpoint is a conservative overestimation (Cartesian product of intervals) of the possible floating-point values for the input variables.

As is usually the case with interval propagation solvers, propagation over floating-point intervals does not ensure that the set of constraints is satisfiable when a fixpoint is reached. Hence, one must resort to enumeration to locate particular solutions. This is done by a *labelling procedure* which tries to systematically assign a floating-point to a variable and initiate propagation through the constraint–store. This process is repeated until all the uninstantiated variables become bound. If this valuation leads to a contradiction then the process backtracks to other possible values or variables.

5.3   Floating-point variable projections

In the proposed approach, each normalized symbolic expression is decomposed into ternary and binary symbolic expressions. These expressions could be directly translated into elementary constraints. Each of these constraints is a ternary or binary constraint and is itself decomposed into projection functions. A ternary symbolic expression $r = a \odot b$ where $\odot$ denotes one of the four arithmetical operations $\oplus, \ominus, \otimes, \oslash$, is decomposed into 3 projections: the direct projection $proj(r, r = a \odot b)$, the first inverse projection $proj(a, r = a \odot b)$ and the second inverse projection $proj(b, r = a \odot b)$. Inverse means that pro-

17

jection is performed on a right operand of an assignment. The variable a in $proj(a, r = a \odot b)$ is called the projected variable. Note that single assignment $r = a$ can be treated as the ternary symbolic expression $r = a \ominus +0.0$ because $a \ominus +0.0 = a$ even when $a = -0.0$. A binary symbolic expression $a = (type)b$ where $type$ is either $float$ or $double$ is decomposed into a direct projection $proj(a, a = (type)b)$ and an inverse one $proj(b, a = (type)b)$. A binary symbolic expression a rel b where rel denotes any of the six relational operators $==, <, =<, >, >=, ! =$ is decomposed into two projections : $proj(a, a \ rel \ b)$ and $proj(b, a \ rel \ b)$.

5.3.1 Computing direct projections for ternary symbolic expressions

Let $[r_l, r_h], [a_l, a_h], [b_l, b_h]$ be the current floating-point domains of r, a, b, then the direct projection $proj(r, r = a \odot b)$ computes new bounds r'_l, r'_h for the domain of r by using the formula of Fig.3.

$$
\begin{aligned}
&[r'_l, r'_h] \leftarrow [a_l \oplus b_l, a_h \oplus b_h] \cap [r_l, r_h] && \text{when } \odot = \oplus \\
&[r'_l, r'_h] \leftarrow [a_l \ominus b_h, a_h \ominus b_l] \cap [r_l, r_h] && \text{when } \odot = \ominus \\
&[r'_l, r'_h] \leftarrow [min(a_l \otimes b_l, a_l \otimes b_h, a_h \otimes b_l, a_h \otimes b_h), max(a_l \otimes b_l, a_l \otimes b_h, a_h \otimes b_l, a_h \otimes b_h)] \cap [r_l, r_h] \\
&\qquad\qquad\qquad \text{when } \odot = \otimes \\
&[r'_l, r'_h] \leftarrow [min(a_l \oslash b_h, a_l \oslash b_l, a_h \oslash b_h, a_h \oslash b_l), max(a_l \oslash b_h, a_l \oslash b_l, a_h \oslash b_h, a_h \oslash b_l)] \cap [r_l, r_h] \\
&\qquad\qquad\qquad \text{when } \odot = \oslash \text{ and } +0.0, -0.0 \text{ do not belong to } [b_l, b_h]
\end{aligned}
$$

Figure 3. Formulae for direct projections $proj(r, r = a \odot b)$

Although this remains implicit, it is important to bear in mind that these formulae are based on the to-the-nearest rounding mode. Note also that they was inspired by interval arithmetic [33,36] but differ from it [11]. Thanks the

---

[11] For example, the expected result over the reals of the sum of two numbers x and y can be captured by the interval $[\underline{z}, \overline{z}]$ where $\underline{z}$ (resp. $\overline{z}$) denotes the rounded toward negative (resp. positive) infinity result of $x + y$ [17].

monotonicity of the to-the-nearest rounding direction, these formula can directly be deduced from the interval arithmetic. The special case where $+0.0$ or -0.0 belongs to the right operand of the $\oslash$ operator can easily be handled by using infinities; this will be explained in the next section. Note also that the intersection of two intervals can be computed by using the formula $[a, b] \cap [c, d] = [max(a, c), min(b, d)]$ as the set of **numerical** floating-point values is totally ordered (even for both -0.0 and $+0.0$). Fig. 4 shows an example of application of the formula for the operator $\oplus$. The intervals of a, b, r are shown with vertical lines and each horizontal arrow represents the actual computation of the new bounds of r, before rounding. In this example, the new inferior bound of r is rounded up although the result over the reals $a_l + b_l$ is strictly less than the to-the-nearest rounded result of $a_l \oplus b_l$. This is due to the fact that $a_l + b_l$ is strictly greater than $mid((a_l \oplus b_l)^-, a_l \oplus b_l)$. This shows that the formula does not usually retain the solutions over the reals but handles all the solutions over the floating-point numbers.
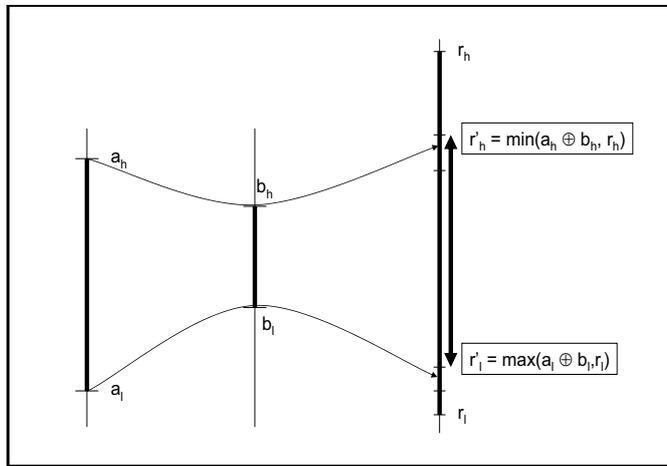

Figure 4. Computation of direct projection $proj(r, r = a \oplus b)$

Note that these formula for direct projections lead to an optimal pruning of the interval of r, because IEEE-754 guarantees that the four arithmetic operations are correctly rounded.

Inverse projections are a little bit more complicated to compute. The **first inverse** projection $proj(a, r = a \odot b)$ computes new bounds a'_l, a'_h for the domain of a whereas the **second inverse** projection $proj(b, r = a \odot b)$ computes new bounds b'_l, b'_h for the domain of b. The formulae to compute these inverse projections are given in Fig. 5. Note that the first and the second projections for $\oplus$ and $\otimes$ are the same. Thus, only one of them is given here.

$$[a'_l, a'_h] \leftarrow [mid(r_l, r_l^-) \ominus b_h, mid(r_h, r_h^+) \ominus b_l] \cap [a_l, a_h] \qquad \text{when } \odot = \oplus$$

$$[a'_l, a'_h] \leftarrow [mid(r_l, r_l^-) \oplus b_l, mid(r_h, r_h^+) \oplus b_h] \cap [a_l, a_h] \qquad \text{when } \odot = \ominus \text{ (first inverse)}$$

$$[b'_l, b'_h] \leftarrow [a_l \ominus mid(r_h, r_h^+), a_h \ominus mid(r_l, r_l^-)] \cap [b_l, b_h] \qquad \text{when } \odot = \ominus \text{ (second inverse)}$$

$$[a'_l, a'_h] \leftarrow \begin{array}{l} [min(mid(r_l, r_l^-) \oslash b_l, mid(r_l, r_l^-) \oslash b_h, mid(r_h, r_h^+) \oslash b_l, mid(r_h, r_h^+) \oslash b_h), \\ max(mid(r_l, r_l^-) \oslash b_l, mid(r_l, r_l^-) \oslash b_h, mid(r_h, r_h^+) \oslash b_l, mid(r_h, r_h^+) \oslash b_h)] \end{array} \cap [a_l, a_h]$$

$$\text{when } \odot = \otimes \text{ and } +0.0, -0.0 \text{ do not belong to } [b_l, b_h]$$

$$[a'_l, a'_h] \leftarrow \begin{array}{l} [min(mid(r_l, r_l^-) \otimes b_l, mid(r_l, r_l^-) \otimes b_h, mid(r_h, r_h^+) \otimes b_l, mid(r_h, r_h^+) \otimes b_h), \\ max(mid(r_l, r_l^-) \otimes b_l, mid(r_l, r_l^-) \otimes b_h, mid(r_h, r_h^+) \otimes b_l, mid(r_h, r_h^+) \otimes b_h)] \end{array} \cap [a_l, a_h]$$

$$\text{when } \odot = \oslash \text{ (first inverse)}$$

$$[b'_l, b'_h] \leftarrow \begin{array}{l} [min(a_l \oslash mid(r_l, r_l^-), a_h \oslash mid(r_l, r_l^-), a_l \oslash mid(r_h, r_h^+), a_h \oslash mid(r_h, r_h^+)), \\ max(a_l \oslash mid(r_l, r_l^-), a_h \oslash mid(r_l, r_l^-), a_l \oslash mid(r_h, r_h^+), a_h \oslash mid(r_h, r_h^+))] \end{array} \cap [b_l, b_h]$$

$$\text{when } \odot = \oslash \text{ (second inverse) and } +0.0, -0.0 \text{ do not belong to } [b_l, b_h]$$

Figure 5. Formula for inverse proj. $proj(a, r = a \odot b)$ and $proj(b, r = a \odot b)$

First, all inverse projections computes the middle of (r_l, r_l^-) and the middle of (r_h, r_h^+). The reason for that is that r is the result of a to-the-nearest rounding. More precisely, as the implemented operations are correctly rounded, they might be seen as the rounding to to-the-nearest of the result $r_{\mathbb{R}}$ over the reals of the same operation over the reals. Thus, if the floating point number r_l is the result of a to-the-nearest rounding, $r_{\mathbb{R}}$ has to belong to the interval [12]

---

[12] $[mid(r_l, r_l^-), mid(r_l, r_l^+)]$ is a conservative overestimation. A more precise interval could be computed if we take into account the value of the least significant bit of r_l (or r_h).

$[mid(r_l, r_l^-), mid(r_l, r_l^+)]$. The same reasoning applies to r_h. The computation of the middle of two single-format or double format floating-point variables can easily be computed as a wider format is almost always available [13] : the middle of two singles is captured by a double and the middle of two doubles is captured by an extended double. Note that the operations themselves are performed over a wider format, such as in the inverse projection of $\oplus : mid(r_l, r_l^-) \ominus b_h$ as shown in Fig. 6. Here, both operands of $\ominus$ are first converted into a greater format, although this remains implicit in the formula.
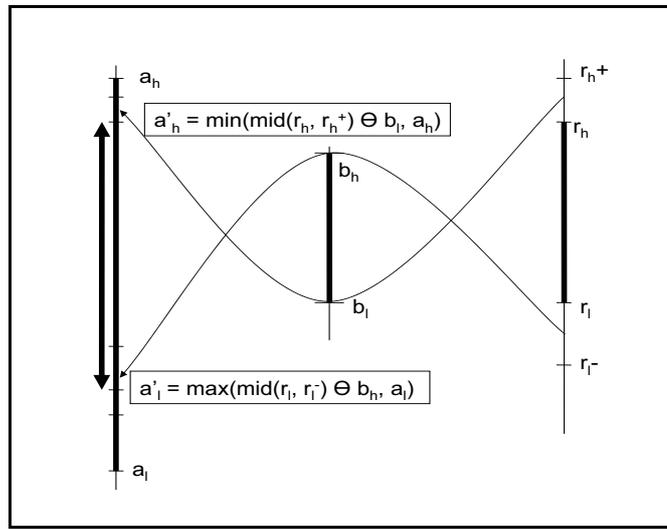

Figure 6. Computing first inverse projection $proj(a, r = a \oplus b)$

Second, special attention must be paid to the computation of the bounds of the projected variable. Operators $\oplus, \ominus, \otimes, \oslash$ are correctly rounded. Thus, they can be used to compute their inverse. The complete proof of this statement can be found in [23] and only an outline of it is given here. Consider the computation of a_h' for the addition in Fig. 6. As explained above, r_h is the

---

[13] Note however that an overestimation of the solution can still be computed using the same format as the operands, but this usually leads to a greater imprecision. For example, $[a_l', a_h'] \leftarrow [r_l^- \ominus b_h, r_h^+ \ominus b_l] \cap [a_l, a_h]$ is a conservative overestimation for the first inverse projection of the addition.

result of a to-the-nearest rounding of the addition of a' and b over the reals. Thus, over the reals, the following inequality holds : $a'_h + b \leq mid(r_h, r_h^+)$ where the floating-point number $b \in [b_l, b_h]$. Over the reals, this inequality leads to $a'_h \leq mid(r_h, r_h^+) - b_l$. In order to obtain a'_h, that is to say, in order to find the greatest floating-point number less or equal to $mid(r_h, r_h^+) - b_l$ (which is nothing but the definition of a rounding to $-\infty$), we would have to compute $mid(r_h, r_h^+) - b_l$ with a rounding to $-\infty$. However, a to-the-nearest rounding computes a conservative value for a'_h, *i.e.* a value that is equal or greater than the optimal value, and avoid the cost of a modification of the rounding mode.

As a consequence, the formula given here for computing the inverse projections are not always optimal but offer a conservative overestimation of the set of floating-point values that satisfy a given normalized symbolic expression. Considering the least significant bit of r_l and r_h can lead to slightly more shrinking [23] but requires changing the rounding mode several times during the computation of each projection function. Note also that interesting results from the literature can be used to improve the computation of inverse projections. For example, a classical result [37] says that if $x \oplus y$ underflows to a denormalized number then $x \oplus y$ is exactly equal to $x + y$. In such a case, the computation of the middle $mid(rh, rh^+)$ might be avoided.

5.4 *Handling comparisons and conversions*

Comparisons. Relational operators such as $==, >, >=, <, <=, != $ are handled by ordered set properties because the finite set of numerical floating-point variables is totally ordered. The formula is similar for the first and the second projections, hence only the first are given in Fig. 7. The floating-point domain

of a (resp. b) is $[a_l, a_h]$ (resp. $[b_l, b_h]$) and the domain of the result a' is $[a'_l, a'_h]$.

$$[a'_l, a'_h] \leftarrow [max(a_l, b_l), min(a_h, b_h)] \qquad \text{for } proj(a, a == b)$$

$$[a'_l, a'_h] \leftarrow [max(a_l, b_l), a_h] \qquad \text{for } proj(a, a \geq b)$$

$$[a'_l, a'_h] \leftarrow [max(a_l, b_l)^+, a_h] \qquad \text{for } proj(a, a > b)$$

$$[a'_l, a'_h] \leftarrow [a_l, min(a_h, b_h)] \qquad \text{for } proj(a, a \leq b)$$

$$[a'_l, a'_h] \leftarrow [a_l, min(a_h, b_h)^-] \qquad \text{for } proj(a, a < b)$$

$$[a'_l, a'_h] \leftarrow [\text{if } (a_l = b_l = b_h) \text{ then } a_l^+ \text{ else } a_l,$$
$$\text{if } (a_h = b_l = b_h) \text{ then } a_h^- \text{ else } a_h] \qquad \text{for } proj(a, a\mathord{!=}b)$$

Figure 7. Formulae for projections coming from comparison operators

These formulae are mainly inspired by interval arithmetic [33] but slightly differ from it for the computation of modified bounds. Here, the computation benefits from the fact that it operates over a finite set of floating-point values. **Conversions.** The simple language described in Sec.3.2 allows only two conversions $r = (float)a$ where a is a double and $r = (double)a$ where a is a single. Formulae that compute the bounds of projected variables with direct and inverse projections of conversion operators are given in Fig. 8. Note that any single-format value can be exactly converted into a double-format value. Thus, some conversions do not require any computation and remain implicit in the formulae.

$$[r'_l, r'_h] \leftarrow [max\_f((float)a_l, r_l), min\_f((float)a_h, r_h)] \qquad \text{for} \quad proj(r^f, r^f = (float)a^d)$$

$$[a'_l, a'_h] \leftarrow [max\_d(a_l, mid(r_l, r_l^-)), min\_d(a_h, mid(r_h, r_h^+))] \qquad \text{for} \quad proj(a^d, r^f = (float)a^d)$$

$$[r'_l, r'_h] \leftarrow [max\_d(a_l, r_l), min\_d(a_h, r_h)] \qquad \text{for} \quad proj(r^d, r^d = (double)a^f)$$

$$[a'_l, a'_h] \leftarrow [max\_f(a_l, r_l), min\_f(a_h, r_h)] \qquad \text{for} \quad proj(a^f, r^d = (double)a^f)$$

where r^f, a^f denote single-format variables, and r^d, a^d denote double-format variables,

max_f, min_f operate over the singles and max_d, min_d operate over the doubles

Figure 8. Formulae for projections coming from conversion operators

6 Handling symbolic values

IEEE-754 distinguishes two kinds of symbolic values: infinities and NaNs. The cases where infinities and NaNs can be produced as the result of a computation are detailed in [17]. However, implementing projection functions over symbolic values requires to further analysis of how to combine infinities, numerical values, zeros and NaNs and how to deal with exceptions [37].

In the proposed approach, the numerical domain is merely extended with both infinities and remains totally ordered. Roughly speaking, the main idea for computing projections consists in isolating the infinities from the numerical values of the domains, computing the projected variable's domain in the numerical case, combining the symbolic values between themselves, and merging the results of both the symbolic and the numerical cases.

To compute the projections of $\oplus$ (direct and inverse), tables 1 and 2 are required. Note that Nv stands for any non-zero numerical value and $\pm INF$ denotes any of the two infinities.

Table 1

Value of r in direct $proj(r, r = a \oplus b)$

$a \backslash b$	$-INF$	-0.0	$+0.0$	Nv	$+INF$
$-INF$	$-INF$	$-INF$	$-INF$	$-INF$	$-$
-0.0	$-INF$	-0.0	$+0.0$	Nv	$+INF$
$+0.0$	$-INF$	$+0.0$	$+0.0$	Nv	$+INF$
Nv	$-INF$	Nv	Nv	$Nv \cup \{\pm INF, +0.0\}$	$+INF$
$+INF$	$-$	$+INF$	$+INF$	$+INF$	$+INF$

Some combinations of symbolic values are impossible. For example, when $r = +0.0$ and $b = +INF$, the first inverse projection $proj(a, r = a \oplus b)$ computes an empty domain for variable a. Thus, there exists no floating-point value of

24

a able to satisfy the equation $+0.0 = a \oplus +INF$. These cases are indicated by the presence of the symbol $-$. When the operands of a projection are known and $-$ is encountered in the tables then the projection is refuted and the constraint store is shown to be contradictory. Note that when the sum of two opposite operands is exactly zero and the rounding mode is the to-the-nearest mode, then the result is $+0.0$ (and not -0.0). The cases where infinity is produced as the result of an operation over two numerical values (such as in $Nv \oplus Nv$) usually correspond to an overflow.

More frequently, operands are just known by their interval of possible values. Hence, when a combination of bounds is $-$, such as in $proj(a, r = a \oplus b)$ where $r \in [-INF, +INF]$ and $b \in [-INF, +INF]$, $-$ is just ignored and the interval of a is leaved unchanged (although $+0.0$ belong to the interval of r). The new bounds of r are computed using the formula of the numerical case ($[r'_l, r'_h] \leftarrow [a_l \oplus b_l, a_h \oplus b_h] \cap [r_l, r_h]$). Signed zeros, infinities and overflows are just special cases of this computation. If signed zeros belongs to the intervals of a or b then the numerical case ($Nv \oplus Nv$) of the table is applied. If an overflow occurs then the bounds are updated with the corresponding infinities.

Table 2

Value of a in first inverse $proj(a, r = a \oplus b)$

$b \backslash r$	$-INF$	-0.0	$+0.0$	Nv	$+INF$
$-INF$	$Nv \cup \{-INF, \pm 0.0\}$	$-$	$-$	$-$	$-$
-0.0	$-INF$	-0.0	$+0.0$	Nv	$+INF$
$+0.0$	$-INF$	$-$	± 0.0	Nv	$+INF$
Nv	$Nv \cup \{-INF\}$	$-$	$Nv \cup \{\pm 0.0\}$	$Nv \cup \{\pm 0.0\}$	$Nv \cup \{+INF\}$
$+INF$	$-$	$-$	$-$	$-$	$Nv \cup \{+INF, \pm 0.0\}$

The same procedure can be used for the computation of the projections of $\ominus, \otimes, \oslash$ using the tables given at the end of the paper. Note that the nega-

tive and positive numerical cases have not been distinguished in these tables. Although this is useful to implement better pruning of domains, these cases are not difficult to determine as simple sign rules remain valid in the context of non-zeros numerical floating-point values. Note that the only cases where NaN is produced when operands are non-NaNs are $\infty - \infty$ for $\oplus, \ominus$ and $0 * \infty, 0/0, \infty/\infty$ for $\otimes$ and $\oslash$.

7 A labelling procedure

As previously said, projection functions only reduce the domains of the variables. Thus, constraint propagation ensures neither the path conditions are satisfiable nor a test datum to be found in the general case. Note however that this process is efficient as it only requires $O(md)$ operations in the worst case where m denotes the number of constraints and d denotes the size of the largest domain [22]. To find a solution, a labelling procedure has to be implemented. Some heuristics are used to choose the variables and the values to be first enumerated. Several heuristics have been discussed in [38] and can easily be implemented. Note that in a symbolic execution framework, only the input variables need to be instantiated as all the other internal variables are computed in terms of these. As soon as a value is given to an uninstantiated variable, the interval propagator wakes up all the projection functions where this variable appears, thereby propagating the choice through the constraint system. In the applications of symbolic execution over floating-point variables, two difficult situations may sometimes occur at the end of the initial propagation step: either the path conditions have no solutions (*i.e.* the corresponding path is non-feasible) but this has not been detected, or the path conditions

26

have solutions but the resulting intervals are too approximate for it to be found. In these two related situations the labelling process is time-consuming and cannot be completed in all the cases. However, note there are always less than 2^{32} (resp. 2^{64}) possible values in the domain of a single-format (resp. double-format) floating-point value. So the process is no more time-consuming than the one used in constraint-based automatic test data generation environments over integers [39,25,11].

8 Implementation and experimental results

We implemented a symbolic execution tool for ANSI C floating-point computations, called FPSE (Floating Point Symbolic Execution). The tool extracts path conditions and symbolic expressions by a forward analysis and tries to solve them using the principles described in this paper. The constraint propagation engine of FPSE is written in Prolog whereas the projection functions are written in C.

FPSE handles floating-point computations that strictly conform to IEEE-754 and are intended to run on Sparc architectures. ANSI C accommodates the IEEE 754 floating point standard by not adopting any constraints on floating point which are contrary to this standard. In particular, it allows operations on `float` to be performed in single precision calculations. Note, however, that ANSI C gives the compiler a large degree of freedom in how to interpret and evaluate a floating-point expression to a precision wider than that normally associated with its type. While compiling the tested programs, it is necessary to avoid the use of compiler options that activate code optimizations as well as options that allow the storage of floating-point values into extended formats.

In practice, it is very difficult to guarantee that the symbolic execution will strictly conform to the actual execution because of several reasons: the lack of documentation of the compiler options and design, the existence of unexpected hardware optimizations such as the fused multiply-add a+b*c, the unexpected change of rounding modes by user actions, the defaults in the compiler implementation and so on. These limitations have to be taken into account when interpreting the results of FPSE.

8.1 Experimental results

To evaluate the approach, we compared the results provided by FPSE with expected floating-point results computed by hand and results obtained with three available solvers over the reals and the rationals. Distributed as part as the ECLIPSE Prolog system, are the following three distinct solvers:

(1) the *IC* library [19] which is an hybrid integer/real interval arithmetic constraint solver based on interval propagation. As in any other interval propagation solver **over the reals** (*e.g.* Ilog solver, RealPaver [20], Interlog [21,22]), each real number is represented by a pair of floating point bounds and any arithmetic operation is performed by using these bounds. The resulting interval is then widened to take into account any possible error in the operation, thus ensuring the resulting interval contains the true answer over the reals. This contrasts with our approach where floating-point numbers and operations are correctly approximated by relations over finite sets of floating-point numbers (also represented by pair of floating point bounds);

(2) the *clpr* library [27] that solves linear constraints **over the reals**. *clpr*

28

makes use of floating-point numbers to approximate computations over the reals;

(3) the *clpq* library [27] that solves linear constraints **over rationals** computed with an arbitrary precision. In *clpq*, each rational is treated as a pair of integers and any arithmetic computation remains exact;

Both solvers *clpr* and *clpq* exploit the simplex method and a Fourier-Motzkin algorithm to solve linear constraints. In addition, they provide several isolation axioms to take into account some restricted shapes of non-linear constraints.
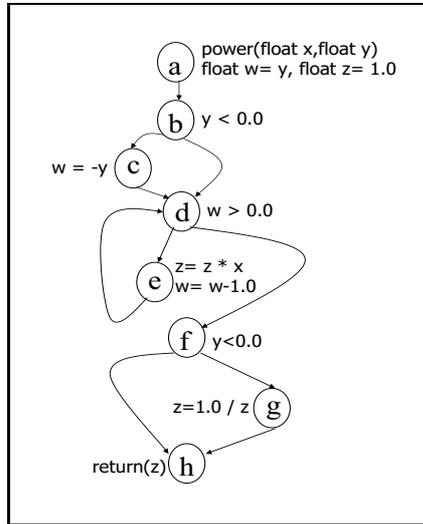
Figure 9. Control flow graph of program power.c

Programs. Several floating-point programs of small size were extracted from the literature to be carefully examined. We considered two distinct uses of symbolic execution: output symbolic expression computation and path feasibility.

Firstly, symbolic expressions were extracted from [17] and implemented in programs `g1.c, g2.c`. `g1.c` contains the C expression $X = ((2.0e-30 + 1.0e30) - 1.0e30) - 1.0e-30$ whereas `g2.c` contains $\delta == B^2 - 4AC$. For this

29

latter, two symbolic expressions were computed: the first one corresponds to **the direct evaluation** of the expression by taking $A = 1.22, B = 3.34, C = 2.28$ whereas the second one corresponds to **the inverse evaluation** where C is unknown and $\Delta == +0.0$. Symbolic expressions were extracted from paths of the program `power.c` that computes x^y, given in Fig.9. The two selected paths contain a number of iterations (40 and 350) that lead to overflows. All these symbolic expressions are given in the top of Tab.3.

Secondly, path feasibility was experimented with FPSE on path conditions extracted from programs `foo1.c` and `foo2.c` given in the introductory part of the paper (Fig.1,2), from the program `howden.c` that is a small-size numeric computation extracted from [40] and from the program `power.c` (Fig.9). For these programs, path conditions are given in the bottom of Tab.3. Second column provides the expressions as they appear in the literature. In particular, note that the path conditions of examples 8,9,10,11 results from a simplification process which has eliminated several redundant constraints. This process, as proposed for several symbolic execution tools [10], is unsound over floating-point variables as algebraic properties (such as associativity and distributivity) are not preserved. Third column of Tab.3 contains the normalized symbolic expressions as they are computed by the FPSE tool in the normalization process (sec.4). Finally, the last column contains the number of constraints present in the normalized path conditions.

All programs were compiled with `gcc` [14] on an ultra Sparc FPU under Solaris 2.7.

---

[14] `gcc-3.3.3 -g -Wall -DFPSE_SPARC -lm -std=gnu89 -ffloat-store -mhard-float -msoft-quad-float -munaligned-doubles` (some default options)

Table 3

Programs and FPSE expressions

	Program	Symbolic expr. over $\mathbb{R}$	Normalized FPSE expression	#
1	g1.c	$X = (2*10^{-30} + 10^{30}) - 10^{30} - 10^{-30}$	$T_2 = 2.0e - 30 \oplus 1.0e30, T_1 = T_2 \ominus 1.0e30,$ $X = T_1 \ominus 1.0e - 30$	3
2	g2.c	$\Delta = B^2 - 4*A*C$ with $A = 1.22, B = 3.34, C = 2.28$	$T_1 = B \otimes B, T_2 = A \otimes C, T_3 = 4.0 \otimes T_2,$ $\Delta = T_1 \ominus T_3$	4
3	g2.c	$\Delta = B^2 - 4*A*C$ with $A = 1.22, B = 3.34, \Delta = 0$	$T_1 = B \otimes B, T_2 = A \otimes C, T_3 = 4.0 \otimes T_2,$ $\Delta = T_1 \ominus T_3, \Delta == 0.0$	5
4	power.c (X=10,Y=-40) a-b-c-{d-e}^{40}-d-f-g-h	$RES = X^Y$ with $X = 10, Y = -40$	$W_1 = 0.0 \ominus Y, Z_1 = 1.0,$ $\{Z_{i+1} = Z_i * X, W_{i+1} = W_i - 1.0\}_{i=1..40},$ $Z_{42} = 1.0 \oslash Z_{41}, RES = Z_{42}$	84
5	power.c (X=10,Y=-350) a-b-c-{d-e}^{350}-d-f-g-h	$RES = X^Y$ with $X = 10, Y = -350$	$W_1 = 0.0 \ominus Y, Z_1 = 1.0,$ $\{Z_{i+1} = Z_i * X, W_{i+1} = W_i - 1.0\}_{i=1..350},$ $Z_{352} = 1.0 \oslash Z_{351}, RES = Z_{352}$	704
	Program	Path condition over $\mathbb{R}$	Normalized FPSE path conditions	#
6	foo1.c	$X > 0, X + 10^{12} = 10^{12}$	$X > 0.0, T_1 = X \oplus 1.0e12, T_1 = 1.0e12$	3
7	foo2.c	$X < 10^4, X + 10^{12} > 10^{12}$	$X < 10000.0, T_1 = X \oplus 1.0e12, T_1 > 1.0e12$	3
8	howden.c	$A*B + 2 > 100, 48 - A*B > 0$	$T_1 = A \otimes B, X_1 = T_1 \oplus 2.0, X_1 > 100.0,$ $X_2 = 100.0 \ominus X_1, X_3 = X_2 \ominus 50.0, X_3 > 50.0$	6
9	power.c (X,Y unknown) a-b-c-d-f-g-h	$Y < 0, Y \geq 0$	$Y < 0.0, W = \ominus Y, W \leq 0.0$	3
10	power.c (X,Y unknown) a-b-c-{d-e}^{40}-d-f-g-h	$Y < 0, Y < -39, Y \geq -40$	$Y < 0.0, W_1 = 0.0 \ominus Y,$ $\{W_i > 0.0, W_{i+1} = W_i - 1.0\}_{i=1..40}, W_{41} \leq 0.0$	83
11	power.c (X,Y unknown) a-b-c-{d-e}^{350}-d-f-g-h	$Y < 0, Y < -349, Y \geq -350$	$Y < 0.0, W_1 = 0.0 \ominus Y,$ $\{W_i > 0.0, W_{i+1} = W_i - 1.0\}_{i=1..350}, W_{351} \leq 0.0$	703

Results. In all the cases, the CPU time required to get the results with any of the four solvers (FPSE, IC, clpr, clpq) is less than a few seconds, so it is not shown. The first column contains the expected results computed either by executions of the C program or by manual analysis. In both cases, we provide the results over the singles and the doubles. Binary floating-point numbers are represented by decimal constants, noted with 16 decimals. The second column contains the results computed by the solvers over the reals and the rationals (IC, clpr and clpq). These solvers do not use single-format floating-point numbers, hence only the results over the double-format or the rationals is given. The last column contains the results computed by FPSE over both formats. Note that for any of the solvers (including FPSE), the labelling process has not been triggered and the results that are shown are obtained just after the constraint propagation step. Note that, as Eclipse IC is based on interval propagation, interval bounds are only changed if the absolute and relative changes of the bound exceed a given propagation threshold, which

is set to 1.0e-8.

Table 4

First experimental results

	Expected	with Eclipse	with FPSE
1	single:X = -1.0000000031710769e-30 double:X = -1.0000000000000001e-30	IC: X ∈ [-1.0e-30, 140737488355328] clpr: X = 0.0 clpq: X = 1/99999999999999879147136483328	single:X = -1.0000000031710769e-30 double:X = -1.0000000000000001e-30
2	single:Δ = 0.029199600219726562 double:Δ = 0.029200000000001225	IC: Δ ∈ [0.029199999999997672, 0.029200000000001225] clpr: Δ = 0.029200000000001152 clpq: Δ = 73/2500 = 0.0292	single:Δ = 0.029199600219726562 double:Δ = 0.029200000000001225
3	single:C = 2.2859835624694824 double:C = 2.2859836065573770	IC: C ∈ [2.2859836065573766, 2.2859839065573771] clpr: C = 2.285983606557377 clpq: C = 27889/12200	single:C ∈ [2.2859833240509033, 2.2859835624694824] double:C ∈ [2.2859836065573766, 2.2859836065573770]
4	single:RES = +0.0 double:RES = 1.00000000000000001e-40	IC: RES ∈ [9.9999999999999871e-41, 1.0000000000000016e-40] clpr: RES = 1.0e-40 clpq: RES = 10^{-40}	single: RES = +0.0 double:RES = 1.00000000000000001e-40
5	single:RES = +0.0 double: RES = +0.0	IC: RES ∈ [0.0, 5.56268464626801e-309] clpr: RES = 1.0e-350 clpq: RES = 10^{-350}	single: RES = +0.0 double: RES = +0.0
	Expected	with Eclipse	with FPSE
6	single:X ∈ [1.4012984643248171e-45, 3.2767998046875000e+04] double:X ∈ [4.9406564584124654e-324, 6.1035156250000000e-05]	IC: infeasible path clpr: infeasible path clpq: infeasible path	single:X ∈ [1.4012984643248171e-45, 3.2768000000000000e+04] double:X ∈ [4.9406564584124654e-324, 6.1035156250000000e-05]
7	single:infeasible path double:X ∈ [6.1035156250000000e-05, 9.9999999999999982e+03]	IC: X ∈ [0.0, 10000.0] clpr: -0.0 < X < 10000.0 clpq: 0 < X < 10000	single:infeasible path double:X ∈ [6.1035156250000000e-05, 9.9999999999999982e+03]
8	single:double:infeasible path	IC: infeasible path clpr: -48.0 + B*A < 0.0, 98.0 - B*A < 0.0 clpq: -48 + B*A < 0, 98 - B*A < 0	single:double:infeasible path
9	single:double:infeasible path	ic,clpr,clpq: infeasible path	single:double:infeasible path
10	single:Y ∈ [-4.0e01, -39.000003814697265625] double:Y ∈ [-4.0e01, -39.000000000000007105]	IC: Y ∈ [-40.0,-39.0] clpr: -40.0 ≤ Y < -39.0 clpq: -40 ≤ Y < -39	single:Y ∈ [-4.0e01, -39.0] double:Y ∈ [-4.0e01, -39.0]
11	single:Y ∈ [-350.0, -349.000030517578125] double:Y ∈ [-350.0, -349.00000000000005684]	IC: Y ∈ [-350.0,-349.0] clpr: -350.0 ≤ Y < -349.0 clpq: -350 ≤ Y < -349	single:Y ∈ [-350.0, -349.0] double:Y ∈ [-350.0, -349.0]

Analysis. First examples illustrate that the four evaluators may produce distinct results. In example 1, the results computed by both *clpr* and *clpq* are incorrect not only w.r.t. the expected result over the floats (first column) but also over the expected solutions over the reals (*i.e.* $+1.0e-30$). The library *IC* provides a correct but useless result over the reals as the superior bound of the

computed interval is greater than 10^{14}. As expected, FPSE provides the result strictly conforming to the evaluation of the program over the floating-point numbers (single and double), without any overestimation. Examples 2 and 3 show that even when expressions are not targeted to exemplify floating-point computation problems (`g2.c` computes the roots of the second order equation), the results given by the three solvers over the reals and the rationals (*IC, clpr, clpr*) do not conform to the ones computed by program executions. In example 3, FPSE returns an interval of 2 floating point values (in both cases) but only one of them satisfy the symbolic expression. Examples 4 and 5 show situations where floating-point numbers are flushed to zero by the computations, leading to a divergence with the computations over the reals (the program returns +0.0 instead of a strict positive quantity). FPSE provides the expected result as $1.0 \oslash +INF$ results in +0.0. Example 6 and 7 have already been discussed in the introduction of the paper. Examples 8 and 9 demonstrate path infeasibility. In example 8, both *clpr* and *clpq* return an unsolved non-linear constraint system. Solvers based on interval propagation (IC,FPSE) are not restricted to deal with linear constraints hence path infeasibility is shown. In example 9, all the four solvers provide the expected result. Finally, examples 10 and 11 illustrate the capacity of the solvers to deal with a realistic number of constraints, even when inverse projections are involved. In examples 8,9,10,11, IC and FPSE return the same (possibly overestimated) correct results at the end of the constraint propagation step, but only FPSE is trustworthy over the floating-point numbers.

To conclude, these experiments demonstrate that the proposed approach is suitable to deal efficiently with small-sized C floating-point computations. Of course, the set of experiments is too restricted to easily extrapolate the results

33

to larger computations but this work is a first attempt to address the problem of floating-point computations in symbolic execution.

9 Further work

In this paper, a new symbolic execution framework able to handle correctly IEEE-754 compliant floating-point computations has been introduced. The definitions of correct and efficient projection functions for solving normalized symbolic expressions have been given. Handling other rounding modes than the to-the-nearest number appears as being a tedious but not difficult extension of the proposed framework. In the same spirit, handling the square root function is straightforward: this function is included in the IEEE-754 standard and is correctly rounded. Dealing with extended formats appears to be an interesting extension as computations require more and more precision. This extension probably requires using multiple-precision floating-point numbers, as exploited in some computer algebra systems. The most difficult extension concerns the transcendental functions as there is nothing to guarantee that the computation is correctly rounded in these cases. This problem known as the table maker dilemma problem is likely to be the more prospective part of future work on this topic.

Acknowledgements

References

[1] King, J.C., "Symbolic execution and program testing", *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, July 1976.

[2] Goldberg, A. and Wang, T. and Zimmerman, D., "Applications of feasible path analysis to program testing", in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'94)*, Seattle, WN, August 1994, pp. 80–92.

[3] Jasper, R. and Brennan, M. and Williamson, K. and Zimmerman, D., "Test data generation and feasible path analysis", in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'94)*, Seattle, WN, August 1994, pp. 95–107.

[4] Weyuker, E., "Translatability and decidability questions for restricted classes of program schemas", *SIAM Journal of Computing*, vol. 8, no. 4, pp. 587–598, November 1979.

[5] Clarke, L., "A system to generate test data and symbolically execute programs", *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 215–222, September 1976.

[6] Howden, W., "Reliability of the path analysis testing strategy", *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 208–214, September 1976.

[7] Boyer, R. and Elspas, B. and Levitt, K., "SELECT - a formal system for testing and debugging programs by symbolic execution", *SIGPLAN Notices*, vol. 10, no. 6, pp. 234–245, June 1975.

[8] Ramamoorthy, C. and Ho, S. and Chen, W., "On the automated generation of program test data", *IEEE Transactions on Software Engineering*, vol. 2, no. 4,

pp. 293–300, December 1976.

[9] Bicevskis, J. and Borzovs, J. and Straujums, U. and Zarins, A. and Miller, E., "SMOTL - a system to construct samples for data processing program debugging", *IEEE Transactions on Software Engineering*, vol. 5, no. 1, pp. 60–66, January 1979.

[10] Coward, D. and Ince, D., *The Symbolic Execution of Software - The SYM-BOL System*, Chapman & Hall, London, UK, 1995.

[11] Meudec, C., "ATGen: automatic test data generation using constraint logic programming and symbolic execution", *Software Testing, Verification and Reliability*, vol. 11, no. 2, pp. 81–96, June 2001.

[12] DeMillo, R.A. and Offut, J.A., "Experimental results from an automatic test case generator", *ACM Transactions on Software Engineering Methodology*, vol. 2, no. 2, pp. 109–127, April 1993.

[13] Coen-Porisini, A. and de Paoli, F. and Ghezzi, C. and Mandrioli, D., "Software specialization via symbolic execution", *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 884–899, September 1991.

[14] Fahringer, T. and Scholz, B., "A unified symbolic evaluation framework for parallelizing compilers", *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 11, pp. 1105–1125, November 2000.

[15] Coen-Porisini, A. and Denaro, G. and Ghezzi, C. and Pezze, M., "Using symbolic execution for verifying safety-critical systems", in *Proceedings of the European Software Engineering Conference (ESEC/FSE'01)*, Vienna, Austria, September 2001, ACM, pp. 142–150.

[16] Chen, T.Y. and Tse, T.H. and Zhou, Z., "Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing", in *Proceedings*

of the International Symposium on Software Testing and Analysis (ISSTA'02), Roma, Italy, July 2002, pp. 191–195.

[17] Goldberg, D., "What every computer scientist should know about floating-point arithmetic", *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, March 1991.

[18] IEEE-754, "Standard for binary floating-point arithmetic", *ACM SIGPLAN Notices*, vol. 22, no. 2, pp. 9–25, February 1985.

[19] Brisset, P. and Sakkout, H. and Fruhwirth, T. and Gervet, C. and Harvey, et al., *ECLiPSe Constraint Library Manual*, International Computers Limited and Imperial College London, UK, 2005, Release 5.8.

[20] Granvilliers, L., *RealPaver User's Manual : Solving Nonlinear Constraints by Interval Computations*, University of Nantes, FR, 2003, Release 0.3.

[21] Botella, B. and Taillibert, P., "Interlog : Constraint logic programming on numeric intervals", in *Third International Workshop on Software Engineering, Artificial Intelligence and Expert Systems*, Oberammergau, October 1993.

[22] Lhomme, O. and Gotlieb, A. and Rueher, M. and Taillibert, P., "Boosting the interval narrowing algorithm", in *Proceeedings of the Joint International Conference and Symposium on Logic Programming (JICSLP'96)*, Bonn, September 1996, MIT Press, pp. 378–392.

[23] Michel, C., "Exact projection functions for floating point number constraints", in *Proceedings of seventh AIMA Symposium*, Fort Lauderdale, FL, USA, 2002.

[24] Miller, W. and Spooner, D., "Automatic generation of floating-point test data", *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 223–226, September 1976.

[25] Gotlieb, A. and Botella, B. and Rueher, M., "Automatic test data generation using constraint solving techniques", in *Proceedings of the International*

Symposium on Software Testing and Analysis (ISSTA'98), Clearwater Beach, FL, USA, March 1998, pp. 53–62.

[26] Gotlieb, A. and Botella, B. and Rueher, M., "A clp framework for computing structural test data", in *Proceedings of Computational Logic (CL'2000)*, London, UK, July 2000, LNAI 1891, pp. 399–413.

[27] Holzbaur, C., *OEFAI clp(q,r) Manual Rev. 1.3.2*, Austrian Research Institute for Artificial Intelligence, Vienna, AU, 1995, TR-95-09.

[28] Muchnick, S. and Jones, N., *Program Flow Analysis: Theory and Applications – Chapter 9 : L. Clarke, D. Richardson*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[29] Coen-Porisini, A. and de Paoli, F., "Array representation in symbolic execution", *Computer Languages*, vol. 18, no. 3, pp. 197–216, 1993.

[30] Dillon, E. and Meudec, C., "Automatic test data generation from embedded c code", in *Proceedings of SAFECOMP'04*, Potsdam, Germany, September 2004, Springer Verlag, LNCS 3219, pp. 180–194.

[31] Benhamou, F. and McAllester, D. and van Hentenryck, P., "CLP(Intervals) revisited", in *Proceedings of the 1994 International Symposium on Logic Programming (ILPS'94)*, Ithaca, New York, November 1994, pp. 124–138, MIT Press.

[32] Benhamou, F. and Older, W., "Applying interval arithmetic to real, integer and boolean constraints", *Journal of Logic Programming*, vol. 32, no. 1, pp. 1–24, July 1997.

[33] Moore, R.A., *Interval Analysis*, Prentice Hall, New Jersey, 1966.

[34] Cleary, J.G., "Logical arithmetic", *Future Computing Systems*, vol. 2, no. 2, pp. 125–149, 1987.

[35] Older, W. and Vellino, A., "Extending prolog with constraints arithmetic on reals intervals", in *Proceedings of IEEE Canadian Conference on Electrical and Computer Engineering*. 1990, IEEE Computer Society Press.

[36] Hickey, T.J. and Ju, Q. and van Emden, M.H., "Interval arithmetic: From principles to implementation", *Journal of ACM*, vol. 48, no. 5, pp. 1038–1068, September 2001.

[37] Hauser, J.R., "Handling floating-point exceptions in numeric programs", *ACM Transactions on Programming Language and Systems*, vol. 18, no. 2, pp. 139–174, March 1996.

[38] Michel, C. and Rueher, M. and Lebbah, Y., "Solving constraints over floating-point numbers", in *Proceedings of Principles and Practices of Constraint Programming (CP'01)*, Paphos, Cyprus, November 2001, Springer Verlag, LNCS 2239, pp. 524–538.

[39] DeMillo, R.A. and Offut, J.A., "Constraint-based automatic test data generation", *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, September 1991.

[40] Howden, W., "Validation of scientific programs", *ACM Computing Surveys*, vol. 14, no. 2, pp. 193–227, June 1982.

Appendix

This appendix contains the tables used in direct and inverse projections when infinities are involved in the computations.

Table 5

Value of r in direct $proj(r, r = a \ominus b)$

$_a\backslash^b$	$-INF$	-0.0	$+0.0$	Nv	$+INF$
$-INF$	$-$	$-INF$	$-INF$	$-INF$	$-INF$
-0.0	$+INF$	$+0.0$	-0.0	Nv	$-INF$
$+0.0$	$+INF$	$+0.0$	$+0.0$	Nv	$-INF$
Nv	$+INF$	Nv	Nv	$Nv \cup \{\pm INF, +0.0\}$	$-INF$
$+INF$	$+INF$	$+INF$	$+INF$	$+INF$	$-$

Table 6

Value of r in direct $proj(r, r = a \otimes b)$

$_a\backslash^b$	$-INF$	-0.0	$+0.0$	Nv	$+INF$
$-INF$	$+INF$	$-$	$-$	$-INF$	$-INF$
-0.0	$-$	$+0.0$	-0.0	$\{\pm 0.0\}$	$-$
$+0.0$	$-$	-0.0	$+0.0$	$\{\pm 0.0\}$	$-$
Nv	$\{\pm INF\}$	$\{\pm 0.0\}$	$\{\pm 0.0\}$	$Nv \cup \{\pm 0.0, \pm INF\}$	$\{\pm INF\}$
$+INF$	$-INF$	$-$	$-$	$+INF$	$+INF$

Table 7

Value of r in direct $proj(r, r = a \oslash b)$

$_a\backslash^b$	$-INF$	-0.0	$+0.0$	Nv	$+INF$
$-INF$	$-$	$+INF$	$-INF$	$\{-INF, +INF\}$	$-$
-0.0	$+0.0$	$-$	$-$	$\{\pm 0.0\}$	-0.0
$+0.0$	-0.0	$-$	$-$	$\{\pm 0.0\}$	$+0.0$
Nv	$\{\pm 0.0\}$	$\{\pm INF\}$	$\{\pm INF\}$	$Nv \cup \{\pm 0.0, \pm INF\}$	$\{\pm 0.0\}$
$+INF$	$-$	$-INF$	$+INF$	$\{-INF, +INF\}$	$-$

Table 8

Value of a in first inverse $proj(a, r = a \ominus b)$

$_b \backslash ^r$	$-INF$	-0.0	$+0.0$	Nv	$+INF$
$-INF$	$-$	$-$	$-$	$-$	$Nv \cup \{+INF, \pm 0.0\}$
-0.0	$-INF$	$-$	$\{\pm 0.0\}$	Nv	$+INF$
$+0.0$	$-INF$	-0.0	$+0.0$	Nv	$+INF$
Nv	$Nv \cup \{-INF\}$	$-$	Nv	$Nv \cup \{\pm 0.0\}$	$Nv \cup \{+INF\}$
$+INF$	$Nv \cup \{-INF, \pm 0.0\}$	$-$	$-$	$-$	$-$

Table 9

Value of a in first inverse $proj(a, r = a \otimes b)$

$_b \backslash ^r$	$-INF$	-0.0	$+0.0$	Nv	$+INF$
$-INF$	$Nv \cup \{+INF\}$	$-$	$-$	$-$	$Nv \cup \{-INF\}$
-0.0	$-$	$Nv \cup \{+0.0\}$	$Nv \cup \{-0.0\}$	$-$	$-$
$+0.0$	$-$	$Nv \cup \{-0.0\}$	$Nv \cup \{+0.0\}$	$-$	$-$
Nv	$Nv \cup \{\pm INF\}$	$\{\pm 0.0\}$	$\{\pm 0.0\}$	Nv	$Nv \cup \{\pm INF\}$
$+INF$	$Nv \cup \{-INF\}$	$-$	$-$	$-$	$Nv \cup \{+INF\}$

Table 10

Value of a in first inverse $proj(a, r = a \oslash b)$

$_b \backslash ^r$	$-INF$	-0.0	$+0.0$	Nv	$+INF$
$-INF$	$-$	$Nv \cup \{+0.0\}$	$Nv \cup \{-0.0\}$	$-$	$-$
-0.0	$Nv \cup \{+INF\}$	$-$	$-$	$-$	$Nv \cup \{-INF\}$
$+0.0$	$Nv \cup \{-INF\}$	$-$	$-$	$-$	$Nv \cup \{+INF\}$
$+INF$	$-$	$Nv \cup \{-0.0\}$	$Nv \cup \{+0.0\}$	$-$	$-$
Nv	$\{\pm INF\}$	$\{\pm 0.0\}$	$\{\pm 0.0\}$	Nv	$\{\pm INF\}$