Correct Handling of Floating-Point Computations in Symbolic Execution

Bernard Botella, Arnaud Gotlieb, Claude Michel

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Correct Handling of Floating-Point Computations in Symbolic Execution

Bernard Botella , Arnaud Gotlieb , Claude Michel

N°5150

March 2004

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--5150--FR+ENG

Correct Handling of Floating-Point Computations in Symbolic Execution

Bernard Botella* , Arnaud Gotlieb† , Claude Michel‡

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lande

Rapport de recherche n ° 5150 — March 2004 — 25 pages

Abstract: Symbolic execution is a program testing technique which evaluates statements with symbolic input data along a selected path of the control flow graph. The process involves the computation of path conditions that tend to be simplified or solved in order either to get test data that sensitize the selected path or to demonstrate infeasibility of the path. In the presence of floating-point computations, the current strategy consists in using a constraint solver based on rationals or reals. Unfortunately, even when the computations conform ANSI/IEEE-754 floating-point arithmetic, this leads not only to approximative results but also to incorrect ones. For example, a path can be labeled as infeasible by using a constraint solver on the rationals although there exist floating-point input data that sensitize it. This paper shows how to evaluate symbolically the expressions when floating-point variables are involved in the computations. The focus is on the design and the implementation of projection functions required to solve path conditions over the floats. The proposed approach handles not only the numeric values of floating-point variables but also the symbolic values, such as infinities and NaNs. A symbolic execution environment of C floating-point computations is currently under development. Some very first experimental results are reported.

Key-words: Program Testing, Floating-point computations, Symbolic execution, IEEE-754

(Résumé : tsvp)

* THALES Airborne Systems, 2 av. Gay-Lussac, 78851 Elancourt Cedex, FRANCE
† Projet Lande – IRISA / INRIA – Campus de Beaulieu, 35042 Rennes Cedex, FRANCE
‡ I3S-CNRS / INRIA, 2004 route des Lucioles, 06902 Sophia Antipolis Cedex, FRANCE

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Traitement correct des calculs à virgule flottante en exécution symbolique

Résumé : L'exécution symbolique est une technique de test qui évalue les instructions d'un programme avec des données d'entrée symboliques, le long d'un chemin sélectionné dans le graphe de flot de contrôle. Ce procédé entraîne le calcul de conditions de chemin qui tendent à être simplifiées ou résolues dans le but de trouver des données de test qui sensibilisent le chemin sélectionné ou de démontrer la non-exécutabilité de ce chemin. En présence de calculs à virgule flottante, la stratégie actuellement en vigueur consiste à utiliser un solveur de contraintes basé sur les rationnels ou les réels. Malheureusement, même lorsque les calculs sont conformes à la norme ANSI/IEEE-754, cela conduit non seulement à l'obtention de résultats approximatifs mais aussi à l'obtention de résultats incorrects. Par exemple, un chemin peut être indiqué comme étant non-exécutable en utilisant un solveur sur les rationnels, tandis qu'il existe des données d'entrée flottantes qui sensibilisent ce chemin. Ce papier montre comment évaluer symboliquement les expressions lorsque des variables à virgule flottante sont impliquées dans le calcul. Le papier se concentre sur la description et l'implantation des fontions de projection requise pour la résolution des conditions de chemin sur les flottants. L'approche proposée prend en compte non seulement les valeurs flottantes numériques mais aussi les valeurs symboliques, telles que les infinités et les NaNs. Un environnement d'exécution symbolique de calculs C est en cours de développement. Nos tous premiers résultats expérimentaux sont présentés.

Mots-clé : Test Logiciel, Calculs à virgule flottante, Exécution symbolique, IEEE-754

1 Introduction

Symbolic execution, as introduced by King in the context of Software Testing [19], consists in evaluating each statement along a selected path of the control flow graph, with symbolic input data. This evaluation yields to formula which only depend on the symbolic input data. Symbolic execution also computes path conditions which are sets of constraints that characterize the selected path. They tend to be simplified or solved to obtain input data that sensitize the path. When one proves that the path conditions have no solution then the selected path is shown to be infeasible. This is not always possible since finding all of the infeasible paths in the general case is a classical undecidable problem [29]. Applications of symbolic execution include automatic test data generation [5, 16, 3, 28, 2, 7, 21], path feasibility analysis [9], program specialization [6], program proving [4] just to name a few.

```
float foo(float x) {
      float y = 1.0e12, z = 0.0 ;
1.    if (x < 10000.0)
2.          z = x + y ;
3.    if (z > y)
4.          ...
```

Figure 1: Program foo

An illustrating example. It is well known that solving path conditions by reasoning over the reals or the rationals leads to some inconsistencies when the results are transposed on the floating point variables [10, 13]. Although its computations are done with floating-point values, an automatic constraint solver over the reals is usually conservative of all the reals solutions but it may loose solutions over the floats [23]. So, even when a program conforms the ANSI/IEEE-754 standard for binary floating-point arithmetic [18], incorrect deductions can occur. Consider the C program given in Fig.1 and the symbolic execution of path 1-2-3-4. Path conditions can be written as $\{x < 10000.0, x \oplus 1.0e12 > 1.0e12\}$, where $\oplus$ denotes the addition operator over the floats[1]. It isn't difficult to see that all the reals x of the open interval $(0, 10000)$ are solutions of this set of constraints if $\oplus$ is interpreted over the reals. However, there is no single floating-point value able to sensitize the path 1-2-3-4 because any single float in $(0, 10000)$ will be absorbed by the $\oplus$ operator. Hence the path 1-2-3-4 is actually infeasible. Conversely, consider the constraints $\{x > 0.0, x \oplus 1.0e12 = 1.0e12\}$ which can easily be derived from small modifications of the same C program. This set of constraints doesn't have any solution over the reals but it has numerous solutions over the floats (all the single-format floating-point numbers in the interval $[1.401298464324817E-45, 32767.9990234]$). In this case, the corresponding path would be labeled as infeasible by a symbolic execution tool over the reals although this is incorrect over the floats. This example shows that solving path conditions over the reals (or the rationals) may not only lead to approximative results but also to incorrect ones.

[1] Type "float" denotes the single format of IEEE-754

Existing work. Although a lot of work has been carried out in the Numerical Analysis field on related problems, only a few studies deals with floating-point numbers in the Software Testing community. To the best of our knowledge, only Miller and Spooner [24] studied how to generate automatically floating-point test data for imperative programs. Their work opened the road of execution-based test data generation methods which don't suffer of the problems mentioned above. At a time when no standard existed for floating-point arithmetic, symbolic execution was pioneered by King [19], Clarke [5], Howden [16] and others [3, 28, 2] in several systems. SELECT [3] and DAVE [5] used Linear Programming algorithms to solve the path conditions by combining several techniques : the Gomory's cutting planes algorithm for integer constraints, the Bender's decomposing algorithm and an algorithm due to Glover [5] for solving mixed linear inequalities, and a conjugate gradient algorithm for non-linear inequalities over the reals [3]. CASEGEN [28] utilized an ad-hoc procedure for solving non-linear equations and inequalities over the reals which was based on a try-by-value method. SMOTL [2] solved integers inequalities by using a domain reduction technique. The system iteratively examined integer inequalities : the domain of each inequality variable was pruned until a fixpoint were reached. More recently, GODZILLA [8] and its extensions [27] revamped this approach in solving path conditions over integers by using sophisticated heuristics for choosing the variables and values to enumerate first. Gotlieb et al. pioneered in [11, 12] the use of Constraint Logic Programming over finite domains to solve constraints extracted from imperative programs. The proposed framework dealt with non-linear constraints over integers to automatically generate test data. Although not strictly based on symbolic execution[2], their approach faced also the problem of floating-point computations. More recently, Meudec followed a similar path in [21] and proposed to solve path conditions over floating-point variables by using a constraint solver over infinite precision rationals, which is based on an extended[3] version of the simplex algorithm.

From 2001, the authors participated to the project INKA which was devoted to the design of an automatic test data generation tool based on Constraint Logic Programming. Within the project, Michel [22] threw the theoretical basis of a constraint solver over an idealized set of floating-point numbers. To face the remaining theoretical and practical difficulties, another project called V3F[4] was launched in 2003 to alleviate the technical problems of verifying programs in the presence of floating-point computations.

Contributions of the paper. This paper introduces new results to accurately solve the path conditions generated by a symbolic executor, when floating-point computations are involved. The paper extends the theoretical work of Michel [22] on the design of exact projection functions that can be embedded into a constraint solver for pruning the domains of floating-point variables. We explain in a practical way how to build efficient projection functions not only for the arithmetic operators [22] but also for the comparison and format-conversion operators. The proposed approach deals also with symbolic values such as infinities and NaNs which haven't been studied elsewhere in the context of sym-

[2] In [11], the proposed framework is rather based on Static Single Assignment form

[3] that handles some restricted form of non-linear constraints

[4] `http://lifc.univ-fcomte.fr/~v3f/`

bolic execution. A symbolic executor for C floating-point computations is currently under development. The very first experimental results we got with it are reported.

Contents. Section 2 contains the basic material coming from the IEEE-754 standard for binary floating-point arithmetic and indicates the restrictions of our approach. Section 3 recalls the main principles of symbolic execution. Section 4 presents the design of efficient projection functions over numeric floating-point variables. Section 5 explains how to deal with special symbolic values such as infinities and NaNs. Section 6 describes our current implementation and reports first experimental results over C floating-point computations. Last section discusses further work to extend our approach.

2 Basic Materials

This section presents some points of the arithmetical model specified by the ANSI/IEEE-754 standard binary floating-point arithmetic [18].

2.1 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 represents the biased exponent and f stands for the significand. The single format occupies 32 bits ($s = 1, e = 8, f = 23$) while the double occupies 64 bits ($s = 1, e = 11, f = 52$). The standard doesn't prescribe the exact precision and size of the extended formats, but it does specify their minimum precision. For example, double extended must occupy at least 79 bits. Each format defines several classes of numbers : normalized numbers, denormalized numbers, signed zeros, infinities and NaNs (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 but 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 to maintain certain algebraic properties [10]. NaNs ($e = 255, f \neq 0$) are used to represent the results of invalid or dubious computations such division by zero or subtraction of infinities. They allow an execution where no exception is raised. IEEE-754 indicates four types of rounding directions: toward the nearest representable value, with "even" values preferred whenever there are two nearest representable values (near-to-even), toward negative infinity (down), toward positive infinity (up) and toward zero (chop). The most important requirement of IEEE-754 arithmetic concerns 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 minimum modification of the exact result w.r.t. the prescribed rounding mode and the result

format destination. It is said that these operators are correctly rounded[5]. For example, the single-format result of $999999995904 \oplus 10000$ is[6] 999999995904 which is the single-format floating point number nearest to the exact result over the reals. This example shows that although the accuracy requirement of IEEE-754 is very strong, inconsistent results over the reals may easily be obtained (the second operand is absorbed by $\oplus$). The maximal error around a floating-point number x is represented by a function $ulp(x)$ (units in the last place). Its value on 1.0 is about 10^{-7}.

2.2 Restrictions and notations

In the rest of the paper, we will suppose that programs and architectures strictly conform to the IEEE-754 standard and manipulate only the single or the double-format. Our framework currently handles only the near-to-even rounding direction, which is the default rounding mode. However, the same principles apply to other rounding modes. When a decimal constant such as $1.0e12$ denotes a floating-point value, we will keep this notation in the examples instead of using its correct form of the nearest binary floating-point value. It is implicit that every constant is interpreted as such in machine when one uses the near-to-even rounding mode. x^+ (resp. x^-) denotes the smallest (resp. greatest) float greater (resp. smaller) than x, depending on the format. When x is exactly located on the middle of (x^-, x^+) then the near-to-even rounding operation requires that the least significant bit of the rounded result is zero. $mid(a, b)$ denotes the floating-point middle[7]. $\oplus, \ominus, \otimes, \oslash$ denote the floating-point operations (i.e. the format dependent near-to-even result of the exact result) whereas $+, -, *, /$ denote the operations over the reals. This paper addresses only the problem of dealing with floating-point variables in symbolic evaluation frameworks. Other issues of symbolic execution such as loops, arrays and pointers aren't discussed here. Hence, we confine ourselves to a simple language over the floats. Its C-like syntax is given in Fig.2. This simple language possesses only floating-point data types because we considered that the combination of integers and floating-point numbers in a symbolic execution framework was outside the scope of this paper.

3 Symbolic execution

Control flow graph and paths. The control flow graph of a program P is a connected oriented graph composed by a set of vertices, a set of edges and two particular nodes, e the unique entry node, and s the unique exit node. Each node represents a basic block and each edge represents the 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. A terminating execution in P follows a single path from e to s. Let $Var(P)$ denote the set of variables of P and $In(P)$ the subset of input variables of P. Variables of $Var(P)$ that are not in $In(P)$

[5] IEEE-754 says equivalently "exactly rounded"

[6] Note that these two decimals can be exactly represented by single binary floating-point numbers

[7] which is a floating-point of an extended format of its two operands a and b

```
pgm ::= type id(decl){  decl stmt return (lexp) }

type ::= float | double
decl ::= ϵ | type lexp | decl ; decl

stmt ::= ϵ | lexp = exp
         | if (exp) {  stmt } else {  stmt }
         | stmt ; stmt

exp ::=  single_constant | double_constant | id
         | ( type ) id          conversions
         | exp op exp           op in {⊕, ⊖, ⊗, ⊘}
         | exp rel exp          rel in {==, ! =, >, >=, <, <=}
lexp ::= id
```

Figure 2: A simple language over the floats

are called internal variables.

Symbolic states and expressions. Symbolic execution works by computing symbolic states for each program's variable. A *symbolic state* for path $e - n_1 - \ldots - n_k$ in P is a triple $(e - n_1 - \ldots - n_k, \{(v, \phi_v)\}_{v \in Var(P)}, c_1 \wedge \ldots \wedge c_n)$ where ϕ_v is a symbolic expression associated with the variable v and $c_1 \wedge \ldots \wedge c_n$ is a conjunction of symbolic expressions, called path conditions. By *symbolic expression*, we mean a well parenthesized expression that hold solely over $In(P)$. In fact, when computing a new symbolic expression, each internal variable reference is replaced by its previous symbolic expression over $In(P)$. For example, the symbolic state of path 1-2-3-4 in the program of Fig.1 is

$$(\text{1-2-3-4}, \{(x, X), (y, 1.0e12), (z, X \oplus 1.0e12)\},$$

$$X < 10000.0 \wedge X \oplus 1.0e12 > 1.0e12)$$

Note that a valuation of $In(P)$ sensitizes $e - n_1 - \ldots - n_k$ if and only if it satisfies the path conditions.

Forward/backward analysis. Symbolic states are computed by induction on their path by a forward or a backward analysis [26]. Each statement of each node of the path is symbolically evaluated by using an evaluation function which computes the symbolic states. In forward analysis, the statements of the selected path are followed within the same direction of actual program execution, whereas the reverse direction is used in backward analysis. Backward analysis is usually preferred when one wants only to compute the path conditions.

Analyzing floating-point computations. In the presence of floating-point computations, special attention must be paid to conform the actual execution of program. The

idea is to conform the expression's shape of the abstract syntax tree build by the compiler without any rearrangement nor any simplification due to optimizations. When symbolic expressions are directly extracted from the abstract syntax tree, then not only the operator's priority is respected but also is the order on which operands are evaluated. This is not necessarily the case when symbolic expressions are extracted from source code by an analyzer. Preserving the order of evaluation in our analyzer is essential with floating-point computations as simple algebraic properties such as associativity or distributivity are lost. We propose here a decomposing approach of expressions that takes into account the above requirements and yields to normalization. This allows the handling of symbolic expressions over the floats independently of their compiling environment.

Normalization. Any of the symbolic expressions is decomposed into a sequence of assignments where fresh temporary variables are introduced keeping 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 $t_1 \oplus v_4 \wedge t_1 = v_1 \otimes t_2 \wedge t_2 = v_2 \otimes v_3$ because $\otimes$ has a higher priority than $\oplus$ and operands are evaluated from the left to the right. Our decomposing approach requires that intermediate results of an operation conforms the type's storage of its operands[8]. In the above example, if v_2 and v_3 are single-format floating-point variable, then the temporary variable t_2 must also be of single-format. As a result of this decomposition, path conditions are only composed of binary or ternary symbolic expressions that have a single operator over a known floating-point format. The resulting form is what we call a normalized form of a symbolic expression.

4 Solving path conditions over the floats

In this section, the floating-point variables possess a numerical value and the computations do not overflow or raise exceptions. The results presented here are based on the theoretical work of [22] where formal proofs can be found. Extensions to symbolic values such as infinities and NaNs are introduced in the following section.

Path conditions are composed of normalized symbolic expressions over floating-point input and temporaries variables. Each of these variables takes its numeric values into a finite domain of possible floating-point values w.r.t. its format. Domains are represented by a couple of bounds (intervals) that can possibly be provided by the user. By default, any numeric single-format floating-point values belongs to $[-3.40282347e38,\ 3.40282347e38]$ and any double-format values belongs to $[-1.7976931348623158e308,\ 1.7976931348623158e308]$. For solving path conditions over floating-point variables, we propose here to follow the classic approach of finite domain constraint solving [20] which consists to prune the domains of their inconsistent values by using constraint propagation, labelling, and floating-point variable projections.

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

4.1 Constraint propagation and labelling

During this process, constraints are incrementally introduced into a propagation queue. A fixpoint algorithm manages each constraint one by one into this queue by filtering the domains of floating-point variables. Filtering algorithms will consider only the bounds of the domains to eliminate inconsistant values. When the domain of a variable is pruned then the algorithm reintroduces in the queue all the constraints where 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 constraint is added into a constraint–store. The constraint–store is contradictory if the domain of at least one variable becomes empty during the propagation. In this case, the selected path is shown to be infeasible. The constraint propagation process reaches a fixpoint because only a finite number of floating-point values can be removed from the domains. In our approach, this fixpoint is a conservative approximation (intervals) of the possible floating-point values for the input variables. However nothing prevents the path conditions to remain contradictory. Hence, one must resort to enumeration to get particular solutions. This is done by a labelling procedure which tries to give a floating-point value to a variable one by one and propagates throughout the constraint store. This process is repeated until all the uninstantiated variables become instantiated. If this valuation leads to a contradiction then the process backtracks to other possible values or variables. The valuation is done according to some choice-heuristics of variables and values. Note that in our symbolic execution framework only the input variables need to be instantiated as all the other internal variables are computed by means of the previous ones.

4.2 Floating-point variable projections

In our approach, each normalized symbolic expression is decomposed into floating-point variable projections. Hence, these projections play the role of constraints in constraint propagation. According to the simple language of Fig.2 and to the normalization procedure, only ternary and binary symbolic expressions have to be considered. A ternary symbolic expression $r = a \odot b$ where $\odot$ denotes any of the four arithmetical operations, is decomposed into 3 projections : the direct projection $proj(r, r = a \odot b)$, the first indirect projection $proj(a, r = a \odot b)$ and the second indirect projection $proj(b, r = a \odot b)$. Indirect means that projection is done on a right operand of an assignment. 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 indirect 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 : $proj(a, a\ rel\ b)$ is the first one and $proj(b, a\ rel\ b)$ is the second one. We switch now to the most interesting part of this work which is made of the design of filtering algorithms for pruning floating-point variables of their inconsistent values.

4.2.1 Computing direct projections for ternary symbolic expression

Let $[rl, rh], [al, ah], [bl, bh]$ be the current floating-point domains of r, a, b, then the filtering algorithm applied to a direct projection $proj(r, r = a \odot b)$ computes new bounds nrl, nrh for the domain of r as shown in Fig.3.

$$
\begin{array}{ll}
[nrl, nrh] \leftarrow [al \oplus bl, ah \oplus bh] \cap [rl, rh] & \text{when } \odot = \oplus \\
[nrl, nrh] \leftarrow [al \ominus bh, ah \ominus bl] \cap [rl, rh] & \text{when } \odot = \ominus \\
[nrl, nrh] \leftarrow [min(al \otimes bl, al \otimes bh, ah \otimes bl, ah \otimes bh), max(al \otimes bl, al \otimes bh, ah \otimes bl, ah \otimes bh)] & \\
\quad \cap [rl, rh] & \text{when } \odot = \otimes \\
[nrl, nrh] \leftarrow [min(al \oslash bh, al \oslash bl, ah \oslash bh, ah \oslash bl), max(al \oslash bh, al \oslash bl, ah \oslash bh, ah \oslash bl)] & \\
\quad \cap [rl, rh] & \text{when } \odot = \oslash \text{ and } +0.0, -0.0 \text{ don't belong to } [bl, bh]
\end{array}
$$

Figure 3: Computations of direct projections for ternary symbolic expressions

Note that these formula for direct projections are inspired of Interval Arithmetic [25, 15] but differ from it because our goal here isn't to conserve the expected result over the reals[9]. They directly come from the monotony of the near-to-even rounding direction. 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 **numeric** floating-point values is totally ordered (even for both -0.0 and $+0.0$). Fig.4 shows the application of the formula for the operator $\oplus$. The intervals of a, b, r are shown with vertical lines and each arrow represents the actual computation of the new bounds of r, before rounding. In this example note that the new inferior bound of r is rounded up although the result over the reals $al + bl$ is strictly inferior to the rounded result $al \oplus bl$. This is due to the fact that $al + bl$ is strictly greater than $mid((al \oplus bl)^-, al \oplus bl)$ in this example.

Fig.5 shows another example for the $\ominus$ operator.

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.

4.2.2 Computing indirect projections

Indirect projections are more complicated to compute. The filtering algorithm applied to a **first indirect** projection $proj(a, r = a \odot b)$ computes new bounds nal, nah for the domain of a whereas it computes new bounds nbl, nbh for b when applied to a **second indirect** projection $proj(b, r = a \odot b)$. The formula for these indirect projections are given in Fig.6. Note that first and second projections for $\oplus$ and $\otimes$ are the same, hence only the first one is given.

[9] 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 $\underline{z}$ is $x \oplus y$ rounded toward negative infinity and $\overline{z}$ is $x \oplus y$ rounded toward positive infinity [10]

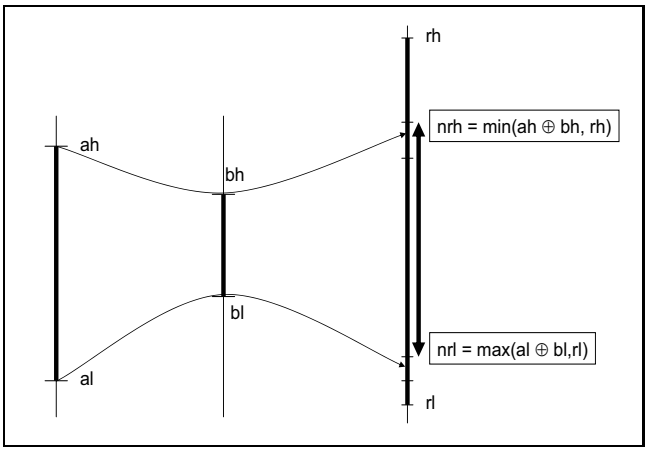

Figure 4: Computation of direct $proj(r, r = a \oplus b)$

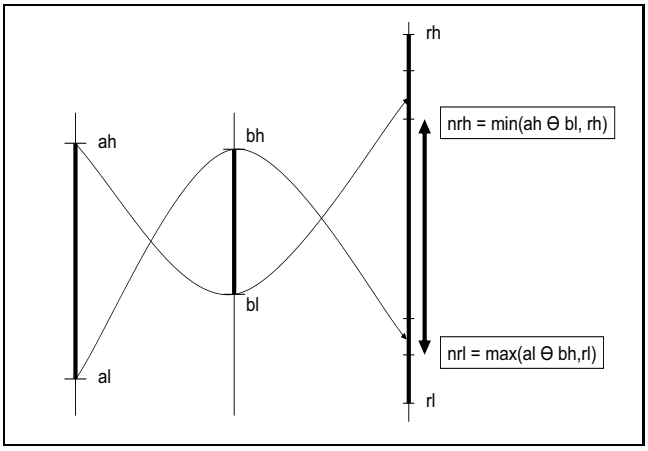

Figure 5: Computation of direct $proj(r, r = a \ominus b)$

Firstly, all indirect projections require to compute the middle of (rl, rl^-) and (rh, rh^+). In fact, any of the real values that strictly belongs to the interval $(mid(rl, rl^-), rl)$ is rounded up to rl and any reals in $(rh, mid(rh, rh^+))$ is rounded down to rh because if such a real value is the exact result of a direct computation then its nearest floating-point value will be either rl or rh. The computation of the middle of two floating-point variables of a given format can easily be done for single and double formats because there always exists a greater format in IEEE-754 : the middle of two singles can be computed by using a double

$$[nal, nah] \leftarrow [mid(rl, rl^-) \ominus bh, mid(rh, rh^+) \ominus bl] \cap [al, ah] \text{ when } \odot = \oplus$$

$$[nal, nah] \leftarrow [mid(rl, rl^-) \oplus bl, mid(rh, rh^+) \oplus bh] \cap [al, ah] \text{ when } \odot = \ominus \text{ (first indirect)}$$

$$[nbl, nbh] \leftarrow [al \ominus mid(rh, rh^+), ah \ominus mid(rl, rl^-)] \cap [bl, bh] \text{ when } \odot = \ominus \text{ (second indirect)}$$

$$[nal, nah] \leftarrow \begin{array}{l}[min(mid(rl, rl^-) \oslash bl, mid(rl, rl^-) \oslash bh, mid(rh, rh^+) \oslash bl, mid(rh, rh^+) \oslash bh), \\ max(mid(rl, rl^-) \oslash bl, mid(rl, rl^-) \oslash bh, mid(rh, rh^+) \oslash bl, mid(rh, rh^+) \oslash bh)]\end{array} \cap [al, ah]$$
$$\text{when } \odot = \otimes \text{ and } +0.0, -0.0 \text{ don't belong to } [bl, bh]$$

$$[nal, nah] \leftarrow \begin{array}{l}[min(mid(rl, rl^-) \otimes bl, mid(rl, rl^-) \otimes bh, mid(rh, rh^+) \otimes bl, mid(rh, rh^+) \otimes bh), \\ max(mid(rl, rl^-) \otimes bl, mid(rl, rl^-) \otimes bh, mid(rh, rh^+) \otimes bl, mid(rh, rh^+) \otimes bh)]\end{array} \cap [al, ah]$$
$$\text{when } \odot = \oslash \text{ (first indirect)}$$

$$[nbl, nbh] \leftarrow \begin{array}{l}[min(al \oslash mid(rl, rl^-), ah \oslash mid(rl, rl^-), al \oslash mid(rh, rh^+), ah \oslash mid(rh, rh^+)), \\ max(al \oslash mid(rl, rl^-), ah \oslash mid(rl, rl^-), al \oslash mid(rh, rh^+), ah \oslash mid(rh, rh^+))]\end{array} \cap [bl, bh]$$
$$\text{when } \odot = \oslash \text{ (second indirect) and } +0.0, -0.0 \text{ don't belong to } [bl, bh]$$

Figure 6: Computations of indirect projections $proj(a, r = a \odot b)$ and $proj(b, r = a \odot b)$

and the middle of two doubles can be computed by using an extended double. Note that the operation over the middle has also to be computed into the greatest format, such as in the indirect projection of $\oplus : mid(rl, rl^-) \ominus bh$ as shown in Fig.7. Hence, both operands of $\ominus$ are first converted into the greatest format, although this isn't explicit in the formula.

Secondly, special attention must be paid to the computation of bounds of projected variable for indirect projections. Operators $\oplus, \ominus, \otimes, \oslash$ are correctly rounded hence they can be used to compute their own inverse. Let us give the flavor of the proof for the formula for indirect projection of $\oplus$. Consider the computation of nah in Fig.7 and suppose that the least significant bit of rh be even. The indirect projection looks for the greatest single-format floating-point value x such as $x + b \leq mid(rh, rh^+)$ where the floating-point b satisfies $bl \leq b \leq bh$. From that, we get $x \leq mid(rh, rh^+) - bl$ recalling that the above inequalities hold over the reals. By computing the second operand over the doubles, we get the following : $d1 \leq mid(rh, rh^+) - bl \leq d2$ where $d1, d2$ are two consecutive doubles and either $mid(rh, rh^+) \ominus bl = d1$ or $mid(rh, rh^+) \ominus bl = d2$ because $\ominus$ is correctly rounded with the near-to-even rounding direction. Hence, taking nah as the smallest single floating-point number greater than $d2$ will conservatively solve the problem.

As a consequence, the formula given for indirect projections aren't optimal but conservatively overestimate the set of floating-point values that satisfy a given normalized symbolic expression. Considering the least significant bit of rl and rh can lead to slight more pruning [22] but we believe that this optimization can be of a high cost when implemented at any step of the projection computation because it requires to change the rounding mode several times during the computation. Note that interesting results coming from the literature may

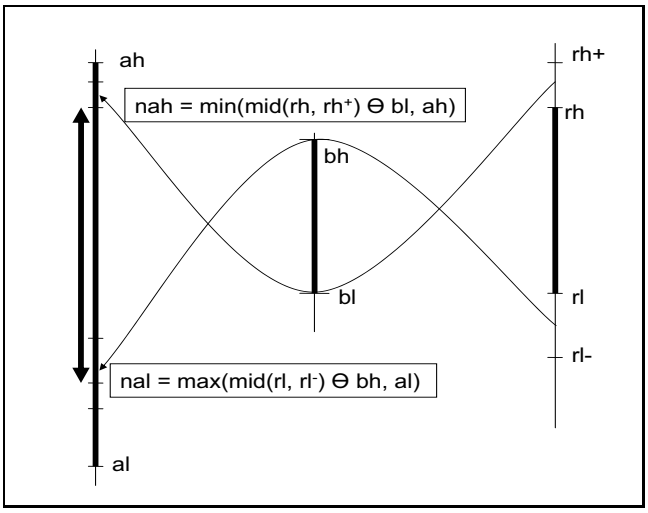

Figure 7: Computing first indirect $proj(a, r = a \oplus b)$

also be used to improve the computation of indirect projections. For example, a classical result [14] says that if $x \oplus y$ underflows to a denormalized number then $x \oplus y = x + y$ exactly, hence the computation of the middle wouldn't be required in this case.

The indirect projections of operator $\ominus$ are illustrated in Fig.8,9 to exemplify other situations.

4.3 Handling comparisons and conversions

Comparisons. Relational operators such as $==, >, \geq, \ldots$ are handled by ordered set properties because the finite set of numeric floating-point variables is totally ordered. The formula are mainly inspired from Interval Arithmetic [25]. They are similar for the first and second projections, hence only the first ones are given in Fig.10. The current floating-point domain of A (resp. B) is $[al, ah]$ (resp. $[bl, bh]$) and the resulting domain of A is $[nal, nah]$. Provided that $-0.0 == +0.0$ evaluates to true in IEEE-754, the computation of the bounds when a zero is involved in a first indirect projection requires that $max(-0.0, +0.0) = max(+0.0, -0.0)$ results to -0.0 and $min(-0.0, +0.0) = min(+0.0, -0.0)$ results to $+0.0$.

Conversions. The simple language of Fig.2 allows only two conversions $r = (float)a$ where a is a double and $r = (double)a$ where a is a single. Formula to compute the bounds of projected variables in direct and indirect projections are given in Fig.11. It is trivial to see that any single-format value can be exactly converted into a double-format value, hence some conversions in the formula don't require any computation and remain implicit.

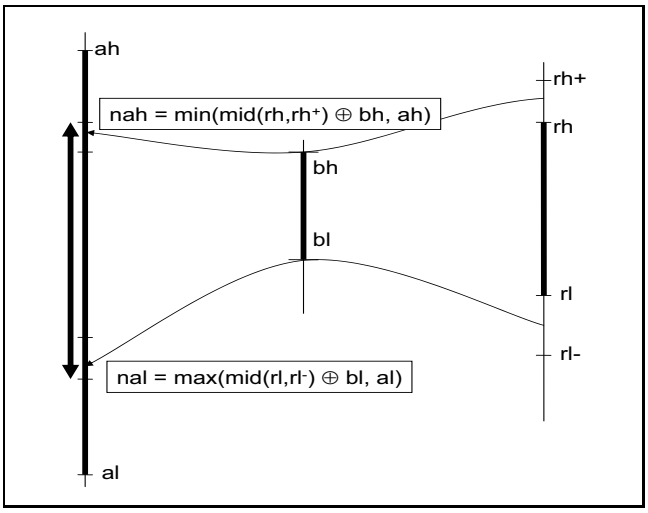

Figure 8: Computing first indirect $proj(a, r = a \ominus b)$

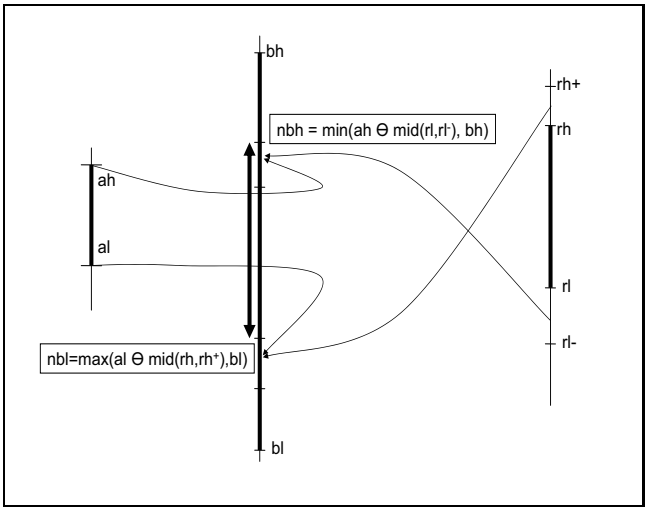

Figure 9: Computing second indirect $proj(b, r = a \ominus b)$

5 Handling symbolic values

IEEE-754 distinguishes 2 kinds of symbolic values : infinities and NaNs. There are two infinities ($-INF$,$+INF$) and we treat all NaNs as a single symbolic value (NaN) though

$[nal, nah] \leftarrow [max(al, bl), min(ah, bh)]$ when $a == b$
$[nal, nah] \leftarrow [max(al, bl), ah]$ when $a \geq b$
$[nal, nah] \leftarrow [max(al, bl)^+, ah]$ when $a > b$
$[nal, nah] \leftarrow [al, min(ah, bh)]$ when $a \leq b$
$[nal, nah] \leftarrow [al, min(ah, bh)^-]$ when $a < b$
$[nal, nah] \leftarrow [\text{if } (al = bl = bh) \text{ then } al^+ \text{ else } al, \text{ if } (ah = bl = bh) \text{ then } ah^- \text{ else } ah]$ when $a!{=}b$

Figure 10: First projection for comparison operators

$[nrl, nrh] \leftarrow [max_f((float)al, rl), min_f((float)ah, rh)]$ for direct $proj(r_f, r_f = (float)a_d)$
$[nal, nah] \leftarrow [max_d(al, mid(rl, rl^-)), min_d(ah, mid(rh, rh^+)]$ for indirect $proj(a_d, r_f = (float)a_d)$
$[nrl, nrh] \leftarrow [max_d(al, rl), min_d(ah, rh)]$ for direct $proj(r_d, r_d = (double)a_f)$
$[nal, nah] \leftarrow [max_f(al, rl), min_f(ah, rh)]$ for indirect $proj(a_f, r_d = (double)a_f)$
where r_f, a_f are singles, r_d, a_d are doubles,
max_f, min_f operate over the singles and max_d, min_d operate over the doubles

Figure 11: Direct and indirect projections of conversion operators

it can result of distinct bits strings. The cases where infinities and NaN can be produced as the result of a computation are well known [10]. However, implementing projection functions over symbolic values requires to further analyze how to combine NaNs, infinities, numerical values and signed zeros and how to deal with exceptions [14].

In our approach, we extend the numerical domain with the two infinities. The resulting set of floating point numbers remains totally ordered and its properties are extended with an arithmetic over infinities. However, this arithmetic requires to use a supplementary variable to represent the NaN. The state of this variable is three-valued : true,false and unknown. If v is a variable then v_N denotes its NaN state. At the beginning of the solving process, all the NaN states of variables are unknown.

The general idea for computing projections consists in isolating the symbolic values of their domains and merging the results of both the symbolic and the numeric cases. In the presence of signed zeros or symbolic values, the computation of the domain of the projected variable is based on specific tables.

5.1 Direct projections

When one computes the direct projection of $\oplus$, the extended arithmetic given in Tab.1 is required. This table and the other ones for direct projections are inspired of [15] but follow the formal specification of [1]. Only slight differences exist in the treatment of zeros as we restricted our approach to the near-to-even rounding direction. For the sake of clarity, when there is a single possible value n, it confounds with its singleton $\{n\}$. Nv stands for any non-zero numeric value whereas $\pm INF$ denotes any of the two infinities. Note that the sign of the entries when both arguments are zeros is uniquely determined when the rounding mode is known. When the sum or the difference of two opposite operands is exactly zero, the sign

Table 1: Value of r in direct $proj(r, r = a \oplus b)$

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

of that sum or difference is $+0.0$ in the near-to-even mode. The cases where an infinity is produced as the result of an operation over 2 numeric values ($Nv \oplus Nv$ for example) usually correspond to an overflow.

Let's explain how to extend the computation of direct $proj(r, r = a \oplus b)$ with symbolic values. Firstly, the NaN state is evaluated by the following tests. If either $a_N = t$ or $b_N = t$ then $r_N \leftarrow t$ because the symbolic NaN is propagated through an operation. When $al = ah = +INF$ and $bl = bh = -INF$ or $al = ah = -INF$ and $bl = bh = +INF$ then we also get $r_N \leftarrow t$, as shown in Tab.1. When r_N evaluates to true, the computation of the projection ends because the result of the operation is NaN. When both a_N and b_N are false and opposite infinities don't belong to the domains of a and b then $r_N \leftarrow f$. But most of the time r_N remains unknown until the labelling process where the possible assignment $a_N = t$ or $b_N = t$ will be considered if a_N, b_N are input variables.

Secondly, the new bounds of r are computed by using the formula of the numeric case ($[nrl, nrh] \leftarrow [al \oplus bl, ah \oplus bh] \cap [rl, rh]$). 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 numeric case ($Nv \oplus Nv$) of the table is applied. If an overflow occurs then the bounds are updated with the corresponding infinities.

The same procedure can be used for the computation of direct projections for $\ominus, \otimes, \oslash$ by using the following tables.

Table 2: Value of r in direct $proj(r, r = a \ominus b)$

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

Note that we didn't distinguish the negative and positive numeric cases in these tables. Although this is useful to implement better pruning of domains, these cases aren't difficult to determine as simple sign rules remain valid in the context of non-zeros numeric 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 $\oslash$.

5.2 Indirect projections

Similar procedures can be used for indirect projections although NaN states cannot be computed as easy as we did for direct projections. Instead, cases where NaNs can be produced have to be given, so NaN appears in the first row of Tab.2 which stands for the first indirect projection of $\oplus$. Other tables for indirect projections are given at the end of the paper.

When considering indirect projections, some combinations of symbolic values are impossible. For example, consider the first indirect projection $proj(a, r = a \oplus b)$ where $r = +0.0$ and $b = +INF$ are known. This indirect projection prunes the domain of a of all its values as there doesn't exist any floating-point value of a that satisfies the constraint $+0.0 = a \oplus +INF$. The symbol $\perp$ denotes the case where the combination of symbolic values is impossible. When the operands of an indirect projection are known and $\perp$ is encountered into the tables then the projection is refuted and the constraint store is shown to be contradictory. More frequently, operands are just known by their interval of possible values. Hence, when a combination of bounds is $\perp$, such as for example in $proj(a, r = a \oplus b)$ where $r \in [-INF, +INF]$ and $b \in [-INF, +INF]$, $\perp$ is just ignored and the interval of a is leaved unchanged.

To summarize our approach, let's give the following guidelines : first, symbolic values are isolated from the intervals ; second, they are combined together by using the tables for indirect projections ; third, the formula for numerical indirect projections are used to compute intermediate results ; finally, the results are unified to set the NaN state and the new bounds for the projected variable.

Table 3: Value of r in direct $proj(r, r = a \otimes b)$

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

Table 4: Value of r in direct $proj(r, r = a \oslash b)$

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

Table 5: Value of a in first indirect $proj(a, r = a \oplus b)$

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

6 The current implementation

InKa and FPSE. We implemented a solver of symbolic expressions over C floating-point computations as part as the automatic test data generator InKa [11, 12]. This tool computes

Table 6: Value of a in first indirect $proj(a, r = a \ominus b)$

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

Table 7: Value of b in second indirect $proj(b, r = a \ominus b)$

$a \backslash r$	$-INF$	-0.0	$+0.0$	Nv	$+INF$	NaN
$-INF$	$Nv \cup \{+INF, \pm 0.0\}$	$\bot$	$\bot$	$\bot$	$\bot$	$\{-INF, NaN\}$
-0.0	$+INF$	$+0.0$	-0.0	Nv	$-INF$	NaN
$+0.0$	$+INF$	$\bot$	$\{\pm 0.0\}$	Nv	$-INF$	NaN
Nv	$Nv \cup \{+INF\}$	$\bot$	Nv	$Nv \cup \{\pm 0.0\}$	$Nv \cup \{-INF\}$	NaN
$+INF$	$\bot$	$\bot$	$\bot$	$\bot$	$Nv \cup \{-INF, \pm 0.0\}$	$\{+INF, NaN\}$

Table 8: Value of a in first indirect $proj(a, r = a \otimes b)$

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

Table 9: Value of a in first indirect $proj(a, r = a \oslash b)$

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

test data insuring a high-level coverage of some structural criteria. When the selected criterion is *all_paths*[10], the tool extracts normalized symbolic expressions by a forward analysis. Symbolic expressions are then solved by using FPSE (Floating Point Symbolic Execution), a current implementation of the projection functions described above.

FPSE handles C floating-point computations that conform to IEEE-754 and run on SPARC and INTEL architectures although their internal representation of floating-point numbers are distinct. When launching the compiler, it is required to set up the compiler options that forbid the storage of floating-point values into registers and extended formats during the computations[11], and options that desactivate code restructuration for optimizations. FPSE itself is intended to run on the same machine as the tested programs to conform the same rounding algorithms. The constraint propagation engine of FPSE is written in Prolog whereas the projection functions are written in C. Note that FPSE combines floating-point and integer constraints into the same propagation engine but this is outside the scope of the paper.

For the moment, only a trivial labelling heuristic has been implemented. It consists in selecting the smallest value of the interval of the first found uninstantiated floating-point variable. Of course, more sophisticated heuristics will be considered in the future. The labelling process is time-consuming and cannot be driven until the end in all the cases. However, 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 isn't more time-consuming than the one used in constraint-based environments over integers [7, 11, 21].

Features. In FPSE, single and double formats normalized and denormalized numeric floating-point numbers are handled. Signed zeros and infinities are also handled but NaNs yet aren't. They are even avoided meaning that our current implementation doesn't produce a test datum on which the computation yields to a NaN. Further, specific conditions such as `if(isNaN(x))` or `if(x!=x)`[12] cannot be satisfied.

In practice, it is very difficult to guarantee that the symbolic execution will strictly conform the actual execution because of several reasons : the lack of documentation of the compiler options and design, the existence of hardware optimizations such as the fused multiply-add a+b*c which is a newly hardware-supported operation, the possible manipulations of the user to change the rounding mode and to control how exceptions are produced,

[10] with a k-limiting scheme to handle loops

[11] option `-ffloat_store` in gcc, option `/Op` in Visual

[12] According to [10] NaNs should satisfy the constraint `x!=x`

Table 10: Value of b in second indirect $proj(b, r = a \oslash b)$

$a \backslash r$	$-INF$	-0.0	$+0.0$	Nv	$+INF$	NaN
$-INF$	$Nv \cup \{+0.0\}$	$\bot$	$\bot$	$\bot$	$Nv \cup \{-0.0\}$	$\{\pm INF, NaN\}$
-0.0	$\bot$	$Nv \cup \{+INF\}$	$Nv \cup \{-INF\}$	$\bot$	$\bot$	$\{\pm 0.0, NaN\}$
$+0.0$	$\bot$	$Nv \cup \{-INF\}$	$Nv \cup \{+INF\}$	$\bot$	$\bot$	$\{\pm 0.0, NaN\}$
$+INF$	$Nv \cup \{-0.0\}$	$\bot$	$\bot$	$\bot$	$Nv \cup \{+0.0\}$	$\{\pm INF, NaN\}$
Nv	$\{\pm 0.0\}$	$\{\pm INF\}$	$\{\pm INF\}$	Nv	$\{\pm 0.0\}$	NaN

Table 11: First experimental results

Pgm	Path conditions	in	*clpr*	*clpq*	FPSE
g1	$T2 = 2.0e-30 \oplus 1.0e30, T1 = T2 \ominus 1.0e30, T3 = T1 \ominus 1.0e-30, X == T3$	X	0.0	1/9999999999-999998791471-36483328	-1.0000000031710769e-30
g2_d	$T1 = B \otimes B, T2 = A \otimes C, T3 = 4.0 \otimes T2, \Delta = T1 \ominus T3, A = 1.22, B = 3.34, C = 2.28$	Δ	0.0292000000-00001225	73/2500 =0.0292	0.029199600219726562
g2_i	$T1 = B \otimes B, T2 = A \otimes C, T3 = 4.0 \otimes T2, \Delta = T1 \ominus T3, A = 1.22, B = 3.34, \Delta = 0$	C	2.28598360-6557377	27889/12200	[2.285982608795166, 2.2859842777252197] and after labeling 2.2859835624694824
foo	$X < 10000.0, T1 = X \oplus 1.0e12, T1 > 1.0e12$	X	$(-0.0,\ 10000.0)$	$(0,\ 10000)$	infeasible
foo_m	$X > 0.0, T1 = X \oplus 1.e12, T1 == 1.0e12$	X	infeasible	infeasible	[1.401298464324817E-45, 32768.0]
h	$T1 = A \otimes B, X1 = T1 \oplus 2.0, X1 > 100.0, X2 = 100.0 \ominus X1, X3 = X2 \ominus 50.0, X3 > 50$	A, B	infeasible	infeasible	infeasible

etc. These limitations have to be taken into account when interpreting the results of FPSE. When a path is labeled as infeasible, the actual meaning is that it's infeasible when several coding and compiling rules are enforced when doing the actual execution. As a consequence, when a test datum is proposed as a solution of path conditions, we check the path followed by the proposed result by executing the program as a final step of the symbolic execution process.

6.1 Experimental results

We only report a selection of the first experimental results we got on several expressions extracted from the literature. Our goal was to compare the results provided by FPSE with results given by a solver over the reals and the rationals : the simplex-based solvers of the SICStus Prolog environment. Both solvers (clpr for the reals and clpq for the rationals) use the simplex algorithm with several isolation axioms to take into account restricted shapes of non-linear constraints.

We selected here five expressions that correspond to symbolic expressions and path conditions that well exemplify the floating-point related-problems into a symbolic execution framework. Our first examples g1,g2_d and g2_i come from [10]. g1 is a normalized version of the symbolic expression $X == ((2.0e-30 + 1.0e30) - 1.0e30) - 1.0e-30$. g2_i corresponds to a direct evaluation of the symbolic expression $B^2 - 4AC$ whereas g2_i corresponds to an indirect evaluation of the same expression. The path conditions of path 1-2-3-4 in the program foo of Fig.1 give the fourth example. foo_m corresponds to the modified version of this program proposed in the introduction of the paper. Finally, the last example is extracted from [17].

The experimental results given in Tab.3 were performed over the singles on a Pentium 4 personal computer for expressions intended to be compiled with VisualC++[13]. The cpu time required to get the results is less than 1sec for the three solvers. The first expression illustrates that the three evaluators can produce distinct results. The results computed by both clpr and clpq are wrong not only w.r.t. the expected result over the reals $+1.0e-30$ but also over the floats. As expected, FPSE provides the result that strictly conforms the evaluation over the singles. The two following expressions exemplify the fact that even when evaluating a symbolic expression over the floats (g2_d) or computing trivial path conditions (g2_i), the results given by the solvers may be inconsistent with the program. In g2_d, we got three distinct results and only FPSE delivers the expected result over the singles. In g2_i, FPSE returns an interval of 8 singles before labelling. This interval is a conservative overestimation of the set of singles that solve the path conditions. After labelling, FPSE finds that only one single conforms the actual execution. The results for foo and foo_m have already been discussed. They illustrate how dangerous incorrect deductions can be done when solving path conditions with clpr or clpq. Finally, the last example shows that there are cases where the three solvers conform the expected infeasibility result but only the one provided by FPSE is trustworthy over the floats.

7 Further work

In this paper, we have introduced a symbolic execution framework able to deal correctly with restricted form of IEEE-754 compliant floating-point computations. We have provided the definitions of projection functions for solving normalized symbolic expressions. Our approach handles operators on numeric values as well as symbolic values and our current implementation has been evaluated on a small set of symbolic execution problems. Of course, more developments and experiments have to be performed to fully validate the proposed approach. Several extensions can be foreseen. Handling other rounding modes than the near-to-even one appears as being a tedious but not too difficult extension of our framework. In the same spirit, handling the square root function is interesting because it allows to compute roots of some polynomials equations. Furthermore, it is included in the IEEE-754 standard as a required correctly rounded function. Dealing with extended formats appear to be interesting for the future but probably requires to use multiple-precision floating-point numbers. More complex is the extension to transcendental functions such as exponential or trigonometric as nothing guarantees the computation of being correctly rounded in these cases. This is likely to be the more prospective part of our future work.

Acknowledgements

Many thanks to Michel Rueher for numerous and fruitful discussions on this work.

[13] In VisualC++ 6.0, types double and long double are equals, hence FPSE cannot easily compute the middle of two doubles with this compiler

References

[1] G. Barrett. Formal method applied to a floating-point number system. *IEEE Trans. on Software Engineering*, 15(5):611–621, May. 1989.

[2] J. Bicevskis, J. Borzovs, U. Straujums, A. Zarins, and E. Miller. SMOTL - A System to Construct Samples for Data Processing Program Debugging. *IEEE Trans. on Software Engineering*, 5(1):60–66, Jan. 1979.

[3] R. Boyer, B. Elspas, and K. Levitt. SELECT - A formal system for testing and debugging programs by symbolic execution. *SIGPLAN Notices*, 10(6):234–245, Jun. 1975.

[4] T. Chen, T. Tse, and Z. Zhou. Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing. In *Proc. of ISSTA'02)*, pages 191–195, Roma, Italy, Jul. 2002.

[5] L. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Trans. on Software Engineering*, 2(3):215–222, Sep. 1976.

[6] A. Coen-Porisini, F. d. Paoli, C. Ghezzi, and D. Mandrioli. Software Specialization Via Symbolic Execution. *IEEE Trans. on Software Engineering*, 17(9):884–899, Sep. 1991.

[7] R. DeMillo and A. Offut. Constraint-Based Automatic Test Data Generation. *IEEE Trans. on Software Engineering*, 17(9):900–910, Sep. 1991.

[8] R. DeMillo and A. Offut. Experimental Results from an Automatic Test Case Generator. *ACM Trans. on Software Engineering Methodology*, 2(2):109–175, 1993.

[9] A. Goldberg, T. Wang, and D. Zimmerman. Applications of Feasible Path Analysis to Program Testing. In *Proc. of ISSTA'94*, pages 80–92, Seattle, WN, Aug. 1994. ACM.

[10] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[11] A. Gotlieb, B. Botella, and M. Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. In *Proc. of ISSTA'98*, pages 53–62, Clearwater Beach, FL, Mar. 1998.

[12] A. Gotlieb, B. Botella, and M. Rueher. A clp framework for computing structural test data. In *Proc. of Computational Logic (CL), in LNAI 1891*, pages 399–413, London, UK, Jul. 2000.

[13] E. Goubault. Static analyses of the precision of floating-point operations. In *Static Analysis Symposium (SAS'01) and also in LNCS 2126*, pages 234–245, Paris, Jul. 2001.

[14] J. Hauser. Handling floating-point exceptions in numeric programs. *ACM Trans. on Programming Languages and Systems*, 18(2):139–174, Mar. 1996.

[15] T. J. Hickey, Q. Ju, and M. H. van Emden. Interval arithmetic: From principles to implementation. *Journal of the ACM*, 48(5):1038–1068, 2001.

[16] W. Howden. Reliability of the Path Analysis Testing Strategy. *IEEE Trans. on Software Engineering*, 2(3):208–214, Sep. 1976.

[17] W. Howden. Validation of scientific programs. *ACM Computing Surveys*, 14(2):193–227, Jun. 1982.

[18] IEEE-754. Standard for binary floating-point arithmetic. *SIGPLAN Notices*, 22(2):9–25, Feb. 1985.

[19] J. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, Jul. 1976.

[20] K. Marriott and P. Stuckey. *Programming with Constraints : An Introduction.* The MIT Press, 1998.

[21] C. Meudec. ATGen: automatic test data generation using constraint logic programming and symbolic execution. *Software Testing, Verification and Reliability*, 11:81–96, Nov. 2001.

[22] C. Michel. Exact projection functions for floating point number constraints. In *Proc. of 7th AIMA Symposium*, Fort Lauderdale (US), 2002.

[23] C. Michel, M. Rueher, and Y. Lebbah. Solving constraints over floating-point numbers. In *Constraint Prog. (CP'01)*, pages 524–538, LNCS 2239, Nov 2001.

[24] W. Miller and D. Spooner. Automatic Generation of Floating-Point Test Data. *IEEE Trans. on Software Engineering*, 2(3):223–226, Sep. 1976.

[25] R. A. Moore. *Interval Analysis.* Prentice Hall, Englewood Cliffs, NJ, 1966.

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

[27] A. Offut, Z. Jin, and J. Pan. The Dynamic Domain Reduction Procedure for Test Data Generation. *Software–Practice and Experience*, 29(2):167–193, 1999.

[28] C. Ramamoorthy, S. Ho, and W. Chen. On the automated generation of program test data. *IEEE Trans. on Software Engineering*, 2(4):293–300, Dec. 1976.

[29] E. Weyuker. Translatability and decidability questions for restricted classes of program schemas. *SIAM Journal of Computing*, 8(4):587–598, Nov. 1979.

Contents

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399