Solving constraints over floating-point numbers *

Claude MICHEL[1], Michel RUEHER[1], and Yahia LEBBAH[2]

[1] {cpjm, rueher}@unice.fr
Université de Nice–Sophia Antipolis, I3S–CNRS, 930 route des Colles, B.P. 145, 06903 Sophia Antipolis Cedex, France
[2] ylebbah@algeriecom.com
Université d'Oran Es-Senia, Faculté des Sciences, Département d'Informatique, B.P. 1524 El-M'Naouar, Oran, Algeria

Abstract. This paper introduces a new framework for tackling constraints over the floating-point numbers. An important application area where such solvers are required is program analysis (e.g., structural test case generation, correctness proof of numeric operations). Albeit the floating-point numbers are a finite subset of the real numbers, classical CSP techniques are ineffective due to the huge size of the domains. Relations that hold over the real numbers may not hold over the floating-point numbers. Moreover, constraints that have no solutions over the reals may hold over the floats. Thus, interval-narrowing techniques, which are used in numeric CSP, cannot safely solve constraints systems over the floats. We analyse here the specific properties of the relations over the floats. A CSP over the floats is formally defined. We show how local-consistency filtering algorithms used in interval solvers can be adapted to achieve a safe pruning of such CSP. Finally, we illustrate the capabilities of a CSP over the floats for the generation of test data.

1 Introduction

This paper introduces a new framework for tackling constraints over the floating-point numbers. Due to the specific properties of the floating-point numbers, neither classical CSP techniques nor interval-narrowing techniques are effective to handle them. The tricky point is that constraints that have no solutions over the reals may hold over the floats. Moreover, relations that hold over the real numbers may not hold over the floating-point numbers. For instance, Equation $16.0 + x = 16.0$ with $x > 0$ has solutions over the floats with a rounding mode set to *near* whereas there is no solution over $\mathbb{R}$. Equation $x^2 = 2$ has no solution over the floats with the usual rounding mode (round to nearest) whereas the solution over $\mathbb{R}$ is $\sqrt{2}$.

An important application area where such solvers are required is program analysis (e.g., structural test case generation, correctness proof of numeric operations). For instance, structural test techniques are widely used in the unit

* This work was partially supported by the RNTL project INKA

testing process of software. A major challenge of this process consists in generating test data automatically, i.e., in finding input values for which a selected point in a procedure is executed. We have shown in [Got00,GBK98] that the later problem can be handled efficiently by translating a non-trivial imperative program into a CSP over finite domains. However, when these programs contain arithmetic operations that involve floating-point numbers, the challenge is to compute test data that are valid even when the arithmetic operations performed by the program to be tested are unsafe. In other word, the constraint solver should not compute the smallest interval that contains the solution in $\mathbb{R}$, but a solution over the floats that complies with the arithmetic evaluation process in imperative language like C or C++.

Thus, when generating test data for programs with numeric operations over the floats, the critical issue is to guaranty that the input data derived from the constraint system are safe, i.e., that the selected instruction will actually be executed when the program to be tested is called with that data (see Section 5). So, what we need is a safe solver over the floats.

In the remainder of this section we first detail our motivations before providing a brief summary of our framework.

1.1 Motivations

Like any constraint over finite domains, a constraint over the floats is a subset of the Cartesian product of the domains, which specifies the allowed combinations of values for the variables. However constraints systems over the floats have some very specific properties:

- The size of the domains is very large: they are more than 10^{18} floating-point numbers in the interval $[-1, 1]$. So, classical CSP techniques are ineffective to handle these constraint systems.
- The evaluation of a constraint is not a trivial task: the result of the evaluation of constraint $c(x_1, \ldots, x_n)$ for an n-uplet of floating-point numbers depends on various parameters (e.g., rounding mode, mathematical library, floating-point unit processor, sequence of evaluation).

Since constraints over the floats are defined by arithmetic expressions, a question which naturally arises is that of the ability of interval solvers –like `PROLOG IV`[Col94], `Numerica`[VHMD97] or `DeClic` [Gua00]– to handle them. Using interval solvers to tackle constraints over the floats yields two major problems:

- *Interval solvers are not conservative over the floats*, i.e., they may remove solutions over the floats, which are not solutions over the reals. For instance, any floating-point value in $[-1.77635683940025046e-15, 1.77635683940025046e-15]$ is a solution of the relation $16.1 = 16.1 + x$ over the floats whereas `PROLOG IV` reduces the domain of x to 0 (with a rounding mode set to near). Likewise, the relation $cos(x) = 1$ holds for any floating-point value in the interval $[-1.05367121277235798e-08, 1.05367121277235798e-08]$[1] while

[1] With $x \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ on a SPARC processor (with the *libm* library and a rounding mode set to near).

`PROLOG IV` and `DeClic` respectively reduce the domain of x to 0 and to $[-4.9406564584124655e-324, +4.9406564584124655e-324]$.
Of course, these problems are amplified by the symbolic transformations that some solvers perform to prune the intervals better.
- Solutions provided by interval solvers are tight intervals that may contain a solution over the reals whereas the solutions we are looking for are n-uplet of floating-point values.

That is why we introduce here a new solver based on a conservative filtering algorithm, i.e., an algorithm that does not remove any solutions over the floats. Roughly speaking, this algorithm takes advantages of the local consistency algorithms used in the interval solvers to identify a subpart of the search space which may not contain any solution. An evaluation process that complies with the arithmetic computations performed in imperative language like C or C++, is used to verify that these subparts do not contain any solution.

1.2 Outline of the paper

The next section introduces the notations, recalls some basic definitions, and states the working hypothesis (e.g., compliance with IEEE 754 norm [ANS85]). A CSP over the floats is formally defined in Section 3. Filtering algorithms for CSP over the floats are detailed in Section 4. Section 5 illustrates the capabilities of a CSP over the floats for test data generation. The last section discusses some extensions of our framework.

2 Notations and basic definitions

This section introduces the notations and recalls some basic definitions that are required in the rest of the paper. Note that the definition of the intervals differs from the classical definition of intervals over the reals.

2.1 Notations

We mainly use the notations suggested by Kearfott [Kea96]. Thus, throughout, boldface will denote intervals, lower case will denote scalar quantities, and upper case will denote vectors and sets. Brackets "[.]" will delimit intervals while parentheses "(.)" will delimit vectors. Underscores will denote lower bounds of intervals and overscores will denote upper bounds of intervals.

We will also use the following notations, which are slightly non-standard :

- $\mathbb{R} = \mathcal{R} \cup \{-\infty, +\infty\}$ denotes the set of real numbers augmented with the two infinity symbols. $\mathbb{F}$ denotes a finite subset of $\mathbb{R}$ containing $\{-\infty, +\infty\}$. Practically speaking, $\mathbb{F}$ corresponds to the set of floating-point numbers;
- v stands for a constant in $\mathbb{F}$, v^+ (resp. v^-) corresponds to the smallest (resp. largest) number of $\mathbb{F}$ strictly greater (resp. lower) than v;
- f, g denote functions over the floats; $c\ :\ \mathbb{F}^n \rightarrow \mathcal{B}ool$ denotes a constraint over the floats; $X(c)$ denotes the variables occurring in constraint c.

2.2 Intervals

Definition 1 (Intervals).
An interval $\mathbf{x} = [\underline{x}, \overline{x}]$*, with* $\underline{x}$ *and* $\overline{x} \in I\!F$*, is the set of floating-point values* $\{v \in I\!F \mid \underline{x} \leq v \leq \overline{x}\}$*.* $\mathcal{I}$ *denotes the set of intervals and is ordered by set inclusion.*

We draw the attention of the reader to the fact that an interval represents here a finite subset of the reals.

2.3 Representation of floating-point numbers

Floating-point numbers provide a discrete representation of the real numbers. This discretisation is needed due to the limited memory resources of the computers. It results in approximations and tricky properties.

The IEEE 754 standard [ANS85] for binary floating-point arithmetic[2] is now widely accepted and most of the available floating-point units comply with it. This section recalls the main features of the IEEE 754 standard that are required to understand the rest of this paper.

IEEE 754 defines two primary representations of floating-point numbers, *simple* and *double*, and offers the possibility to handle two others representations *simple extended* and *double extended.* If the two first representations are well defined and always available within the IEEE 754 compliant floating-point units, the latter may vary among the different implementations[3].

Differences between representations could be captured by two parameters: the size t (in bits) of the exponent e and the size p (in bits) of the significant m. Thus a set of floating-point numbers is well defined by $I\!F_{(t,p)}$.

Each floating-point number is fully defined by a 3-uples $\langle s, m, e\rangle$ where s is a bit which denotes the sign, m is the represented part of the significant and e is the biased exponent[4].

The standard distinguishes different classes of floating-point numbers. Assuming e_{max} is the maximal value an exponent could take in $I\!F_{(t,p)}$ (i.e., with all its t-bits set to 1), the standard defines the following classes of numbers:

- *normalized* numbers defined by $0 < e < e_{max}$. This class of numbers represent the following real number $(-1)^s \times 1.m \times 2^{(e-bias)}$.
- *denormalized* numbers defined by $m \neq 0$ and $e = 0$. Denormalized numbers are used to fill regularly the gaps between zero and the first normalized numbers. Their value is $(-1)^s \times 0.m \times 2^{(-bias+1)}$.

[2] IEEE 854 extends floating-point number representation by allowing the use of decimal instead of binary numbers. In this paper we restrict ourselves to the binary format.

[3] The standard just fixes some minimal requirement over the precision offered by the representation

[4] Exponents in IEEE 754 follow a quite unusual convention to represent negative values: a *bias* is subtracted from the stored value in order to get its real value.

- *infinites* defined by $m = 0$ and $e = e_{max}$ and represented by the two symbols $+\infty$ and $-\infty$.
- *signed zero* defined by $m = 0$ and $e = 0$. The IEEE 754 standard has chosen to sign the zero to handle cases where a signed zero is required to get a correct result.
- *Not-a-Number (NaN's)* defined by $m \neq 0, e = e_{max}$. The `NaN's` allows to handle exceptional cases —like a division by zero— without stopping the computation.

2.4 Floating-point arithmetic: rounding modes and exceptions

Rounding is necessary to close the operations over $\mathbb{F}$. The usual result of the evaluation of an expression over floating-point numbers is not a floating-point number. The rounding function maps the result of evaluations to available floating-point numbers. Four rounding modes are available:

- to $+\infty$ which maps x to the least floating-point number x_k such that $x \leq x_k$.
- to $-\infty$ which maps x to the greatest floating-point number x_k such that $x \geq x_k$.
- to *0* which is equivalent to rounding to $-\infty$ if $x \geq 0$ and to rounding to $+\infty$ if $x < 0$. This rounding mode has to be compared with truncation.
- to *the nearest even* which maps x to the nearest floating point number. When x is equidistant from two floating-point numbers, then x is mapped to the one that has a 0 as its least significant bit in its mantissa.

To provide a better accuracy, the standard requires exact rounding of the basic operations. Exactly rounded means that the computation of the result must be done exactly before rounding. More formally, let $\odot \in \{\oplus, \ominus, \otimes, \oslash\}$ be a binary operator, $x \in \mathbb{F}$, $y \in \mathbb{F}$, two floating-point numbers, and *Round* a rounding function, then, if $\odot$ is exactly rounded: $x \odot y =_{def} Round(x \ . \ y)$. The square root also belongs to exactly rounded functions. Functions which do not work with the exactly rounded mode may yield significant round off error (e.g., Intel 387 provides transcendental functions with up to 4.5 *ulps*[5] error).

IEEE 754 also defines *exceptions* and exceptions flags. They denote events which might occur during the computation. Such events are *underflow*, *overflow*, *inexact result*, etc. The handling of these exceptions is out of the range of this paper.

Almost none of the nice algebraic properties of the reals is preserved by floating-point arithmetic. For example, the basic operations do not have an inverse operation.

[5] Roughly speaking, an *ulps* corresponds to the size of the gap between two consecutive floating-point numbers.

2.5 Working hypothesis

In the rest of this paper we assume that all computations are done with the same rounding mode and comply with the IEEE 754 recommendations.
Floating-point numbers will be understood as *double* (i.e. $\mathbb{F}_{(11,52)}$), and neither NaN's nor exceptions flags will be handled. That's to say, the set $\mathbb{F}$ is defined by the union of the set *normalized* numbers, the set of the *denormalized* numbers, the *infinites* and the *signed zero*.
The level of reasoning is that of the FPU (Floating Point Unit). So, when source code is considered, we assume that the compiler complies both with the ANSI C and IEEE754 standard, and that the compiler does not perform any optimization.

3 Constraint systems over floating-point numbers

In this section we formally define constraint systems over floating-point numbers (named `FCSP` in the rest of the paper). We investigate the capabilities of local consistencies —like $2B$–consistency and Box–consistency— for pruning the search space. We show that the filtering algorithms that achieves these consistencies require a relaxation of the projection functions that prevents them to ensure that all solutions of the `FCSP` are preserved.

3.1 Floating-point number CSPs

A **FCSP** (floating-point constraint system) $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is defined by:

- a set of *variables* $\mathcal{X} = \{x_1, ..., x_n\}$;
- a set $\mathcal{D} = \{D_1, ..., D_n\}$ of current *domains* where D_i is a finite set of possible floating-point values for variable x_i;
- a set $\mathcal{C}$ of constraints between the variables.

$|\mathcal{C}|$ denotes the number of constraints while $|\mathcal{X}|$ denotes the number of variables. A **constraint** c on the ordered set of variables $X(c) = (x_1, ..., x_r)$ is a subset $T(c)$ of the Cartesian product $(D_1 \times ... \times D_r)$ that specifies the *allowed* combinations of values for variables $(x_1, ..., x_r)$.

The syntactical expression of a constraint $c_j : \mathbb{F}^k \rightarrow Bool$ is denoted by $f_j(x_1, ..., x_n) \quad \diamond \quad 0$ where $\diamond \in \{=, \ \leq, \ \geq\}$ and $f_j : \mathbb{F}^k \rightarrow \mathbb{F}$. Note that any expression of the form $f_j(x_1, ..., x_n) \quad \diamond \quad g_j(x_1, ..., x_m)$ can be rewritten in $f_j(x_1, ..., x_n) - g_j(x_1, ..., x_m) \ \diamond \ 0$ since the following property : $x = y \leftrightarrow x - y = 0$ over the set of considered floating point numbers [Gol91].

Let $eval(f(v_1, \ldots, v_n), r)$ be the arithmetic evaluation of expression f over the n-uplet of floating-point numbers $< v_1, \ldots, v_n >$ with a rounding mode $r \in \{+\infty, -\infty, 0, near\}$.

A constraint c holds for a n-uplet $< v_1, \ldots, v_n >$ if $eval(f(v_1, ..., v_n), r) \ \diamond \ 0$ is true. A solution of a $FCSP$ defined by the 3-uplet $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ is a n-uplet $<$

$v_1, \ldots, v_n >$ of floating-point values such that $\forall c_j \in \mathcal{C}$, $eval(f_j(v_1, ..., v_n), r) \diamond 0$ is true.

Next section outlines the limit of standard local consistency algorithms for a safe pruning of the search space of an `FCSP`.

3.2 Limits of local consistencies

Local filtering algorithms over reals are based upon $2B$-consistency and Box-consistency. Formal definitions of $2B$-consistency and Box-consistency can be found in [CDR98]. We will just recall here the basic idea of these local consistencies.

$2B$-consistency [Lho93] states a local property on the bounds of the domains of a variable at a single constraint level. Roughly speaking, a constraint c is $2B$-consistent if, for any variable x, there exist values in the domains of all other variables which satisfy c when x is fixed to $\underline{x}$ and $\overline{x}$.

Algorithms achieving $2B$-filtering work by narrowing domains and, thus, need to compute the projection of a constraint c_j over the variable x_i in the space delimited by the domains of all variables but x_i occurring in c_j. Exact projection functions cannot be computed in the general case [CDR98]. Thus, $2B$-filtering decomposes the initial constraints in ternary basic constraints for which it is trivial to compute the projection [Dav87,Lho93].

Unfortunately, the inverse projection functions introduced by the $2B$-filtering are not conservative. For example, the equation $16.0 = 16.0 + x$, is handled by the solver as $x = 16.0 - 16.0$, which results in $x = 0$, whereas the floating-point solutions is any float belonging to $[-8.88178419700125232e-16, 1.77635683940025046e-15]$.

Box-consistency [BMVH94,HS94] is a coarser relaxation of Arc-consistency than $2B$-consistency. It generates univariate relations by replacing all existentially quantified variables but one with their intervals in the initial constraints. Contrary to $2B$-filtering, Box-filtering does not require any constraint decomposition of the initial constrain systems.

Effective implementations of Box-consistency (e.g., `Numerica`[VHMD97], `DeClic` [Gua00]) use the interval Newton method to compute the leftmost and the rightmost 0 of the generated univariate relations. Again, some solutions over the floats may be lost due to the Taylor manipulation introduced by the interval Newton method. For example, consider the equation $f(x, y, z) = x + y + z = 0$, with $x \in X = [-1, 1], y \in Y = [16.0, 16.0], z \in Z = [-16.0, -16.0]$. Interval Newton iteration $X := X \cap (m(X) - \frac{f(m(X), Y, Z)}{\frac{\partial f}{\partial x}(X, Y, Z)})$ immediately yields $X = [0, 0]$, whereas on floating-point numbers the solution is much more wider.

However, the definition of the Box-consistency does not mention the interval Newton method. Next section shows how the definition of the Box-consistency can be extended to handle interval of floating point numbers.

4 Solving FCSP

This section shows that interval analysis provides a decision procedure for checking whether a given interval contain no solution of an FCSP. We also introduce a new algorithm, which exploit local consistency algorithms as heuristics. Roughly speaking, local consistency algorithms are very helpful to identify a part of the search space which may not contain any solution; Interval analysis being used to verify that these spaces do actually contain no solution.

4.1 A decision procedure based upon interval analysis

The basic property that a filtering algorithm of an FCSP must satisfy is the conservation of all the solutions. So, to reduce interval $\mathbf{x} = [\underline{x}, \overline{x}]$ to $\mathbf{x} = [x_m, \overline{x}]$ we must check that there exists no solution for some constraint $f_j(x, x_1, ..., x_n) \diamond 0$ when $\mathbf{x}$ is set to $[\underline{x}, x_m]$. This job can be done by using interval analysis techniques to evaluate $f_j(x, x_1, ..., x_n)$ over $[\underline{x}, x_m]$ when all computations comply with the IEEE 754 recommendations.

To establish this nice property let us recall some basics on interval analysis over reals.

Let $\mathbf{x} = [\underline{x}, \overline{x}]$, with $\underline{x}$ and $\overline{x} \in I\!\!F$, be an interval of $\mathcal{I}$. We note $\mathbf{X} = [\underline{x}, \overline{x}]$ the corresponding interval over $I\!\!R$, i.e., $\mathbf{X}$ is the set of reals $\{v \in I\!\!R \mid \underline{x} \leq v \leq \overline{x}\}$. $\mathcal{I}_{\mathcal{R}}$ denotes the set of intervals over the reals. Functions defined over $I\!\!R$ will be subscripted by $\mathcal{R}$.

Definition 2 (Interval Extension [Moo66,Han92]).
Let $\tilde{x}$ denotes any value in interval $\mathbf{X}$.
• $\mathbf{f} : {\mathcal{I}_{\mathcal{R}}}^n \rightarrow \mathcal{I}_{\mathcal{R}}$ *is an interval extension of* $f_{\mathcal{R}} : \mathcal{R}^n \rightarrow \mathcal{R}$ *iff* $\forall\ \mathbf{X_1}, \ldots, \mathbf{X_n} \in \mathcal{I}_{\mathcal{R}} : f_{\mathcal{R}}(\tilde{x}_1, \ldots, \tilde{x}_n) \in \mathbf{f}(\mathbf{X_1}, \ldots, \mathbf{X_n})$.
• $\mathbf{c} : {\mathcal{I}_{\mathcal{R}}}^n \rightarrow \mathcal{B}ool$ *is an interval extension of* $c : \mathcal{R}^n \rightarrow \mathcal{B}ool$ *iff* $\forall\ \mathbf{X_1}, \ldots, \mathbf{X_n} \in \mathcal{I}_{\mathcal{R}} : c(\tilde{x}_1, \ldots, \tilde{x}_n) \Rightarrow \mathbf{c}(\mathbf{X_1}, \ldots, \mathbf{X_n})$

Definition 3 (Set Extension).
Let S be a subset of $\mathcal{R}$. The Hull *of S —denoted $\Box S$— is the smallest interval $\mathbf{I}$ such that $S \subseteq \mathbf{I}$.*

The term "smallest subset" (w.r.t. inclusion) must be understood according to the precision of floating-point operations. We consider that results of floating-point operations are outward-rounded when a function is evaluated over an interval.

Similarly, $\mathbf{f}$ is the *natural* interval extension of $f_{\mathcal{R}}$ (see [Moo66]) if $\mathbf{f}$ is obtained by replacing in $f_{\mathcal{R}}$ each constant k with the smallest interval containing k, each variable x with an interval variable $\mathbf{X}$, and each arithmetic operation with its optimal interval extension [Moo66]. $\mathbf{c}$ denotes the natural interval extension of c.

Now, let us recall a fundamental result of interval analysis :

Proposition 1. *[Moo66]*
Let $\mathbf{f}: \mathcal{I_R}^n \rightarrow \mathcal{I_R}$ *be the natural interval extension of* $f_{\mathcal{R}} : \mathcal{R}^n \rightarrow \mathcal{R}$*, then* $\Box\{f_{\mathcal{R}}(\tilde{x}_1, \ldots, \tilde{x}_n)\} \subseteq \mathbf{f}(\mathbf{X_1}, \ldots, \mathbf{X_n})$ *where* $\tilde{x}_i$ *denotes any value in* $\mathbf{X}_i$*.*

Proposition 1 states that $\mathbf{f}(\mathbf{X_1}, \ldots, \mathbf{X_n})$ contains at least all solutions in $\mathcal{R}$.

Proposition 2.
Let $\mathbf{f}: \mathcal{I_R}^n \rightarrow \mathcal{I_R}$ *be the natural interval extension of* $f_{\mathcal{R}} : \mathcal{R}^n \rightarrow \mathcal{R}$*, then* $\Box\{f(\tilde{v}_1, \ldots, \tilde{v}_n)\} \subseteq \mathbf{f}(\mathbf{X_1}, \ldots, \mathbf{X_n})$ *where* $\tilde{v}_i$ *denotes any value in* $\mathbf{x}_i$*.*

Sketch of the proof: It is trivial to show that $\mathbf{f}(\mathbf{X_1}, \ldots, \mathbf{X_n})$ contains all solutions over the floats when f is a basic operation, i.e., operations for which an optimal interval exists [Moo66]. Indeed, if f is a basic operation, the bounds of $\mathbf{f}(\mathbf{X_1}, \ldots, \mathbf{X_n})$ correspond respectively to $\min(eval(f(x_1, ..., x_n), -\infty)$ and $\max(eval(f(x_1, ..., x_n), +\infty))$ for $x_i \in \mathbf{x}_i$ and $i \in \{1, n\}$.

So, it results from the properties of the rounding operations[6] that

$$\underline{\mathbf{f}(\mathbf{X_1}, \ldots, \mathbf{X_n})} \leq eval(f(x_1, ..., x_n), r) \leq \overline{\mathbf{f}(\mathbf{X_1}, \ldots, \mathbf{X_n})}$$

for $r \in \{+\infty, -\infty, 0, near\}$, $x_i \in \mathbf{x}_i$ and $i \in \{1, n\}$. That is to say, whatever rounding mode is used, there exist no floating-point value $v_x \in \mathbf{x}$ such that $eval(f(v_x, v_1, \ldots, v_k), r) \notin \mathbf{f}(\mathbf{X}, \mathbf{X_1}, \ldots, \mathbf{X_k})$.

It is straightforward to show by induction that this property still holds when $f(x_1, \ldots, x_n)$ is a composition of basic operations. The essential observation is that the computation of $eval(f_j(x_1, ..., x_n), r)$ and of $\mathbf{f}(\mathbf{X_1}, \ldots, \mathbf{X_n})$ are performed by evaluating the same sequence of basic operations (based on the same abstract tree). $\Box$

Thus, interval analysis based evaluation provides a safe procedure to check whether a constraint c may contain a solution in some interval.

Now, we are in position to introduce a "conservative" local consistency for interval of floating point numbers.

Definition 4 (FP-Box–Consistency). *Let* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ *be an FCSP and* $c \in \mathcal{C}$ *a k-ary constraint c is FB-Box–Consistent if, for all* x_i *in* $X(c)$ *such that* $D_{x_i} = [a, b]$*, the following relations hold :*
1. $\mathbf{c}(\mathbf{D}_{x_1}, \ldots, \mathbf{D}_{x_{i-1}}, [a, a], \mathbf{D}_{x_{i+1}}, \ldots, \mathbf{D}_{x_k})$,
2. $\mathbf{c}(\mathbf{D}_{x_1}, \ldots, \mathbf{D}_{x_{i-1}}, [b, b], \mathbf{D}_{x_{i+1}}, \ldots, \mathbf{D}_{x_k})$.

Next section describes a filtering algorithm which enforces FP-Box–Consistency.

4.2 A new Filtering algorithm

In the following, we introduce a new algorithm for pruning the domain of an `FCSP`. This algorithm is adapted from the "Branch and Prune Algorithm Newton" algorithm introduced in [VMK97]

[6] It follows from the definition of rounding [ANS85] that: $Round_{-\infty}(x\ .\ y) \leq Round_r(x\ .\ y) \leq Round_{+\infty}(x\ .\ y)$ where $Round_r$ is rounding toward r, for all $r \in \{-\infty, 0, near, +\infty\}$.

Algorithm 1 prunes the domain of a variable by increasing the lower bound. The function that computes the upper bound can be written down in a similar way.
The function $\texttt{guess}(c_j, \mathbf{x})$ searches for the left most 0 of the constraint c_j. The simplest way to implement $\texttt{guess}(c_j, \mathbf{x})$ consists in using a dichotomy algorithm to split the domain. Of course, such a process would be very inefficient. That is why we suggest to use 2B or Box–consistency to implement function $\texttt{guess}(c_j, \mathbf{x})$. Of course, different heuristics may be used to choose x_m. For instance, a value closer from $\overline{x}$ than the midpoint could be selected.

Algorithm 1 Computing Lower bound

Function `Lower-bound(IN:` $c_j, \mathbf{x}$`)` **return** Lower bound of $\mathbf{x}$
% ϵ: minimal reduction

$\overline{x} \leftarrow$**guess**$(c_j, \mathbf{x})$
if $\mathbf{c}_j([\underline{x}, \overline{x}], \mathbf{X}_1, \ldots, \mathbf{X}_n)$ & $\underline{x} < \overline{x}$
then
$x_m \leftarrow \frac{\underline{x}+\overline{x}}{2}$
if $\mathbf{c}_j([\underline{x}, x_m], \mathbf{X}_1, \ldots, \mathbf{X}_n)$
then return *Lower-bound*$(c_j, [\underline{x}, x_m])$
else return *Lower-bound* $(c_j, [x_m, \overline{x}])$
endif
else return $\overline{x}$
endif

end `Lower-bound`

The scheme of the standard narrowing algorithm —derived from AC3 [Mac77]– is given by algorithm 2. $narrow(c, \mathbf{X})$ is a function which prunes the domains of all the variables occurring in c. Implementation of $narrow(c, \mathbf{X})$ consists just in a call of the functions `Lower_bound` and `Upper_bound` for each variable. FB–Filtering achieves FP–Box consistency. An approximation of FP–Box consistency can be computed by replacing the test $\underline{x} < \overline{x}$ by $|\underline{x} - \overline{x}| > \epsilon$ where ϵ is an arbitrary value.

Algorithms that achieve stronger consistency filtering can be derived from algorithm 2 in the same way as 3B–consistency [Lho93] (resp. Bound–consistency [VHMD97,PVH98]) algorithms have been derived from the 2B–consistency (resp. Box-consistency) algorithms.

4.3 Labelling and search

Solving a constraint system requires to alternate filtering steps and search steps. In the applications where constraints over the floats occur, we often have to deal with very different types of problems:

Algorithm 2 FB–Filtering

```
Procedure FB-Filtering(IN C, INOUT X)

    Queue ← C
    while Queue ≠ ∅
        c ← POP(Queue)
        X' ← narrow(c, X)
        if X' ≠ X then
            X ← X'
            Queue ←Queue ∪ {c' ∈ C | X(c) ∩ X(c') ≠ ∅}
        endif

end FB-Filtering
```

- Problems without solutions, e.g. , the point to reach correspond to so-called dead code[7] in the test case generation application;
- Problems with a huge number of solutions, e.g. , the point to reach correspond to a standard instruction;
- Problems with very few solutions, e.g. , the point to reach correspond to very specific exceptions handling.

Stronger consistencies are a key issue to prove that a problem has no solution. Although we could define a complete labelling and filtering process, practically we may fail to prove that some problems do not have any solution. Since in most cases numerous solutions exists, we suggest to start by a labelling process which "fairly" selects values in the domain of a variable (see section 5.2).

5 Experimentations and applications

We have implemented the `FB-filtering` algorithm and various labelling strategies in a solver named FPics (which stands for floating-point numbers interval constraint solver). In the first subsection, we compare the results of `FB-filtering`, `DeClic` and `PROLOG IV` on several small examples. In the second subsection, we compare these different solvers on a small test case generation problem. The labelling strategy developed for that application is also described.
Unless otherwise specified, the constraints are solved with a rounding mode set to *near*.

5.1 Naive examples

A simple example already given to argue for floating-point CSP is the following equation : $x + y = y$ where y is a constant. This equation was used to show that solvers over $\mathbb{R}$ do not take into account addition cancellation.

[7] Dead code is a piece of code in a program, which can never be executed [Kor90].

Consider the equation $x+16.1 = 16.1$. Table 1 and table 2 show the computation results of `DeClic`, `PROLOG IV`, and `FB-filtering` on two different instances of such an equation.

Table 1. Computing results for $x + 16.1 = 16.1$

results	$[\underline{x}, \overline{x}]$
DeClic	$[-7.10542735760100186e - 15, \ +7.10542735760100186e - 15]$
PROLOG IV	$[0, 0]$
FB-filtering	$[-3.55271367880050053e - 15, \ 3.55271367880050053e - 15]$

Table 2. Computing results for $x + 16.0 = 16.0$

result	$[\underline{x}, \ \overline{x}]$
DeClic	$[0, \ 0]$
PROLOG IV	$[0, \ 0]$
FB-filtering	$[-1.77635683940025027e - 15, \ 3.55271367880050053e - 15]$

The two different constants used in these examples illustrate well the behaviours of solvers over $\mathbb{R}$. `FB-filtering` preserve all the solutions over $\mathbb{F}$ in both cases. Note that the second result provided by the `FB-filtering` is not symmetric around 0. This is due to the fact that the exponent of the first floating-point number strictly smaller than 16 and the exponent of 16 are different.

`DeClic` converts decimal numbers to binary floating-point numbers whenever an interval contains only one float. This conversion extends the interval to up to three floats unless the float maps an integer. That is why it yields a larger interval for the first example.

The reader can easily check that numerous solutions exist in the intervals yield by `FB-filtering`. Consider for instance the subpart $X = [1.0e-200, 1.0e-15]$ of the interval computed by `FB-filtering`. The evaluation of $(16.0 - x) - 16.0$ yields an interval which actually contains 0. The following lines of C code compute that interval R :

```
round_down();
R.low = (16.0 + X.low) - 16.0;
round_up();
R.high = (16.0 + X.high) - 16.0;
```

5.2 Application to automatic data test generation

In this subsection we investigate a small test case generation example. In automatic test case generation applications, a problem is defined by the set of constraints that represent all the executable path that goes through some point or instruction[8]. Before going into the details of the example, let us first introduce the labelling process.

Labelling

The labelling process is based on an uniform exploration of the whole domain. It is parameterised by a *depth* which defines the number p of levels of exploration. Figure 1 illustrates the different steps of this enumeration process on one variable. The number correspond to the levels.

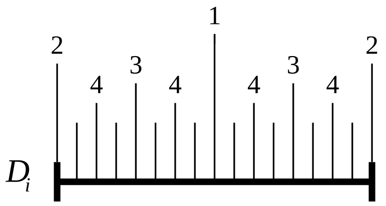

Fig. 1. Barycentric enumeration schema

Such an enumeration process is applied in a round robin fashion to all variables; each labelling step being followed with a filtering step.

The solver `FPics` is based on such a labelling process and on the `FB-filtering` algorithm. It also propagates the constant values after each labelling step.

The cubic example

Consider the piece of `C` code in figure 2, which is extracted from a program that computes the cubic roots of a number.

Here is the constraint system that defines the path going through `point 1`:

$Q = (3.0 \times B - A \times A)/3.0$

$\wedge\ R = (2.0 \times A \times A \times A - 9.0 \times A \times B + 27.0 \times C)/27.0$

$\wedge\ DELTA = (Q \times Q \times Q/27.0 + R \times R/4.0)$

$\wedge\ abs(DELTA) < 1.0e - 40$

where *abs* stands for the function that returns the absolute value of a number.

We have generated test data which satisfy these constraints with `DeClic` and `FPics`. For both solvers, we have used the same labelling process with a depth of 6.

DeClic could generate 35 sets of input values, 10 of them were wrong. The problems mainly came from the combination of outward rounding and the con-

[8] The generation of this constraint system is not always possible: it is undecidable in the general case since it can be reduced to the halting problem.

```
int solve(double a, double b, double c) {
   double q, r, delta;
   ...

   q = (3.0*b - a*a)/3.0;
   r = (2.*a*a*a - 9.0*a*b + 27.0*c)/27.0;
   delta = q*q*q/27.0 + r*r/4.0;
   if(fabs(delta) < 1.0e-40) {
     /** point 1 **/
     ...
   } else {
     ...
   }
   ...
 }
```

Fig. 2. Code of cubic-roots example

straint $abs(DELTA) < 1.0e-40$: outward rounding transforms floating-point values in intervals and the constraints $DELTA = (Q \times Q \times Q/27.0 + R \times R/4.0)$ holds as long as $DELTA$ contains values in $(-1.0e-40, 1.0e-40)$, even if the evaluation of C expression is out of range.

`FPics` did generate 337 sets of input values. `FPics` could generate much more test data because it preserves all floating-point solutions. However the point is that, for all of them, the `C` program did reach '`point 1`'. Moreover, both the `C` program and `FPics` generate the sames values for the local variables.

6 Conclusion

This paper has introduced a new framework for tackling constraint systems over the floating-point numbers, which are required to model imperative programs. After a detailed analysis of the specificity of constraints systems over the floats, we have introduced algorithms that achieve a safe filtering of the domains of floating point valued variables. Experimentations with the FPics solver are promising and provide a first validation of the proposed approach.

Further works concerns the improvement of the guess function of algorithm 1, the handling of specific values like NaNs or exception flags as well as efficiency issues. It would also be worthwhile to investigate the problems that raise implementations where functions are not exactly rounded.

Acknowledgements: Many thanks to Bernard Bottela and Arnaud Gotlieb for numerous and enriching discussions on this work. We also gratefully thank an anonymous reviewer for his constructive remarks, and Gilles Trombettoni for his careful reading of this paper.

References

[ANS85] ANSI/IEEE, New York. *IEEE Standard for Binary Floating Point Arithmetic*, Std 754-1985 edition, 1985.

[BMVH94] F. Benhamou, D. McAllester, and P. Van-Hentenryck. Clp(intervals) revisited. In *Proceedings of the International Symposium on Logic Programming*, pages 124–138, 1994.

[CDR98] H. Collavizza, F. Delobel, and M. Rueher. A note on partial consistencies over continuous domains solving techniques. In *Proc. CP98 (Fourth International Conference on Principles and Practice of Constraint Programming), Pisa, Italy, October 26-30*, 1998.

[Col94] A. Colmerauer. Spécifications de prolog iv. Technical report, GIA, Faculté des Sciences de Luminy,163, Avenue de Luminy 13288 Marseille cedex 9 (France), 1994.

[Dav87] E. Davis. Constraint propagation with interval labels. *Journal of Artificial Intelligence*, pages 32:281–331, 1987.

[GBK98] A. Gotlieb, B. Botella, and Rueher K. A clp framework for computing structural test data. In *Proc. ISSTA 98 (Symposium on Software Testing and Analysis),*. ACM SIGSOFT, vol. 2, pp. 53-62, 1998.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

[Got00] A. Gotlieb. *Automatic Test Data Generation using Constraint Logic Programming*. PhD thesis, Université de Nice — Sophia Antipolis, France, 2000.

[Gua00] F. Gualard. *Langages et environnements en programmation par contraintes d'intervalles*. PhD thesis, Université de Nantes — 2, rue de la Houssinière, F-44322 NANTES CEDEX 3, France, 2000.

[Han92] E. Hansen, editor. *Global optimization using interval analysis*. Marcel Dekker, 1992.

[HS94] H. Hong and V. Stahl. Safe starting regions by fixed points and tightening. *Computing*, pages 53:323–335, 1994.

[Kea96] R. Baker Kearfott. *Rigorous Global Search: Continuous Problems*. Number 13 in Nonconvex optimization and its applications. Kluwer Academic Publishers Group, Norwell, MA, USA, and Dordrecht, The Netherlands, 1996.

[Kor90] Bogdan Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, august 1990.

[Lho93] O. Lhomme. Consistency techniques for numeric csps. In *Proceedings of IJCAI'93*, pages 232–238, 1993.

[Mac77] A. Mackworth. Consistency in networks of relations. *Journal of Artificial Intelligence*, pages 8(1):99–118, 1977.

[Moo66] R. Moore. *Interval Analysis*. Prentice Hall, 1966.

[PVH98] J.F. Puget and P. Van-Hentenryck. A constraints satisfaction approach to a circuit design problem. *Journal of global optimization*, pages 13(1):75–93, 1998.

[VHMD97] P. Van-Hentenryck, L. Michel, and Y. Deville. *Numerica : a Modeling Languge for Global Optimization*. MIT press, 1997.

[VMK97] P. Van Hentenryck, D. McAllester, and D. Kapur. Solving polynomial systems using a branch and prune aprroach. *SIAM Journal*, 34(2), 1997.