Symbolic Path-Oriented Test Data Generation for Floating-Point Programs

Roberto Bagnara*, Matthieu Carlier†, Roberta Gori‡, Arnaud Gotlieb§
*BUGSENG srl and Department of Mathematics and Computer Science, University of Parma, Italy
Email: roberto.bagnara@bugseng.com
†INRIA Rennes Bretagne Atlantique, France
‡Department of Computer Science, University of Pisa, Italy
Email: gori@di.unipi.it
§Certus Software V&V Center, SIMULA Research Laboratory, Norway
Email: arnaud@simula.no

***Abstract*—Verifying critical numerical software involves the generation of test data for floating-point intensive programs. As the symbolic execution of floating-point computations presents significant difficulties, existing approaches usually resort to random or search-based test data generation. However, without symbolic reasoning, it is almost impossible to generate test inputs that execute many paths with floating-point computations. Moreover, constraint solvers over the reals or the rationals do not handle the rounding errors. In this paper, we present a new version of FPSE, a symbolic evaluator for C program paths, that specifically addresses this problem. The tool solves path conditions containing floating-point computations by using correct and precise projection functions. This version of the tool exploits an essential filtering property based on the representation of floating-point numbers that makes it suitable to generate path-oriented test inputs for complex paths characterized by floating-point intensive computations. The paper reviews the key implementation choices in FPSE and the labeling search heuristics we selected to maximize the benefits of enhanced filtering. Our experimental results show that FPSE can generate correct test inputs for selected paths containing several hundreds of iterations and thousands of executable floating-point statements on a standard machine: this is currently outside the scope of any other symbolic-execution test data generator tool.**

I. Introduction

During the last decade, the use of floating-point computations in the design of critical systems has become increasingly acceptable. Even in the civil and military avionics domain, which are among the most critical domains for software, floating-point numbers are now seen as a sufficiently-safe, faster and cheaper alternative to fully-controlled, implementation-based fixed-point arithmetic.

Acceptance of floating-point computations in the design of critical systems took a long time. In fact, rounding errors are difficult to predict and control, and can lead to catastrophic failures. For instance, during the first Persian Gulf War, the failure of a Patriot missile battery in Dhahran was traced to an accumulating rounding error in the continuous execution of tracking and guidance software, causing the death of several civilians [26]. A careful analysis of this failure revealed that, even though the rounding error obtained at each step of the floating-point computation was very small, the propagation during a long loop-iterating path could lead to dramatic imprecision.

Adoption of floating-point computations in critical systems involves the use of thorough unit testing procedures that are able to exercise complex chains of floating-point operations. In particular, a popular practice among software engineers in charge of the testing of floating-point-intensive computations consists in executing carefully chosen loop-iterating paths in programs. They usually pay more attention to the paths that are most likely to expose the system to unstable numerical computations. For critical systems, a complementary requirement is to demonstrate the infeasibility of certain paths, in order to convince a third-party certification authority that certain unsafe behaviors of the systems cannot be reached. As a consequence, software engineers face two difficult problems:

1) How to accurately predict the expected output of a given floating-point computation?[1]
2) How to find a test input that is able to exercise a given path, the execution of which depends on the results of floating-point computations, or to guarantee that such a path is infeasible?

The first problem has been well addressed in the literature [15] through several techniques, either based on multiple related program executions [1], [9], or on statically-extracted properties of programs [12], or on perturbation techniques to evaluate the stability of a numerical program [27]. In contrast, the second problem received only little attention. Beyond the seminal work of W. Miller and D. L. Spooner [23], who proposed to guide the search of floating-point inputs to execute a selected path, few approaches try to exactly reason about floating-point computations. The work in [23] paved the way to the development of *search-based test data generation* techniques, which consist in searching test inputs by minimizing a cost function, evaluating the distance between the currently executed path and a targeted selected path [2], [14], [16], [20]. Although these techniques

[1]This is the the well-known *oracle problem* [28].

enable quick and efficient coverage of testing criteria such as "all decisions," they are unfortunately sensible to the rounding-errors incurred in the computation of the branch distance [2]. Moreover, search-based test data generation cannot be used to study path feasibility, i.e., to decide whether a possible execution path involving floating-point computations is feasible or not in the program. In addition, these techniques can be stuck in local minima without being able to provide a meaningful result [2]. An approach to tackle these problems combines program execution and symbolic reasoning [10]. This kind of reasoning requires solving constraints over floating-point numbers in order to generate test inputs that exercise a selected behavior of the program under test. However, solving floating-point constraints is hard and requires dedicated filtering algorithms [21], [22]. According to our knowledge, this approach is currently implemented in four solvers only: ECLAIR[2], FPCS [5], FPSE[3] [7], and Gatel, a test data generator for Lustre programs [5], [18].

A promising approach to improve the filtering capabilities of constraints over floating-point variables consists in using some peculiar numerical properties of floating-point numbers. For linear constraints, this led to a relaxation technique where floating-point numbers and constraints are converted into constraints over the reals by using linear programming approaches [4]. For interval-based consistency approaches, B. Marre and C. Michel identified a property of the representation of floating-point numbers and proposed to exploit it in filtering algorithms for addition and subtraction constraints [19]. In [8], a reformulation of the Marre-Michel property in terms of filtering by maximum ULP (*Units in the Last Place*) was proposed in order to ease its implementation in constraint solvers such as FPSE. In addition, the authors sketched a generalization of the property to multiplication and division constraints. This paper is concerned with this challenge. More precisely, it addresses the question of whether the Marre-Michel property can be useful for the automatic solution of realistic test input generation problems. The contributions of the paper are:

1) The filtering algorithm proposed in [19] for addition and subtraction is reformulated and corrected.
2) The plan anticipated in [8] is brought to completion: a uniform framework is thoroughly defined that generalizes the property identified by Marre and Michel to the case of multiplication and division.
3) Our implementation of filtering by maximum ULP in FPSE is presented and critical design choices (e.g., to avoid slow convergence phenomena) are explained.
4) Experimental results are presented on constraint systems that have been extracted from programs engaging into intensive floating-point computations. These results show that the Marre-Michel property and its generalization defined in this paper speed up the test inputs generation process.

[2]http://bugseng.com/products/eclair

[3]http://www.irisa.fr/celtique/carlier/fpse.html

The rest of the paper is organized as follows. Next section presents the IEEE 754 standard of binary floating-point numbers and introduces the notations used throughout the paper. Section III recalls the basic principles of interval-based consistency techniques over floating-point variables and constraints. Section IV presents our generalization of the Marre-Michel property, while Section V details our implementation of the property in FPSE. Section VI presents our experimental results and analysis. Section VII discusses related work and Section VIII concludes the paper.

II. Preliminaries

A. IEEE 754

This section recalls the arithmetic model specified by the IEEE 754 standard for binary floating-point arithmetic [13].

IEEE 754 binary floating-point formats are uniquely identified by: $p \in \mathbb{N}$, the number of significant digits (precision); $e_{\max} \in \mathbb{N}$, the maximum exponent; $e_{\min} \in \mathbb{N}$, the minimum exponent (usually $1 - e_{\max}$). The *single precision* format has $p = 24$ and $e_{\max} = 127$, the *double precision* format has $p = 53$ and $e_{\max} = 1023$ (IEEE 754 also defines extended precision formats). An IEEE 754 floating-point number z has the form $(-1)^s a.m \times 2^e$ where s is the *sign bit*, a is the *hidden bit*, m is the *significand* and the *exponent* e is also denoted by e_z. Each format defines several classes of numbers: normal numbers, subnormal numbers, signed zeroes, infinities and NaNs (*Not a Number*). The smallest positive *normal* floating-point number is $f_{\min}^{\mathrm{nor}} = 1.0 \cdots 0 \times 2^{e_{\min}} = 2^{e_{\min}}$ and the largest is $f_{\max} = 1.1 \cdots 1 \times 2^{e_{\max}} = 2^{e_{\max}}(2 - 2^{1-p})$; normal numbers have the hidden bit $a = 1$. The non-zero floating-point numbers whose absolute value is less than $2^{e_{\min}}$ are called *subnormals*: they always have fewer than p significant digits as their hidden bit is $a = 0$. Every finite floating-point number is an integral multiple of the smallest subnormal magnitude $f_{\min} = 0.0 \cdots 01 \times 2^{e_{\min}} = 2^{e_{\min}+1-p}$. There are two infinities, denoted by $+\infty$ and $-\infty$, and two *signed zeros*, denoted by $+0$ and -0: they allow some algebraic properties to be maintained [11]. NaNs are used to represent the results of invalid computations such as a division or a subtraction of two infinities. They allow the program execution to continue without being halted by an exception.

IEEE 754 defines five rounding directions: toward negative infinity (*down*), toward positive infinity (*up*), toward zero (*chop*) and toward the nearest representable value (*near*); the latter comes into two flavors: *tail-to-even* or *tail-to-away* in which values with even mantissa or values away from zero are preferred, respectively. This paper is only concerned with round-to-nearest, tail-to-even, which is, by far, the most widely used. The round-to-nearest, tail-to-even value of a real number x will be denoted by $\circ(x)$.

All rounding modes are monotonic; in particular, for each $x, y \in \mathbb{R}$, $x \le y$ implies $\circ(x) \le \circ(y)$.

The most important requirement of IEEE 754 arithmetic is the accuracy of floating-point computations: add, subtract, multiply, divide, square root, remainder, conversion and comparison operations must deliver to their destination the exact result rounded as per the rounding mode in effect and the format of the destination. It is said that these operations are "exactly rounded."

The accuracy requirement of IEEE 754 can still surprise the average programmer: for example the single precision, round-to-nearest addition of 999999995904 and 10000 (both numbers can be exactly represented) gives 999999995904, i.e., the second operand is absorbed. The maximum error committed by representing a real number with a floating-point number under some rounding mode can be expressed in terms of the function $\mathrm{ulp}\colon \mathbb{R} \to \mathbb{R}$ [24]. Its value on 1.0 is about 10^{-7} for the single precision format.

The *chop* and *near* rounding modes are *symmetric*, i.e., the value after rounding does not depend on the sign: for each $x \in \mathbb{R}$, $\circ(x) = -\circ(-x)$.

B. Notation

$\mathbb{R}$ denotes the set of real numbers while $\mathcal{F}_{p,e_{\max}}$ denotes an idealized set of binary floating-point numbers, defined from a given IEEE 754 format: this excludes subnormals and NaNs, but includes $-\infty$, $+\infty$ and zeroes. This restriction allows to considerably simplify the presentation (e.g., avoiding all technical details concerning subnormals); yet, everything can be generalized to any IEEE 754 binary floating-point format [3]. The exposition is also much simplified by allowing $e_{\max}$ to be ∞, i.e., by considering an idealized set of floats where the exponent is unbounded. $\mathcal{F}^+_{p,e_{\max}}$ denotes the "positive" subset of $\mathcal{F}_{p,e_{\max}}$, i.e., with $s = 0$. When the format is clear from the context, a real decimal constant (such as 10^{12}) denotes the corresponding round-to-nearest, tail-to-even floating-point value (i.e., 999999995904 for 10^{12}). Henceforth, x^+ (resp., x^-) denotes the smallest (resp., greatest) floating-point number strictly greater (resp., smaller) than x w.r.t. the considered IEEE 754 format. Of course, we have ${f_{\max}}^+ = +\infty$ and $(-f_{\max})^- = -\infty$.

Binary arithmetic operations over the floats will be denoted by $\oplus$, $\ominus$, $\otimes$ and $\oslash$, corresponding to $+$, $-$, $\cdot$ and $/$ over the reals, respectively. According to IEEE 754, they are defined with the rounding operator $\circ$ by $x \oplus y = \circ(x + y)$, $x \ominus y = \circ(x - y)$, $x \otimes y = \circ(x \cdot y)$ and $x \oslash y = \circ(x/y)$. As IEEE 754 floating-point numbers are closed by negation, we will denote the negation of $x \in \mathcal{F}_{p,e_{\max}}$ simply by $-x$. The symbol $\odot$ denotes any of $\oplus$, $\ominus$, $\otimes$ or $\oslash$. A floating-point variable x is associated to an interval of possible floating-point values; we will write $\mathrm{x} \in [\underline{\mathrm{x}}, \overline{\mathrm{x}}]$, where $\underline{\mathrm{x}}$ and $\overline{\mathrm{x}}$ denote the smallest and greatest value of the the interval, $\underline{\mathrm{x}} \le \overline{\mathrm{x}}$ and either $\underline{\mathrm{x}} \ne +0$ or $\overline{\mathrm{x}} \ne -0$.

III. Background on Constraint Solving over Floating-Point Variables

A. Interval-based Consistency on Arithmetic Constraints

Program analysis usually starts with the generation of an intermediate code representation in a form called *three-address code* (TAC). In this form, complex arithmetic expressions and assignments are decomposed into sequences of assignment instructions of the form $\mathtt{result} :=$ $\mathtt{operand}_1$ $\mathtt{operator}$ $\mathtt{operand}_2$. A further refinement consists in the computation of the *static single assignment form* (SSA) whereby, labeling each assigned variable with a fresh name, assignments can be considered as if they were equality constraints. For example, the TAC form of the floating-point assignment $\mathtt{z} := \mathtt{z} * \mathtt{z} + \mathtt{z}$ is $\mathtt{t} := \mathtt{z} * \mathtt{z}$; $\mathtt{z} := \mathtt{t} + \mathtt{z}$, which in SSA form becomes $\mathtt{t}_1 := \mathtt{z}_1 * \mathtt{z}_1$; $\mathtt{z}_2 := \mathtt{t}_1 + \mathtt{z}_1$, which, in turn, can be regarded as the conjunction of the constraints $t_1 = z_1 \otimes z_1$ and $z_2 = t_1 \oplus z_1$.

In an interval-based consistency approach to constraint solving over the floats, constraints are used to iteratively refine the intervals associated to each variable. A *projection* is a function that, given a constraint and the intervals associated to two of the variables occurring in it, computes a possibly refined interval for the third variable (the projection is said to be *over* the third variable). Taking $z_2 = t_1 \oplus z_1$ as an example, the projection over z_2 is called *direct projection* (it goes in the same sense of the TAC assignment it comes from), while the projections over t_1 and z_1 are called *indirect projections*. Non-optimal projections for the four arithmetic operations can be found in [7], [21].[4]

B. The Marre-Michel Property

In [19], Marre and Michel published an idea to improve the filtering of the addition/subtraction projectors. This is based on a property of the distribution of floating-point numbers among the reals: the greater a float is, the greater the distance between it and its immediate successor is. More precisely, for a given float x with exponent e_x, if $\Delta = x^+ - x$, then for y of exponent $e_x + 1$ we have $y^+ - y = 2\Delta$.

Proposition 3.1: [19, Proposition 1] Let $z \in \mathcal{F}_{p,\infty}$ be such that $0 < z < +\infty$; let also

$$\begin{aligned} z &= 1.b_2 \cdots b_i \overbrace{0 \cdots 0}^{k} \times 2^{e_z}, && \text{with } b_i = 1; \\ \alpha &= 1.1 \cdots 1 \times 2^{e_z + k}, && \text{with } k = p - i; \\ \beta &= \alpha \oplus z. \end{aligned}$$

Then, for each $x, y \in \mathcal{F}_{p,\infty}$, $z = x \ominus y$ implies $x \le \beta$ and $y \le \alpha$. Moreover, $\beta \ominus \alpha = \beta - \alpha = z$.

[4] E.g., for the constraint $\mathrm{z} = \mathrm{x} \oplus \mathrm{y}$ we have $\overline{\mathrm{z}} = \overline{\mathrm{x}} \oplus \overline{\mathrm{y}}$ and $\underline{\mathrm{z}} = \underline{\mathrm{x}} \oplus \underline{\mathrm{y}}$ (direct), $\overline{\mathrm{x}} = \mathrm{mid}(\overline{\mathrm{z}}, \overline{\mathrm{z}}^+) \ominus \underline{\mathrm{y}}$ and $\underline{\mathrm{x}} = \mathrm{mid}(\underline{\mathrm{z}}, \underline{\mathrm{z}}^-) \ominus \overline{\mathrm{y}}$ (1st indirect), $\overline{\mathrm{y}} = \mathrm{mid}(\overline{\mathrm{z}}, \overline{\mathrm{z}}^+) \ominus \underline{\mathrm{x}}$ and $\underline{\mathrm{y}} = \mathrm{mid}(\underline{\mathrm{z}}, \underline{\mathrm{z}}^-) \ominus \overline{\mathrm{x}}$ (2nd indirect). Here, for finite $x, y \in \mathcal{F}_{p,e_{\max}}$, we denote by $\mathrm{mid}(x, y)$ the number that is exactly halfway between x and y; note that either $\mathrm{mid}(x, y) \in \mathcal{F}_{p,e_{\max}}$ or $\mathrm{mid}(x, y) \in \mathcal{F}_{p+1,e_{\max}}$.

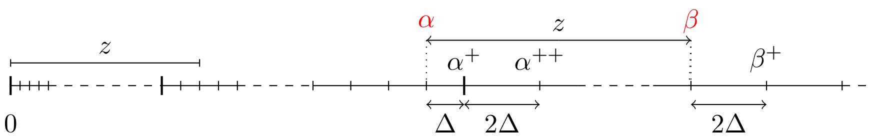

Figure 1. An illustration of the Marre-Michel property

This property, which can be generalized to subnormals, can intuitively be explained on Figure 1 as follows. Let $z \in \mathcal{F}_{p,\infty}$ be a strictly positive constant such that $z = x \ominus y$, where $x, y \in \mathcal{F}_{p,\infty}$ are unknown. The Marre-Michel property says that y cannot be greater than α. In fact, α is carefully positioned so that $\alpha^{++} - \alpha^{+} = 2(\alpha^{+} - \alpha)$, $e_\alpha + 1 = e_\beta$ and $z = \beta - \alpha$; if we take $y = \alpha^+$ we need $x > \beta$ if we want $z = x - y$; however, the smallest element of $\mathcal{F}_{p,\infty}$ that is greater than β, β^+, is 2Δ away from β, i.e., too much. Going further with y does not help: if we take $y \geq \alpha^+$, then $y - \alpha$ is an odd multiple of Δ (one Δ step from α to α^+, all the subsequent steps being even multiples of Δ), whereas for each $x \geq \beta$, $x - \beta$ is an even multiple of Δ. Hence, if $y > \alpha$, $\bigl|z - (x - y)\bigr| \geq \Delta = 2^{e_z+1-i}$. However, since $k \neq p-1$, $z^+ - z = z - z^- = 2^{e_z+1-p} \leq \Delta$. The last inequality, which holds because $p \geq i$, implies $z \neq x \ominus y$. A similar reasoning allows to see that x cannot be greater than β independently from the value of y.

In order to improve the filtering of the addition/subtraction projectors, in [19], Marre and Michel presented an algorithm to maximize the values of α and β over an interval. That algorithm and the main ideas behind the work presented in [19] will be revisited and discussed in detail in Section IV-C.

IV. Filtering by Maximum ULP

This section reformulates the Marre-Michel property so as to generalize it to multiplication and division. The filtering algorithms that result from this generalization are collectively called *filtering by maximum ULP*.

A. Upper Bound

For each floating-point operation $\odot \in \{\oplus, \ominus, \otimes, \oslash\}$, we will define the sets $\mathcal{F}_\odot \subseteq \mathcal{F}_{p,e_{\max}}$ and $\overline{\mathcal{F}}_\odot \subseteq \mathcal{F}^+_{p,\infty}$. Then we will define a function $\overline{\delta}_\odot \colon \mathcal{F}_\odot \to \overline{\mathcal{F}}_\odot$ satisfying the following properties, for each $z \in \mathcal{F}_\odot \setminus \{-0, +0, -\infty\}$:

$$\exists y \in \overline{\mathcal{F}}_\odot \,.\, \overline{\delta}_\odot(z) \odot y = z; \quad (1)$$

$$\forall z' \in \overline{\mathcal{F}}_\odot : z' > \overline{\delta}_\odot(z) \implies \nexists y \in \overline{\mathcal{F}}_\odot \,.\, z' \odot y = z. \quad (2)$$

In words, $\overline{\delta}_\odot(z)$ is the greatest float in $\overline{\mathcal{F}}_\odot$ that can be the left operand of $\odot$ to obtain z. Remark that we may have $\overline{\mathcal{F}}_\odot \nsubseteq \mathcal{F}_{p,e_{\max}}$: properties (1) and (2) refer to an idealized set of floating-point numbers with unbounded exponents.

Since we are interested in finding the upper bound of $\overline{\delta}_\odot(z)$ for $z \in [\underline{z}, \overline{z}]$, we need the following

Proposition 4.1: Let $w, v_1, \ldots, v_n \in \mathcal{F}_\odot \setminus \{-0, +0, -\infty\}$ be such that, for each $i = 1, \ldots, n$, $\overline{\delta}_\odot(w) \geq \overline{\delta}_\odot(v_i)$. Then, for each $i = 1, \ldots, n$ and $w' \in \overline{\mathcal{F}}_\odot$ such that $w' > \overline{\delta}_\odot(w)$, there does not exist a float $y \in \overline{\mathcal{F}}_\odot$ such that $w' \odot y = v_i$.

Let $\mathrm{z} = \mathrm{x} \odot \mathrm{y}$ be a constraint where $-0, +0, -\infty \notin [\underline{z}, \overline{z}]$ and let $w \in [\underline{z}, \overline{z}]$ be such that $\overline{\delta}_\odot(w) \geq \overline{\delta}_\odot(v)$ for each $v \in [\underline{z}, \overline{z}]$: then no element of x that is greater than $\overline{\delta}_\odot(w)$ can participate to a solution of the constraint.

Dually, in order to refine the upper bound of y subject to the constraint $\mathrm{z} = \mathrm{x} \odot \mathrm{y}$, it is possible to define a function $\overline{\delta}'_\odot$ that satisfies properties similar to (1) and (2). In this paper we will focus on bounds for x only. Note, though, that when $\odot$ is commutative (i.e., $\oplus$ or $\otimes$), $\overline{\delta}_\odot = \overline{\delta}'_\odot$.

B. Lower bound

For computing the lower bound, we will introduce functions $\underline{\delta}_\odot \colon \mathcal{F}_\odot \to \overline{\mathcal{F}}_\odot$ for each $z \in \mathcal{F}_\odot \setminus \{-0, +0, +\infty\}$:

$$\exists y \in \overline{\mathcal{F}}_\odot \,.\, \underline{\delta}_\odot(z) \odot y = z; \quad (3)$$

$$\forall z' \in \overline{\mathcal{F}}_\odot : z' < \underline{\delta}_\odot(z) \implies \nexists y \in \overline{\mathcal{F}}_\odot \,.\, z' \odot y = z. \quad (4)$$

These properties entail a result similar to Proposition 4.1: given the constraint $\mathrm{z} = \mathrm{x} \odot \mathrm{y}$ where $-0, +0, +\infty \notin [\underline{z}, \overline{z}]$ and $w \in [\underline{z}, \overline{z}]$ such that $\underline{\delta}_\odot(w) \leq \underline{\delta}_\odot(v)$ for each $v \in [\underline{z}, \overline{z}]$, the float $\underline{\delta}_\odot(w)$ is a possibly refined lower bound for x.

C. Filtering by Maximum ULP on Addition/Subtraction

In this section we introduce the functions $\underline{\delta}_\oplus$ and $\overline{\delta}_\oplus$. The functions $\underline{\delta}_\ominus$ and $\overline{\delta}_\ominus$ can be deduced by symmetry, as explained in Section III-B and [19]. Using the Michel and Marre property (Proposition 3.1) we formally define the function $\overline{\delta}_\oplus$ as follows.

Definition 4.1: Let us define $\mathcal{F}_\oplus = \mathcal{F}_{p,e_{\max}}$, $\overline{\mathcal{F}}_\oplus = \mathcal{F}^+_{p,\infty}$, and let $z \in \mathcal{F}_\oplus$ be such that $|z| = 1.b_2 \cdots b_i 0 \cdots 0 \times 2^{e_z}$, with $b_i = 1$. Similarly to Proposition 3.1, let $k = p - i$, $\alpha = 1.1 \cdots 1 \times 2^{e_z+k}$ and $\beta = \alpha \oplus z$. Then $\overline{\delta}_\oplus \colon \mathcal{F}_\oplus \to \overline{\mathcal{F}}_\oplus$ is defined, for each $z \in \mathcal{F}_\oplus$, as follows:

$$\overline{\delta}_\oplus(z) = \begin{cases} +\infty, & \text{if } z = -\infty \text{ or } z = +\infty; \\ \alpha, & \text{if } -\infty < z < 0; \\ +0, & \text{if } z = -0 \text{ or } z = +0; \\ \beta, & \text{if } 0 < z < +\infty. \end{cases}$$

Theorem 4.1: $\overline{\delta}_\oplus$ is well-defined and satisfies (1) and (2).

The function $\underline{\delta}_\oplus \colon \mathcal{F}_\oplus \to \overline{\mathcal{F}}_\oplus$ is defined dually: for each $z \in \mathcal{F}_\oplus \setminus \{-0, +0, +\infty\}$, $\underline{\delta}_\oplus(z) = -\overline{\delta}_\oplus(-z)$. It is easy to see that properties (1) and (2) of $\overline{\delta}_\oplus$ entail properties (3) and (4) of $\underline{\delta}_\oplus$.

We now need algorithms to maximize $\overline{\delta}_\oplus$ and minimize $\underline{\delta}_\oplus$ over an interval of floating-point values. Since the two problems are dual to each other, we will focus on the maximization of $\overline{\delta}_\oplus$. As $\overline{\delta}_\oplus$ is not monotonic, a nontrivial analysis of its range over an interval is required. When the interval contains only finite, nonzero and positive (resp., negative) values, the range of $\overline{\delta}_\oplus$ has a simple shape. We are thus brought to consider an interval $[\underline{\mathrm{z}}, \overline{\mathrm{z}}]$ such that $\underline{\mathrm{z}} \notin \{-\infty, -0, +0\}$ and $\overline{\mathrm{z}} \notin \{-0, +0, +\infty\}$ have the same sign. We will now revisit and correct the algorithm proposed by Michel and Marre in [19] to maximize $\overline{\delta}_\oplus$ over $[\underline{\mathrm{z}}, \overline{\mathrm{z}}]$.

The idea presented in [19] is the following. When dealing with an interval $[\underline{\mathrm{z}}, \overline{\mathrm{z}}]$ with $\underline{\mathrm{z}} > 0$, α (and thus β and our $\overline{\delta}_\oplus$) grows with the exponent and the number of successive 0 bits to the right of the mantissa, i.e., k in Proposition 3.1 and in Definition 4.1. Thus, maximizing these two criteria allows to maximize α over the interval.

Definition 4.2: Let z be a variable over $\mathcal{F}_{p,e_{\max}}$ such that $\underline{\mathrm{z}} \notin \{-\infty, -0, +0\}$ and $\overline{\mathrm{z}} \notin \{-0, +0, +\infty\}$ have the same sign and $\underline{\mathrm{z}} < \overline{\mathrm{z}}$. Then $\mu_\oplus(\mathrm{z}) \in \mathcal{F}_{p,e_{\max}}$ is given by $1.0 \cdots 0 \times 2^{e_{\overline{\mathrm{z}}}}$, if $e_{\underline{\mathrm{z}}} \neq e_{\overline{\mathrm{z}}}$; otherwise we define $\mu_\oplus(\mathrm{z}) = 1.b_2 \cdots b_i a 0 \cdots \times 2^{e_{\overline{\mathrm{z}}}}$, where, for some $b_{i+1} \neq b'_{i+1}$,

$$
\begin{aligned}
|\underline{\mathrm{z}}| &= 1.b_2 \cdots b_i b_{i+1} \cdots \times 2^{e_{\overline{\mathrm{z}}}};\\
|\overline{\mathrm{z}}| &= 1.b_2 \cdots b_i b'_{i+1} \cdots \times 2^{e_{\overline{\mathrm{z}}}};\\
a &= \begin{cases} 0, & \text{if } 1.b_2 \cdots b_i 0 \cdots 0 \times 2^{e_{\overline{\mathrm{z}}}} = |\underline{\mathrm{z}}|;\\ 1, & \text{otherwise.} \end{cases}
\end{aligned}
$$

Theorem 4.2: Let z be as in Definition 4.2. Then, for each $z \in [\underline{\mathrm{z}}, \overline{\mathrm{z}}]$, $\overline{\delta}_\oplus(z) \leq \overline{\delta}_\oplus\big(\mu_\oplus(\mathrm{z})\big)$.

As we have already pointed out, the algorithm of Definition 4.2 is very similar to the algorithm presented in [19]. There is an importance difference, though: in the case when $\underline{\mathrm{z}} = 1.b_2 \cdots b_i b_{i+1} 0 \cdots 0 \times 2^{e_{\overline{\mathrm{z}}}}$, $\overline{\mathrm{z}} = 1.b_2 \cdots b_i b'_{i+1} \cdots \times 2^{e_{\overline{\mathrm{z}}}}$ and $b_j = 1$, for some $j \leq i$. In this case, the algorithm of [19] returns $1.b_2 \cdots b_i 1 0 \cdots 0 \times 2^{e_{\overline{\mathrm{z}}}}$. Note, however, that the value that maximizes α is $\underline{\mathrm{z}}$, which is different from $1.b_2 \cdots b_i 1 0 \cdots 0 \times 2^{e_{\overline{\mathrm{z}}}}$.

Definition 4.2 can be extended to intervals that include subnormals [3], but not to intervals containing zeroes. So, when z's interval contains zeroes, only the classical filtering is applied. For efficiency reasons, filtering by maximum ULP is only applied when $\overline{\delta}_\oplus\big(\mu_\oplus(\mathrm{z})\big) \leq f_{\max}$ so as to avoid the use of wider floating-point formats.

Example 4.1: Consider $\mathrm{z} = \mathrm{x} \oplus \mathrm{y}$ with $\mathrm{z} \in [1.0, 2.0]$, $\mathrm{x} \in [-1.0 \times 2^{50}, 1.0 \times 2^{50}]$ and $\mathrm{y} \in [-1.0 \times 2^{30}, 1.0 \times 2^{30}]$. With classical filtering we obtain $\mathrm{x}, \mathrm{y} \in [-1.0 \times 2^{30}, 1.0 \times 2^{30}]$, whereas with filtering by maximum ULP we obtain the much tighter interval $\mathrm{x}, \mathrm{y} \in [-1.1 \cdots 1 \times 2^{24}, 1.0 \times 2^{25}]$.

This example shows that filtering by maximum ULP can be stronger than classical interval-consistency based filtering. However, there are trivial examples that show the opposite phenomenon so that classical and maximum ULP are orthogonal: both should be applied, through interval intersection, in order to obtain optimal results.

D. Filtering by Maximum ULP on Multiplication

Let $z \in \mathcal{F}_{p,e_{\max}}$ be a strictly positive constant such that $z = x \otimes y$, where $x, y \in \mathcal{F}_{p,e_{\max}}$ are unknown. As for Property 3.1, there exists a greatest float $x_m \in \mathcal{F}_{p,e_{\max}}$ such that there exists $y \in \mathcal{F}_{p,e_{\max}}$ satisfying $z = x_m \otimes y$. More precisely, x_m is the float such that $z = x_m \otimes f_{\min}^{\mathrm{nor}}$. Such a float always exists because multiplication by $f_{\min}^{\mathrm{nor}} = 2^{e_{\min}}$ is equivalent to an exponent shifting. Now let us consider $x' \in \mathcal{F}_{p,e_{\max}}$ such that $x' > x_m$. By monotonicity of $\otimes$, $z < x' \otimes f_{\min}^{\mathrm{nor}}$ and there is no other float $y \neq f_{\min}^{\mathrm{nor}}$ such that $z = x' \otimes y$. In fact, by monotonicity, such float y should be smaller than $f_{\min}^{\mathrm{nor}}$. On the other hand, y must be greater than $+0$ for otherwise $x' \otimes y$ would not be strictly positive. However, for no $y \in \mathcal{F}_{p,e_{\max}}$ we have $+0 < y < f_{\min}^{\mathrm{nor}}$. Therefore, the value x_m such that $z = x_m \otimes f_{\min}^{\mathrm{nor}}$ is the greatest value for x that can satisfy $z = x_m \otimes y$ for some y.

Definition 4.3: $\mathcal{F}_\otimes = \left\{ z \in \mathcal{F}_{p,e_{\max}} \,\middle|\, f_{\min}^{\mathrm{nor}} \cdot |z| \leq f_{\max} \right\}$ and $\overline{\mathcal{F}}_\otimes = \mathcal{F}^+_{p,e_{\max}}$ are the domain and codomain of $\overline{\delta}_\otimes \colon \mathcal{F}_\otimes \to \overline{\mathcal{F}}_\otimes$, defined for each $z \in \mathcal{F}_\otimes$ as follows:

$$\overline{\delta}_\otimes(z) = |z| \cdot 2^{-e_{\min}}$$

Theorem 4.3: Function $\overline{\delta}_\otimes$ is well-defined and satisfies (1) and (2).

The function $\underline{\delta}_\otimes$ is simply defined as $\underline{\delta}_\otimes = -\overline{\delta}_\otimes(z)$. Moreover, Definition 4.3 and Theorem 4.3 can be extended to intervals that include subnormals replacing all occurrences of $f_{\min}^{\mathrm{nor}}$ by $f_{\min}$ [3].

The value $M \in [\underline{\mathrm{z}}, \overline{\mathrm{z}}]$ that maximizes $\overline{\delta}_\otimes$ is the one with the greatest absolute value, i.e., $M = \max\big\{|\underline{\mathrm{z}}|, |\overline{\mathrm{z}}|\big\}$. Since $\underline{\delta}_\otimes$ is defined as $-\overline{\delta}_\otimes(z)$, the value that minimizes $\underline{\delta}_\otimes$ is again M. Hence, if $[\underline{\mathrm{z}}, \overline{\mathrm{z}}]$ does not contain zeroes, $\overline{\delta}_\otimes(M)$ (resp., $\underline{\delta}_\otimes(M)$) is an upper bound (resp., a lower bound) of x w.r.t. the constraint $\mathrm{z} = \mathrm{x} \otimes \mathrm{y}$. The restriction to intervals not containing zeroes is justified by the fact that, e.g., if $z = 0$ then $z = x \otimes y$ holds with $x = f_{\max}$ and $y = 0$, hence no useful filtering can be applied to x.

As the product is commutative, the function of Definition 4.3 can be used for filtering y as well. Note that this filtering can only be applied when $\max\big\{|\underline{\mathrm{z}}|, |\overline{\mathrm{z}}|\big\} \in \mathcal{F}_\otimes$.

Example 4.2: Consider the IEEE 754 single-precision constraint $\mathrm{z} = \mathrm{x} \otimes \mathrm{y}$ with $\mathrm{z} \in [1.0 \times 2^{-50}, 1.0 \times 2^{-30}]$ and $\mathrm{x}, \mathrm{y} \in [-\infty, +\infty]$. We have

$$
\begin{aligned}
\overline{\delta}_\otimes(1.0 \times 2^{-30}) &= 1.0 \times 2^{-30} \cdot 2^{-(-126)} = 1.0 \times 2^{96},\\
\underline{\delta}_\otimes(1.0 \times 2^{-30}) &= -1.0 \times 2^{-30} \cdot 2^{-(-126)} = -1.0 \times 2^{96},
\end{aligned}
$$

so, while classical filtering does not prune the intervals for x and y, filtering by maximum ULP yields the refined intervals $\mathrm{x}, \mathrm{y} \in [-1.0 \cdots 0 \times 2^{96}, 1.0 \cdots 0 \times 2^{96}]$.

E. Filtering by Maximum ULP on Division

On the $\mathcal{F}_{p,e_{\max}}$ domain, a role similar to the one of $f_{\min}^{\mathrm{nor}}$ in the definition of filtering by ULP max on multiplication is played by $f_{\max}$ in the definition of filtering by ULP max on division.

Definition 4.4: Let

$$\mathcal{F}_{\oslash} = \left\{ z \in \mathcal{F}_{p,e_{\max}} \mid |z| \otimes f_{\max} \leq f_{\max} \right\}$$

and $\overline{\mathcal{F}}_{\oslash} = \mathcal{F}^{+}_{p,e_{\max}}$. Then $\overline{\delta}_{\oslash} \colon \mathcal{F}_{\oslash} \to \overline{\mathcal{F}}_{\oslash}$ is defined, for each $z \in \mathcal{F}_{\oslash}$, by $\overline{\delta}_{\oslash}(z) = |z| \otimes f_{\max}$.

Theorem 4.4: Function $\overline{\delta}_{\oslash}$ is well-defined and satisfies (1) and (2).

A similar result can be obtained for intervals that include subnormals by a suitable modification of Definition 4.4 [3].

The function $\underline{\delta}_{\oslash}$ is simply defined, for each $z \in \mathcal{F}_{\oslash}$, by $\underline{\delta}_{\oslash} = -\overline{\delta}_{\oslash}(z)$.

The value $M \in [\underline{z}, \overline{z}]$ that maximizes $\overline{\delta}_{\oslash}$ is the one that has the greatest absolute value, i.e., $M = \max\bigl\{|\underline{z}|, |\overline{z}|\bigr\}$. Since $\underline{\delta}_{\oslash}$ is defined as $-\overline{\delta}_{\oslash}(z)$, M is also the value that minimizes $\underline{\delta}_{\oslash}$. Hence, if $[\underline{z}, \overline{z}]$ does not contain zeroes, $\overline{\delta}_{\oslash}(M)$ (resp., $\underline{\delta}_{\oslash}(M)$) is an upper bound (resp. a lower bound) of x w.r.t. the constraint $\mathrm{z} = \mathrm{x} \oslash \mathrm{y}$. Once again, the restriction to intervals not containing zeroes is justified by the fact that, e.g., if $z = 0$ then $z = x \oslash y$ holds with $x = f_{\max}$ and $y = \infty$, hence, also in this case, no useful filtering can be applied to x. Note that this filtering can only be applied when $\max\bigl\{|\underline{z}|, |\overline{z}|\bigr\} \in \mathcal{F}_{\oslash}$.

Example 4.3: Consider the IEEE 754 single-precision constraint $\mathrm{z} = \mathrm{x} \oslash \mathrm{y}$ with $\mathrm{z} \in [-1.0 \times 2^{-110}, -1.0 \times 2^{-121}]$ and $\mathrm{x}, \mathrm{y} \in [-\infty, +\infty]$. We have

$$\begin{aligned}
\overline{\delta}_{\oslash}(1.0 \times 2^{-110}) &= 1.0 \times 2^{-110} \cdot 1.1 \cdots 1 \times 2^{127} \\
&= 1.1 \cdots 1 \times 2^{17}, \\
\underline{\delta}_{\oslash}(1.0 \times 2^{-110}) &= -1.0 \times 2^{-110} \cdot 1.1 \cdots 1 \times 2^{127} \\
&= -1.1 \cdots 1 \times 2^{17}.
\end{aligned}$$

Again, filtering by maximum ULP improves upon classical filtering with $\mathrm{x} \in [-1.1 \ldots 1 \times 2^{17}, 1.1 \ldots 1 \times 2^{17}]$.

F. Synthesis

Table I provides a compact presentation of filtering by maximum ULP under the assumption $e_{\min} = 1 - e_{\max}$, where the required functions can be summarized as follows:

$$\begin{aligned}
\overline{\delta}_{\oplus}(z) &= \begin{cases} \beta, & \text{if } 0 < z < +\infty, \\ \alpha, & \text{if } -\infty < z < 0; \end{cases} & \underline{\delta}_{\oplus}(z) &= -\overline{\delta}_{\oplus}(-z); \\
\overline{\delta}_{\otimes}(z) &= |z| \cdot 2^{-e_{\min}}; & \underline{\delta}_{\otimes}(z) &= -\overline{\delta}_{\otimes}(z); \\
\overline{\delta}_{\oslash}(z) &= |z| \otimes f_{\max}; & \underline{\delta}_{\oslash}(z) &= -\overline{\delta}_{\oslash}(z).
\end{aligned}$$

V. Implementation in FPSE

A. FPSE

FPSE [7] is a constraint solver based on interval consistency filtering dedicated to the analysis of IEEE 754 floating-point computations coming from C programs. The tool takes a *path condition* as input, which is a quantifier-free conjunction of constraints extracted from a path of a C function. Constraints hold over the input variables of the program, including global variables, as well as temporary variables introduced by classical compiler code transformations. For a given path condition, FPSE can either return the first solution found or show that there is no solution. In the former case, the result can be interpreted as a test data that activates the selected path; in the latter case, infeasibility of the path is proved. Of course, solving the constraints in reasonable time is not always possible since the search space can be huge.

The constraints in FPSE are based on expressions built over $\oplus$, $\ominus$, $\otimes$, $\oslash$ and the relations $=$, $\neq$, $<$, $\leq$. Interval constraints (e.g., $x \in [a, b]$) are allowed as well as IEEE 754 type casting constraints, namely float-to-double, double-to-float, long-to-double and double-to-long. FPSE works under some hypotheses that are now summarized. The tool deals only with the *near tail-to-even* rounding mode, which is used by default in almost all C implementations and is the most difficult to handle in constraint solving over floating-point variables [21]. To model floating-point computations, special attention is paid to conform to the actual execution of programs. In order to capture the semantics of the program, it is of course necessary to respect the precedence of expression operators as specified by the C language as well as the evaluation order realized by the language implementation at hand. FPSE respects the shape of expressions as represented in the abstract syntax tree of the program without any rearrangement or simplification. The order in which operands are evaluated by a C implementation can be matched by using a preprocessor like CIL [25].

Any symbolic expression is decomposed into a sequence of TAC assignments where fresh temporary variables are introduced bearing in mind that the order of evaluation must be preserved.[5] This decomposition requires that intermediate results of an operation conform to the type of storage of its operands.[6] Constraint solving is implemented by using interval consistency combined with search heuristics. Several heuristics with static and dynamic choice of variable have been considered. FPSE is implemented with about 10 KLOC of SICStus Prolog (for the high-level constraint-solving machinery) and C (for the projection functions).

[5] The introduction of temporary variables does not change the semantics of floating-point computations as long as it reflects the behavior of the compiler and of the floating-point unit.

[6] This is not always true: e.g., on Intel's architectures based on the 387 floating-point coprocessor, registers have more precision than the IEEE 754 `float` and `double` types; this makes rounding unpredictable. Luckily, SSE instruction sets, which do not pose this problem, are superseding 387.

Table I
FILTERING BY MAXIMUM ULP SYNOPSIS

Constraint	$\mathtt{x} \subseteq \cdot$	$\mathtt{y} \subseteq \cdot$	Condition(s)
$\mathtt{z} = \mathtt{x} \oplus \mathtt{y},\ 0 < \mathtt{z} \leq f_{\max}$	$[\ \underline{\delta}_{\oplus}(\zeta),\ \ \overline{\delta}_{\oplus}(\zeta)]$	$[\ \underline{\delta}_{\oplus}(\zeta),\ \ \overline{\delta}_{\oplus}(\zeta)]$	$\zeta = \mu_{\oplus}(\mathtt{z}),\ \ -f_{\max} \leq \underline{\delta}_{\oplus}(\zeta),\ \ \overline{\delta}_{\oplus}(\zeta) \leq f_{\max}$
$\mathtt{z} = \mathtt{x} \oplus \mathtt{y},\ -f_{\max} \leq \mathtt{z} < 0$	$[-\overline{\delta}_{\oplus}(\zeta'), -\underline{\delta}_{\oplus}(\zeta')]$	$[-\overline{\delta}_{\oplus}(\zeta'), -\underline{\delta}_{\oplus}(\zeta')]$	$\zeta' = \mu_{\oplus}(-\mathtt{z}),\ \ -f_{\max} \leq \underline{\delta}_{\oplus}(\zeta'),\ \ \overline{\delta}_{\oplus}(\zeta') \leq f_{\max}$
$\mathtt{z} = \mathtt{x} \ominus \mathtt{y},\ 0 < \mathtt{z} \leq f_{\max}$	$[\ \underline{\delta}_{\oplus}(\zeta),\ \ \overline{\delta}_{\oplus}(\zeta)]$	$[-\overline{\delta}_{\oplus}(\zeta), -\underline{\delta}_{\oplus}(\zeta)]$	$\zeta = \mu_{\oplus}(\mathtt{z}),\ \ -f_{\max} \leq \underline{\delta}_{\oplus}(\zeta),\ \ \overline{\delta}_{\oplus}(\zeta) \leq f_{\max}$
$\mathtt{z} = \mathtt{x} \ominus \mathtt{y},\ -f_{\max} \leq \mathtt{z} < 0$	$[-\overline{\delta}_{\oplus}(\zeta'), -\underline{\delta}_{\oplus}(\zeta')]$	$[\ \underline{\delta}_{\oplus}(\zeta'),\ \ \overline{\delta}_{\oplus}(\zeta')]$	$\zeta' = \mu_{\oplus}(-\mathtt{z}),\ \ -f_{\max} \leq \underline{\delta}_{\oplus}(\zeta'),\ \ \overline{\delta}_{\oplus}(\zeta') \leq f_{\max}$
$\mathtt{z} = \mathtt{x} \otimes \mathtt{y},\ 0 < \lvert z \rvert \leq 2(2 - 2^{1-p})$	$[\ \underline{\delta}_{\otimes}(m),\ \ \overline{\delta}_{\otimes}(m)]$	$[\ \underline{\delta}_{\otimes}(m),\ \ \overline{\delta}_{\otimes}(m)]$	$m = \max\{\lvert\underline{\mathtt{z}}\rvert, \lvert\overline{\mathtt{z}}\rvert\},$
$\mathtt{z} = \mathtt{x} \oslash \mathtt{y},\ 0 < \lvert\mathtt{z}\rvert \leq 1$	$[\ \underline{\delta}_{\oslash}(m),\ \ \overline{\delta}_{\oslash}(m)]$		$m = \max\{\lvert\underline{\mathtt{z}}\rvert, \lvert\overline{\mathtt{z}}\rvert\}$

B. *Relative ϵ*

Slow convergence phenomena typically arise when few values are continuously pruned in a constraint propagation cycle. For example, the constraint $\mathtt{x} < \mathtt{y} \land \mathtt{y} \leq \mathtt{x}$ causes a slow convergence phenomenon over floating-point (and, for that matter, also integer) variables. Each time a projection function is woken, a single float is pruned from the domain of x and y. Unsatisfiability is ultimately proved, but not in reasonable time. To avoid slow convergence phenomena, we implemented a procedure that stops the filtering under a given threshold called *relative* ϵ. For a given floating-point variable, if a filtering function does not reduce its domain of more than $\epsilon\%$, then we withdraw that filtering function and do not prune the domain. An important difference with respect to interval-propagation-based constraint solvers over continuous domains is that we differentiate the treatment of direct and indirect projection functions. For direct projection functions this threshold is positioned at 0%, while for indirect projection functions it is positioned at 10% (a value that was experimentally determined to provide a good compromise). The idea is to benefit from the structure of the problem. When all the input variables of the program under test are instantiated, direct projection functions are sufficient to get a solution: thus applying them unconditionally is advantageous. In contrast, when a local or an output variable is instantiated first, this may lead to a propagation cycle that has to be cut very early: we discovered that cutting it using the relative ϵ on indirect projection functions is very effective as every potential propagation cycle involves at least one indirect projection function. For example, in $\mathtt{x} < \mathtt{y} \land \mathtt{y} \leq \mathtt{x}$ only indirect projection functions are involved on both variables, and propagation cycles do not occur. Of course, in this pathological case there is no propagation cycle but the system is still partially consistent at the end of the initial filtering: hopefully other constraints will allow to prune many values from the domains of x and y so that enumeration will not have to try all the values to prove unsatisfiability.

VI. EXPERIMENTAL EVALUATION

The aim of our experimental study was to evaluate filtering by maximum ULP (in brief *ULP Max*) and to determine whether it is an effective, practical property for solving constraints over the floats with an acceptable overhead. For presentation of the results in this paper we selected two C functions performing intensive floating-point computations. The first one is a small C function that computes a root of a polynomial equation within a given range: `dichotomic()` in Figure 2. Its computations are dominated by single-precision floating-point computations. The second program is a real-world program embedded on unmanned airplanes to avoid fly-to-fly collision.

We implemented several search heuristics with static and dynamic variable orderings. For the choice of values, we implemented a domain-splitting strategy adapted to floating-point variables that proved to be very effective. For a given variable, our strategy selects first the floating-point value v that separates the domain of x in two equivalently-sized sub-domains, i.e., two domains containing the same number of floats; then it considers $x = v$, $x < v$ and $x > v$ by successively backtracking on these choices.

For the `dichotomic()` function we selected at random a path that iterates 12 times in the loop and considered all its path prefixes from iteration 1 to 12. For each path, we used FPSE to automatically generate a test input (an instantiation of all the input variables) that covers the path. We considered two versions of FPSE: a version that implements *ULP Max* filtering as defined in this paper, and a version without that. We measured: the number of elementary (ternary) constraints on the path (*NbC*); the number of uninstantiated floating-point variables on the path and the number of variables involved in the solution path of the search tree (*NbV*); the number of times *ULP Max* filtering takes place, globally and on the solution path (*NbE*); the number of floats pruned by *ULP Max*, in millions, globally and on the solution path (*NbD*); the percentage of domains pruned by *ULP Max* over all the variables involved in the solution path (%); the CPU time for generating test inputs with the standard version of FPSE (*w/o*) and with FPSE augmented

Table II
EXPERIMENTAL RESULTS FOR `dichotomic()` (TIMEOUT = 30 MIN)

#	NbC	NbV	Global results		On the solution path				ULP Max		Speedup
			NbE	NbD	NbV	NbE	NbD	%	w/o	w/	factor
1	17	12	62	17,515	12	1	864	20.2	0.142	0.080	1.775
2	31	22	3,948	484,128	22	0	0	0.00	12.326	3.536	3.486
3	45	32	461	102,522	32	3	1,174	9.15	3.969	0.872	4.552
4	59	42	544,377	9,208,097	42	0	0	0.00	timeout	847.778	∞
5	73	52	510	158,716	52	5	1,895	8.86	2.370	1.506	1.574
6	87	62	799	209,621	62	0	0	0.00	timeout	2.050	∞
7	101	72	494	87,934	72	7	2,625	8.77	6.087	0.983	6.192
8	115	82	timeout	timeout	timeout	timeout	timeout	0.00	timeout	timeout	∞
9	129	92	258	83,166	92	9	3,338	8.67	2.352	0.978	2.405
10	143	102	637	157,421	102	0	0	0.00	timeout	2.482	∞
11	157	112	224	73,702	112	11	4,034	8.57	2.471	0.724	3.413
12	171	122	635	153,318	122	0	0	0.00	4.924	2.642	1.864

```
float f(float x) { return x*x - 2.0F; }
float dichotomic(float xL, float xR) {
   float xM = 1.0F;
   while ((xR - xL) > 0.0001F) {
      xM = (xR + xL) / 2.0F;
      if ((f(xL) * f(xM)) > -1.0F) { xL = xM; }
      else { xR = xM; } }
   return xM; }
```

Figure 2. The `dichotomic()` function

with *ULP Max* (*w/*); the ratio between *w/o* and *w/* (*Speedup factor*).

For CPU time, we took the average of 10 runs of the same test input generation process. All results were computed on a system equipped with an Intel Core 2 Duo 3.00 GHz and running Linux 2.6 with 4 GB of RAM.

The results for `dichotomic()`, are presented in Table II. These show that FPSE with *ULP Max* is effective enough to solve at least 3 constraint systems that the standard version cannot solve. Moreover, *ULP Max* is able to prune the domains of floating-variables in all the cases and CPU time gains are due to this extra pruning power. On 6 paths, FPSE with *ULP Max* was able to prune between 8% and 20% of the variable domains.

The second example is a real-world example extracted from a critical embedded system. The function `tcas_periodic_task_1Hz()`, an excerpt of which is presented in Figure 3, is the core of a TCAS system (Traffic Collision Avoidance System) embedded into unmanned aircrafts.[7] The system receives the speed and direction of other aircrafts and, based on floating-point computations, it modifies the speed and direction of the host aircraft in order to avoid collisions. This program is interesting because determining the feasibility of its paths is hard and requires precise reasoning on non-linear floating-point computations.

For `tcas_periodic_task_1Hz()`, we selected all the possible paths up to 5 iterations of the main loop. This corresponds to about 130 paths among which 51% were

[7]The complete source code is available at http://paparazzi.enac.fr

```
void tcas_periodic_task_1Hz(void) {
  ...
  for (i = 2; i < NB_ACS; i++) {
    uint32_t dt = gps_itow - acs[i].itow;
    ...
    float dx = acs[i].east - estimator_x;
    float dy = acs[i].north - estimator_y;
    float dz = acs[i].alt - estimator_z;
    float dvx
      = vx - acs[i].gspeed * sinf(acs[i].course);
    float dvy
      = vy - acs[i].gspeed * cosf(acs[i].course);
    float dvz = estimator_z_dot - acs[i].climb;
    float scal = dvx*dx + dvy*dy + dvz*dz;
    float ddh = dx*dx + dy*dy;
    float ddv = dz*dz;
    float tau = TCAS_HUGE_TAU;
    ...
    switch (tcas_acs_status[i].status) {
    case TCAS_RA: ... break;
    case TCAS_TA: ... break;
    case TCAS_NO_ALARM: ... break;
} } }
```

Figure 3. An excerpt of `tcas.c`

shown to be infeasible by FPSE, regardless of whether *ULP Max* was used or not. For the remaining 49% of feasible paths, *ULP Max* has effects on 27 paths. We generated test inputs for all these paths with both versions of FPSE (i.e., w and w/o *ULP Max*). The results, which are reported in Table III, show that *ULP Max* always prunes the search space by more than 40 millions of single-precision floating-point values. In other words, *ULP Max* effectively prunes the search space in most cases. However, this extra-pruning does not always result in the overall speedup of the test input generation process: with reference to Table III, when the speedup is below 1 the overhead of computing *ULP Max* is not compensated by the gains it offers. On the other hand, it must be observed that those cases are not too frequent (9 cases out of 27 in this example) and the implementation of *ULP Max* in FPSE has much room for improvement.

Table III
EXPERIMENTAL RESULTS FOR `tcas_periodic_task_1Hz()`

#	NbC	NbV	Global results		On the solution path				ULP Max		Speedup factor
			NbE	NbD (M)	NbV	NbE	NbD (M)	%	w/o	w/	
1	157	191	5	765	191	1	11	0.28	1.200	1.212	0.99
2	152	191	1	45	191	1	45	1.07	3.261	3.313	0.98
3	152	191	1	45	191	1	45	1.07	3.688	3.715	0.99
4	152	191	4	753	191	0	0	0.00	0.039	0.032	1.22
5	152	191	4	753	191	0	0	0.00	0.041	0.037	1.11
6	157	191	4	955	191	0	0	0.00	0.060	0.048	1.25
7	157	191	4	955	191	0	0	0.00	0.071	0.078	0.91
8	157	191	25	1,884	191	20	1,884	2.20	0.046	0.046	1.00
9	157	191	25	1,884	191	20	1,884	2.20	0.369	0.382	0.97
10	157	191	25	1,884	191	20	1,884	2.20	0.068	0.068	1.00
11	157	191	25	1,884	191	20	1,884	2.20	0.706	0.698	1.01
12	152	191	25	1,884	191	20	1,884	2.20	0.029	0.027	1.05
13	152	191	25	1,884	191	20	1,884	2.20	0.027	0.029	0.93
14	157	191	3	387	191	1	10	0.24	0.076	0.030	2.53
15	157	191	3	395	191	0	0	0.00	0.081	0.039	0.93
16	157	191	1	43	191	1	43	1.01	0.071	0.076	0.93
17	157	191	3	387	191	1	10	0.24	0.074	0.032	2.31
18	157	191	3	395	191	0	0	0.00	0.083	0.040	2.08
19	157	191	1	43	191	1	43	1.01	0.075	0.076	0.99
20	152	191	1	43	191	1	43	1.01	0.079	0.079	1.00
21	152	191	1	43	191	1	43	1.01	0.075	0.075	1.00
22	152	191	8	521	191	6	144	0.56	0.077	0.033	2.33
23	152	191	8	521	191	6	144	0.56	0.077	0.033	2.33
24	157	191	2	477	191	0	0	0.00	0.079	0.031	2.55
25	157	191	2	477	191	0	0	0.00	0.074	0.032	2.31
26	152	191	1	43	191	1	43	1.01	0.075	0.077	0.97
27	152	191	1	43	191	1	43	1.01	0.078	0.077	1.01

VII. DISCUSSION

In this paper we brought to completion the plan anticipated in [8]. This is part of a long-term research effort concerning the correct, precise and efficient handling of floating-point constraints [4], [5], [7], [8], [19], [21], [22].

Other authors have considered using search-based test data generation with a specific notion of distance in their fitness function [16], [17]. For instance, search-based tools like AUSTIN and FloPSy can generate a test input for a specific path by evaluating the path covered by some current input with respect to a targeted path in the program. However, they cannot *solve* the constraints of path conditions, since: 1) they cannot determine unsatisfiability when the path is infeasible, and 2) they can fail to find a test input while the set of constraints is satisfiable.[8]

Recently, Borges et al. [6] combined a search-based test data generation engine with the RealPaver interval constraint solver, which is well-known in the Constraint Programming community. Even though FPSE and RealPaver are based on similar principles, their treatment of intervals is completely different. While FPSE preserves the solutions over the floats, RealPaver preserves the solutions over the reals by making the appropriate choices in the rounding modes used for computing the interval bounds. [7] contains small examples showing that an interval constraint solver over the reals can miss floating-point solutions to constraints over floating-point variables. However, as RealPaver can treat transcendental functions with high precision, the approach followed in [6] allows the generation of floating-point inputs for programs that use such functions in a nontrivial way, something that is outside the scope of this paper.

[8]The floating-point intensive programs shown in the previous section seem to be outside the reach of the search-based tool AUSTIN [16]: for example, AUSTIN seems to die pseudo-randomly on `dichotomic()`; both for `dichotomic()` and `tcas_periodic_task_1Hz()` we were not able to produce any test input data within 2 days of CPU time.

VIII. CONCLUSION

This paper concerns constraint solving over floating-point numbers and its application to automatic test data generation. Interval-based consistency techniques are very effective for the solution of such numerical constraints, provided precise and efficient filtering algorithms are available. We reformulated and corrected the filtering algorithm proposed by Marre and Michel in [19] for addition and subtraction. We proposed a uniform framework that generalizes the property identified by Marre and Michel to the case of multiplication and division. The main ideas of this article were roughly sketched in [8]: in this paper they have been revised, corrected and extended. The new filtering algorithms have been implemented in the FPSE system and the experimental evaluation show that they definitely improve the state-of-the-art of automatic test data generation for floating-point programs. Future work includes the exploration of other properties based on linearization of floating-point computations, such as those proposed in [4].

Acknowledgment

We are grateful to Abramo Bagnara (BUGSENG srl, Italy) for the many fruitful discussions we had on the subject of this paper, and to Paul Zimmermann (INRIA Lorraine, France) for the help he gave us proving a crucial result.

References

[1] P. E. Ammann and J. C. Knight. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers*, 37(4):418–425, 1988.

[2] A. Arcuri. Theoretical analysis of local search in software testing. In *Proc. of the 5th Int'l Symp. on Stochastic Algorithms: Foundations and Applications*, volume 5792 of *LNCS*, pages 156–168, 2009.

[3] R. Bagnara, M. Carlier, R. Gori, and A. Gotlieb. Filtering floating-point constraints by maximum ULP. Submitted for publication, 2013.

[4] M. S. Belaid, C. Michel, and M. Rueher. Boosting local consistency algorithms over floating-point numbers. In *Proc. of the 18th Int'l Conf. on Principles and Practice of Constraint Programming*, volume 7514 of *LNCS*, pages 127–140, 2012.

[5] B. Blanc, F. Bouquet, A. Gotlieb, B. Jeannet, T. Jeron, B. Legeard, B. Marre, C. Michel, and M. Rueher. The V3F project. In *Proc. of the 1st Workshop on Constraints in Software Testing, Verification and Analysis*, 2006.

[6] M. Borges, M. d'Amorim, S. Anand, D. Bushnell, and C. S. Pasareanu. Symbolic execution with interval solving and meta-heuristic search. In *Proc. of the 5th IEEE Int'l Conf. on Software Testing, Verification and Validation*, pages 111–120, 2012.

[7] B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability*, 16(2):97–121, 2006.

[8] M. Carlier and A. Gotlieb. Filtering by ULP maximum. In *Proc. of the 23rd IEEE Int'l Conf. on Tools with Artificial Intelligence*, pages 209–214, 2011.

[9] F. T. Chan, T. Y. Chen, S. C. Cheung, M. F. Lau, and S. M. Yiu. Application of metamorphic testing in numerical analysis. In *Proc. of the IASTED Int'l Conf. on Software Engineering*, pages 191–197, 1998.

[10] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. of the ACM SIGPLAN 2005 Conf. on Programming Language Design and Implementation*, pages 213–223, 2005.

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

[12] E. Goubault. Static analyses of the precision of floating-point operations. In *Static Analysis: 8th Int'l Symp., SAS 2001*, volume 2126 of *LNCS*, pages 234–259, 2001.

[13] The Institute of Electrical and Electronics Engineers, Inc. *IEEE Standard for Floating-Point Arithmetic*, IEEE Std 754-2008 (revision of IEEE Std 754-1985), August 2008.

[14] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[15] V. V. Kuliamin. Standardization and testing of mathematical functions. In *Perspectives of Systems Informatics*, volume 5947 of *LNCS*, pages 257–268, 2010.

[16] K. Lakhotia, M. Harman, and H. Gross. AUSTIN: A tool for search based software testing for the C language and its evaluation on deployed automotive systems. In *Proc. of the 2nd Int'l Symp. on Search Based Software Engineering*, pages 101–110, 2010.

[17] K. Lakhotia, N. Tillmann, M. Harman, and J. De Halleux. FloPSy: Search-based floating point constraint solving for symbolic execution. In *Proc. of the 22nd IFIP WG 6.1 Int'l Conference on Testing Software and Systems*, pages 142–157, 2010.

[18] B. Marre and B. Blanc. Test selection strategies for Lustre descriptions in GATeL. In *Proc. of the Workshop on Model Based Testing*, volume 111 of *Electronic Notes in Theoretical Computer Science*, pages 93–111, 2005.

[19] B. Marre and C. Michel. Improving the floating point addition and subtraction constraints. In *Proc. of the 16th Int'l Conf. on Principles and Practice of Constraint Programming*, volume 6308 of *LNCS*, pages 360–367, 2010.

[20] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[21] C. Michel. Exact projection functions for floating point number constraints. In *Proc. of the 7th Int'l Symp. on Artificial Intelligence and Mathematics*, 2002.

[22] C. Michel, M. Rueher, and Y. Lebbah. Solving constraints over floating-point numbers. In *Proc. of the 7th Int'l Conf. on Principles and Practice of Constraint Programming*, volume 2239 of *LNCS*, pages 524–538, 2001.

[23] W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223–226, 1976.

[24] J.-M. Muller. On the definition of $\mathrm{ulp}(x)$. Rapport de recherche 5504, INRIA, 2005.

[25] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction: Proc. of the 11th Int'l Conf.*, volume 2304 of *LNCS*, pages 213–228, 2002.

[26] R. Skeel. Roundoff error and the Patriot missile. *SIAM News*, 25(4):11, July 1992.

[27] E. Tang, E. T. Barr, X. Li, and Z. Su. Perturbing numerical calculations for statistical analysis of floating-point program (in)stability. In *Proc. of the 19th Int'l Symp. on Software Testing and Analysis*, pages 131–142, 2010.

[28] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.