Refining Abstract Interpretation-based Approximations with Constraint Solvers

Olivier Ponsini, Claude Michel, Michel Rueher

Refining Abstract Interpretation-based Approximations with Constraint Solvers

Olivier Ponsini, Claude Michel, and Michel Rueher

University of Nice–Sophia Antipolis, I3S/CNRS
BP 121, 06903 Sophia Antipolis Cedex, France
firstname.lastname@unice.fr

Abstract. Programs with floating-point computations are tricky to develop because floating-point arithmetic differs from real arithmetic and has many counterintuitive properties. A classical approach to verify such programs consists in estimating the precision of floating-point computations with respect to the same sequence of operations in an idealized semantics of real numbers. Tools like Fluctuat—based on abstract interpretation—have been designed to address this problem. However, such tools compute an over-approximation of the domains of the variables, both in the semantics of the floating-point numbers and in the semantics of the real numbers. This over-approximation can be very coarse on some programs. In this paper, we show that constraint solvers over floating-point numbers and real numbers can significantly refine the approximations computed by Fluctuat. We managed to reduce drastically the domains of variables of C programs that are difficult to handle for abstract interpretation techniques implemented in Fluctuat.

Key words: Program verification; Floating-point computation; C programs; Abstract interpretation-based approximation; Interval-based constraint solvers over real and floating-point numbers

1 Introduction

Numerous critical software applications rely on floating-point computations: in particular, simulation or control applications for physical systems in various domains as transportation, nuclear energy, medicine, or avionics and aerospace. These applications are often based on algorithms and mathematical models designed for real numbers. Floating-point numbers are then an additional possible source of errors and famous computer bugs are due to errors in the floating-point computations, e.g., the Ariane 5 rocket explosion or the Patriot missile failure. Indeed, for a same sequence of operations, floating-point numbers do not behave identically to real numbers. The finite nature of floating-point numbers has consequences that are counterintuitive with regards to real number arithmetic [12, 22]. For instance, some decimal real numbers are not representable (e.g., 0.1 has no exact representation with binary floating-point numbers), arithmetic operators are not associative and may be subject to phenomena such as absorption

(e.g., $a + b$ is rounded to a when a is far greater than b) or cancellation (subtraction of nearly equal operands after rounding that only keeps the rounding error). Despite the IEEE 754 standard, these intrinsic phenomena of floating-point arithmetic depend on various factors such as compilers, operating systems, or computer hardware architectures.

Although these phenomena are well known, ensuring that a program with floating-point computations reasonably approximates its original mathematical model on reals remains difficult. Formal verification methods have been developed to estimate the accuracy of floating-point computations and to show the absence of some runtime errors like arithmetic overflows, invalid operations, or division by zero. Existing automatic tools are mainly based on abstract interpretation techniques. In particular, the static analyzer Fluctuat [9] was successfully applied to study rounding error propagation in critical industrial C programs[1]. Fluctuat computes two over-approximations of the domains of the variables of a program respectively considered with a semantics on real numbers and with a semantics on floating-point numbers. Fluctuat is then able to bound the error made by floating-point computations. The computed approximations are precise enough for the analysis needs in many cases. However, some program constructs may lead to over-approximations so large that the analysis is not conclusive.

In this paper, we show that constraint solvers over floating-point numbers and real numbers can significantly refine the approximations computed by Fluctuat. The goal is to take advantage of the refutation capabilities of the filtering algorithms to reduce the domains computed by abstract interpretation. Implementation uses interval-based constraint solvers: RealPaver [15] which is a safe and correct solver for constraints over real numbers, and FPCS [20, 19] which is a safe and correct solver for constraints over floating-point numbers. We managed to reduce drastically the domains of variables of C programs that are difficult to handle for the abstract interpretation techniques implemented in Fluctuat.

Section 2 illustrates our approach with a small example and discusses related works. Section 3 details our approach and the tools it relies on: Fluctuat, RealPaver, and FPCS. In Sect. 4, we discuss the results of our approach on several representative programs.

2 Motivation

In this section, we illustrate our approach with a motivating example, then we discuss how our approach relates to existing works.

2.1 Example

The program in Fig. 1 is mentioned in [11] as a difficult program for abstract interpretation based analyses. On floating-point numbers, as well as on real

[1] Critical applications of interest for formal verification of floating-point computations are often embedded and predominantly written in C language.

Fig. 1. Abstract domain intersection.

```
1 /* Pre−condition :  x ∈ [0, 10] */
2 double conditional(double x) {
3   double y = x*x - x;
4   if (y >= 0)
5     y = x/10;
6   else
7     y = x*x + 2;
8   return y;
9 }
```

numbers, this function returns a value in the interval $[0, 3]$. From the conditional statement of line 4, we can derive the following information:

- `if` branch: $x = 0$ or $x \geq 1$, hence $y \in [0, 1]$;
- `else` branch: $x \in]0, 1[$, hence $y \in [2, 3]$.

However, classical abstract domains (e.g., intervals, polyhedra), as well as the abstract domain of *zonotopes* used in Fluctuat, fail to obtain a good approximation of this value. The best interval obtained with these abstractions is $[0, 102]$, both over the real and the floating-point numbers. The difficulty for these analyses is to intersect the abstract domains computed for y at lines 3 and 4. Actually, they are unable to derive from these statements any constraint on x. As a consequence, in the `else` branch, they still estimate that x ranges over $[0, 10]$.

In the approach we propose here, we compute an approximation of the domains in both execution paths. CSP filtering techniques are strong enough to reduce the domains of the variables for the generated constraints systems. Consider for instance the constraint system over the real numbers $\{y_0 = x_0 * x_0 - x_0, y_0 < 0, y_1 = x_0 * x_0 + 2, x_0 \in [0, 10]\}$ which corresponds to the execution path[2] through the `else` branch of the function `conditional`. From the constraints $y_0 = x_0 * x_0 - x_0$ and $y_0 < 0$, an interval solver can reduce the initial domain of x_0 to $[0, 1]$. This reduced domain is then used to compute the one of y_1 *via* the constraint $y_1 = x_0 * x_0 + 2$, which yields $y_1 \in [2, 3.001]$. Likewise, a constraint solver over the floating-point numbers will reduce x_0 to $[4.94 \times 10^{-324}, 1.026]$ and y_1 to $[2, 3.027]$.

To sum up, we build the constraint systems that correspond to each executable path in a function and use filtering techniques to reduce the domains of the variables computed by Fluctuat. Execution paths are explored on-the-fly and interrupted as soon as the inconsistency of the associated constraint system is detected. Over real numbers, we use the combination of hull and box consistencies implemented in RealPaver [15]. Over floating-point numbers, we use

[2] Statements are converted into SSA (Static Single Assignment) form where each variable is assigned exactly once on each program path.

Table 1. Return domain of the `conditional` function.

	Domain	Time
Exact real and floating-point domains	$[0, 3]$	n.a.
FLUCTUAT (real and floating-point domains)	$[0, 102]$	0.1 s
Constrained zonotopes (real domain)	$[0, 9.72]$	n.a.
FPCS (floating-point domain)	$[0, 3.027]$	0.2 s
REALPAVER (real domain)	$[0, 3.001]$	0.3 s

$3B$-consistency [18] as implemented in FPCS [20], which is correct over floating-point numbers. Table 1 collects the results obtained by the different techniques on the example of the function `conditional`. On this example, contrary to FLUCTUAT, our approach computes very good approximations of both floating-point and real domains. Analysis times are very similar (n.a. stands for not available). In [11], the authors proposed an extension to the zonotopes—named *constrained zonotopes*—which attempts to overcome the issue due to program conditional statements. This extension is defined for the real numbers and is not yet implemented in FLUCTUAT. The approximation computed with *constrained zonotopes* is better than the one of FLUCTUAT but remains still less precise than the one computed with REALPAVER.

2.2 Related works

Different methods address static validation of programs with floating-point computations: abstract interpretation based analyses, proofs of programs with proof assistants or with decision procedures in automatic solvers.

Analyses based on abstract interpretation represent rounding errors due to floating-point computation in their abstract domains. They are usually fast, automatic, and scalable. However, they may lack of precision and they do not generate any counter-example. ASTRE [7] is probably one of the most famous tool in this family of methods. The tool estimates the value of the program variables at every program point and can show the absence of runtime errors, that is the absence of behaviors not defined by the programming language (e.g., division by zero, arithmetic overflow). FLUCTUAT, which is detailed in Sect. 3.1, estimates in addition the accuracy of the floating-point computations, that is, a bound on the difference between the values taken by variables when the program is given a real semantics and when it is given a floating-point semantics [9].

A second group of methods endeavors to formalize floating-point arithmetic in proof assistants like Coq [2] or HOL [16]. Proofs of program properties are done manually in the proof assistant which guarantees proof correctness. These formalisms are not suitable for estimating the domains of program variables. Moreover, a property that cannot be proven is not necessarily false. Therefore, in these approaches, no counter-example can be generated. The Gappa tool [10] combines interval arithmetic and term rewriting from a base of theorems. The theorems rewrite arithmetic expressions so as to compensate for the shortcomings

of interval arithmetic, e.g., loss of dependency between variables. Whenever the computed intervals are not precise enough, theorems can be manually introduced or the input domains can be subdivided. The cost of this semi-automatic method is then considerable. In [1], the authors propose an axiomatization of floating-point arithmetic within first-order logic to automate the proofs conducted in proof assistants such as Coq by calling external SMT (Satisfiability Modulo Theories) solvers and Gappa. Their experiments show that human interaction with the proof assistant is still required. Because of the size of the domains of floating-point variables, the classical bit-vector approach of SAT solvers is ineffective. An abstraction technique was devised for CBMC in [4]. It is based on under and over-approximation of floating-point numbers with respect to a given precision expressed as a number of bits of the mantissa. However, this technique remains slow.

3 Proposed approach

The approach we propose here consists in refining the intervals computed by Fluctuat with constraint solvers over real and floating-point numbers. Before going into the details, we recall the characteristics of Fluctuat (Sect. 3.1), RealPaver (Sect. 3.2) and FPCS that are useful to understand the rest of the paper. (Sect. 3.3). The whole process we propose is described in Sect. 3.4.

3.1 Fluctuat

Fluctuat [9] is a static analyzer for C programs specialized in estimating the precision of floating-point computations. The tool compares the behavior of the analyzed program over real numbers and over floating-point numbers. In other words, Fluctuat allows to specify ranges of values for the program input variables and computes for each program variable:

- bounds for the domain of the variable considered as a real number;
- bounds for the domain of the variable considered as a floating-point number;
- bounds for the maximum error between real and floating-point values;
- the contribution of each statement to the error associated with the variable;
- the contribution of the input variables to the error associated with the variable.

Fluctuat proceeds by abstract interpretation. It uses the weakly relational abstract domain of zonotopes [13], which is a good trade-off between performance and precision. Zonotopes are sets of affine forms that improve over interval arithmetic: linear correlations between variables are preserved. To increase the analysis precision, the tool allows to use arbitrary precision numbers or to subdivide input variable intervals.

FLUCTUAT is developed by CEA-LIST[3] and was successfully used for industrial applications of several tens of thousands of lines of code in transportation, nuclear energy, or avionics areas.

3.2 RealPaver

REALPAVER[4] [15] is an interval solver for numerical constraint systems over the real numbers. Constraints can be non-linear and can contain the usual arithmetic operations and transcendental elementary functions. Not equal and strict inequality operators are not handled, neither is disjunction of constraints.

The solver computes reliable approximations of continuous solution sets using correctly rounded interval methods and constraint satisfaction techniques. More precisely, the computed domains are closed intervals bounded by floating-point numbers. REALPAVER implements several partial consistencies: box, hull, and $3B$ consistencies for instance. An approximation of a solution is described by a box, i.e., the Cartesian product of the domains of the variables. REALPAVER either proves the unsatisfiability of the constraint system or computes a union of boxes that contains all the solutions of the system.

3.3 FPCS

FPCS [20, 19] is a constraint solver that was designed to solve correctly, i.e., without losing any solution, a set of constraints over floating-point numbers. To this end, FPCS uses a $2B$-consistency [18] along with projection functions adapted to floating-point arithmetic [21, 3]. The main difficulty lies in computing inverse projection functions that preserve all the solutions. Indeed, if direct projections, i.e., computing the domain of y from the domain of x for a constraint like $y = f(x)$, only requires a slight adaptation of classical results on interval arithmetic, inverse projections, i.e., computing the domain of x from the one of y, do not follow the same rules because of the properties of floating-point arithmetic. All these results are described in [20] and extended in [3]. FPCS also implements stronger consistencies—e.g., kB-consistencies [18]—to deal with the classical issues of multiple occurrences and to reduce more substantially the bounds of the domains of the variables.

Contrary to most of the works mentioned in Sect. 2.2, the floating-point domains handled by FPCS are not limited to numerical values, but they also include infinities. Moreover, FPCS handles all the basic arithmetic operations, as well as most of the usual mathematical functions. Type conversions are also correctly processed.

The behavior of programs containing floating-point computations may vary with the programming language or the compiler used, but also, with the operating system or the hardware architecture on which the program is executed.

[3] FLUCTUAT web site: `http://www-list.cea.fr/labos/fr/LSL/fluctuat/index.html`

[4] REALPAVER web site: `http://pagesperso.lina.univ-nantes.fr/info/perso/permanents/granvil/realpaver/`

FPCS targets C programs, compiled with GCC without any optimization option and intended to be run on an x86 architecture managed by a 32-bit Linux operating system.

3.4 Process

We can now detail our approach. The steps of the process we propose are the following ones:

1. For a given C program, we compute with FLUCTUAT a first approximation of the domains of the variables over the real and over the floating-point numbers.
2. We parse the C program and build two constraint systems for each executable path (see details below): one with floating-point variables and one with real variables. We assign the domains estimated by FLUCTUAT to the program variables.
3. For each constraint system, we filter the domains with a partial consistency.
   - We use FPCS and a $3B(w)$-consistency on systems with floating-point variables.
   - We use REALPAVER and its $BC5$-consistency on systems with real variables. $BC5$-consistency is a combination of interval Newton method, hull-consistency and box-consistency.
4. For each output variable, we compute the union of all the domains that were obtained at step 3.

During step 2, we explore each execution path of a program, i.e., each path of the control flow graph, using a forward analysis (going from the beginning to the end of the program). Statements are converted into SSA (Static Single Assignment) form where each variable is assigned exactly once on each program path [8]. Lengths of the paths are bounded since recursive function calls are forbidden and loops are unfolded a user-defined number of times. Possible states of the program at any point of an execution path are represented by a constraint system made up of a finite set of variables and constraints over these variables. With FPCS, variables have domains that correspond to the implementation on machine of the types of the C language (`int`, `float` and `double`); with REALPAVER, domains are intervals over the reals. Rules define how each program statement modifies the possible program states by adding new constraints and variables.

Execution paths are explored on-the-fly and interrupted as soon as the inconsistency of the associated constraint system is detected: simple $3B$-consistency filtering with FPCS and hull $HC4$-consistency filtering with REALPAVER. This allows to limit the combinatorial explosion of the number of paths by only exploring those that are executable. This technique for representing programs by constraint systems was introduced for bounded verification of programs in CPBPV [5]. The implementation of the approach proposed in this paper relies on libraries developed for CPBPV.

RealPaver modeling language does not provide strict inequality and not equal operators, which can be found in program conditional expressions. As a consequence, in the constraint systems generated for RealPaver, strict inequalities are replaced by non strict ones and constraints with a not equal operator are ignored. This may lead to over-approximations, but this is safe since no solution is lost.

4 Experiments and discussion

In this section, we present the results of our approach on characteristic programs that show when our approach is able to improve the approximation computed by Fluctuat. Results are rounded to three decimal places for the sake of readability and were obtained on an Intel Core 2 Duo at 2.8 GHz with 4 Go memory running Linux. We used Fluctuat version 3.8.22_opt and RealPaver version 0.4. In the tables that follow, we denote real and floating-point domains with the symbols $\mathbb{R}$ and $\mathbb{F}$ respectively.

4.1 Conditionals

We have presented in Sect. 2.1 the issue of intersecting abstract domains in abstract interpretation based analyses. Here, we illustrate how our approach performs in this case with a program that computes the roots of a quadratic equation. This program is listed in Fig. 2 and was extracted from the GNU scientific library (GSL [24]). Roots are computed in variables `x0` and `x1`. The program calls two functions from the C library: `fabs` and `sqrt`, the absolute value and the square root function, respectively. The function `sqrt` is incidentally one of the few functions defined in the IEEE 754 standard. These functions are directly handled by FPCS and thus can appear in constraints as is. RealPaver has a predefined `sqrt` function, but `fabs(x)` was replaced by `max(x,-x)`.

Table 2 shows analysis times and the approximations of the domains of variables `x0` and `x1` obtained with two configurations of the domains of the input variables. The first two rows in the table present the results of Fluctuat and RealPaver over the reals. The next two rows present the results of Fluctuat and FPCS over the floating-point numbers.

In the first configuration, where $a \in [-1, 1]$, $b \in [0.5, 1]$ and $c \in [0, 2]$, the Fluctuat over-approximation is so large that it does not give any information on the domain of the roots, whereas our approach is able to drastically reduce these domains both over $\mathbb{R}$ and $\mathbb{F}$. Nevertheless, intersection of abstract domains does not always impact so significantly on the bounds of all domains. This is illustrated by the domain over $\mathbb{F}$ of `x0` in the second configuration where $a, b, c \in [1, 1 \times 10^6]$. Even though the domain computed by Fluctuat is still an over-approximation, our approach does not succeed in reducing it. In contrast, for this same configuration, our approach performs again a very good reduction of the domain of `x1`.

Fig. 2. Quadratic equation roots.

```
int quadratic(double a, double b, double c) {
  double r, sgnb, temp, r1, r2
  double disc = b * b - 4 * a * c;
  if (a == 0) {
    if (b == 0)
      return 0;
    else {
      x0 = -c / b;
      return 1;
    }
  }
  if (disc > 0) {
    if (b == 0) {
      r = fabs (0.5 * sqrt (disc) / a);
      x0 = -r;
      x1 =  r;
    } else {
      sgnb = (b > 0 ? 1 : -1);
      temp = -0.5 * (b + sgnb * sqrt (disc));
      r1 = temp / a;
      r2 = c / temp;
      if (r1 < r2) {
        x0 = r1;
        x1 = r2;
      } else {
        x0 = r2;
        x1 = r1;
      }
    }
    return 2;
  } else if (disc == 0) {
    x0 = -0.5 * b / a;
    x1 = -0.5 * b / a;
    return 2;
  } else
    return 0;
}
```

In order to increase the analysis precision, FLUCTUAT allows to divide the domains of at most two input variables into a user-defined number of sub-domains. Analyses are then run over each combination of sub-domains and the results are merged. Without any *a priori* knowledge of which sub-domains should be divided, all the combinations of one and, next, two domains should be tried if necessary for the required precision. The number of subdivisions of each domain is difficult to determine too: it must be the largest possible while maintaining an

Table 2. Domains of the roots of the `quadratic` function.

		conf. #1: $a \in [-1,1]$ $b \in [0.5,1]$ $c \in [0,2]$			conf. #2: $a,b,c \in [1, 1\times10^6]$		
		`x0`	`x1`	Time	`x0`	`x1`	Time
$\mathbb{R}$	FLUCTUAT	$[-\infty,\infty]$	$[-\infty,\infty]$	0.1 s	$[-2\times10^6,0]$	$[-1\times10^6,0]$	0.1 s
	REALPAVER	$[-\infty,0]$	$[-8.011,\infty]$	1.5 s	$[-1\times10^6,0]$	$[-5.186\times10^5,0]$	0.5 s
$\mathbb{F}$	FLUCTUAT	$[-\infty,\infty]$	$[-\infty,\infty]$	0.1 s	$[-2\times10^6,0]$	$[-1\times10^6,0]$	0.1 s
	FPCS	$[-\infty,0]$	$[-8.064,\infty]$	0.3 s	$[-2\,503.709,0]$		0.3 s

Table 3. Domains over $\mathbb{F}$ for the `quadratic` function with input domains subdivided.

	conf. #1		conf. #2	
	`x0`	Time	`x1`	Time
FLUCTUAT a subdivided	$[-\infty,-0]$	>1 s	$[-1\times10^6,0]$	>1 s
FLUCTUAT b subdivided	$[-\infty,\infty]$	>1 s	$[-5\times10^5,0]$	>1 s
FLUCTUAT c subdivided	$[-\infty,\infty]$	>1 s	$[-1\times10^6,0]$	>1 s
FLUCTUAT a & b subdivided	$[-\infty,-0]$	>10 s	$[-1.834\times10^5,0]$	>10 s
FLUCTUAT a & c subdivided	$[-\infty,-0]$	>10 s	$[-1\times10^6,0]$	>10 s
FLUCTUAT b & c subdivided	$[-\infty,\infty]$	>10 s	$[-5\times10^5,0]$	>10 s

acceptable analysis time, and this without guarantee of improving the precision of the analysis. In Tab. 3, we set the subdivisions to 50 when only one domain is divided; otherwise, we set them to 30 for each domain. We only report the results over $\mathbb{F}$ in the table. Over $\mathbb{R}$, in the first configuration, the subdivisions yield no improvement and, in the second configuration, the results are identical to those over $\mathbb{F}$. The cost in time of these subdivisions can be significant compared to the gain in precision:

- In the first configuration, subdivisions of the domain of a lead to a significant reduction of the domain of `x0` (identical to what is obtained with our approach). No subdivision combination could reduce the domain of `x1`.
- In the second configuration, the best reduction of the domain of `x1` is obtained by subdividing the domains of both a and b. The gain remains however quite small compared to the reduction performed by our approach. No subdivision combination could reduce the domain of `x0`.

Whenever it is necessary to subdivide all the input domains, the cost is prohibitive. Our approach turns out to be more efficient: it often improves the precision of the approximation, and its cost is low, whether the precision is improved or not. Moreover, our approach could also take advantage of the subdivision technique.

Fig. 3. 7^{th}-order Taylor series of function sinus.

```
double sinus(double x)
{ return x - x*x*x/6 + x*x*x*x*x/120 + x*x*x*x*x*x*x/5040; }
```

Fig. 4. Rump's polynomial.

```
double rump(double x, double y) {
  double f;
  f = 333.75*y*y*y*y*y*y;
  f = f + x*x*(11*x*x*y*y - y*y*y*y*y*y - 121*y*y*y*y - 2);
  f = f + 5.5*y*y*y*y*y*y*y*y;
  f = f + x / (2*y);
  return f;
}
```

4.2 Non-linearity

The abstract domain used by FLUCTUAT is based on affine forms which do not allow an exact representation of non-linear operations: the image of a zonotope by a non-linear function is not a zonotope in general. Non-linear operations are thus over-approximated in FLUCTUAT by introducing an error term. The constraint solvers we use handle the non-linear expressions better. This is illustrated on the program of the 7^{th}-order Taylor series of function sinus (see Fig. 3) where our approach improves significantly the approximation of FLUCTUAT (see Tab. 4, column `sinus`).

The constraint solvers we use handle most of the non-linear expressions well but they also use approximations. This is illustrated on the `rump` polynomial (see Fig. 4) extracted from [23]. This very particular polynomial was designed for showing a catastrophic cancellation phenomenon on some specific hardware architectures whatever the floating-point number precision is. Neither FPCS nor REALPAVER succeeded in reducing the domain computed by FLUCTUAT for the `rump` polynomial.

4.3 Loops

FLUCTUAT unfolds loops a bounded number of times before applying the widening operator of abstract interpretation (default is ten times). The widening operator allows to find a fixed point and terminate quickly. However, this operator may lead to very large over-approximations. This situation occurs in the analysis of the return value of `sqrt`, a program that computes an approximate value with an error of 1×10^{-2} of the square root of a number greater than 4. The algorithm of `sqrt` is based on the so-called Babylonian method (see Fig. 5).

Table 4. Domains of the return value of `sinus` and `rump` functions.

		`sinus` $x \in [-1, 1]$		`rump` $x \in [7 \times 10^4, 8 \times 10^4]$ $y \in [3 \times 10^4, 4 \times 10^4]$	
		Domain	Time	Domain	Time
$\mathbb{R}$	FLUCTUAT	$[-1.009, 1.009]$	0.1 s	$[-1.168 \times 10^{37}, 1.992 \times 10^{37}]$	0.1 s
	REALPAVER	$[-0.842, 0.843]$	0.3 s	$[-1.144 \times 10^{36}, 1.606 \times 10^{37}]$	1.2 s
$\mathbb{F}$	FLUCTUAT	$[-1.009, 1.009]$	0.1 s	$[-1.168 \times 10^{37}, 1.992 \times 10^{37}]$	0.1 s
	FPCS	$[-0.853, 0.852]$	0.2 s	$[-1.168 \times 10^{37}, 1.992 \times 10^{37}]$	0.2 s

Fig. 5. Square root function.

```
double sqrt(double x) {
  double xn, xn1;
  xn = x/2;
  xn1 = 0.5*(xn + x/xn);
  while (xn-xn1 > 1e-2) {
    xn = xn1;
    xn1 = 0.5*(xn + x/xn);
  }
  return xn1;
}
```

FLUCTUAT cannot perform any reduction of the return value of `sqrt` over $\mathbb{F}$ for the configuration $x \in [5, 10]$ (see Tab. 5).

In our approach, we do not try to analyze the behavior of loops: we just unfold the loops N times, where N is a user-defined parameter[5]. We only try to reduce the domains computed by FLUCTUAT if the entry conditions of the loops are false for a number of unfoldings k less than N.

Table 5 shows the results we obtain with $N = 10$. Unfolding can quickly become time-consuming, but the gain in precision can be significant too: over $\mathbb{F}$, in the configuration $x \in [5, 10]$, we compute for `sqrt` the domain $[2.232, 3.168]$ instead of the domain $[-\infty, \infty]$ obtained by FLUCTUAT. Note that REALPAVER does not terminate in a reasonable time if we use $[-\infty, \infty]$ as initial domain. REALPAVER is faster than FPCS because it uses a weaker consistency for pruning unreachable paths when unfolding the loop (i.e., hull-consistency *versus* $3B$-consistency).

4.4 Discussion

Abstract interpretation techniques compute approximations of variable domains over a relaxation of the initial problem. In the case of FLUCTUAT, sets of affine

[5] We can also use FLUCTUAT to estimate a bound on the number of necessary unfoldings [14].

Table 5. Domain of the return value of the `sqrt` function.

		conf. #1: $x \in [4.5, 5.5]$		conf. #2: $x \in [5, 10]$	
		Domain	Time	Domain	Time
$\mathbb{R}$	FLUCTUAT	$[2.116, 2.354]$	0.1 s	$[2.098, 3.435]$	0.1 s
	REALPAVER	$[2.121, 2.346]$	0.3 s	$[2.232, 3.165]$	0.5 s
$\mathbb{F}$	FLUCTUAT	$[2.116, 2.354]$	0.1 s	$[-\infty, \infty]$	0.1 s
	FPCS	$[2.120, 2.347]$	1 s	$[2.232, 3.168]$	1.6 s

forms abstract non-linear expressions and constraints. This often yields a first approximation small enough to allow efficient filtering with partial consistencies not relying on the same relaxation.

$3B$-consistency filtering works well with FPCS. $2B$-consistency is not strong enough to reduce the domains computed by FLUCTUAT whereas a stronger kB-consistency is too time-consuming. We experimented also various consistencies implemented in REALPAVER. Table 6 reports the most significant results. $BC5$ is a combination of hull and box consistencies with interval Newton method and $HC4$ is hull-consistency over the user constraints. The timeout limit (T.O.) was set to 5 minutes. $3B$-consistency was difficult to tune through its width parameter that is why we introduced $3B$ timer, a $3B$-consistency interrupted after 0.5 second of filtering. It appears that $3B$-consistency is too time-consuming. A weaker consistency such as the $BC5$-consistency provides a better trade-off between time cost and domain reduction.

Even though the same domain reductions can sometimes be achieved without starting from the approximation computed by FLUCTUAT (i.e., starting from $[-\infty, \infty]$), our experiments show that our approach usually benefit from the approximation computed by FLUCTUAT. For example, with the `sqrt` function of Fig. 5, REALPAVER does not filter the domains in a reasonable time if they are all set to $[-\infty, \infty]$ initially; whereas this takes less than half a second if the domains are set to the bounds computed by FLUCTUAT. Likewise, with the `quadratic` function of Fig. 2, FPCS computes the domain $[-5.001 \times 10^{11}, 0]$ when starting from $[-\infty, \infty]$, whereas FLUCTUAT produces the better approximation of $[-2 \times 10^6, 0]$.

Of course, constraint techniques cannot always refine the approximations computed by FLUCTUAT. Especially when the relaxation done by FLUCTUAT is precise enough to compute good approximations of the domains (e.g., linear systems). Even when the filtering can refine the domains, the computation time may be too long; for instance, when a large number of loop unfoldings is required or when slow convergence phenomena occur in FPCS [19].

5 Conclusion

In this paper, we introduced a new approach for refining the approximations of the domains of the variables computed by the analyzer of C programs FLUCTUAT. This approach relies on constraint solvers, FPCS and REALPAVER, which are

Table 6. Comparison of RealPaver consistencies.

	`quadratic` x1 conf. #1		`quadratic` x1 conf. #2		`sqrt` conf. #1	
	Domain	Time	Domain	Time	Domain	Time
$3B$	n.a.	T.O.	n.a.	T.O.	n.a.	T.O.
$3B$ timer (0.5 s)	$[-11.444, \infty]$	3 s	$[-749\,999.721, 0]$	2.4 s	$[2.12, 2.346]$	1.3 s
weak$3B$	$[-8.004, \infty]$	9.3 s	$[-206\,746.455, 0]$	5 s	$[2.121, 2.346]$	1 s
$BC5$	$[-8.011, \infty]$	1.5 s	$[-518\,518.519, 0]$	0.5 s	$[2.121, 2.346]$	0.3 s
$HC4$	$[-8.223, \infty]$	0.8 s	$[-518\,518.519, 0]$	0.5 s	$[2.121, 2.346]$	0.3 s

correct over floating-point and real numbers. We exploit the refutation capabilities of partial consistencies to reduce the domains computed by Fluctuat. We showed that this approach is fast and efficient on programs that are representative of the difficulties of Fluctuat (conditional constructs and non-linearities).

However, our approach does not substitute for Fluctuat. Tools based on abstract interpretation like Fluctuat are efficient for computing a *global* approximation of the domains; in other words, abstract interpretation computes at each program point the merge of the domains over all paths analysis of the program [6]. Global approximation of conditional statements and widening facilitate scaling but at the price of over-approximations that can be very rough. Moreover, zonotopes constitute better approximations of linear constraint systems than the boxes used in interval-based constraint solvers. Nonetheless, zonotopes are less adapted for non-linear constraint systems. Filtering techniques used in numeric CSP offer a flexible and extensible framework for handling non-linear constraint systems. Distinct exploration of each executable path is a critical issue for computing sharp approximations. However, to limit the combinatorial explosion, we have to bound the length of the paths. Therefore, the approach proposed in this paper is complementary to the one of Fluctuat: our approach gives better results when Fluctuat reduces the domains of the variables beforehand.

The natural extension of this work is to study how Fluctuat can be combined with constraint solvers in a tighter way. Also, stronger consistencies could be used to further improve the approximation precision, e.g., global constraints like the Quad constraint on non-linear expressions [17] and kB-consistency applied to the domains of output variables only.

Acknowledgments. The authors gratefully acknowledge Sylvie Putot and Éric Goubault from CEA-List for their advice and help on using Fluctuat.

References

1. Ayad, A., Marché, C.: Multi-prover verification of floating-point programs. In: IJCAR. LNCS, vol. 6173, pp. 127–141. Springer (2010)
2. Boldo, S., Filliâtre, J.C.: Formal verification of floating-point programs. In: 18th IEEE Symposium on Computer Arithmetic. pp. 187–194. IEEE (2007)

3. Botella, B., Gotlieb, A., Michel, C.: Symbolic execution of floating-point computations. Software Testing, Verification and Reliability 16(2), 97–121 (2006)
4. Brillout, A., Kroening, D., Wahl, T.: Mixed abstractions for floating-point arithmetic. In: 9th International Conference on Formal Methods in Computer-Aided Design. pp. 69–76. IEEE (2009)
5. Collavizza, H., Rueher, M., Hentenryck, P.V.: A constraint-programming framework for bounded program verification. Constraints Journal 15(2), 238–264 (2010)
6. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: 6th ACM Symposium on Principles of Programming Languages. pp. 269–282 (1979)
7. Cousot, P., Cousot, R., Feret, J., Miné, A., Mauborgne, L., Monniaux, D., Rival, X.: Varieties of static analyzers: A comparison with ASTRÉE. In: 1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering. pp. 3–20. IEEE (2007)
8. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems 13(4), 451–490 (1991)
9. Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an industrial use of fluctuat on safety-critical avionics software. In: FMICS. LNCS, vol. 5825, pp. 53–69. Springer (2009)
10. de Dinechin, F., Lauter, C.Q., Melquiond, G.: Assisted verification of elementary functions using Gappa. In: ACM Symposium on Applied Computing. pp. 1318–1322. ACM (2006)
11. Ghorbal, K., Goubault, E., Putot, S.: A logical product approach to zonotope intersection. In: CAV. LNCS, vol. 6174, pp. 212–226. Springer (2010)
12. Goldberg, D.: What every computer scientist should know about floating point arithmetic. ACM Computing Surveys 23(1), 5–48 (1991)
13. Goubault, E., Putot, S.: Static analysis of numerical algorithms. In: SAS. LNCS, vol. 4134, pp. 18–34. Springer (2006)
14. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: VMCAI. LNCS, vol. 6538, pp. 232–247. Springer (2011)
15. Granvilliers, L., Benhamou, F.: Algorithm 852: Realpaver: an interval solver using constraint satisfaction techniques. ACM Transactions on Mathematical Software 32(1), 138–156 (2006)
16. Harrison, J.: A machine-checked theory of floating-point arithmetic. In: TPHOLs. LNCS, vol. 1690, pp. 113–130. Springer-Verlag (1999)
17. Lebbah, Y., Michel, C., Rueher, M.: A rigorous global filtering algorithm for quadratic constraints. Constraints 10(1), 47–65 (2005)
18. Lhomme, O.: Consistency techniques for numeric CSPs. In: 13th International Joint Conference on Artificial Intelligence. pp. 232–238 (1993)
19. Marre, B., Michel, C.: Improving the floating point addition and subtraction constraints. In: CP. LNCS, vol. 6308, pp. 360–367. Springer (2010)
20. Michel, C.: Exact projection functions for floating-point number constraints. In: 7th International Symposium on Artificial Intelligence and Mathematics (2002), `http://rutcor.rutgers.edu/~amai/aimath02/PAPERS/21.ps`
21. Michel, C., Rueher, M., Lebbah, Y.: Solving constraints over floating-point numbers. In: CP. LNCS, vol. 2239, pp. 524–538. Springer Verlag (2001)
22. Monniaux, D.: The pitfalls of verifying floating-point computations. ACM Transactions on Programming Languages and Systems 30(3), 12:1–12:41 (2008)
23. Rump, S.M.: Verification methods: Rigorous results using floating-point arithmetic. Acta Numerica 19, 287–449 (2010)
24. The GSL Team: GNU Scientific Library Reference Manual, 1.14 edn. (2010), `http://www.gnu.org/software/gsl/`