Verifying floating-point programs with constraint programming and abstract interpretation techniques

Olivier Ponsini, Claude Michel, Michel Rueher

► To cite this version:

Olivier Ponsini, Claude Michel, Michel Rueher. Verifying floating-point programs with constraint programming and abstract interpretation techniques. Automated Software Engineering, 2016, 10.1007/s10515-014-0154-2 . hal-00860681v2

HAL Id: hal-00860681
https://hal.science/hal-00860681v2

Submitted on 3 Mar 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Automated Software Engineering manuscript No.
(will be inserted by the editor)

Verifying floating-point programs with constraint programming and abstract interpretation techniques

Olivier Ponsini · **Claude Michel** ·
Michel Rueher

Received: date / Accepted: date

Abstract Static value analysis is a classical approach for verifying programs with floating-point computations. Value analysis mainly relies on abstract interpretation and over-approximates the possible values of program variables. State-of-the-art tools may however compute over-approximations that can be rather coarse for some very usual program expressions. In this paper, we show that constraint solvers can significantly refine approximations computed with abstract interpretation tools. More precisely, we introduce a hybrid approach combining abstract interpretation and constraint programming techniques in a single static and automatic analysis. This hybrid approach benefits of the strong points of abstract interpretation and constraint programming techniques, and thus, it is more effective than static analysers and constraint solvers, when used separately. We compared the efficiency of the system we developed—named RAICP—with state-of-the-art static analyzers: RAICP produces substantially more precise approximations and is able to check program properties on both academic and industrial benchmarks.

Keywords Program verification · Floating-point computation · Constraint solving over floating-point numbers · Constraint solving over real number intervals · Abstract interpretation-based approximation

This work was partially supported by ANR VACSIM (ANR-11-INSE-0004), ANR AEOLUS (ANR-10-SEGI-0013), and OSEO ISI PAJERO projects. Part of this work was done while Michel Rueher was visiting Professor at NII (National Institute of Informatics) 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430

A very preliminary version of this paper was published in Constraint Programming (CP), 2012.

Olivier Ponsini · Claude Michel · Michel Rueher
Université de Nice Sophia Antipolis, CNRS, I3S, 06900 Sophia Antipolis, France
E-mail: firstname.lastname@unice.fr

1 Introduction

Programs with floating-point computations control complex and critical systems in numerous domains, including cars and other transportation systems, nuclear energy plants, or medical devices. Floating-point computations are derived from mathematical models on real numbers (Goldberg, 1991), but computations on floating-point numbers are different from computations on real numbers. For instance, with binary floating-point numbers, some real numbers are not representable (for example, 0.1 does not have any exact representation). Floating point arithmetic operators are neither associative nor distributive, and may be subject to phenomena such as absorption and cancellation. Furthermore, the behavior of programs containing floating-point computations varies with the programming language, the compiler, the operating system, or the hardware architecture.

For all these reasons, floating-point computations are an additional source of errors in embedded programs. But there is much more, including the fact that most programs are written with the semantics of real numbers in mind. That is why it is very important to estimate the accuracy of floating-point computations with respect to the same sequence of operations in an idealized semantics of real numbers. The goal of this estimation is to identify suspicious values, that is values for which the behavior of the program over the floating-point numbers is different from the behavior one could expect over the real numbers. Identifying such suspicious values is a critical issue in embedded program verification.

1.1 Value analysis

Static value analysis is a classical approach for verifying programs with floating-point computations. Value analysis can deal with properties ranging from the absence of run-time errors to simple user assertions (Cousot et al, 2007). Value analysis consists in approximating variable *domains*, that is the set of possible values that each variable can take at a program point. Approximations are mainly worked out with abstract interpretation techniques. They are used to estimate the accuracy of floating-point computations with respect to the same sequence of operations in an idealized semantics of real numbers.

Value analysis is also used to check program properties: if none of the values in variable domains can violate a property, then the property holds. But value analysis over-approximates domains and thus, some values in a domain may not actually be reachable for the corresponding variable. Therefore, value analysis usually cannot ensure that a property is violated: when some values may violate a property, the analysis just reports a potential error in the program. If such a potential error is reported for a property that turns out to be true, it is called a false alarm. This issue is intensified by the fact that state-of-the-art systems for value analysis may compute rather coarse approximations for very usual programming constructs and expressions (Ghorbal et al (2010); see also example in Sect. 1.3).

1.2 Contribution

The main goal of the approach introduced in this paper is to compute tight approximations for value analysis, and thus to reduce the number of false alarms. To achieve this goal, we introduce a hybrid approach for value analysis of floating-point programs that combines abstract interpretation (AI) and constraint programming (CP) techniques. This hybrid approach benefits of the strong points of AI and CP techniques, and thus, it is more effective than static analyser and CP solvers, when used separately. On one hand we exploit the refutation capabilities of CP solvers to refine domains computed by abstract interpretation, on the other hand we take advantage of AI's capabilities for approximating loops and some specific constraints with multiple occurrences of variables. We show that CP solvers over floating-point numbers and over real numbers can significantly improve the precision of the value analysis. Experiments on academic programs demonstrate that our system—named rAiCp—is substantially more precise than FLUCTUAT (Delmas et al, 2009), a state-of-the-art AI analyzer dedicated to floating-point computations; especially on programs that are difficult to handle with abstract interpretation techniques.

We also evaluated rAiCp on a set of 55 benchmarks proposed by D'Silva et al (2012) to demonstrate the capabilities of CDFL, a program analysis tool that embeds an abstract domain in the conflict driven clause learning algorithm of a SAT solver. rAiCp was on average three-fold faster than CDFL and four-fold slower than FLUCTUAT but did not produce any false alarm whereas FLUCTUAT did generate 11 false alarms.

Eventually, we applied rAiCp to check a property of a real time software application embedded in a car provided by Geensys/Dassault Systems[1]. rAiCp proved the property for a realistic system service time. rAiCp also compared well on this example with CBMC, a state-of-the-art bounded model checker using on a SAT solver.

To sum up, rAiCp is a promising framework for computing precise domain approximations in floating-point programs and thus for proving properties of hybrid systems with floating-point and integer computations.

Before going into the details, we illustrate in the next subsection how our approach works on a small example.

1.3 Motivating example

The program in Fig. 1 contains only linear expressions and a sequence of two conditional statements. This quite simple program is difficult to handle for AI-based analyses. On floating-point numbers—as well as on real numbers—this function returns a value in the interval $[0,50]$. Indeed, the pre-condition and the assignment of line 5 state the following constraints on g and y: $\{x = f + 2 * g \text{ and } f, g \in [-10,10]\}$. Thus, from the conditional statement of line 7 we can derive the following information:

- `then` branch, line 8: $g \in [-10,5]$, and thus $y \in [-10,5]$ at the end of this branch;
- `else` branch, line 11: $g \in]-5,10]$, and thus $y \in [-10,5[$ at the end of this branch.

[1] `http://www.3ds.com/`

```
1 /* Pre−condition: f,g ∈ [−10,10] */
2 float foo(float f, float g) {
3   float x, y, z;
4
5   x = f + 2 * g;
6
7   if (x <= 0) {
8     y = g;
9   }
10  else {
11    y = -g;
12  }
13
14  if (y >= 0) {
15    z = 10*y;
16  }
17  else {
18    z = -y;
19  }
20
21  return z;
22 }
```

Fig. 1 Function `foo`

Then, from the conditional statement of line 14, we obtain $z \in [0,50]$.

Conversely FLUCTUAT fails to compute a good approximation for z. With *zonotope*-based abstract domains, FLUCTUAT over-approximates z to $[0,100]$, both over the real numbers and the floating-point numbers. The difficulty for AI techniques is to intersect the abstract domains computed for x at lines 5 and 7. Actually, AI techniques are unable to derive from these statements any constraint on g. As a consequence, FLUCTUAT estimates that g ranges over $[-10,10]$ in the `then` and `else` branches of the first conditional statement. FLUCTUAT's analysis of the second conditional statement is more precise, but the upper bound of z is overestimated since it relies on the coarse over-approximation of g and y computed previously.

On this example, RAICP managed to shrink the domain of z to $[0,50]$. To do so, it successively used AI techniques and CP techniques between consecutive merge points of the control flow graph of the program. The key idea is to build one constraint system for each path between successive merge points, and to apply CP filtering techniques on each of these systems to refine the approximations computed by AI on the corresponding path. Merge points of program `foo` are at lines 13 and 21; for the sake of uniformity, we consider also the program's entry point as a merge point.

There are two paths between the program's entry point and the first merge point. Consider the path through the `then` branch of the first conditional statement. AI techniques compute on this path the following approximations: $f,g,y \in [-10,10]$, $x \in [-10,0]$. So, the constraint system built for this path is:

$$C_1 = \{x = f + 2*g \wedge x \leq 0 \wedge y = g \wedge -10 \leq f \wedge f \leq 10 \wedge -10 \leq g \wedge g \leq 10 \\ \wedge -10 \leq y \wedge y \leq 10 \wedge -10 \leq x \wedge x \leq 0\}$$

CP filtering techniques reduce the domain of g to $[-10,5]$ and shrink the domain of y to $[-10,5]$ with constraint system C_1. In a similar way, for the path going through the `else` branch of the first conditional statement, we obtain the constraint system:

$$C_2 = \{x = f + 2*g \land x > 0 \land y = -g \land -10 \leq f \land f \leq 10 \land -10 \leq g \land g \leq 10 \\ \land -10 \leq y \land y \leq 10 \land 0 < x \land x \leq 10\}$$

Here, CP techniques shrink the domain of y to $[-10,5[$ with constraint system C_2.

We merge the domains computed for every variable on the different paths reaching a merge point. So, at line 13, the domain of y becomes $[-10,5]$, that is the smallest closed interval including all the values in $[-10,5] \cup [-10,5[$. Note that this domain is sharper than the one computed by FLUCTUAT, that is $y \in [-10,10]$. These new domains are then used for analyzing the rest of the program.
On program `foo`, the analysis goes on from line 13 to the next merge point at line 21. Again, we generate a constraint system for each of the two paths. For the path through the `then` branch of the second conditional statement, AI techniques shrink the domain of z to $[0,50]$. Hence, the constraint system for this path is $\{y \geq 0 \land z = 10*y \land -10 \leq y \land y \leq 5 \land 0 \leq z \land z \leq 50\}$. CP filtering techniques cannot reduce anymore the domain of z with this constraint system. Likewise, for the path going through the second conditional statement, RAICP builds the constraint system $\{y < 0 \land z = -y \land -10 \leq y \land y \leq 5 \land 0 < z \land z \leq 50\}$. Here, CP filtering techniques reduce the domain of z to $[0,10]$. Finally, at the last merge point, RAICP computes the union of domains and we obtain $z \in [0,50] \cup]0,10] = [0,50]$.

It is worth noting that RAICP does not generate one constraint system for each execution path in the control flow graph (CFG) of a program. We split programs according to the merge points in the CFG and we generate one constraint system per path going from one merge point to the next merge point. Thus, for a program with a succession of n conditional statements, we would only generate $2n$ constraint systems whereas the program includes 2^n execution paths. At each merge point, we use CP filtering techniques to shrink the domains computed by abstract interpretation. Then the analysis goes on with the reduced domains. Note also that the CFG exploration is performed on-the-fly: exploration stops as soon as we detect that the constraint system of the current path is inconsistent, that is when we detect that the current path is infeasible.

Now, assume we want to verify a post-condition p_1 that states that the value returned by function `foo` is always less than 75. Since AI-based analysis approximates the domain of z by $\in [0,100]$, it would infer that the post-condition may not hold, and hence generating a false alarm. In contrast, RAICP can ensure that post-condition p_1 holds. Here, the proof is trivial since the upper bound of z is strictly smaller than 75. In practice this proof may be more difficult and we apply the following process: to check a post-condition defined over the program variables, we add the negation of this post-condition to each of the constraint systems generated between the last merge point and the end of the program. If all these systems are inconsistent, we can conclude that the post-condition holds; otherwise, the post-condition may be violated.

In program `foo`, we have two paths from the merge point at line 13 to the end of the program. So, to prove post-condition p_1, we generate the following constraint

systems:

$$\{y \geq 0 \wedge z = 10 * y \wedge z \geq 75 \wedge z \leq 50 \wedge \cdots\}$$
$$\{y < 0 \wedge z = -y \wedge z \geq 75 \wedge z \leq 50 \wedge \cdots\}$$

Both systems are trivially inconsistent and thus, we can ensure that post-condition p_1 holds.

For program properties specified as assertions inside the program, we apply the same reasoning as for a post-condition: we consider the constraint systems that correspond to paths reaching the assertion from previous merge points together with the negation of the assertion.

1.4 Outline of the paper

In Sect. 2, we recall basics on abstract interpretation and constraint programming. Sect. 3 concerns related works. RAICP is described in details in Sect. 4. Section 5 gives some insights into the implementation and analyses the experiments and their results.

2 Background

Before going into the details, we recall basics on abstract interpretation and constraint programming techniques that are useful to understand the rest of this paper. Readers familiar with these techniques may skip the corresponding sections.

2.1 Abstract interpretation

The main intuition behind abstract interpretation[2] is that we do not need to know the exact state of a program to prove some property: a conservative approximation is sufficient. That's why abstract interpretation methods (Cousot and Cousot, 1977, 1979; Cousot, 2001) define an abstract semantics that approximates the concrete semantics of programs. In other words, given a programming or specification language, abstract interpretation consists of giving several semantics linked by relations of abstraction. A semantics is a mathematical characterization of a possible behaviour of the program. The most precise semantics, describing very closely the actual execution of the program, is called the concrete semantics. An abstract semantics is built upon an abstract domain that determines a trade-off between precision and speed of the analysis.

An abstract domain approximates the concrete state of a program by considering only some specific properties of the state. Then, all concrete operations are mapped to corresponding operations in the abstract domain. More precisely, abstract values

[2] See also `http://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www`

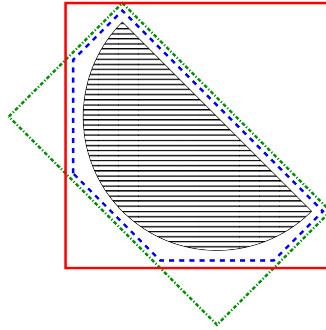

Fig. 2 Half-disk approximations by intervals (red straight lines), zonotope (green pointed lines), and polyhedron (blue dashed lines)

name sets of concrete values, i.e., a function α, maps each set to the abstract value that best describes it. The inverse function γ maps abstract value to set of concrete values it represents. Abstraction followed by concretization demonstrates that α is sound but not exact. This is formalized by a Galois connexion.

Predicate abstraction is used for conditional instructions. An acceleration method is used to enforce the convergence of fixed-point computations and to approximate the effect of loops. It relies of a widening operator in the general case. Local narrowing techniques are used on the control points where widening has been applied to yield a better fixed point, and thus a more precise over approximation than the one computed by widening. Widening and narrowing operations approximate program loops in very short time.

The choice of an abstract domain is a critical issue. As we can observe on Fig. 2, the approximation of the half-disk in black by a polyhedron is much more precise than the approximation by a box of intervals. The issue is that operations like intersection between polyhedra require computationally expensive algorithms whereas these operations are trivial on intervals. Zonotopes (Goubault and Putot, 2006) offer a good trade-off between performance and precision. Zonotope abstract domain affine arithmetic is an extension of Interval Arithmetic that keeps track of affine relations between values of variables. An affine form expresses a set of values as a central value plus a sequence of deviation terms over symbolic symbols, called noise symbols. In other words, zonotopes are sets of affine forms that preserve linear correlations between variables and keep track of the statements involved in the loss of precision of floating-point computations. Zonotopes have nevertheless some drawbacks: approximations of some common program constructs, such as conditionals and nonlinear expressions, are not very precise. Zonotopes were successfully applied elsewhere, such as for reachability analysis in the model-checking of hybrid systems (Girard, 2005).

An abstract semantics is a super-set of the concrete program semantics, and thus AI-based analyses are sound but incomplete. In other words, since the domains of the variables are over-approximated by value analysis, properties proved true with the abstract semantics are actually true on the concrete one, but properties violated with the abstract semantics may hold with the concrete one. The latter case is called

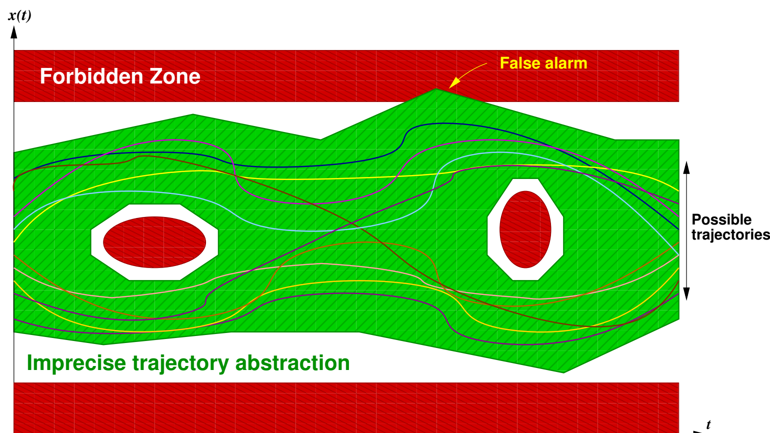

Fig. 3 A false alarm occurs when the abstract semantics intersects the forbidden zone while the concrete semantics does not intersect this zone. Forbidden zones are in red, the abstract semantics is in green, and the concrete semantics is the set of curves.

a *false alarm* when properties represent desired behaviors of the program (see Fig. 3 extracted from Cousot's informal introduction to abstract interpretation[3]).

To sum up, AI techniques provide a good trade-off between precision and performance. They scale well but they lack precision for programs with non-linear expressions and with numerous conditionals.

2.2 Constraint programming

Constraint Programming (CP) is a *way of modeling and solving combinatorial optimization problems*. CP combines techniques from artificial intelligence, logic programming, and operations research. The CP framework is basically a *branch & prune* schema inspired by the traditional branch and bound approach used in optimisation problems. In this paper we mainly use constraint techniques over continuous domains[4] where this *branch & prune* schema is best viewed as an iteration of two steps (Hentenryck et al, 1997; Rossi et al, 2006):

1. Pruning the search space;
2. Branching on variables.

The *branching step* splits the interval associated to some variable in two or more intervals (often with the same width), that is, it generates two (or more) sub-problems. This step relies on *search strategies* that try to exploit the structure of the problem to guide the splitting and instantiation process.

Pruning techniques on continuous domains rely on local consistencies. Local consistencies are conditions that filtering algorithms must satisfy. A filtering algorithm

[3] `http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html`

[4] For an informal introduction, see `http://www.it.uu.se/research/group/astra/CPmeetsCAV/slides/rueher_Continuous_Domains.pdf`

$$\mathscr{D}_k = \begin{cases} \mathscr{D} & \text{if } k = 0 \\ Op(\mathscr{D}_{k-1}) & \text{if } k > 0 \end{cases}$$

Fig. 4 Filtering algorithms as fixed point algorithms

can be seen as a fixed point algorithm (see Figure 4) defined by the sequence $\{\mathscr{D}_k\}$ of domains generated by the iterative application of an operator $Op : \mathbb{I}^n \longrightarrow \mathbb{I}^n$ where $\mathbb{I}^n$ stands for a Cartesian product of intervals over the real numbers $\mathbb{R}$, and $\mathscr{D}$ denotes the set of the initial domains of the variables of the constraint satisfaction problem.

The operator Op of a filtering algorithm generally satisfies the following three properties:

- $Op(\mathscr{D}) \subseteq \mathscr{D}$ (contractancy)
- $Op(Op(\mathscr{D})) \equiv \mathscr{D}$ (idempotency)
- $\mathscr{D}' \subseteq \mathscr{D} \Rightarrow Op(\mathscr{D}') \subseteq Op(\mathscr{D})$ (monotonicity)

Moreover, Op is conservative; that is, it cannot remove any solution. Under those conditions, the limit of the sequence $\{\mathscr{D}_k\}$, which corresponds to the greatest fixed point of the operator Op, exists and is called a *closure*. A fixed point for Op may be characterised by a property lc-consistency, called a local consistency. The algorithm achieving filtering by lc-consistency is denoted lc-filtering. A constraint system C is said to be lc-satisfiable if lc-filtering of C does not produce an empty domain.

$2B$-consistency (Lhomme, 1993) states a local property on the bounds of the domains of a variable at a single constraint level: the domain of variable x is $2B$-consistent if, for any constraint c, at least one value exists in the domains of all other variables such that c can be satisfied when x is set to the upper or lower bound of its domain. Practically, $2B$-consistency reduces an interval when it can prove that the upper bound or the lower bound does not satisfy some constraint. Stronger consistencies have also been defined. For instance, $3B$-consistency (Lhomme, 1993) checks whether $2B$-Consistency can be enforced when the domain of a variable is reduced to the value of one of its bounds in the whole system. Roughly speaking, $3B$-pruning algorithms rely on a shaving process that tries to shrink the interval of a given variable.

To sum up, the strong points of CP are its refutation capabilities and its great flexibility. The pruning algorithm may, however, be time consuming on large domains.

2.3 Complementarity of CP and AI techniques

AI and CP are complementary techniques for computing tight approximations. First, CP solvers cannot handle loops whereas the widening and narrowing techniques used in AI can often compute good approximations of the domains of variables occurring in loops, especially when the approximation of the input values of the loop are precise enough. Program `bigLoop` (see fig. 5a, p. 23) illustrates well the advantage of this collaboration : without the approximation computed by the CP solver, the AI-based static analyser is unable to compute a precise fixed-point of the loop. Second, both CP and AI techniques may be very sensitive to the syntactic form of the constraints,

```
1  /* Pre-condition:
2     x, y, and t ∈ [-10,10] */
3  float foo2(float x,
4                 float y, float t) {
5    float u, z, k;
6
7    u = t*t - t;
8    k = 2.0;
9
10   if (u < 0.0) {
11     if (t > 1.2) {
12       k = 6.0;
13     }
14   }
15
16   z = (x - y)*(x + y);
17   z = k*z;
18
19   return z;
20 }
```

(a)

```
1  /* Pre-condition:
2        x, y, and t ∈ [-10,10] */
3  float foo3(float x,
4                 float y, float t) {
5    float u, z, k;
6
7    u = t*t - t;
8    k = 2.0;
9
10   if (u < 0.0) {
11     if (t > 1.2) {
12       k = 6.0;
13     }
14   }
15
16   z = k*(x - y)*(x + y);
17
18
19   return z;
20 }
```

(b)

Fig. 5 Two variations, (a) and (b), of a simple program

especially for non-linear constraints and constraints with multiple occurrences of a same variable. Programs `foo2` and `foo3` (see fig. 5) illustrate well the advantage of a cooperation of both techniques in the presence of such constraints. Both programs differ only by syntactic expression used for calculating `z` (see lines 16 and 17 of fig. 5). CP handles better than FLUCTUAT the conditional statements (line 7 –14) whereas FLUCTUAT approximates better than CP the evaluation of z in program `foo2` but not in program `foo3`. Let us detail this process now.

Handling the conditional statements. After the first two assignments CP reduces the domain of `u` to the interval $[-0.28300884, 110.0]$ whereas FLUCTUAT can only reduce it to $[-10.0, 110.0]$. At this stage, CP detects that the condition of the `if` statement at line 11 cannot hold. More precisely, filtering of the constraint system $C_{foo} = \{u = t*t - t \wedge k = 2.0 \wedge u < 0.0 \wedge t > 1.2\}$ yields an empty domain, and thus CP detects that C_{foo} does not hold. This prevents assigning `k` by 6, and so, at line 15, before the evaluation of `z`, the value of `k` is still 2.0 for CP filtering, but FLUCTUAT enlarged the domain of `k` to the interval $[2, 6]$.

Approximating z. FLUCTUAT approximates better than CP the expression of `z` (line 16 in program 5a). This is due to the fact that filtering techniques consider the different occurrences of variables `x` and `y` as different variables with the same domain. So, when considering expression `k*z` at line 17, FLUCTUAT has already reduced the domain of `z` to $[-100, 100]$ whereas CP only shrunk it to $[-400.0, 400.0]$. Hence, at the exit of program 5a, CP returns a domain of $[-800.0, 800.0]$ and FLUCTUAT returns a domain of $[-600.000245, 600.000245]$. But for program 5b, FLUCTUAT will consider `k` as a variable in the expression `k*(x - y)*(x + y)` and therefore can only reduce the domain of `z` to $[-1200.00037, 1200.00037]$ whereas CP filtering will still reduce the domain of `z` to $[-800.0, 800.0]$.

For both programs, RAICP can take advantage of the reductions of k by CP filtering and of z by FLUCTUAT to shrink the final domain of z to $[-200.00008, 200.0008]$.

3 Related works

Various 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 relying on abstract interpretation capture rounding errors due to floating-point computation in their abstract domains. They are usually fast, automatic, and scalable. They may, however, lack precision. ASTRÉE (Cousot et al, 2007) is one of the most famous tool in this family of methods: it estimates the value of the program variables at every program point and can show the absence of run-time errors, for example, division by zero, arithmetic overflow. FLUCTUAT (Delmas et al, 2009) estimates in addition the accuracy of the floating-point computations: it bounds 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.

Proof assistants like Coq (Boldo and Filliâtre, 2007) or HOL (Harrison, 1999) serve to formalize floating-point arithmetic. Proofs of program properties are done manually in the proof assistants that only guarantee the correctness of the proof. Even though some parts of the proofs may be automatized, the user must still make a significant effort to conduct the proof. Moreover, when a proof strategy fails to prove a property, the user often does not know whether the property is false or whether he could prove it with another strategy. The Gappa tool (de Dinechin et al, 2011) 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, for example, 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. Again, the cost for the user of this semi-automatic method is considerable. Ayad and Marché (2010) propose axiomatizing 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.

The classical bit-vector approach of SAT solvers is ineffective on programs with floating-point computations because of the size of the domains of floating-point variables and the cost of bit-vector operations. An abstraction technique was devised for CBMC by Brillout et al (2009). It relies on under and over-approximation of floating-point numbers with respect to a given precision expressed as a number of bits of the mantissa. This technique remains however slow. D'Silva et al (2012) developed recently CDFL, a program analysis tool that embeds an abstract domain in the conflict driven clause learning algorithm of a SAT solver. CDFL relies on a sound and complete analysis for determining the range of floating-point variables in loop-free control software. The authors state that CDFL is more than 200-fold faster than

CBMC (D'Silva et al, 2012). In Section 5.3, we compare the performances of CDFL and RAICP on a set of benchmarks proposed by D'Silva et al.

Links between abstract interpretation and constraint logic programming have been studied at a theoretical level for a long time (Codognet and Filé, 1992). More recently, Denmat et al (2007) introduced a new global constraint to model iterative arithmetic relations between integer variables. The associated filtering algorithms rely on abstract interpretation over polyhedra. Pelleau et al (2013) designed a generic constraint solver parametrized by abstract domains. They focus on mixed discrete-continuous problems over the integer and real numbers. In this paper, we show how abstract interpretation and constraint programming techniques can complement each other for the static analysis of floating-point programs.

4 RAICP, a hybrid approach

As said before, RAICP, the approach we introduce in this paper, relies on a piece-wise exploration of a program CFG that alternates path analysis and merging steps. Nodes of the CFG where two branches join are selected as merge points. We build one constraint system per path between two successive merge points. We use CP filtering techniques on these systems to reduce variable domains first computed with AI techniques. At merge points, the reduced domains for the different paths are merged and exploration goes on with the next part of the CFG.

In Sect. 4.1, we detail the notions of merge point and path exploration of a CFG. Then, in Sect. 4.2, we give the algorithms implemented in RAICP to perform piece-wise exploration of the CFG and compute domain approximations on each piece of the CFG.

4.1 Control flow graph exploration

A control flow graph is a standard graph representation of computations and control flow in a program. Nodes in the graph are basic blocks of the program, that is a sequence of consecutive statements without any branching in it. Directed edges represent possible flow of control from the end of one block to the beginning of another one. A control flow graph contains one entry node i.e., a node without incoming edge, and one exit node i.e., a node without outgoing edge.

In our CFGs, we will consider the following types of nodes:

- *assignment* nodes containing a program assignment;
- *assertion* nodes containing a logical expression to be checked;
- *while* nodes containing a loop condition and a loop body;
- *if* nodes containing a branching condition;

Exploring all paths would be intractable since there are 2^n paths in a program with n sequential `if` statements. That's why we defined specific locations in the program that correspond to nodes in the CFG where two branches join. We call these locations *merge points*. In addition, the exit node is always a merge point. Our CFGs are acyclic

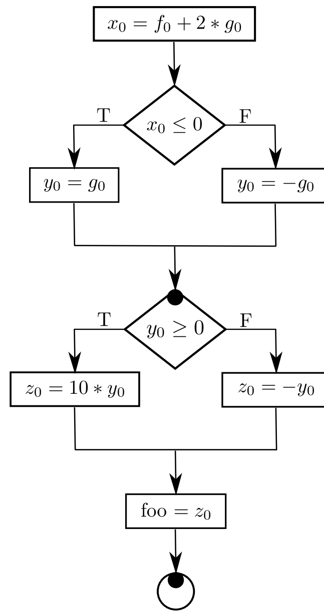

Fig. 6 CFG of program `foo` from Fig 1: nodes with black circles are merge points

graphs since we unfold loops a bounded number of times before enclosing them in a *while* node (see paragraph on *while* node in sub-section 4.2.1). For instance, the graph in Fig. 6 is the CFG of function `foo` described in Fig. 1. Edges labeled T (resp. F) represent the control flow when the associated condition is true (resp. false). The merge points are the nodes with a black circle. The second merge point is not the assignment node that follows the branch junction but the exit node: a merge point is always the last node of a straight sequence of nodes after a junction. Note that program expressions were put into DSA-like form[5] to facilitate constraint generation.

As said before, we only explore paths between two successive merge points. Of course, this process may be less accurate than an exploration of the full length paths, but it is sound: variable domains are over-approximated for value analysis and properties found to be true hold over the full length paths too. In addition, it is easy to parametrize the length of the paths in term of the number of choice points. Our experiments have, however, shown that exploring more than one merge point at once quickly results in time explosion and only yields a limited gain in precision. An exploration strategy using one merge point at once seems to be a good trade off between time and precision.
The CFG is explored using a forward analysis starting at the beginning of the program. We generate one constraint system for each path between two consecutive merge points. At any point of a path, the possible states of the program are represented by a constraint system over the program variables. To this end, the semantics

[5] DSA (Dynamic Single Assignment) is a semantic preserving program transformation in which each variable is assigned at most once on each program path (Barnett and Leino, 2005).

Algorithm 1: RAICP

Data:
- Q, a queue of merge points.
- D, an array of sets s.t. D[m] is the set of variable domains at merge point m.
- $\mathscr{D}^I$, the initial variable domains.
- n^I, the CFG's entry node.
- n^E, the CFG's exit node.

Result:
D[n^E] is the set of variable domains at the end of the program.
error is a set of domains when an assertion may be violated; otherwise it is the empty set.

```
1 error ⟵ explorePaths(n^I, D^I, ∅)
2 while error = ∅ and Q ≠ ∅ do
3 |   n ⟵ pop(Q)
4 |   if n ≠ n^E then
5 |   |   error ⟵ explorePaths(n, D[n], ∅)
6 |   end
7 end
```

of each program statement is expressed by constraints. Variable domains are intervals over the integers, the floating-point numbers, or the real numbers depending on the type of the variable. This technique for representing programs by constraint systems was introduced for bounded program verification in CPBPV by Collavizza et al (2010).

4.2 RAICP algorithm

In this section, we detail how RAICP explores the CFG between consecutive merge points. We also describe the process for computing domain approximations.

4.2.1 Exploring paths

Algorithm 1 launches the exploration of the paths from each merge point. It uses a queue of merge points ordered by increasing depth in the CFG, that is the number of nodes from the entry node to the merge point. RAICP stores in array D the result of the value analysis of the program at each merge point. Initially, all elements of D are empty sets and at the end D[m] contains the domain approximations computed at merge point m made of the union of the domains of each paths that belong to the given merge point.

When the program contains a user assertion, RAICP stores in set error the result of the assertion checking process: error is empty if the assertion holds; otherwise, error contains values that may violate the assertion.

Algorithm 1 calls the recursive function `explorePaths` to explore all the paths between a given node and the next merge points. Exploration of the CFG stops when a property may be violated or all merge points were considered. Function `explorePaths` updates the domains stored in D during path exploration. To this end, the function generates on-the-fly one constraint system per path while visiting successively the

Function explorePaths

```
Data:
    Q,    a queue of merge points ordered by increasing depth in the CFG.
    D,    an array of sets s.t. D[m] is the set of variable domains at merge point m.
Input:
    n,      a CFG node.
    D_csp,  a set of variable domains.
    csp,    a set of constraints.
Output: A set of domains when an assertion may be violated; ∅ otherwise.
1  Function explorePaths(n, D_csp, csp) is
2      if n ∈ Q then   // merge point
3          D_a ⟵ approximate(D_csp, csp)
4          D[n] ⟵ D[n] ∪ D_a
5          return ∅
6      else if n is assignment assign then
7          return explorePaths(next(n), D_csp, csp ∪ {assign})
8      else if n is assertion assert then
9          error ⟵ check(D_csp, csp, assert)
10         if error ≠ ∅ then return error   // assertion is violated
11         else return explorePaths(next(n), D_csp, csp)
12     else if n is a 'while' node with condition cond then
13         D_a ⟵ approximate(D_csp, csp)
14         D_AI ⟵ approx_loop_AI(D_a, n)
15         return explorePaths(next(n), D_AI, {¬cond})
16     else if n is an 'if' node with condition cond then
17         error ⟵ ∅
18         if feasible(D_csp, csp ∪ {cond}) then
19             error ⟵ explorePaths(nextThen(n), D_csp, csp ∪ {cond})
20         end
21         if error = ∅ and feasible(D_csp, csp ∪ {¬cond}) then
22             error ⟵ explorePaths(nextElse(n), D_csp, csp ∪ {¬cond})
23         end
24         return error
25     end
26 end
```

nodes of the path. At an *if* node, `explorePaths` explores successively the paths in each branch of the control flow. Note that the function checks the consistency of the constraint system of a branch before exploring it.

At each merge point m, `explorePaths` calls function `approximate` for computing an approximation of the domains for the current path. Function `approximate` combines AI and CP techniques (see Sect. 4.2.2). Function `explorePaths` updates $\mathsf{D}[m]$ with the smallest closed interval including all the values in the union of the domains computed for the different paths.

For *while* nodes, `explorePaths` relies on AI analyser to approximate the domains at the end of the loop. More precisely, `explorePaths` calls an AI-based analyser that uses standard widening and narrowing techniques for computing an over-approximation of the state at the end of the loop. By default, loops are unfolded at most 10 times. The function then goes on exploring the path with these domains and the negation of the loop condition in the constraint system. Approximating loops

Function approximate

Input:
$\mathscr{D}$, current variable domains.
$\mathscr{C}$, current set of constraints.
Output: A set of domains. If $\mathscr{C}$ is found inconsistent, the returned set is empty.

```
1  Function approximate(𝒟, 𝒞) is
2      if ¬consistent_synt(𝒞) then
3          return ∅
4      else
5          D_AI ⟵ filter_AI(𝒟, 𝒞)
6          if D_AI = ∅ then
7              return ∅
8          else
9              return filter_CP(D_AI, 𝒞)
10         end
11     end
12 end
```

with AI techniques ensures that the length of paths are bounded, and as a result the constraint system generation always terminates.

When `explorePaths` reaches an *assertion* node, it will check whether the assertion holds on the current path. To this end, `explorePaths` calls function `approximate` with a constraint system made up of the negation of the assertion to check and of the constraints collected along the path starting at the previous merge point. When function `approximate` can detect an inconsistency, the assertion holds and exploration goes on with the next node on the path. Otherwise, the property checking process is inconclusive: path exploration stops and the domains computed by `approximate` are returned.

4.2.2 Computing approximations

Function `approximate` computes an approximation of the variable domains for a given path between two successive merge points. It takes the domains defined at the beginning of the path ($\mathscr{D}$) and the constraints collected on the path ($\mathscr{C}$). The function returns domains reduced according to the constraints, or an empty set if an inconsistency of the constraint system has been detected.

Function `approximate` starts by checking whether the set of constraints $\mathscr{C}$ is not trivially inconsistent: $consistent_{synt}$ just checks whether a constraint and its syntactic negation are in $\mathscr{C}$. This removes some slow convergence issues that may occur when trying to solve pathological systems such as $\{a \geq b \wedge a < b\}$. Note that a and b must be identical expressions in both constraints: we do not perform any formal expression simplification because they would likely be unsafe.

Function $filter_{AI}$ calls an AI library to analyze the part of the program corresponding to the path between the two considered merge points. It returns an empty set when it detects that the path is infeasible. Function $filter_{CP}$ applies strong partial consistencies to the constraint system of the path updated with the domains computed by $filter_{AI}$.

5 Experiments

In this section, we first describe the prototype of RAICP we have implemented. Then, we report the experiments we have performed to evaluate RAICP. We compare RAICP with FLUCTUAT on academic programs, and we evaluate the property checking capabilities of RAICP both on a set of academic benchmarks provided by the authors of CDFL and on an industrial benchmark. Academic programs are available at `http://users.polytech.unice.fr/~rueher/Benchs/ASE_RAICP`.

All results were obtained on an Intel I7-2720QM at 2.2 GHz with 8 GB of memory running Linux using FLUCTUAT version 3.1247, REALPAVER version 0.4, CPLEX version 12.6, CBMC version 4.7 and the downloadable version of CDFL.

5.1 Implementation

We implemented a prototype of RAICP that uses:

- FLUCTUAT for AI-based computations,
- REALPAVER for constraint solving over real numbers, and
- FPCS for constraint solving over floating-point numbers.

More precisely, RAICP takes as input a C program and builds the corresponding CFG. Each explored path of the CFG between two merge points is transformed into both a set of constraints and a C program. RAICP calls the FLUCTUAT library on these generated C programs. Then, RAICP passes the domains returned by FLUCTUAT and the set of constraints to the constraint solver FPCS (resp. REALPAVER) to reduce the domains over the floating-point numbers (resp. the real numbers). The domains returned by the constraint solver will be used by RAICP for the next steps of the analysis.

Neither REALPAVER nor FPCS can deal with constraints over integers. As a workaround, the prototype handles constraints over integers with the MILP solver IBM ILOG CPLEX in separate constraint systems. The current prototype does not yet handle variables that appear both in constraints over integers and floating-points.

Our prototype uses $2B$-like partial consistencies[6] to cut infeasible paths during CFG exploration and $3B$-like partial consistencies[7] to reduce domains at merge points. This choice is motivated by performance: $2B$-like consistency algorithms are much faster than $3B$-like consistency algorithms, but the latter may achieve a much stronger pruning.

As said before, the length of the paths in term of the number of choice points can be parametrized in RAICP. We experimented various lengths on the different benchmarks, but we did not obtain a significant reduction of the domains; however this may come from the fact that all this benchmarks –except the one containing loops– have a limited number of nested `if` statements.

[6] The prototype uses REALPAVER's $HC4$-consistency or FPCS's $2B(w)$-consistency.

[7] The prototype uses REALPAVER's $BC5$-consistency in paving mode or FPCS's $3B(w)$-consistency.

RAICP analyzes C programs that conform to IEEE 754 standard with the following restrictions: size of arrays are bounded; pointers, bit-wise operators and statements that interrupt the control flow (`goto`, `continue`, and `break`) are not handled. All aspects of computations over floating-point numbers are unfortunately not specified in the IEEE 754 standard and so are implementation-dependent. We assume here that the C programs will be compiled with GCC without any optimization option and run on an x86 architecture managed by a 64-bit Linux operating system[8]. In the current implementation, we handle basic arithmetic operations, comparisons and some classical functions like square root.

The RAICP prototype works automatically without any user interaction: the user only needs to provide domains for the input variables.

5.1.1 AI-based static analyzer

FLUCTUAT is a static analyzer for C programs that proceeds by abstract interpretation. It is specialized in estimating the precision of floating-point computations (Delmas et al, 2009). FLUCTUAT is developed by CEA-LIST[9] and was successfully used for industrial applications of several tens of thousands of lines of code in transportation, nuclear energy, or avionics areas. FLUCTUAT compares the behavior of the analyzed program over real numbers and over floating-point numbers. In other words, it determines ranges of values for the program input variables and computes for each program variable v:

- bounds for the domain of variable v considered as a real number;
- bounds for the domain of variable v 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 variable v ;
- the contribution of the input variables to the error associated with variable v.

FLUCTUAT uses the weakly relational abstract domain of zonotopes (Goubault and Putot, 2006). Zonotopes are sets of affine forms that preserve linear correlations between variables. They offer a good trade-off between performance and precision for floating-point and real number computations. Indeed, the analysis is fast and scales well, processes accurately linear expressions, and keeps track of the statements involved in the loss of accuracy of floating-point computations. To increase the analysis precision, FLUCTUAT uses arbitrary precision numbers or to subdivide up to two input variable intervals. Over-approximations computed by FLUCTUAT may, unfortunately, be very large because the abstract domains do not handle well conditional statements and non-linear expressions.

[8] Computations are done using SSE float and double operations according to the targeted type.

[9] `http://www-list.cea.fr/validation_en.html`

5.1.2 Constraint solver over the real numbers

REALPAVER is an interval solver for numerical constraint systems over the real numbers[10] (Granvilliers and Benhamou, 2006). It handles non-linear constraints defined with the usual arithmetic operations as well as transcendental elementary functions.

REALPAVER 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. An approximation of a solution is described by a box, that is the Cartesian product of the domains of the variables. REALPAVER either proves the unsatisfiability of the constraint system or computes small boxes that contains all the solutions of the system.

The REALPAVER modeling language does not provide strict inequality and not-equal operators, which can be found in conditional expressions in programs. 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 it is safe since no solutions are lost.

We experimented with various consistencies implemented in REALPAVER: $BC5$, a combination of $2B$ and box consistencies with interval Newton method, provided the best trade-off between time cost and domain reduction.

5.1.3 Constraint solver over the floating-point numbers

FPCS is a constraint solver designed to solve a set of constraints over floating-point numbers without losing any solution (Michel, 2002; Marre and Michel, 2010). Note that constraint solvers over the real numbers relying on interval arithmetic cannot handle constraints over the floating-point numbers because of the specific properties of the floating-point numbers. The tricky point is that constraints that do not have any solutions over the real numbers may hold over the floating-point numbers. Moreover, relations that hold over the real numbers may not hold over the floating-point numbers. Finite domain solvers are ineffective for handling constraints over the floating-point numbers due to the huge size of the domains.

FPCS implements $2B$-consistency with projection functions adapted to floating-point arithmetic (Michel et al, 2001; Botella et al, 2006). Inverse projection functions that keep all the solutions are the most difficult to implement. Indeed, direct projections only requires a slight adaptation of classical results on interval arithmetic but inverse projections do not follow the same rules because of the properties of floating-point arithmetic. More precisely, each constraint is decomposed into an equivalent binary or ternary constraint by introducing new variables if necessary. A ternary constraint $x = y \odot_f z$, where $\odot_f$ is an arithmetic operator over the floating-point numbers, is decomposed into three projection functions:

- the direct projection, $\Pi_x(x = y \odot_f z)$;
- the first inverse projection, $\Pi_y(x = y \odot_f z)$;

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

– the second inverse projection, $\Pi_z(x = y \odot_f z)$.

A binary constraint of the form $x \odot_f y$, where $\odot_f$ is a relational operator among `==`, `!=`, `<`, `<=`, `>`, and `>=`, is decomposed into two projection functions: $\Pi_x(x \odot_f y)$ and $\Pi_y(x \odot_f y)$. The computation of the approximation of these projection functions is mainly derived from interval arithmetic and benefits from floating-point numbers being a totally ordered finite set. FPCS also implements stronger consistencies—for example, *kB*-consistencies (Lhomme, 1993)—to deal with the classical issues of multiple occurrences and to reduce more substantially the bounds of the domains of the variables.

The floating-point domains handled by FPCS 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.

On our experiments, 3*B*-consistency pruning worked well with FPCS whereas 2*B*-consistency was not strong enough to reduce the domains computed by FLUCTUAT.

5.2 Comparison with FLUCTUAT for value analysis

We report here experiments on a set of academic programs with conditionals, non-linearities, and loops. The experiments show that RAICP yields more accurate results than FLUCTUAT alone on these benchmarks. Computation times are not really meaningful on these benchmarks.

5.2.1 Conditionals

Function `gsl_poly_solve_quadratic` (see Fig. 7) comes from the GNU scientific library. It computes the two real roots of a quadratic equation $ax^2 + bx + c$ and puts the results in variables `x0` and `x1`. The function `gsl_poly_solve_quadratic` contains several conditional statements for which abstract domains need to be intersected with the condition of the conditional statement.

Table 1 shows analysis times and approximations of the domains of variables `x0` and `x1` for a given configuration of the input variables. The first two rows present the results of FLUCTUAT and RAICP (with REALPAVER) over the real numbers. The next two rows present the results of FLUCTUAT and RAICP (with FPCS) over the floating-point numbers. FLUCTUAT's over-approximation is so large that it does not give any information on the domain of the roots, whereas RAICP drastically reduce these domains both over $\mathbb{R}$ and $\mathbb{F}$.

5.2.2 Non-linearity

The abstract domains of FLUCTUAT use affine forms that 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 very

```
/* Pre-condition : a ∈ [−1.0, 1.0],
                   b ∈ [0.01, 1.0],
                   c ∈ [0.01, 1.0]   */
int gsl_poly_solve_quadratic (double a, double b, double c,
                                       double *x0, double *x1) {
  if (a == 0) /* Handle linear case */ {
    if (b == 0) {
      return 0;
    } else {
      *x0 = -c / b;
      return 1;
    };
  }

  {
    double disc = b * b - 4 * a * c;

    if (disc > 0) {
      if (b == 0) {
        double r = sqrt (-c / a);
        *x0 = -r;
        *x1 = r;
      } else {
        double sgnb = (b > 0 ? 1 : -1);
        double temp = -0.5 * (b + sgnb * sqrt (disc));
        double r1 = temp / a ;
        double 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;
    }
  }
}
```

Fig. 7 Program `gsl_poly_solve_quadratic` from the GNU Scientific Library

roughly. FPCS handles the non-linear expressions better. This is illustrated on function `sinus` (see Table 2, column `sinus`). This function computes the 7th-order Taylor series of function sinus: $x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040}$.

Table 1 Domains of the roots of the `gsl_poly_solve_quadratic` function

		$a \in [-1,1]$ $b \in [0.01,1]$ $c \in [0.01,1]$		
		`x0`	`x1`	Time
$\mathbb{R}$	FLUCTUAT	$[-\infty,\infty]$	$[-\infty,\infty]$	0.062 s
	RAICP	$[-\infty,0]$	$[-200.1,\infty]$	2.11 s
$\mathbb{F}$	FLUCTUAT	$[-\infty,\infty]$	$[-\infty,\infty]$	0.062 s
	RAICP	$[-\infty,0]$	$[-312.51,\infty]$	0.97 s

Table 2 Domain of the return value of `sinus` function

		$x \in [-1,1]$	
		Domain	Time
$\mathbb{R}$	FLUCTUAT	$[-1.009,1.009]$	0.051 s
	RAICP	$[-0.842,0.842]$	0.91 s
$\mathbb{F}$	FLUCTUAT	$[-1.009,1.009]$	0.051 s
	RAICP	$[-0.855,0.85]$	0.78 s

Table 3 Domain of the return value of the `sqrt` and `bigLoop` functions

		sqrt #1: $x \in [4.5,5.5]$		sqrt #2: $x \in [5,10]$		bigLoop	
		Domain	Time	Domain	Time	Domain	Time
$\mathbb{R}$	FLUCTUAT	$[2.116,2.354]$	0.062 s	$[2.098,3.435]$	0.079 s	$[-\infty,\infty]$	0.064 s
	RAICP	$[2.121,2.346]$	0.97 s	$[2.232,3.165]$	1.06 s	$[0,10]$	1.48 s
$\mathbb{F}$	FLUCTUAT	$[2.116,2.354]$	0.062 s	$[-\infty,\infty]$	0.079 s	$[-\infty,\infty]$	0.064 s
	RAICP	$[2.120,2.351]$	1.5 s	$[2.232,3.193]$	4.97 s	$[0,10]$	1.35s s

5.2.3 Loops

FLUCTUAT unfolds loops a bounded number of times[11] before applying a widening operator to find a fixed point for the domains at the end of the loop. In RAICP, by default, we let FLUCTUAT compute the domains for a loop. RAICP can also unfold loops until either the exit condition of the loop becomes true or a given bound is reached. In the latter case, we rely again on FLUCTUAT to compute the domains for the loop after the unfolding process.

Program `sqrt` (see Fig. 8a) relies on the so-called Babylonian method that computes an approximate value, with an error of 1×10^{-2}, of the square root of a number greater than 4 (see Table 3). FLUCTUAT obtains accurate results except in the second configuration over $\mathbb{F}$ where it could not achieve any reduction. In this second configuration RAICP shrinks the domain over $\mathbb{F}$ to $[2.232,3.193]$ because the pruning achieved by the CP solver is strong enough to compute a sharp approximation of the remainder of the loop with FLUCTUAT. Note that the CP solver works here on a constraint system derived from the initialization statements and the first ten unfoldings of the loop.

Program `bigLoop` (see Fig. 8b) contains very simple non-linear expressions followed by a loop that iterates one million times. FLUCTUAT alone fails to analyze accurately the loop in this program because of the over-approximation of the non-linear expressions before the loop. CP techniques alone run out of time and memory

[11] Default value is ten times.

```
/* Pre-condition: x ∈ [4.5,5.5] */
double sqrt(double x) {
    double xn, xn1;

    xn = x/2.0;
    xn1 = 0.5*(xn + x/xn);

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

(a)

```
/* Pre-condition: x ∈ [0,10]
                  N ∈ [1,1000000] */
double bigLoop(double x, int N) {
  double a = 0.1;
  int i = 1;
  double y = x*x-x;

  if (y < 0) {
    if (x > 1.2) {
      a = -2;
    }
  }

  while (N > i) {
    x = a * x;
    i = i + 1;
  }

  return x;
}
```

(b)

Fig. 8 Programs (a) `sqrt` with input domain #1 and (b) `bigLoop`

since it is far too expensive to unfold completely such loops. Inversely, CP techniques computed a good approximation of the non-linear expressions at the beginning of the program. That's why RAICP refined significantly the domains of the variables. This example illustrates well the potential benefits of a tight cooperation between CP and AI techniques.

5.3 Property checking on academic benchmarks

We used RAICP to check simple assertions that state numeric bounds on floating-point program variables. These assertions come from benchmarks proposed by D'Silva et al (2012) to evaluate CDFL[12]. CDFL is a program analysis tool that embeds the interval abstract domain in the conflict driven clause learning algorithm of a SAT solver. The benchmarks are derived from 12 programs by varying the input variable domains, the loop bounds, and the constants in the properties to check. All the programs come from academic numerical algorithms, except `Sac` which is generated from a Simulink controller model. We discarded 2 out of 57 benchmarks: one that is related to integers only, and another one that merge integers and floats in the same expressions.

On these benchmarks, CDFL was much more efficient than CBMC and much more accurate than ASTRÉE for approximating floating-point variable domains (D'Silva et al, 2012). Table 4 reports the results of RAICP, FLUCTUAT and CDFL on these benchmarks. RAICP is on average 3.5 times slower than FLUCTUAT used alone but

[12] These benchmarks are available at `http://www.cprover.org/cdfpl`

Table 4 Execution times and number of false alarms of CDFL, FLUCTUAT and RAICP

	CDFL	FLUCTUAT	RAICP
Total execution time	153.02 s	12.94 s	49.79 s
False alarms	0	11	0

it is much more accurate than FLUCTUAT: FLUCTUAT produced 11 false alarms whereas RAICP successfully eliminated all these false alarms and reported correctly all the 33 true properties.
In other words, RAICP is as effective as CDFL on these benchmarks for checking assertions that state numeric bounds on floating-point program variables. On top of it, RAICP is on average three-fold faster than CDFL.

Note that all of these systems may produce false alarms in the general case.

5.4 Property checking on an industrial benchmark

Finally, we applied RAICP to an industrial system provided by Geensys/Dassault Systems. The anti-lock braking system (ABS) is a real time software application running on an electronic unit embedded in a car. The system was designed with Simulink and the embedded code was automatically generated from the Simulink model. The code contains computations over integer and floating-point variables and consists of an infinite loop that repeatedly reads inputs and computes the output every 0.01 s. Since we bound the number of unfoldings of the real-time loop, we can only check assertions for a limited service time of the system. ABS will be active for at most 20 s when braking on a wet road with a maximum vehicle speed of 180 kilometers per hour and a cautious deceleration value of 2.5 meters per squared second. This means that at most 2 000 unfoldings of the real-time loop are required.

ABS prevents wheel lock when braking. It monitors wheel speed through sensors and acts on an hydraulic valve. ABS looks for the tendency to lock of a wheel. It computes the skidding rate of the slowest wheel as $r_s = 1 - \frac{v_{\text{slow}}}{v_{\text{car}}}$ where v_{slow} is the speed of the slowest wheel and v_{car} is the speed of the car. ABS tries to maintain the optimal rate $r_o = 20\%$[13]. When r_s is greater than r_o, ABS starts controlling braking.

Our industrial partner had specified property P_I as follows: ABS enters controlled braking as soon as skidding rate is greater than 20%. The state of the ABS is an internal variable, `abs_state`, that can take two predefined values: `CONTROLLED` or `UNCONTROLLED`. The assertion to be checked for P_I is then:

$$(v_{\text{slow}} < 0.8 * v_{\text{car}}) \implies (\texttt{abs_state} = \texttt{CONTROLLED})$$

We tried to experiment CBMC, FLUCTUAT, and RAICP on the checking of property P_I but we did not manage to run CDFL on these benchmarks. For checking this property, the user was only interested by the behavior of the program with a semantics over the floating-point numbers. We fixed a time-out of one hour. Table 5 shows that RAICP could prove quite efficiently that property P_I holds up to the fixed 2 000 unfoldings limit. Property P_I trivially holds at the first unfolding which corresponds

[13] Actually, optimal rate depends on the road surface and varies between 30% and 10% .

Table 5 Validity results and execution times of CBMC, FLUCTUAT and RAICP on property P_I when varying unfoldings

Number of unfoldings	CBMC		FLUCTUAT		RAICP	
	Validity	Time	Validity	Time	Validity	Time
1	valid	0.19 s	valid	0.05 s	valid	0.90 s
2	valid	0.36 s	unknown	0.06 s	valid	0.98 s
100	-	> 3600 s	unknown	1.14 s	valid	17.99 s
1000	-	-	unknown	55.77 s	valid	302.35 s
2000	-	-	unknown	316.04 s	valid	1167.57 s

to the initialization phase of the ABS. CBMC reached the time limit after few dozen unfoldings. This is probably due to the fact that CBMC falls into a slow convergence process. FLUCTUAT is very fast but computes such coarse over-approximations that one cannot determine whether the property holds or not.

6 Conclusion

In this paper, we introduced a new approach for computing tight intervals of floating-point variables of C programs. The prototype of RAICP we developed relies on the static analyzer FLUCTUAT, on the floating-point solver FPCS, and the real number solver REALPAVER. Thanks to these solvers, RAICP can exploit the refutation capabilities of constraint techniques to refine the domains computed by FLUCTUAT.

This integration of AI and CP works well because the approximation of variable bounds computed by AI is often small enough to prune the domains efficient with partial consistencies. Even though the same domain reductions could sometimes be achieved without starting from the approximation computed by FLUCTUAT, our experiments show that the approximation computed by FLUCTUAT is not only required in programs with loops, but can also be very useful for programs containing expressions with multiple occurrences of some variables. In FLUCTUAT, sets of affine forms abstract non-linear expressions and constraints. These sets approximate better linear constraint systems than the boxes used in interval-based constraint solvers. Nevertheless, they are less adapted for non-linear constraint systems where filtering techniques used in numeric CP solvers offer a more flexible and extensible framework.

Of course, RAICP is slower than FLUCTUAT but is still quite efficient on programs that are representative of the difficulties of FLUCTUAT (conditional constructs and non-linearities). The computed approximations both over the real numbers and the floating-point numbers are much sharper than the ones computed by AI techniques. The user has therefore more facilities to identify suspicious values for which the behavior of the program over the floating-point numbers is different from the behavior the user could expect over the real numbers. Experiments on a significant set of benchmarks showed also that RAICP is as accurate and faster than CDFL, a state-of-the-art tool for bound analysis and assertion checking on programs with floating-point computations. These experiments demonstrate that even limited domain reductions can be critical for discarding false alarms.

Further work concerns a tighter integration of abstract interpretation and constraint solvers and the generation of counter-examples, a critical issue for debugging. For instance, the integration of AI and CP could be done at the abstract domain level instead of the interval domain level. Likewise, the constraint systems generated by RAICP could be used for generating counter-examples when we cannot prove that a property holds. Extending the capabilities of RAICP for handling pointers would also be useful. Likewise, RAICP could be improved for efficiently handling arithmetic expressions over other domains like integers. Note that the latter extension would require to develop a dedicated constraint solver since standard CP solvers over finite domains are not well adapted for handling large domains, and they do not take into account some specific features of arithmetic over the integer, for example, modulo operations.

Acknowledgements The authors gratefully acknowledge Sylvie Putot, Éric Goubault and Franck Védrine for their advice and help on using FLUCTUAT. We also gratefully acknowledge Laurent Arditi and Hélène Collavizza who carefully read this paper and provided critical comments.

References

Ayad A, Marché C (2010) Multi-prover verification of floating-point programs. In: IJCAR, Springer, LNCS, vol 6173, pp 127–141

Barnett M, Leino KRM (2005) Weakest-precondition of unstructured programs. Information Processing Letters 93(6):281–288

Boldo S, Filliâtre JC (2007) Formal verification of floating-point programs. In: 18th IEEE Symposium on Computer Arithmetic, IEEE, pp 187–194

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

Brillout A, Kroening D, Wahl T (2009) Mixed abstractions for floating-point arithmetic. In: 9th International Conference on Formal Methods in Computer-Aided Design, IEEE, pp 69–76

Codognet P, Filé G (1992) Computations, abstractions and constraints in logic programs. In: International Conference on Computer Languages (ICCL'92), IEEE, pp 155–164

Collavizza H, Rueher M, Hentenryck PV (2010) A constraint-programming framework for bounded program verification. Constraints Journal 15(2):238–264

Cousot P (2001) Abstract interpretation based formal methods and future challenges. In: Wilhelm R (ed) Informatics - 10 Years Back. 10 Years Ahead., Springer, Lecture Notes in Computer Science, vol 2000, pp 138–156

Cousot P, Cousot R (1977) Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, ACM, pp 238–252

Cousot P, Cousot R (1979) Systematic design of program analysis frameworks. In: POPL, ACM Press, pp 269–282

Cousot P, Cousot R, Feret J, Miné A, Mauborgne L, Monniaux D, Rival X (2007) Varieties of static analyzers: A comparison with ASTRÉE. In: 1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, IEEE, pp 3–20

Delmas D, Goubault E, Putot S, Souyris J, Tekkal K, Védrine F (2009) Towards an industrial use of FLUCTUAT on safety-critical avionics software. In: FMICS, Springer, LNCS, vol 5825, pp 53–69

Denmat T, Gotlieb A, Ducassé M (2007) An abstract interpretation based combinator for modeling while loops in constraint programming. In: Principles and Practices of Constraint Programming (CP'07), Springer Verlag, LNCS, vol 4741, pp 241–255

de Dinechin F, Lauter CQ, Melquiond G (2011) Certifying the floating-point implementation of an elementary function using Gappa. IEEE Transactions on Computers 60(2):242–253

D'Silva V, Haller L, Kroening D, Tautschnig M (2012) Numeric bounds analysis with conflict-driven learning. In: Proc. TACAS, Springer, Lecture Notes in Computer Science, vol 7214, pp 48–63

Ghorbal K, Goubault E, Putot S (2010) A logical product approach to zonotope intersection. In: CAV, Springer, LNCS, vol 6174, pp 212–226

Girard A (2005) Reachability of uncertain linear systems using zonotopes. In: HSCC, Springer, Lecture Notes in Computer Science, vol 3414, pp 291–305

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

Goubault E, Putot S (2006) Static analysis of numerical algorithms. In: SAS, Springer, LNCS, vol 4134, pp 18–34

Granvilliers L, Benhamou F (2006) Algorithm 852: RealPaver: an interval solver using constraint satisfaction techniques. ACM Transactions on Mathematical Software 32(1):138–156

Harrison J (1999) A machine-checked theory of floating-point arithmetic. In: TPHOLs, Springer-Verlag, LNCS, vol 1690, pp 113–130

Hentenryck PV, Mcallester D, Kapur D (1997) Solving polynomial systems using a branch and prune approach. SIAM Journal on Numerical Analysis 34:797–827

Lhomme O (1993) Consistency techniques for numeric CSPs. In: 13th International Joint Conference on Artificial Intelligence, pp 232–238

Marre B, Michel C (2010) Improving the floating point addition and subtraction constraints. In: CP, Springer, LNCS, vol 6308, pp 360–367

Michel C (2002) Exact projection functions for floating-point number constraints. In: 7th International Symposium on Artificial Intelligence and Mathematics

Michel C, Rueher M, Lebbah Y (2001) Solving constraints over floating-point numbers. In: CP, Springer Verlag, LNCS, vol 2239, pp 524–538

Pelleau M, Miné A, Truchet C, Benhamou F (2013) A constraint solver based on abstract domains. In: Giacobazzi R, Berdine J, Mastroeni I (eds) 14th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2013), Springer, Lecture Notes in Computer Science, vol 7737, pp 434–454

Rossi F, van Beek P, Walsh T (eds) (2006) Handbook of Constraint Programming, 1st edn. Elsevier Science