VACSIM

Validation de la commande des systèmes critiques par couplage simulation et méthodes d'analyse formelles

Tâche 5

Validation formelle de propriétés quantitatives :
Approche par contraintes

Livrable L5.2

Vérification de propriétés quantitatives par résolution de systèmes de contraintes sur les flottants

Version 1

Appel :	**PROGRAMME INGENIERIE NUMERIQUE & SECURITE 2011**
Numéro d'agrément :	A**NR-11-INSE-004**
Thématique:	**Systèmes embarqués et ingénierie du logiciel**
Objectif:	**Validation par simulation de partie opérative**
Date de démarrage du projet:	**01.10.2011**
Durée :	**42 mois**






Synthèse

Le projet **VACSIM** (Validation de la commande des systèmes critiques par couplage simulation et méthodes d'analyse formelle), référencé ANR-11-INSE-004, étudie les avantages respectifs des techniques de simulation, en incluant des modèles des processus commandés, et des méthodes d'analyse formelles, pour la validation de la commande des systèmes critiques. Ce projet est structuré en 6 tâches.

La tâche 5, « Validation formelle de propriétés quantitatives : Approche par contraintes », a pour objectif de contribuer à l'avancée des techniques de validation par résolution de systèmes de contraintes. Elle est découpée en deux sous-tâches qui abordent des aspects complémentaires de la la validation des systèmes critiques : la vérification de propriétés quantitatives pour la sous-tâche T5.1 et la localisation d'erreur pour la sous-tâche T5.2.

Ce livrable L5.2 du projet VACSIM est issu des travaux de la sous-tâche T5.1 : « Génération et résolution des systèmes de contraintes ». Ce livrable décrit la méthode rAiCp pour la vérification de propriétés quantitatives par résolution de systèmes de contraintes sur les flottants. Il comprend la réalisation d'un prototype logiciel implémentant cette méthode et l'évaluation du prototype sur des exemples académiques et sur une étude de cas fournie par Dassault Systèmes, un partenaire du projet VACSIM. Ce livrable décrit aussi l'algorithme FPLP pour le filtrage des contraintes sur les flottants qui exploite des relaxations sur les nombres réels.

Les travaux effectués pour ce livrable visent à lever deux verrous scientifiques identifiés dans le projet VACSIM. Le premier verrou concerne la résolution de systèmes de contraintes non-linéaires sur les flottants. Pour traiter ce problème, nous nous sommes appuyés et avons amélioré un solveur de contraintes sur les flottants développé dans l'équipe (FPCS) : nouveaux algorithmes de filtrage, nouvelles stratégies de recherche, portage sur architecture 64 bits. Nous avons aussi mis au point de nouvelles techniques de résolution des contraintes sur les flottants qui se basent sur une approximation sur les réels des opérations en virgule flottante (FPLP). Le second verrou concerne le passage à l'échelle de la méthode de vérification de propriétés quantitatives pour des systèmes embarqués temps-réel. Pour répondre à cette problématique, nous avons conçu une approche hybride d'analyse des valeurs des variables des programmes qui couple techniques d'interprétation abstraite et techniques de programmation par contraintes (rAiCp). Le passage à l'échelle est aussi amélioré par une stratégie d'exploration du graphe de flot de contrôle des programmes qui fusionne les valeurs des variables à certains nœuds de jonction du graphe pour limiter l'explosion du nombre de chemins d'exécution des programmes à explorer.

Le livrable se présente sous la forme de trois articles joints en annexe de ce document et des liens vers les pages Web des outils développés suivants :

- rAiCp, le prototype logiciel implémentant la méthode de vérification de propriétés quantitatives par résolution de systèmes de contraintes sur les flottants (https://sourcesup.renater.fr/cpbpv).
- FPCS, le solveur de contraintes sur les flottants utilisé dans rAiCp (http://www.i3s.unice.fr/~cpjm/misc/fpcs.html).

Ce livrable L5.2 a été rédigé par l'I3S.

Annexes

Les annexes sont constituées de trois articles :

- Le premier article, en Annexe A p. 4, décrit la méthode de vérification de propriétés quantitatives par résolution de systèmes de contraintes sur les flottants rAiCp. Il s'agit d'un article publié dans les actes de la 18ème conférence internationale « Principles and Practice of Constraint Programming » [1].
- Le second article, en Annexe B p. 20, donne une présentation plus détaillée de la méthode rAiCp ainsi que du prototype l'implantant. L'article décrit aussi l'application du prototype à un cas d'étude industriel. Il s'agit d'un article soumis pour publication et déposé sur HAL (http://hal.archives-ouvertes.fr/hal-00860681).
- Le troisième article, en Annexe C p. 45, décrit l'algorithme de filtrage des contraintes sur les flottants FPLP. Il s'agit d'un article publié dans les actes de la 18ème conférence internationale « Principles and Practice of Constraint Programming » [2].

Références

[1] Refining abstract interpretation based value analysis with constraint programming techniques

Olivier Ponsini, Claude Michel and Michel Rueher.

CP2012 (18th International Conference on Principles and Practice of Constraint Programming), LNCS 7514, pp. 593-607, 2012

[2] Boosting local consistency algorithms over floating-point numbers

Said Mohammed Belaid, Claude Michel and Michel Rueher.

CP2012 (18th International Conference on Principles and Practice of Constraint Programming), LNCS 7514, pp. 127-140, 2012

Annexe A

Titre :

Refining abstract interpretation based value analysis with constraint programming techniques

Auteurs :

Olivier Ponsini, Claude Michel et Michel Rueher

Résumé :

L'analyse des valeurs des variables par interprétation abstraite est une approche classique du problème de la vérification des programmes comportant des calculs en virgule flottante. Cependant, les outils de l'état de l'art calculent une sur-approximation des valeurs des variables qui peut être très grossière. Dans cet article, nous montrons que les solveurs de contraintes peuvent affiner significativement les approximations calculées par les outils basés sur l'interprétation abstraite. Nous introduisons une approche hybride qui combine techniques d'interprétation abstraite et de programmation par contraintes en une unique analyse statique et automatique. Le système que nous avons développé, rAiCp, est substantiellement plus précis que Fluctuat, un analyseur statique de l'état de l'art. Ainsi, rAiCp a pu éliminer les treize fausses alarmes générées par Fluctuat sur un jeu de tests standard.

Accepted for publication at CP'12 (18th International Conference on Principles and Practice of Constraint Programming)

Refining abstract interpretation based value analysis with constraint programming techniques*

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. Abstract interpretation based value analysis is a classical approach for verifying programs with floating-point computations. However, state-of-the-art tools compute an over-approximation of the variable values that can be very coarse. In this paper, we show that constraint solvers can significantly refine the approximations computed with abstract interpretation tools. We introduce a hybrid approach that combines abstract interpretation and constraint programming techniques in a single static and automatic analysis. rAiCp, the system we developed is substantially more precise than Fluctuat, a state-of-the-art static analyser. Moreover, it could eliminate 13 false alarms generated by Fluctuat on a standard set of benchmarks.

Key words: Program verification, Floating-point computation, Constraint solvers over floating-point numbers, Constraint solvers over real number intervals, Abstract interpretation-based approximation

1 Introduction

Programs with floating-point computations control complex and critical physical systems in various domains such as transportation, nuclear energy, or medicine. Floating-point computations are an additional source of errors and famous computer bugs are due to errors in floating-point computations, e.g., the Patriot missile failure. Floating-point computations are usually derived from mathematical models on real numbers [14]. However, real and floating-point computation models are different: for the same sequence of operations, floating-point numbers do not behave identically to real numbers. For instance, with binary floating-point numbers, some decimal real numbers are not representable (e.g., 0.1 has no exact representation), 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).

* This work was partially supported by ANR VACSIM (ANR-11-INSE-0004), ANR AEOLUS (ANR-10-SEGI-0013), and OSEO ISI PAJERO projects.

Value analysis is often used to check the absence of run-time errors, such as invalid integer or floating-point operations, as well as simple user assertions [8]. Value analysis can also help with estimating the accuracy of floating-point computations with respect to the same sequence of operations in an idealized semantics of real numbers. Existing automatic tools are mainly based on abstract interpretation techniques. For instance, Fluctuat [9], a state-of-the-art static analyzer, computes an over-approximation of the domains of the variables for a C program considered with a semantics on real numbers. It also computes an over-approximation of the error due to floating-point operations at each program point. However, these over-approximations may be very coarse even for usual programming constructs and expressions. As a consequence, numerous false alarms[1]—also called false positives—may be generated.

In this paper, we introduce a hybrid approach for the value analysis of floating-point programs that combines abstract interpretation (AI) and constraint programming techniques (CP). We show that constraint solvers over floating-point and real numbers can significantly refine the over-approximations computed by abstract interpretation. rAiCp, the system we developed, uses both Fluctuat and the following constraint solvers:

- RealPaver [17], a safe and correct solver for constraints over real numbers,
- FPCS [21, 20], a safe and correct solver for constraints over floating-point numbers.

Experiments show that rAiCp is substantially more precise than Fluctuat, especially on C programs that are difficult to handle with abstract interpretation techniques. This is mainly due to the refutation capabilities of filtering algorithms over the real numbers and the floating-point numbers used in rAiCp. rAiCp could also eliminate 13 false alarms generated by Fluctuat on a set of 57 standard benchmarks proposed by D'Silva et al [12] to evaluate CDFL, a program analysis tool that embeds an abstract domain in the conflict driven clause learning algorithm of a SAT solver. Moreover, rAiCp is on average at least 5 times faster than CDFL on this set of benchmarks.

Section 2 illustrates our approach on a small example. Basics on the techniques and tools we use are introduced in Section 3. Next section is devoted to related work. Section 5 details our approach whereas experiments are analysed in Section 6.

2 Motivation

In this section, we illustrate our approach on a small example. The program in Fig. 1 is mentioned in [13] as a difficult program for abstract interpretation based

[1] A false alarm corresponds to the case when the abstract semantics intersects the forbidden zone, i.e., erroneous program states, while the concrete semantics does not intersect this forbidden zone. So, a potential error is signaled which can never occur in reality (see `http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html`).

Fig. 1. Example 1.

```
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; }
```

analyses. On floating-point numbers, as well as on real numbers, this function returns a value in the interval $[0, 3]$. Indeed, from the conditional statement of line 4, we can derive the following information:

- `if` branch: $x = 0$ or $x \geq 1$, and thus $y \in [0, 1]$ at the end of this branch;
- `else` branch: $x \in]0, 1[$, and thus $y \in]2, 3[$ at the end of this branch.

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 numbers 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 $[0, 10]$.

We propose here to compute an approximation of the domains in both execution paths. On this example, CSP filtering techniques are strong enough to reduce the domains of the variables. 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 $y_0 < 0$, the interval solver over the real numbers we use 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, our 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 explore the control flow graph (CFG) of a program and stop each time two branches join. There, we build one constraint system per branch that reaches the join point. Then, we use filtering techniques on these systems to reduce the domains of the variables computed by FLUCTUAT at this join point. Exploration goes on with the reduced domains. CFG exploration is performed on-the-fly. Branches are cut as soon as an inconsistency of the constraint system

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

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

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

is detected by a local filtering algorithm. 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. Analysis times are very similar. In [13], the authors proposed an extension to the zonotopes—named *constrained zonotopes*—which attempts to overcome the issue of 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 (the upper bound is reduced to 9.72) but remains less precise than the one computed with REALPAVER.

3 Background

Before going into the details, we recall basics on abstract interpretation and FLUCTUAT, as well as on the constraint solvers REALPAVER and FPCS used in our implementation.

Abstract interpretation[3] consists in considering an abstract semantics, that is a super-set of the concrete program semantics. The abstract semantics covers all possible cases, thus, if the abstract semantics is safe (i.e. does not intersect the forbidden zone) then so is the concrete semantics.

FLUCTUAT is a static analyzer for C programs specialized in estimating the precision of floating-point computations[4] [9]. FLUCTUAT compares the behavior of the analyzed program over real numbers and over floating-point numbers. In other words, it allows to specify 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 ;

[3] See `http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html` for a nice informal introduction.

[4] FLUCTUAT is developed by CEA-LIST (`http://www-list.cea.fr/validation_en.html`) and was successfully used for industrial applications of several tens of thousands of lines of code in transportation, nuclear energy, or avionics areas.

– the contribution of the input variables to the error associated with variable v.

FLUCTUAT proceeds by abstract interpretation. It uses the weakly relational abstract domain of zonotopes [15]. 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 allows to use arbitrary precision numbers or to subdivide up to two input variable intervals. However, over-approximations computed by FLUCTUAT may be very large because the abstract domains do not handle well conditional statements and non-linear expressions.

REALPAVER is an interval solver for numerical constraint systems over the real numbers[5] [17]. Constraints can be non-linear and can contain the usual arithmetic operations and 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: box, hull, and $3B$ consistencies. 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 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 this is safe since no solution is lost.

FPCS is a constraint solver designed to solve a set of constraints over floating-point numbers without losing any solution [21, 20]. It uses $2B$-consistency [19] along with projection functions adapted to floating-point arithmetic [22, 4].

The main difficulty lies in computing inverse projection functions that keep all the solutions. 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:

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

– the direct projection, $\Pi_x(x = y \odot_f z)$;
– the first inverse projection, $\Pi_y(x = y \odot_f z)$;
– 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 inspired from interval arithmetic and benefits from floating-point numbers being a totally ordered finite set.

FPCS also implements stronger consistencies—e.g., kB-consistencies [19]—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.

4 Related work

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 capture 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 are not tailored for automatically generating a counter-example, that is to say, input variable values that violate some assertion in a program. Astrée [8] 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 run-time errors, that is the absence of behavior not defined by the programming language, e.g., division by zero, arithmetic overflow. As said before, Fluctuat 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].

Proof assistants like Coq [3] or HOL [18] allow their users to formalize floating-point arithmetic. Proofs of program properties are done manually in the proof assistants which guarantee proof correctness. Even though some parts of the proofs may be automatized, these tools usually require a lot of user interaction. Moreover, when a proof strategy fails to prove a property, the user often does not know whether the property is false or another strategy could prove it. Like abstract interpretation, proof assistants usually do not provide automatic generation of counter-examples for false properties. The Gappa tool [11] 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 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 in [5]. 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. D'Silva et al [12] developed recently CDFL, a program analysis tool that embeds an abstract domain in a conflict driven clause learning algorithm of a SAT solver. CDFL is based on a sound and complete analysis for determining the range of floating-point variables in control software. In [12] the authors state that CDFL is more than 200 times faster than CBMC. In Section 6 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 (e.g., [6]) and recent work investigate the use of abstract interpretation and abstract domains in the context of constraint programming. In [10], the authors introduce a new global constraint to model iterative arithmetic relations between integer variables. The associated filtering algorithm is based on abstract interpretation over polyhedra. In [23], the authors propose to use the octagonal abstract domain, which proved efficient in abstract interpretation, to represent the variable domains in a continuous constraint satisfaction problem. Then, they generalize local consistency and domain splitting to this octagonal representation. In this paper, we show how abstract interpretation and constraint programming techniques can complement each other for the static analysis of floating-point programs.

5 rAiCp, a hybrid approach

The approach we propose here is based on successive explorations and merging steps. More precisely, we call Fluctuat to compute a first approximation of the variable values at the first program node of the CFG where two branches join. Then, we build one constraint system per branch and use filtering techniques to reduce the domains of the variables computed by Fluctuat. Reduced domains obtained for each branch are merged and exploration goes on with the result of the merge.

5.1 Control flow graph exploration

The CFG of a program is explored using a forward analysis going from the beginning to the end of the program. Statements are converted into DSA (Dynamic

Single Assignment) form where each variable is assigned exactly once on each program path [2]. Lengths of the paths are bounded since loops are unfolded a bounded number of times, after which they are abstracted by the domains computed by abstract interpretation. At any point of an execution path, the possible states of a program are represented by a constraint system over the program variables. Domains of the variables are intervals over the real numbers in the constraint store of RealPaver; domains are intervals over the floating-point numbers that correspond to the `int`, `float` and `double` machine types of the C language[6] in the constraint store of FPCS. Each program statement adds new constraints and variables to these constraint stores. This technique for representing programs by constraint systems was introduced for bounded verification of programs in CPBPV [7]. The implementation of the approach proposed in this paper relies on libraries developed for CPBPV.

CFG exploration is performed on-the-fly and unreachable branches are interrupted as soon as an inconsistency is detected in the constraint store. We collect constraints between two join points in the CFG. If, for all executable paths between these points, the constraint systems are inconsistent for some interval I of an output variable x, then we can remove the interval I from the domain of x. Note that we differentiate between program *input* variables, whose domains cannot be reduced, and program *output* variables, whose domains depend on the program computations and input variable domains, and thus can be reduced.

Merging program states at each join point not only allows a tight cooperation between Fluctuat and the constraint solvers but also limits the number of executable paths to explore.

5.2 Filtering techniques

We use constraint filtering techniques for two different purposes in rAiCp:

- elimination of unreachable branches during CFG exploration;
- reduction of the domain of the variables at CFG join points.

On floating-point numbers constraint systems, we perform $3B(w)$-consistency filtering with FPCS; on real numbers constraint systems, we perform a $BC5$-consistency filtering in paving mode with RealPaver[7].

6 Experiments

In this section, we compare in detail Fluctuat and rAiCp on programs that are representative of Fluctuat limitations. We also compare rAiCp to a state-

[6] Note that the behavior of programs containing floating-point computations may vary with the programming language or the compiler, but also, with the operating system or the hardware architecture. We consider here 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.

[7] $BC5$-consistency is a combination of interval Newton method, hull-consistency and box-consistency.

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: $a, b, c \in [1, 1 \times 10^6]$		
		`x0`	`x1`	Time	`x0`	`x1`	Time
$\mathbb{R}$	FLUCTUAT	$[-\infty, \infty]$	$[-\infty, \infty]$	0.14 s	$[-2 \times 10^6, 0]$	$[-1 \times 10^6, 0]$	0.14 s
	RAICP	$[-\infty, 0]$	$[-8.006, \infty]$	1.55 s	$[-1 \times 10^6, 0]$	$[-5.186 \times 10^5, 0]$	0.58 s
$\mathbb{F}$	FLUCTUAT	$[-\infty, \infty]$	$[-\infty, \infty]$	0.13 s	$[-2 \times 10^6, 0]$	$[-1 \times 10^6, 0]$	0.13 s
	RAICP	$[-\infty, 0]$	$[-8.125, \infty]$	0.39 s	$[-1 \times 10^6, 0]$	$[-3\,906.26, 0]$	0.39 s

of-the-art tool, CDFL on the benchmarks provided by the authors of the latter system.

All results were obtained on an Intel Core 2 Duo at 2.8 GHz with 4 GB of memory running Linux using FLUCTUAT version 3.8.73, REALPAVER version 0.4 and the downloadable version of CDFL. All the programs are available at `http://users.polytech.unice.fr/~rueher/Benchs/RAICP`.

6.1 Improvements over Fluctuat

We show here how our approach improves the approximations computed by FLUCTUAT on programs with conditionals, non-linearities, and loops.

Conditionals: The first benchmark concerns conditional statements, for which abstract domains need to be intersected with the condition of the conditional statement. The function `gsl_poly_solve_quadratic` comes from the GNU scientific library and contains many of these conditional statements. It computes the real roots of a quadratic equation ax^2+bx+c and puts the results in variables `x0` and `x1`.

Table 2 shows analysis times and approximations of the domains of variables `x0` and `x1` for two configurations 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.

In the first configuration, 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 $\mathbb{F}$. However, intersection of abstract domains has not always such a significant impact on the bounds of all domains as illustrated by the domain over $\mathbb{F}$ of `x0` in the second configuration.

To increase analysis precision, FLUCTUAT allows to divide the domains of at most two input variables into a given number of sub-domains. Analyses are then run over each combination of sub-domains and the results are merged. Finding appropriate subdivisions of the domains is a critical issue: subdividing may not improve the analysis precision, but it always increases the analysis time. Table 3 reports the results with 50 subdivisions when only one domain is divided, and 30 when two domains are divided. Over $\mathbb{R}$, in the first configuration, the subdivisions

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	$[-1 \times 10^6, 0]$	> 1 s
FLUCTUAT b subdivided	$[-\infty, \infty]$	> 1 s	$[-5 \times 10^5, 0]$	> 1 s
FLUCTUAT c subdivided	$[-\infty, \infty]$	> 1 s	$[-1 \times 10^6, 0]$	> 1 s
FLUCTUAT a & b subdivided	$[-\infty, -0]$	> 10 s	$[-1.834 \times 10^5, 0]$	> 10 s
FLUCTUAT a & c subdivided	$[-\infty, -0]$	> 10 s	$[-1 \times 10^6, 0]$	> 10 s
FLUCTUAT b & c subdivided	$[-\infty, \infty]$	> 10 s	$[-5 \times 10^5, 0]$	> 10 s

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	$[-1.009, 1.009]$	0.12 s	$[-1.168 \times 10^{37}, 1.992 \times 10^{37}]$	0.13 s
	RAICP	$[-0.842, 0.843]$	0.34 s	$[-1.144 \times 10^{36}, 1.606 \times 10^{37}]$	1.26 s
$\mathbb{F}$	FLUCTUAT	$[-1.009, 1.009]$	0.12 s	$[-1.168 \times 10^{37}, 1.992 \times 10^{37}]$	0.13 s
	RAICP	$[-0.853, 0.852]$	0.22 s	$[-1.168 \times 10^{37}, 1.992 \times 10^{37}]$	0.22 s

yield no improvement and, in the second configuration, the results are identical to those over $\mathbb{F}$.

Subdividing domains can be quite time consuming with little gains in precision:

- In the first configuration, subdivisions of the domain of a lead to a significant reduction of the domain of `x0`. 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 and no subdivision combination could reduce the domain of `x0`.

RAICP turns out to be more efficient: it often improves the precision of the approximation and requires less time than the subdividing process of FLUCTUAT. Moreover, RAICP could also take advantage of the subdivision technique.

Non-linearity: The abstract domain used by FLUCTUAT is based on 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-

Table 5. 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	$[2.116, 2.354]$	0.13 s	$[2.098, 3.435]$	0.2 s	$[-\infty, \infty]$	0.15 s
	RAICP	$[2.121, 2.346]$	0.35 s	$[2.232, 3.165]$	0.57 s	$[0, 10]$	0.8 s
$\mathbb{F}$	FLUCTUAT	$[2.116, 2.354]$	0.13 s	$[-\infty, \infty]$	0.2 s	$[-\infty, \infty]$	0.15 s
	RAICP	$[2.121, 2.347]$	0.81 s	$[2.232, 3.168]$	1.59 s	$[0, 10]$	0.7 s

linear operations are thus over-approximated. FPCS handles the non-linear expressions better. This is illustrated on the 7^{th}-order Taylor series of function sinus (see Table 4, column `sinus`).

FPCS and REALPAVER also use approximations to handle non-linear terms, and thus, are not always more precise than FLUCTUAT.The second row of Table 4 shows that RAICP could not reduce the domain computed by FLUCTUAT for the `rump` polynomial program [24], a very particular polynomial designed to outline a catastrophic cancellation phenomenon.

Loops: FLUCTUAT unfolds loops a bounded number of times[8] before applying the widening operator of abstract interpretation. The widening operator allows to find a fixed point for a loop without unfolding it completely. In RAICP, we also unfold loops a user-defined number of times, after which the loop is abstracted by the invariant computed by abstract interpretation. Note that we can also use FLUCTUAT to estimate an upper bound on the number of necessary unfoldings [16].

`sqrt` is a program based 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. For the analysis of this program with two different input domains (see Table 5), ten unfoldings are sufficient to exit the loop. Both FLUCTUAT and RAICP obtain accurate results over $\mathbb{R}$. Over $\mathbb{F}$, in the second configuration RAICP shrinks the domain to $[2.232, 3.168]$ whereas FLUCTUAT couldn't achieve any reduction.

Program `bigLoop` contains non-linear expressions followed by a loop that iterates one million times. On such programs, it is not possible to completely unfold loops. FLUCTUAT fails to analyze accurately the loop in this program because of over-approximations of the non-linear expressions. RAICP refines significantly the over-approximations computed by FLUCTUAT, even without any initial unfoldings. This example shows that a tight cooperation between CP and AI techniques can be very efficient.

Contributions of AI and CP FLUCTUAT often yields a first approximation that is tight enough to allow efficient filtering with partial consistencies. Even though the same domain reductions can sometimes be achieved without starting

[8] Default value is ten times.

from the approximation computed by FLUCTUAT (i.e., starting from $[-\infty, \infty]$), our experiments show that our approach usually benefits from the approximation computed by FLUCTUAT.

$3B$-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 with various consistencies implemented in REALPAVER: $BC5$, a combination of hull and box consistencies with interval Newton method, $HC4$, $3B$-consistency. $3B$-consistency was in general too time-consuming. $BC5$-consistency provided the best trade-off between time cost and domain reduction.

6.2 Comparison with CDFL

CDFL [12] is a program analysis tool designed for proving the absence of runtime errors in critical programs. In [12], the authors show that CDFL is much more efficient than CBMC and much more precise than ASTRÉE [8] for determining the range of floating-point variables on various programs.

We compare here RAICP and CDFL on the set of benchmarks[9] proposed in [12]. The set consists of 57 benchmarks made from 12 programs by varying the input variable domains, the loop bounds, and the constants in the properties to check. We discarded two benchmarks as they are related to integer computations which are not the focus of this work. All the programs are based on academic numerical algorithms, except `Sac` which is generated from a Simulink controller model. The program properties are simple assertions on program variable domains.

Table 6 provides the runing time of RAICP, FLUCTUAT and CDFL. RAICP was only run with FPCS since the properties and the programs are both defined over the floating-point numbers.

All three analyses may report false alarms: i.e., they may answer a property is false while it is not. Actually, RAICP and CDFL correctly reported all the 33 true properties. FLUCTUAT gave 11 false alarms that are noted with * in FLUCTUAT columns of Table 6. The domain refinements performed by RAICP successfully eliminated the false alarms produced by FLUCTUAT.

On average, RAICP is 5 times faster than CDFL for the same precision. On some benchmarks, we observe a speed-up factor of 25. On average, RAICP is 2.2 times slower than FLUCTUAT used alone but this is largely compensated by the gain in precision.

7 Conclusion

In this paper, we introduced a new approach for computing tight intervals of floating-point variables of C programs. RAICP, the prototype we developed, relies on the static analyser FLUCTUAT and on FPCS and REALPAVER, two con-

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

Table 6. Execution times (s) of CDFL, FLUCTUAT and RAICP.

	CDFL	FLUCTUAT	RAICP		CDFL	FLUCTUAT	RAICP
newton.1.1	0.5	0.12	0.62	eps_line1	0.12	0.11	0.28
newton.1.2	1.64	0.13	0.68	muller	0.13	0.11	0.2
newton.1.3	4.6	0.21	1.89	sac.10	2.49	1.25	1.6
newton.2.1	0.95	0.11	1.47	sac.20	2.46	1.38	1.75
newton.2.2	3.44	0.14	0.82	sac.30	2.49	1.39	1.68
newton.2.3	9.32	0.21	1.79	sac.40	2.47	1.38	1.68
newton.3.1	1.95	0.12*	1.3	sac.50	2.46	1.38	1.71
newton.3.2	5.61	0.13	1.13	sac.60	2.48	1.4	1.76
newton.3.3	15.9	0.22	2.35	sac.70	2.46	1.37	1.7
newton.4.1	1.07	0.12	1.74	sac.80	2.48	1.37	1.7
newton.4.2	8.4	0.13	1.82	sac.90	2.47	1.37	1.67
newton.4.3	23.63	0.22	2.49	sine.1	0.68	0.12	0.31
newton.5.1	1.76	0.12	1.83	sine.2	0.96	0.11	0.28
newton.5.2	14.61	0.13*	2.68	sine.3	0.5	0.11	0.28
newton.5.3	38.19	0.23*	4.01	sine.4	7.89	0.12*	0.3
newton.6.1	1.28	0.12	2.15	sine.5	0.68	0.12*	0.23
newton.6.2	2.33	0.13	8.85	sine.6	0.3	0.12*	0.26
newton.6.3	3.59	0.15	4.76	sine.7	0.13	0.12*	0.22
newton.7.1	1.8	0.12	2.23	sine.8	0.08	0.12	0.23
newton.7.2	1.57	0.14	1.59	square.1	0.16	0.12	0.26
newton.7.3	19.45	0.15	1.68	square.2	0.32	0.12	0.25
newton.8.1	0.41	0.11	0.86	square.3	0.7	0.11	0.25
newton.8.2	1.67	0.12	0.88	square.4	1.05	0.12*	0.22
newton.8.3	7.49	0.12	1.05	square.5	0.68	0.12*	0.22
GC4	0.04	0.14	0.23	square.6	0.55	0.11*	0.23
Poly	0.16	0.11	0.23	square.7	0.36	0.12*	0.23
Rump	0.02	0.11	0.21	square.8	0.06	0.12	0.21
Sterbenz	0	0.12	0.2	**Total**	**208.99**	**18.37**	**40.55**

straint solvers which are respectively correct over floating-point and real numbers. So, RAICP can exploit the refutation capabilities of partial consistencies to refine the domains computed by FLUCTUAT.

We showed that RAICP is fast and efficient on programs that are representative of the difficulties of FLUCTUAT (conditional constructs and non-linearities). Experiments on a significant set of benchmarks showed also that RAICP is as precise and faster than CDFL, a state-of-the-art tool for bound analysis and assertion checking on programs with floating-point computations.

This integration of AI and CP works well because often the first approximation of variable bounds computed by AI is small enough to allow efficient filtering with partial consistencies. In the case of FLUCTUAT, sets of affine forms abstract non-linear expressions and constraints. These sets constitute better approximations of linear constraint systems than the boxes used in interval-based constraint solvers. Nonetheless, they are less adapted for non-linear constraint

systems where filtering techniques used in numeric CSP solving offer a more flexible and extensible framework.

Further work concerns a tighter integration of abstract interpretation and constraint solvers, for instance, at the abstract domain level instead of the interval domain level.

Acknowledgments. The authors gratefully acknowledge Sylvie Putot, Éric Goubault and Franck Védrine 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. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. Information Processing Letters 93(6), 281–288 (2005)
3. Boldo, S., Filliâtre, J.C.: Formal verification of floating-point programs. In: 18th IEEE Symposium on Computer Arithmetic. pp. 187–194. IEEE (2007)
4. Botella, B., Gotlieb, A., Michel, C.: Symbolic execution of floating-point computations. Software Testing, Verification and Reliability 16(2), 97–121 (2006)
5. 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)
6. Codognet, P., Filé, G.: Computations, abstractions and constraints in logic programs. In: International Conference on Computer Languages (ICCL'92). pp. 155–164. IEEE (1992)
7. Collavizza, H., Rueher, M., Hentenryck, P.V.: A constraint-programming framework for bounded program verification. Constraints Journal 15(2), 238–264 (2010)
8. 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)
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. Denmat, T., Gotlieb, A., Ducassé, M.: An abstract interpretation based combinator for modeling while loops in constraint programming. In: Principles and Practices of Constraint Programming (CP'07). LNCS, vol. 4741, pp. 241–255. Springer Verlag (2007)
11. de Dinechin, F., Lauter, C.Q., Melquiond, G.: Certifying the floating-point implementation of an elementary function using Gappa. IEEE Transactions on Computers 60(2), 242–253 (2011)
12. D'Silva, V., Haller, L., Kroening, D., Tautschnig, M.: Numeric bounds analysis with conflict-driven learning. In: Proc. TACAS. Lecture Notes in Computer Science, vol. 7214, pp. 48–63. Springer (2012)
13. Ghorbal, K., Goubault, E., Putot, S.: A logical product approach to zonotope intersection. In: CAV. LNCS, vol. 6174, pp. 212–226. Springer (2010)
14. Goldberg, D.: What every computer scientist should know about floating point arithmetic. ACM Computing Surveys 23(1), 5–48 (1991)

15. Goubault, E., Putot, S.: Static analysis of numerical algorithms. In: SAS. LNCS, vol. 4134, pp. 18–34. Springer (2006)
16. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: VMCAI. LNCS, vol. 6538, pp. 232–247. Springer (2011)
17. Granvilliers, L., Benhamou, F.: Algorithm 852: RealPaver: an interval solver using constraint satisfaction techniques. ACM Transactions on Mathematical Software 32(1), 138–156 (2006)
18. Harrison, J.: A machine-checked theory of floating-point arithmetic. In: TPHOLs. LNCS, vol. 1690, pp. 113–130. Springer-Verlag (1999)
19. Lhomme, O.: Consistency techniques for numeric CSPs. In: 13th International Joint Conference on Artificial Intelligence. pp. 232–238 (1993)
20. Marre, B., Michel, C.: Improving the floating point addition and subtraction constraints. In: CP. LNCS, vol. 6308, pp. 360–367. Springer (2010)
21. Michel, C.: Exact projection functions for floating-point number constraints. In: 7th International Symposium on Artificial Intelligence and Mathematics (2002)
22. Michel, C., Rueher, M., Lebbah, Y.: Solving constraints over floating-point numbers. In: CP. LNCS, vol. 2239, pp. 524–538. Springer Verlag (2001)
23. Pelleau, M., Truchet, C., Benhamou, F.: Octagonal domains for continuous constraints. In: Principles and Practice of Constraint Programming (CP'11). LNCS, vol. 6876, pp. 706–720 (2011)
24. Rump, S.M.: Verification methods: Rigorous results using floating-point arithmetic. Acta Numerica 19, 287–449 (2010)

Annexe B

Titre :

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

Auteurs :

Olivier Ponsini, Claude Michel et Michel Rueher

Résumé :

L'analyse des valeurs des variables est une approche classique de la vérification des programmes comportant des calculs en virgule flottante. Les analyses existantes procèdent principalement par interprétation abstraite et sur-approximent les valeurs possibles des variables des programmes. Cependant, les sur-approximations calculées par les outils de l'état de l'art peuvent s'avérer assez grossières pour des expressions couramment employées dans les programmes. Dans cet article, nous montrons que les solveurs de contraintes peuvent significativement raffiner les approximations calculées par les techniques d'interprétation abstraite. Plus précisément, nous introduisons une approche hybride qui combine techniques d'interprétation abstraite et techniques de programmation par contraintes dans une unique analyse statique et automatique. Nous avons comparé l'efficacité du système rAiCp qui implémente cette approche avec l'état de l'art des analyseurs statiques : rAiCp produit des approximations substantiellement plus précises et est capable de vérifier des propriétés de programme sur des ensembles de programmes de test aussi bien académiques qu'industriels.

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

Olivier Ponsini · Claude Michel · Michel Rueher

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 that combines abstract interpretation and constraint programming techniques in a single static and automatic analysis. 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

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 (e.g. 0.1 does not

This work was partially supported by ANR VACSIM (ANR-11-INSE-0004), ANR AEOLUS (ANR-10-SEGI-0013), and OSEO ISI PAJERO projects.

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

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's 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, i.e. 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*, i.e. 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. However, 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 it may generate. 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. More precisely, we propose to exploit the refutation

capabilities of constraint solvers to refine domains computed by abstract interpretation. We show that constraint 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 4 times faster than CDFL, and it did not produce any false alarm whereas FLUCTUAT did generate 11 false alarms.

We also 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 based on a SAT solver.

To sum up, RAICP is a promising framework for computing accurate 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: $\{x = f + 2 * g$ 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.

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

However, 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.

[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`

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:

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

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, i.e. $y \in [-10,10]$. 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 y to $[0,5]$ and of z to $[0,50]$. Hence, the constraint system for this path is $\{y \geq 0 \wedge z = 10 * y \wedge -10 \leq y \wedge y \leq 5 \wedge 0 \leq z \wedge z \leq 50\}$. 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 $\{y < 0 \wedge z = -y \wedge -10 \leq y \wedge y \leq 5 \wedge 0 < z \wedge z \leq 10\}$. Here again, CP filtering techniques cannot achieve any reduction of the domain of z. 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, i.e. 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. However, in practice this proof may be more difficult and we apply the following process: to check a property defined over the program variables, we add the negation of this property 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 cor-

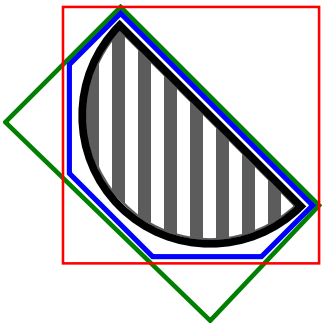

Fig. 2 Half-disk approximations by intervals (in red), zonotope (in green), and polyhedron (in blue)

respond 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

Abstract interpretation methods define an abstract semantics that approximates the concrete semantics of programs. 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. Special operations allow to approximate program loops in very short time. These operations are designed to preserve all the concrete behaviors of the program.

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

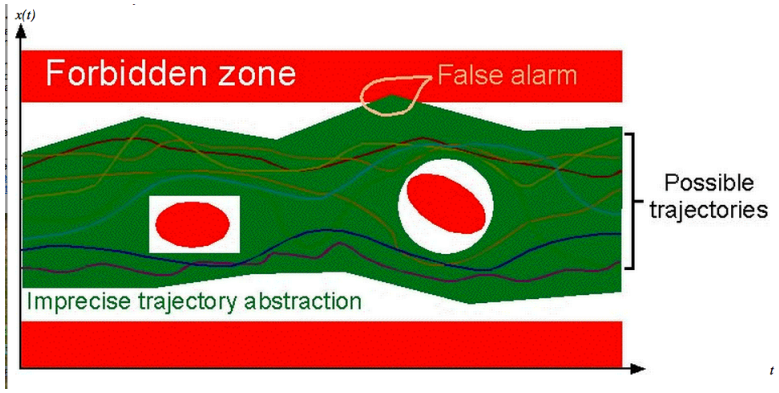

Fig. 3 A false alarm occurs when the abstract semantics intersects the forbidden zone while the concrete semantics does not intersect this zone. Forbidden zones are in red, the abstract semantics is in green, and the concrete semantics is the set of curves.

trade-off between performance and precision. Zonotopes are sets of affine forms that preserve linear correlations between variables and keep track of the statements involved in the loss of accuracy of floating-point computations. Zonotopes have nevertheless some drawbacks: approximations of some common program constructs, such as conditionals and nonlinear expressions, are not accurate.

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 a *false alarm* when properties represent desired behaviors of the program (see Fig. 3 extracted from Cousot's informal introduction to abstract interpretation[2]).

To sum up, AI techniques provide a good trade-off between precision and performance. They scale well, but they lack of 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. Several industrial solvers and academic solvers are available, e.g. `ILOG/IBM`[3], `Gecode`[4]. There are many successful industrial applications, e.g. timetabling (Dutch railway), hardware verification (Hentenryck et al, 2009), scheduling, planning (Rossi et al, 2006, Part II).

[2] `http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html`
[3] `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer`
[4] `http://www.gecode.org`

The key features of CP are *domain filtering* and *search strategies*. Domain filtering algorithms consider each constraint separately and remove values that are trivially inconsistent. Search strategies try to exploit the structure of the problem to guide the variable instantiation process. In this paper we mainly use constraint techniques over continuous domains[5].

A numeric constraint satisfaction problem $(\mathscr{X},\mathscr{D},\mathscr{C})$ is defined by:

- $\mathscr{X} = \{x_1,\ldots,x_n\}$, a set of variables;
- $\mathscr{D} = \{D_{x_1},\ldots,D_{x_n}\}$, a set of domains. D_{x_i} contains all acceptable values for variable x_i;
- $\mathscr{C} = \{c_1,\ldots,c_m\}$, a set of constraints

The constraint programming framework over continuous domains is based on a *branch & prune* schema which is best viewed as an iteration of two steps:

1. Pruning the search space
2. Making a choice to generate two (or more) sub-problems

The *pruning step* reduces an interval when it can prove that the upper bound or the lower bound does not satisfy some constraint. The *branching step* splits the interval associated to some variable in two or more intervals (often with the same width).

Pruning techniques on continuous domains are based on partial consistencies, that is to say a consistency of a relaxation of the system. More precisely, it is a property that holds on a subset of variables or constraints and that is associated with a filtering algorithm. Informally speaking, a constraint system C satisfies a partial consistency property if *a relaxation of C is consistent*. For instance, consider $X = [\underline{x},\overline{x}]$ and $C(x,x_1,...,x_n) \in \mathscr{C}$: if $C(x,x_1,...,x_n)$ does not hold for any values $v \in [\underline{x},x']$, then X may be shrunken to $X = [x',\overline{x}]$.

2B-consistency (Lhomme, 1993) states a local property on the bounds of the domains of a variable at a single constraint level. In other words, the domain of variable x is 2B-consistent if, for any constraint c, there exists at least one value 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.

Example: Let $S_1 = \{x+y=2, y \leq x-1, x \in [0,100], y \in [0,100]\}$ and $S_2 = \{x+y=2, y \leq x-1, x \in [1,2], y \in [0,1]\}$ be two constraint systems. S_1 is not 2B-consistent. Indeed, the domain of x is not 2B-consistent since $100+y=2$ is not satisfiable when $D_y = [0,100]$. S_2 is 2B-consistent. Indeed, D_x is 2B-consistent since $1+y=2$, $2+y=2$, $y \leq 1-1$ and $y \leq 2-1$ are all satisfiable when $D_y = [0,1]$. A similar reasoning can show that D_y is 2B-consistent in S_2.

2B-consistency pruning algorithms successively narrow the domains of the variables. The approximation of the projection of a constraint c on its variables is the basic tool for narrowing domains. The projection $\Pi_x(c)$ of the constraint $c(x,x_1,\cdots,x_n)$ on x is the set defined as follows:

$$\Pi_x(c) = \{v \in D_x \mid \exists (v_1,\cdots,v_n) \in D_{x_1} \times \cdots \times D_{x_n} \text{ s.t. } c(v,v_1,\cdots,v_n) \text{ holds}\}.$$

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

The approximation of $\Pi_x(c)$ is the interval $[\min(\Pi_x(c)), \max(\Pi_x(c))]$. In practice, this approximation is often computed on constraints where the multiple occurrences of variables have been replaced by fresh variables with the same domain.
When the domain of a variable has been narrowed, all the constraints in which this variable occurs will be processed again. The filtering ends when none of the domains cannot be narrowed anymore. If at least one domain becomes empty, the system is not $2B$-consistent.

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 are based on a shaving process that tries to shrink the interval of a given variable. For instance, let x be a variable the domain of which is $X = [\underline{x}, \overline{x}]$. A standard $3B$-pruning algorithm will first try to show that the constraint system is not $2B$-consistent when X is set to $[\underline{x}, \frac{\underline{x}+\overline{x}}{2}]$; if it succeeds, it will remove $[\underline{x}, \frac{\underline{x}+\overline{x}}{2}]$ from X and the process goes on; otherwise, the pruning process restarts with a smaller domain for X, e.g., $[\underline{x}, \frac{\underline{x}+\overline{x}}{4}]$. The process stops when the domain of X becomes smaller than a given ε.

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

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 based on abstract interpretation capture rounding errors due to floating-point computation in their abstract domains. They are usually fast, automatic, and scalable. However, they may lack of 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, e.g. 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) allow their users 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, 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. 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 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. 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 is based 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 times 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 algorithm is based 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, is based on a piecewise 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 piecewise exploration of the CFG and compute domain approximations on each piece of the CFG.

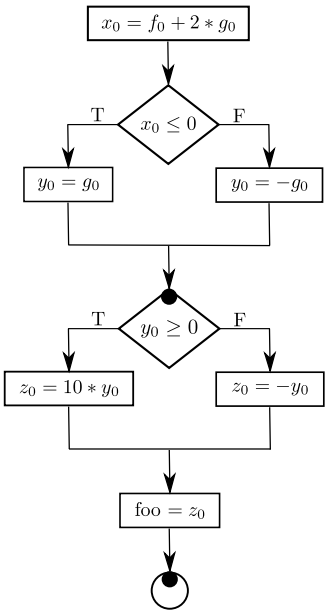

Fig. 4 CFG of program `foo` from Fig 1: nodes with black circles are merge points

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. A control flow graph contains one entry node, a node without incoming edge, and one exit node, 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;

We define 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 graphs since we unfold loops a bounded number of times before enclosing them in a *while* node.

For instance, the graph in Fig. 4 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

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
```

junction. Note that program expressions were put into DSA-like form[6] to facilitate constraint generation.

As said before, we only explore paths between two successive merge points. Of course, this process may be less precise 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.
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 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, i.e. the number of nodes from the entry node to the merge point. RAICP stores in D the result of the

[6] 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).

value analysis of the program. Initially, all elements of D are empty sets and at the end D[m] contains the domain approximations computed at merge point m.

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 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 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 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` uses AI techniques to approximate the domains at the end of the loop. The function then goes on exploring the path with these domains and the negation of the loop condition in the constraint system. Approximating loops 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 on 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.

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
```

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 Core 2 Duo at 2.8 GHz with 4 GB of memory running Linux using FLUCTUAT version 3.1247, REALPAVER version 0.4, CPLEX version 12.3, CBMC version 4.5 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 2*B*-like partial consistencies[7] to cut infeasible paths during CFG exploration and 3*B*-like partial consistencies[8] to reduce domains at merge points. This choice is motivated by performance: 2*B*-like consistency algorithms are much faster than 3*B*-like consistency algorithms, but the latter may achieve a much stronger pruning.

RAICP analyzes C programs that conform to IEEE 754 standard with the following restrictions: size of arrays are bounded; pointers, bitwise operators and statements that interrupt the control flow (`goto`, `continue`, and `break`) are not handled. However, all aspects of computations over floating-point numbers are 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 32-bit Linux operating system[9]. In the current implementation, we handle basic arithmetic operations, comparisons and some classical functions like square root.

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[10] 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 allows to specify 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 ;

[7] The prototype uses REALPAVER's *HC*4-consistency or FPCS's $2B(w)$-consistency.

[8] The prototype uses REALPAVER's *BC*5-consistency in paving mode or FPCS's $3B(w)$-consistency.

[9] All computations are done using 80 bits floating point numbers (long double). Inputs are first converted from their base type to long double while the computation result is converted from double to the awaited result base type.

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

– 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 allows to use arbitrary precision numbers or to subdivide up to two input variable intervals. However, over-approximations computed by FLUCTUAT may be very large because the abstract domains do not handle well conditional statements and non-linear expressions.

5.1.2 Constraint solver over the real numbers

REALPAVER is an interval solver for numerical constraint systems over the real numbers[11] (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, i.e., 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: *BC*5, a combination of 2*B* 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

Constraint solvers over the real numbers based 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.

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

That's why we use here FPCS, a constraint solver designed to solve a set of constraints over floating-point numbers without losing any solution (Michel, 2002; Marre and Michel, 2010). FPCS implements 2*B*-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, $\Pi_y(x = y \odot_f z)$;
- 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—e.g., *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. We show that RAICP is more efficient than FLUCTUAT alone on these benchmarks.

5.2.1 Conditionals

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

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

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	$[-\infty,\infty]$	$[-\infty,\infty]$	0.02 s
	RAICP	$[-\infty,0]$	$[-200.1,\infty]$	1.66 s
$\mathbb{F}$	FLUCTUAT	$[-\infty,\infty]$	$[-\infty,\infty]$	0.02 s
	RAICP	$[-\infty,0]$	$[-312.51,\infty]$	0.95 s

Table 2 Domains of the return value of `sinus` and `rump` functions

		`sinus` $x \in [-1,1]$		`rump` $x \in [7\times10^4, 8\times10^4]$ $y \in [3\times10^4, 4\times10^4]$	
		Domain	Time	Domain	Time
$\mathbb{R}$	FLUCTUAT	$[-1.009, 1.009]$	0.02 s	$[-1.168\times10^{37}, 1.992\times10^{37}]$	0.02 s
	RAICP	$[-0.842, 0.842]$	0.93 s	$[-1.144\times10^{36}, 1.606\times10^{37}]$	1.82 s
$\mathbb{F}$	FLUCTUAT	$[-1.009, 1.009]$	0.02 s	$[-1.168\times10^{37}, 1.992\times10^{37}]$	0.02 s
	RAICP	$[-0.855, 0.85]$	0.86 s	$[-1.168\times10^{37}, 1.992\times10^{37}]$	0.86 s

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

FPCS and REALPAVER also use approximations to handle non-linear terms and thus cannot always achieve a significant pruning. This is outlined in Table 2 by program `rump`. This program computes a very particular polynomial designed by Rump (2010) to illustrate a catastrophic cancellation phenomenon:

$$333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + \frac{x}{2y}$$

5.2.3 Loops

FLUCTUAT unfolds loops a bounded number of times[12] 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. However, RAICP can also

[12] Default value is ten times.

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	$[2.116, 2.354]$	0.02 s	$[2.098, 3.435]$	0.06 s	$[-\infty, \infty]$	0.03 s
	RAICP	$[2.121, 2.346]$	0.97 s	$[2.232, 3.165]$	1.06 s	$[0, 10]$	1.09 s
$\mathbb{F}$	FLUCTUAT	$[2.116, 2.354]$	0.02 s	$[-\infty, \infty]$	0.06 s	$[-\infty, \infty]$	0.03 s
	RAICP	$[2.120, 2.351]$	1.5 s	$[2.232, 3.193]$	4.97 s	$[0, 10]$	0.94 s

```
/* 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. 5 Programs (a) `sqrt` with input domain #1 and (b) `bigLoop`

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. 5a) is based 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. Ten unfoldings are sufficient to exit the loop with the two different input domains used in this benchmark (see Table 3). FLUCTUAT obtains accurate results except in the second configuration over $\mathbb{F}$ where it could not achieve any reduction: the different interpretation of a conditional statement over $\mathbb{R}$ and over $\mathbb{F}$ leads to different paths in the program. With an unfolding bound of ten—like in FLUCTUAT—RAICP shrinks the domain over $\mathbb{F}$ to $[2.232, 3.193]$ in the second configuration.

Program `bigLoop` (see Fig. 5b) 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

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

	CDFL	FLUCTUAT	RAICP
Total execution time	208.99 s	16.06 s	56.25 s
False alarms	0	11	0

since it is far too expensive to unfold completely such loops. However, 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 that a tight cooperation between CP and AI techniques can be very efficient.

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[13]. 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 made from 12 programs by varying the input variable domains, the loop bounds, and the constants in the properties to check. All the programs are based on 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 precise than ASTRÉE for approximating floating-point variable domains (D'Silva et al, 2012). We compare in Table 4 the efficiency of RAICP, FLUCTUAT and CDFL on these benchmarks. RAICP is on average 3.5 times slower than FLUCTUAT used alone, but it is much more precise 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 3.5 times faster than CDFL.

It is however important to 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

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

Table 5 Validity results and execution times of CBMC, FLUCTUAT and RAICP on property P_1 when varying unfoldings

Number of unfoldings	CBMC		FLUCTUAT		RAICP	
	Validity	Time	Validity	Time	Validity	Time
1	valid	0.4 s	valid	0.04 s	valid	0.9 s
2	-	$>$ 3600 s	unknown	0.03 s	valid	1 s
100	-	-	unknown	1.47 s	valid	19.2 s
1000	-	-	unknown	77.3 s	valid	338.7 s
2000	-	-	unknown	413.9 s	valid	1217.8 s

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 $r_s = 1 - \frac{v_{\text{slow}}}{v_{\text{car}}}$. ABS tries to maintain the optimal rate $r_o = 20\%$[14]. When r_s is greater than r_o, ABS starts controlling braking.

Our industrial partner had specified property P_1 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_1 is then:

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

We compared CBMC, FLUCTUAT, and RAICP on the checking of property P_1. 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_1 holds up to the fixed 2 000 unfoldings limit. Property P_1 trivially holds at the first unfolding which corresponds to the initialization phase of the ABS. CBMC reached the time limit on the second unfolding. This is probably due to the fact that CBMC falls into a slow convergence process. FLUCTUAT is very fast, but it 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

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

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 allow efficient pruning 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 required in programs with loops. In FLUCTUAT, sets of affine forms abstract non-linear expressions and constraints. These sets constitute better approximations of 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 CSP solving offer a more flexible and extensible framework.

We showed that RAICP is fast and 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 precise and faster than CDFL, a state-of-the-art tool for bound analysis and assertion checking on programs with floating-point computations.

Further work concerns a tighter integration of abstract interpretation and constraint solvers and the generation of counter-examples. 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.

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, 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

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, Coffrin C, Gutkovich B (2009) Constraint-based local search for the automatic generation of architectural tests. In: Gent IP (ed) 15th International Conference on Principles and Practice of Constraint Programming (CP 2009), Springer, Lecture Notes in Computer Science, vol 5732, pp 787–801

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
Rump SM (2010) Verification methods: Rigorous results using floating-point arithmetic. Acta Numerica 19:287–449

Annexe C

Titre :

Boosting local consistency algorithms over floating-point numbers

Auteurs :

Saïd Mohammed Belaïd, Claude Michel et Michel Rueher

Résumé :

La résolution de systèmes de contraintes sur les nombres à virgule flottante est un problème critique pour de nombreuses applications et notamment pour la vérification de programme. Les algorithmes de filtrage pour les contraintes sur les nombres à virgule flottante se limitent à la 2b-consistance et ses dérivées. Ces algorithmes sont corrects mais souffrent des problèmes bien connus inhérents aux consistances locales, e.g., l'incapacité à traiter efficacement les occurrences multiples des variables. Ces limitations trouvent aussi leur origine dans les restrictions de l'arithmétique sur les nombres à virgule flottante. Pour contourner les faibles propriétés de l'arithmétique sur les nombres à virgule flottante, nous proposons dans cet article un nouvel algorithme de filtrage, appelé FPLP, qui repose sur des relaxations sur les nombres réels du problème sur les nombres à virgule flottante. Des bornes correctes des domaines sont calculées avec un solveur d'optimisation linéaire en nombres entiers mixte (MILP) par des linéarisations conservatives de ces relaxations. Les expérimentations sur un jeu de tests pertinent se révèlent prometteuses et montrent que cette approche peut améliorer les algorithmes de consistance locale sur les nombres à virgule flottante.

Accepted for publication at CP'12 (18th International Conference on Principles and Practice of Constraint Programming)

Boosting local consistency algorithms over floating-point numbers*

Mohammed Said Belaid, Claude Michel, and Michel Rueher

I3S (UNS/CNRS)
2000, route des Lucioles - Les Algorithmes - bât. Euclide B - BP 121
06903 Sophia Antipolis Cedex - France
{MSBelaid, Claude.Michel}@i3s.unice.fr, Michel.Rueher@gmail.com

Abstract. Solving constraints over floating-point numbers is a critical issue in numerous applications notably in program verification. Capabilities of filtering algorithms over the floating-point numbers ($\mathcal{F}$) have been so far limited to 2b-consistency and its derivatives. Though safe, such filtering techniques suffer from the well known pathological problems of local consistencies, e.g., inability to efficiently handle multiple occurrences of the variables. These limitations also have their origins in the strongly restricted floating-point arithmetic. To circumvent the poor properties of floating-point arithmetic, we propose in this paper a new filtering algorithm, called FPLP, which relies on various relaxations over the real numbers of the problem over $\mathcal{F}$. Safe bounds of the domains are computed with a mixed integer linear programming solver (MILP) on safe linearizations of these relaxations. Preliminary experiments on a relevant set of benchmarks are promising and show that this approach can be effective for boosting local consistency algorithms over $\mathcal{F}$.

1 Introduction

Critical systems are more and more relying on floating-point (FP) computations. For instance, embedded systems are typically controlled by software that store measurements and environment data as floating-point number ($\mathcal{F}$). The initial values and the results of all operations must therefore be rounded to some nearby float. This rounding process can lead to significant changes, and, for example, can modify the control flow of the program. Thus, the verification of programs performing FP computations is a key issue in the development of critical systems.

Methods for verifying programs performing FP computations are mainly derived from standard program verification methods. Bounded model checking (BMC) techniques have been widely used for finding bugs in hardware design [3] and software [11]. SMT solvers are now used in most of the state-of-the-art BMC tools to directly work on high level formula (see [2, 9, 11]). The bounded model checker CBMC encodes each FP operation of the program with a set of logic

* This work was partially supported by ANR VACSIM (ANR-11-INSE-0004), ANR AEOLUS (ANR-10-SEGI-0013) and OSEO ISI PAJERO projects.

functions on bit-vectors which requires thousands of additional variables and becomes quickly intractable [6]. Tools based on abstract interpretation [10, 22] can show the absence of run-time errors (e.g., division by zero) on program working with FP numbers. Tools based on abstract interpretation are safe since they over-approximate FP computations. However, over-approximations may be very large and these tools may generate many false alerts, and thus reject many valid programs. For instance, Chen's polyhedral abstract domains [7] rely on coarse approximations of floating-point operations that do not take advantage of the rounding mode. Constraint programming (CP) has also been used for program testing [13, 14] and verification [8]. CP offers many benefits like the capability to deduce information from partially instantiated problems or to exhibit counter-examples. The CP framework is very flexible and simplifies the integration of new solvers for handling a specific domain, for instance FP solvers. However, it is important to understand that solvers over real numbers ($\mathcal{R}$) cannot correctly handle FP arithmetic. Dedicated constraint solvers are required in safe CP-based framework and BMC-SMT tools for testing or verifying numerical software[1].

Techniques to solve FP constraints are based on adaptations of classical consistencies (e.g. box-consistency and 2B-consistency) over $\mathcal{R}$ [21], [20, 5]. However FP solvers based on these techniques do not really scale up to large constraint systems. That is why we introduce here a new method to handle constraints over the FP numbers by taking advantage of solvers over $\mathcal{R}$. The basic tenet is to build correct but tight relaxations over $\mathcal{R}$ of the FP operations. To ensure the tightness of the result, each FP operation is approximated according to its rounding mode. For example, assume that x and y are positive normalized FP numbers[2], then the FP product $x \otimes y$ with a rounding mode set to $-\infty$, is bounded by $\alpha \times (x \times y) < x \otimes y \leq x \times y$ where $\alpha = 1/(1 + 2^{-p+1})$ and p is the size of the significand. Approximations for special cases have also been refined, e.g., for the addition with a rounding mode set to zero, or for the multiplication by a constant.

Using these relaxations, a problem over the FP numbers is first translated into a set of nonlinear constraints over $\mathcal{R}$. A linearization of the nonlinear constraints is then applied to obtain a mixed integer linear problem (MILP) over $\mathcal{R}$. In this process, binary variables are used to handle concave domains to prevent too loose over-approximations. This last set of constraints can directly be solved by available MILP solvers over $\mathcal{R}$ which are relieved from the drawbacks of FP arithmetic. Efficient MILP solvers rely on FP computations and thus, might miss some solutions. In order to ensure a safe behavior of our algorithm, correct rounding directions are applied to the relaxation coefficients [19, 4] and a

[1] See FPSE (http://www.irisa.fr/celtique/carlier/fpse.html), a solver for FP constraints coming from C programs.

[2] A FP number is a triple (s, e, m) where s is the sign, e the exponent and m the significand. Its value is given by $(-1)^s \times 1.m \times 2^e$. r and p are the size of the exponent and the significand. The IEEE standard 754 defines the single format with $(r, p) = (8, 23)$ and the double format with $(r, p) = (11, 52)$. A normalized FP number's significand has no non-zero digits to the left of the decimal point and a non-zero digit just to the right of the decimal point.

procedure [23] to compute a safe minimizer from the unsafe result of the MILP solver is also applied. Preliminary experiments are promising and this new filtering technique should really help to scale up all verifications tools that uses a FP solver.

Our method relies on a high level representation of the FP operations and, thus, does not suffer from the same drawbacks than bit vector encoding. The bit vector encoding used in CBMC generates thousands of additional binary variables for each FP operation of the program. For example, an addition of two 32 bits floats requires 2554 binary variables [6]. The mixed approximations proposed in [6] reduce the number of additional binary variables significantly but the resulting system remains expensive in memory consumption. For instance, a single addition with only 5 bits of precision still requires 1035 additional variables. Our method does also generate additional variables: temporary variables are used to decompose complex expressions into elementary operations over the FP numbers and some binary variables are used to handle the different cases of our relaxations. However, the number of generated variables is negligible compared to the ones required by a bit vector encoding.

1.1 An illustrative example

Before going into the details, let us illustrate our approach on a very simple example. Consider the simple constraint

$$z = x \oplus y \ominus x \tag{1}$$

where x, y and z are 32 bits FP variables, and $\oplus$ and $\ominus$ are the addition and the subtraction over $\mathcal{F}$, respectively. Over the real numbers, such an expression can be simplified to $z = y$. However, this is not true with FP numbers. For example, over $\mathcal{F}$ and with a rounding mode set to the nearest, $10.0 \oplus 10.0^{-8} \ominus 10.0$ is not equal to 10.0^{-8} but to 0. This absorption phenomenon illustrates why expressions over the FP numbers cannot be simplified in the same way than expressions over the real numbers.

Now, let us assume that $x \in [0.0, 10.0]$, $y \in [0.0, 10.0]$ and $z \in [0.0, 10.0^8]$. FP2B, a 2B-consistency [16] algorithm adapted to FP constraints [20], first performs forward propagation of the domains of x and y on the domain of z using an interval arithmetic where interval bounds are computed with a rounding mode set to the nearest. Backward propagation being of no help here, the filtering process yields:

$$x \in [0.0, 10.0],\ y \in [0.0, 10.0],\ z \in [0.0, 20.0]$$

This poor filtering is due to the fact that 2B-consistency algorithms cannot handle efficiently constraints with multiple occurrences of the variables. A stronger consistency like 3B-consistency [16] will reduce the domain of z to the interval $[0.0, 10.01835250854492188]$. However, 3B-consistency will fail to reduce the domain of z when x and y occur more than two times, like in $z = x \oplus y \ominus x \ominus y \oplus x \oplus y \ominus x$.

Algorithm FPLP, introduced in this paper, first builds safe nonlinear relaxations over $\mathcal{R}$ of the constraints over $\mathcal{F}$ derived from the program. Of course, these relaxations are computed according to the rounding mode. Applied to constraint (1), it yields the following relaxations over $\mathcal{R}$:

$$\begin{cases} (1 - \frac{2^{-p}}{(1-2^{-p})})(x+y) \leq tmp1 \\ tmp1 \leq (1 + \frac{2^{-p}}{(1+2^{-p})})(x+y) \\ (1 - \frac{2^{-p}}{(1-2^{-p})})(tmp1 - x) \leq tmp2 \\ tmp2 \leq (1 + \frac{2^{-p}}{(1+2^{-p})})(tmp1 - x) \\ z = tmp2 \end{cases}$$

where p is the size of the significand of the FP variables. $tmp1$ approximates the result of the operation $x \oplus y$ by means of two planes over $\mathcal{R}$ which encompass all the results of this addition over $\mathcal{F}$. $tmp2$ does the same for the subtraction. Some relaxations, like the one of the product, include nonlinear terms. In such a case, a linearization process is applied to get a MILP. Once the problem is fully linear, a MILP solver is used to shrink the domain of each variable, respectively, minimizing and maximizing it.

FPLP, which stands for Floating-Point Linear Program, implements the algorithm previously sketched. A call to FPLP on constraint (1) immediately yields:

$$x \in [0, 10],\ y \in [0, 10],\ z \in [0, 10.0000023841859]$$

which is a much tighter result than the one computed by FP2B. Contrary to 3B-consistency, FPLP still gives the same result with FPLP provides the same result for constraint $z = x \oplus y \ominus x \ominus y \oplus x \oplus y \ominus x$ whereas 3B-consistency cannot reduce the upper bound of z on the latter constraint.

1.2 Outline of the paper

The rest of this paper is organized as follows: the next section introduces the nonlinear relaxations over $\mathcal{R}$ of the constraints over $\mathcal{F}$. The following section shows how the nonlinear terms of the relaxations are linearized. Then, the filtering algorithm is detailed and the results of our experiments are given before concluding the paper.

2 Relaxations of FP constraints

This section introduces nonlinear relaxations over $\mathcal{R}$ of the FP constraints from the initial problem. These relaxations are the cornerstone of the filtering process described in this paper. They must be *correct*, i.e., they must preserve the whole set of solutions of the initial problem, and *tight*, i.e., they should enclose the smallest amount of non FP solutions.

These relaxations are built using two techniques: the *relative error* and the *correctly rounded* operations. The former is a technique frequently used to analyze the precision of the computation. The latter property is ensured by any

IEEE 754 compliant implementation of the FP arithmetic: a correctly rounded operation is an operation whose result over $\mathcal{F}$ is equal to the rounding of the result of the equivalent operation over $\mathcal{R}$. In other word, let x and y be two FP numbers, $\odot$ and $\cdot$, respectively, an operation over $\mathcal{F}$ and its equivalent over $\mathcal{R}$, if $\odot$ is correctly rounded then, $x \odot y = round(x \cdot y)$.

In the rest of this section, we first detail how to build these relaxations for a specific case before defining the relaxations in the general cases. Then, we will show how the different cases can be simplified.

2.1 A specific case

In order to explain how these relaxations are built, let us consider the case where an operation is computed with a rounding mode set to $-\infty$ and the result of this operation is a positive and normalized FP number. Such an operation, denoted $\odot$, could be any of the four basic binary operations from the FP arithmetic. The operands are all supposed to have the same FP type, i.e., either float, double or long double. Then, the following property holds:

Proposition 1. *Let x and y be two FP numbers whose significand is represented by p bits. Assume that the rounding mode is set to $-\infty$ and that the result of $x \odot y$ is a normalized positive FP numbers smaller than max_f, the biggest FP number, then the following property holds:*

$$\frac{1}{1+2^{-p+1}}(x \cdot y) < x \odot y \leq (x \cdot y)$$

where $\odot$ is a basic operation over the FP numbers and, $\cdot$ is the equivalent operation over the real numbers.

Proof. Since IEEE 754 basic operations are correctly rounded and the rounding mode is set to $-\infty$, we have:

$$x \odot y \leq x \cdot y < (x \odot y)^+ \tag{2}$$

$(x \odot y)^+$, the successor of $(x \odot y)$ within the set of FP numbers, can be computed by

$$(x \odot y)^+ = (x \odot y) + ulp(x \odot y)$$

as, ulp, which stands for unit in the last place, is defined by $ulp(x) = x^+ - x$. Thus, it results from (2) that

$$x \odot y \leq x \cdot y < (x \odot y) + ulp(x \odot y)$$

From the second inequality, we have

$$\frac{1}{x \odot y + ulp(x \odot y)} < \frac{1}{x \cdot y}$$

By multiplying each side of the inequality by $x \odot y$ – which is a positive number – we get

$$\frac{x \odot y}{x \odot y + ulp(x \odot y)} < \frac{x \odot y}{x \cdot y}$$

By multiplying each side of the above inequality by -1 and by adding one to each side, we obtain

$$1 - \frac{x \odot y}{x \cdot y} < 1 - \frac{x \odot y}{x \odot y + ulp(x \odot y)} = \frac{ulp(x \odot y)}{x \odot y + ulp(x \odot y)} \tag{3}$$

Now, consider ϵ, the relative error defined by

$$\epsilon = \left| \frac{real\_value - float\_value}{real\_value} \right|$$

ϵ is the absolute value of the difference between the result over $\mathcal{R}$ and the result over $\mathcal{F}$ divided by the result over $\mathcal{R}$. In the considered case, the result of $x \odot y$ being a positive normalized floating-point number and $x \cdot y \geq x \odot y$, the relative error is given by

$$0 \leq \epsilon = \frac{x \cdot y - x \odot y}{x \cdot y} = 1 - \frac{x \odot y}{x \cdot y}$$

Thus, thanks to (3), we have

$$0 \leq \epsilon < \frac{ulp(x \odot y)}{x \odot y + ulp(x \odot y)}$$

z, the result of the operation $x \odot y$, is a binary positive and normalized FP number that can be written $z = 1.m_z 2^{e_z}$, where m_z has p bits. Moreover, $ulp(z) = 2^{-p+1}2^{e_z}$. Therefore,

$$0 \leq \epsilon < \frac{2^{-p+1}2^{e_z}}{m_z 2^{e_z} + 2^{-p+1}2^{e_z}} = \frac{2^{-p+1}}{m_z + 2^{-p+1}}$$

The value of the significand of a normalized FP number belongs to the interval $[1.0, 2.0[$. An upper bound of the relative error ϵ is given by the minimum of $m_z + 2^{-p}$ which is reached when $m_z = 1$. Thus

$$0 \leq \epsilon < \frac{2^{-p+1}}{1 + 2^{-p+1}}$$

Since we have

$$\epsilon = \frac{x \cdot y - x \odot y}{x \cdot y}$$

we have

$$0 \leq \frac{x \cdot y - x \odot y}{x \cdot y} < \frac{2^{-p+1}}{1 + 2^{-p+1}}$$

and

$$0 \leq x \cdot y - x \odot y < (x \cdot y)\frac{2^{-p+1}}{1 + 2^{-p+1}}$$

Rounding mode	Negative normalized	Negative denormalized	Positive denormalized	Positive normalized
to $-\infty$	$[(1+2^{-p+1})z_r, z_r]$	$[z_r - min_f, z_r]$	$[z_r - min_f, z_r]$	$[\frac{1}{(1+2^{-p+1})}z_r, z_r]$
to $+\infty$	$[z_r, \frac{1}{(1+2^{-p+1})}z_r]$	$[z_r, z_r + min_f]$	$[z_r, z_r + min_f]$	$[z_r, (1+2^{-p+1})z_r]$
to 0	$[z_r, \frac{1}{(1+2^{-p+1})}z_r]$	$[z_r - min_f, z_r]$	$[z_r, z_r + min_f]$	$[\frac{1}{(1+2^{-p+1})}z_r, z_r]$
to nearest	$[(1+\frac{2^{-p}}{(1+2^{-p})})z_r,$ $(1-\frac{2^{-p}}{(1-2^{-p})})z_r]$	$[z_r - \frac{min_f}{2},$ $z_r + \frac{min_f}{2}]$	$[z_r - \frac{min_f}{2}$ $z_r + \frac{min_f}{2}]$	$[(1-\frac{2^{-p}}{(1-2^{-p})})z_r,$ $(1+\frac{2^{-p}}{(1+2^{-p})})z_r]$

Table 1. Relaxations of $x \odot y$ for each rounding mode where $z_r = x \cdot y$.

By multiplying each side of the inequality by -1 and adding $x \cdot y$ to each side, we finally obtain

$$\frac{1}{1+2^{-p+1}}(x \cdot y) < x \odot y \leq x \cdot y$$

□

2.2 Generalization

Table 1 summarizes the relaxations for each rounding mode in the different cases, i.e., positive or negative FP numbers, as well as, normalized and denormalized FP numbers. Each case has a dedicated correct and tight approximation built in a way similar to the one of the case detailed in the previous subsection.

Note that tighter approximations for specific cases could also be computed. For example, the approximation of an addition with a rounding mode sets to $\pm\infty$ could be slightly improved. In a similar way, the structure of the problem is another source of improvements of the approximations. For example, $2 \otimes x$ being exactly computed[3], it can directly be evaluated over $\mathcal{R}$.

2.3 Simplified relaxations

The main issue with the previous relaxations is that the solving process will have to handle the different cases. As a result, for n basic operations, the solver has to deal with 4^n potential combinations of the relaxations. To decrease substantially this complexity, we provide here a combination of the four cases of each rounding mode into a single case.

Let us first consider the case where the rounding mode is set to $-\infty$:

Proposition 2. *Let x and y be two FP numbers whose significand size is p and, assume that the rounding mode is set to $-\infty$ and, that $-max_f < x \odot y < max_f$, then,*

$$z_r - 2^{-p+1}|z_r| - min_f \leq x \odot y \leq z_r$$

[3] Provided that no overflow occurs.

where min_f is the smallest positive FP number, $\odot$ and $\cdot$ are respectively a basic binary operation over $\mathcal{F}$ and its equivalent over $\mathcal{R}$, and $z_r = x \cdot y$.

Proof. In a first step, the normalized and denormalized approximations are combined. If $z_r > 0$ then $\frac{1}{1+2^{-p+1}} z_r < z_r$. Thus,

$$\frac{1}{1+2^{-p+1}} z_r - min_f < z_r - min_f$$

and

$$\frac{1}{1+2^{-p+1}} z_r - min_f < \frac{1}{1+2^{-p+1}} z_r$$

Therefore,

$$\frac{1}{1+2^{-p+1}} z_r - min_f < x \odot y \leq z_r,\ z_r \geq 0$$

When $z_r \leq 0$, we get

$$(1+2^{-p+1}) z_r - min_f < x \odot y \leq z_r,\ z_r \leq 0$$

These two approximations can be rewritten as follows,

$$\begin{cases} z_r - \frac{2^{-p+1}}{1+2^{-p+1}} z_r - min_f < x \odot y \leq z_r, z_r \geq 0 \\ z_r + 2^{-p+1} z_r - min_f < x \odot y \leq z_r, \quad z_r \leq 0 \end{cases}$$

To combine the negative and positive approximations together we can use the absolute value:

$$\begin{cases} z_r - \frac{2^{-p+1}}{1+2^{-p+1}} |z_r| - min_f < x \odot y \leq z_r, z_r \geq 0 \\ z_r - 2^{-p+1} |z_r| - min_f < x \odot y \leq z_r, \quad z_r \leq 0 \end{cases}$$

As $max\{\frac{2^{-p+1}}{1+2^{-p+1}}, 2^{-p+1}\} = 2^{-p+1}$, we get

$$z_r - 2^{-p+1} |z_r| - min_f \leq x \odot y \leq z_r$$

□

The same reasoning holds for other rounding modes. Table 2 summarizes the simplified relaxations for each rounding mode. Note that these approximations define concave sets.

3 Linearization of the relaxations

The relaxations introduced in the previous section contain nonlinear terms that cannot be directly handled by a MILP solver. In this section, we describe how these terms are approximated by sets of linear constraints.

Rounding mode	The approximation of $x \odot y$
to $-\infty$	$[z_r - 2^{-p+1}\lvert z_r\rvert - min_f, z_r]$
to $+\infty$	$[z_r, z_r + 2^{-p+1}\lvert z_r\rvert + min_f]$
to 0	$[z_r - 2^{-p+1}\lvert z_r\rvert - min_f,$ $z_r + 2^{-p+1}\lvert z_r\rvert + min_f]$
to the nearest	$[z_r - \frac{2^{-p}}{(1-2^{-p})}\lvert z_r\rvert - \frac{min_f}{2},$ $z_r + \frac{2^{-p}}{(1-2^{-p})}\lvert z_r\rvert + \frac{min_f}{2}]$

Table 2. Simplified relaxations of $x \odot y$ for each rounding mode (with $z_r = x \cdot y$).

3.1 Absolute value linearization

Simplified relaxations that allow to handle all numerical FP values with a single set of two inequalities require absolute values. Absolute values can either be loosely approximated by three linear inequalities or by a tighter decomposition based on big M rewriting method:

$$\begin{cases} z = z_p - z_n \\ |z| = z_p + z_n \\ 0 \leq z_p \leq M \times b \\ 0 \leq z_n \leq M \times (1-b) \end{cases}$$

where b is a boolean variable, z_p and z_n are real positive variables and, M is a FP number such that $M \geq max\{|\underline{z}|, |\overline{z}|\}$. The method separates z_p, the positive values of z, from z_n, its negative values. When $b = 1$, z gets its positive values and we have $z = z_p = |z|$. If $b = 0$, z gets its negative values and we have $z = -z_n$ and $|z| = z_n$.

If the underlying MILP solver allows indicator constraints, the two last set of inequalities can be replaced by:

$$\begin{cases} b = 0 \rightarrow z_p = 0 \\ b = 1 \rightarrow z_n = 0 \end{cases}$$

3.2 Linearization of nonlinear operations

Bilinear terms, square terms, and quotient linearizations are based on standard techniques used by Sahinidis et al [24]. They have been also used in the Quad system [15] designed to solve constraints over the real numbers. $x \times y$ is linearized according to Mc Cormick [18]:

Let $x \in [\underline{x}, \overline{x}]$ and $y \in [\underline{y}, \overline{y}]$, then

$$\begin{cases} z - \underline{x}y - \underline{y}x + \underline{x}\underline{y} \geq 0 \\ -z + \underline{x}y + \overline{y}x - \underline{x}\overline{y} \geq 0 \\ -z + \overline{x}y + \underline{y}x - \overline{x}\underline{y} \geq 0 \\ z - \overline{x}y - \overline{y}x + \overline{x}\overline{y} \geq 0 \end{cases}$$

These linearizations have been proved to be optimal by Al-Khayyal and Falk [1].

Each time $x = y$, i.e., in case of $z = x \otimes x$, the linearization can be improved. x^2 convex hull is underestimated by all the tangents at x^2 curve between $\underline{x}$ and $\overline{x}$ and overestimated by the line that join $(\underline{x}, \underline{x}^2)$ to $(\overline{x}, \overline{x}^2)$. A good balance is obtained with the two tangents at the bounds of x. Thus, x^2 linearization yields:

$$\begin{cases} z + \underline{x}^2 - 2\underline{x}x \geq 0 \\ z + \overline{x}^2 - 2\overline{x}x \geq 0 \\ (\underline{x} + \overline{x})x - z - \underline{x}\overline{x} \geq 0 \\ z \geq 0 \end{cases}$$

The division takes advantage of the properties of real arithmetic: the essential observation is that $z = x/y$ is equivalent to $x = z \times y$. Therefore, Mc Cormick [18] linearizations can be used here. These linearizations need the bounds of z which can directly be computed by interval arithmetic:

$$[\underline{z}, \overline{z}] = [\nabla(min(\underline{x}/\underline{y}, \underline{x}/\overline{y}, \overline{x}/\underline{y}, \overline{x}/\overline{y})), \\ \Delta(max(\underline{x}/\underline{y}, \underline{x}/\overline{y}, \overline{x}/\underline{y}, \overline{x}/\overline{y}))]$$

where ∇ and Δ are respectively the rounding modes towards $-\infty$ and $+\infty$.

4 Filtering algorithm

The proposed filtering algorithm relies on the linearizations of the relaxations over $\mathcal{R}$ of the initial problem to attempt to shrink the domain of the variables by means of a MILP solver. Algorithm 1 details the steps of this filtering process.

First, function **Approximate** relaxes initial FP constraints to nonlinear constraints over $\mathcal{R}$. Then, function **Linearize** linearizes the nonlinear terms of these relaxations to get a MILP.

The filtering loop starts with a call to FP2B, a filtering process relying on an adaptation of 2B-consistency to FP constraints that attempts to reduce the bounds of the variables. FP2B propagates bound values to intermediate variables. The cost of this filtering process is quite light: it stops as soon as domain size reduction between two iterations is less than 10%. Thanks to function **UpdateLinearizations**, newly computed bounds are used to tighten the MILP. Note that this function updates variable domains as well as linearization coefficients.

After that, MILP is used to compute a lower bound and an upper bound of the domain of each variable by means of function **safeMin**. This function computes a safe global minimizer of the MILP.

This process is repeated until the percentage of reduction of the domains of the variables is lower than a given ϵ.

Algorithm 1 FPLP

1: **Function** $FPLP(\mathcal{V}, \mathcal{D}, \mathcal{C}, \epsilon)$
2: % $\mathcal{V}$: FP variables
3: % $\mathcal{D}$: Domains of the variables
4: % $\mathcal{C}$: Constraints over FP numbers
5: % ϵ: Minimal reduction between two iterations
6: $\mathcal{C}' \leftarrow$ **Approximate** $(\mathcal{C})$;
7: $\mathcal{C}'' \leftarrow$ **Linearize** $(\mathcal{C}', \mathcal{D})$;
8: $boxSize \leftarrow \sum_{x \in \mathcal{V}} (\overline{x}_{\mathcal{D}} - \underline{x}_{\mathcal{D}})$;
9: **repeat**
10: $\quad \mathcal{D}' \leftarrow$ **FP2B**$(\mathcal{V}, \mathcal{D}, \mathcal{C}, \epsilon)$;
11: $\quad$ **if** $\emptyset \in \mathcal{D}'$ **then**
12: $\quad\quad$ **return** $\emptyset$;
13: $\quad$ **end if**
14: $\quad \mathcal{C}'' \leftarrow$ **UpdateLinearizations**$(\mathcal{C}'', \mathcal{D}')$;
15: $\quad$ **for all** $x \in \mathcal{V}$ **do**
16: $\quad\quad [\underline{x}_{\mathcal{D}'}, \overline{x}_{\mathcal{D}'}] \leftarrow [$**safeMin**$(x, \mathcal{C}''),\ -$**safeMin**$(-x, \mathcal{C}'')]$;
17: $\quad\quad$ **if** $[\underline{x}_{\mathcal{D}'}, \overline{x}_{\mathcal{D}'}] = \emptyset$ **then**
18: $\quad\quad\quad$ **return** $\emptyset$;
19: $\quad\quad$ **end if**
20: $\quad$ **end for**
21: $\quad oldBoxSize \leftarrow boxSize$;
22: $\quad boxSize \leftarrow \sum_{x \in \mathcal{V}} (\overline{x}_{\mathcal{D}'} - \underline{x}_{\mathcal{D}'})$;
23: $\quad \mathcal{D} \leftarrow \mathcal{D}'$
24: **until** $boxSize \geq oldBoxSize * (1 - \epsilon)$;
25: **return** $\mathcal{D}$;

4.1 Getting a safe minimizer

Using an efficient MILP solver like CPLEX to filter the domains of the variables raises two important issues related to FP computations.

First, linearization coefficients are computed with FP arithmetic and are subject to rounding errors. Therefore, to avoid the loss of solutions, special attention must be paid to the rounding directions. Correct linearizations rely on FP computations done using the right rounding directions. For instance, consider the linearization of x^2 where $\underline{x} \geq 0$ and $\overline{x} \geq 0$:

$$\begin{cases} y + \Delta(\underline{x}^2) - \Delta(2\underline{x})x \geq 0 \\ y + \Delta(\overline{x}^2) - \Delta(2\overline{x})x \geq 0 \\ \Delta(\underline{x} + \overline{x})x - y - \nabla(\underline{x}\overline{x}) \geq 0 \\ y \geq 0 \end{cases}$$

This process that ensures that all the linearizations are safe is called within the **Linearize** and **UpdateLinearizations** functions. For more details on how to compute safe coefficients see [19, 4].

				2B	3B		FPLP (without 2B)		FPLP	
Program	n	n_T	n_B	$t(ms)$	$t(ms)$	%(2B)	$t(ms)$	%(2B)	$t(ms)$	%(2B)
`Absorb1`	2	1	1	TO	TO	-	3	98.91	5	98.91
`Absorb2`	2	1	1	1	24	0.00	3	100.00	4	100.00
`Fluctuat1`	3	12	2	4	156	99.00	264	99.00	172	99.00
`Fluctuat2`	3	10	2	1	4	0.00	29	0.00	21	0.00
`MeanValue`	4	28	6	3	82	97.45	530	97.46	78	97.46
`Cosine`	5	33	7	5	153	33.60	104	33.61	43	33.61
`SqrtV1`	11	140	29	9	27198	99.63	1924	100.00	1187	100.00
`SqrtV2`	21	80	17	7	TO	-	2337	100.00	1321	100.00
`SqrtV3`	5	46	8	5	573	53.80	185	54.83	82	54.83
`Sine taylor`	6	44	9	5	452	63.29	313	63.29	227	63.29
`Sine iter`	16	109	21	8	4503	39.20	5885	39.31	165	39.31
`Qurt`	6	21	3	4	26	43.56	163	43.56	38	43.56
`Poly`	6	51	9	5	1569	49.17	765	76.66	309	76.66
`Newton`	7	69	14	5	1542	45.16	479	45.16	195	45.16

Table 3. Experiments

Second, efficient MILP solvers use FP arithmetic. Thus, the computed minimizer might be wrong. The unsafe MILP solver is made safe thanks to the correction procedure introduced in [23]. It consists in computing a safe lower bound of the global minimizer. The **safeMin** function implements these corrections and return a safe minimizer of the MILP.

5 Experiments

This section compares the results of different filtering techniques for FP constraints with the method introduced in this paper. Experiments have been done on a laptop with an Intel Duo Core at 2.8Ghz and 4Gb of memory running under Linux.

Our experiments are based on the following set of benchmarks:

- `Absorb 1` detects if, in a simple addition, x absorbs y while `Absorb 2` checks if y absorbs x.
- `Fluctuat1` and `Fluctuat2` are program pathes that come from a presentation of the Fluctuat tool in [12].
- `MeanValue` returns true if an interval contains a mean value and false otherwise.
- `Cosine` is a program that computes the function $cos()$ with a Taylor formula.
- `SqrtV1` computes sqrt in $[0.5, 2.5]$ using a two variable iterative method.
- `SqrtV2` computes sqrt with a Taylor formula.

- `SqrtV3` computes the square root of $(x+1)$ using a Taylor formula. This program comes from CDFPL benchmarks[4].
- `Sine taylor` computes the function sine using a Taylor formula.
- `Sine iter` computes the function sine with an iterative method and comes from the SNU real time library[5].
- `Qurt` computes the real and imaginary roots of a quadratic equation and also comes from the SNU library.
- `Poly` tries to compare two different writings of a polynomial. This program is available on Eric Goubault web page[6]
- `Newton` computes one or two iterations of a Newton on the polynomial $x - x^3/6 + x^5/120x^7/5040$ and comes from CDFPL benchmarks.

Table 3 summarizes experiment results for the following filtering methods: FP2B, an adaptation of 2B-consistency to FP constraints that takes advantage of the property described in [17] to avoid some slow convergences, FP3B, an adaptation of 3B-consistency to FP constraints, FPLP(without FP2B), an implementation of algorithm 1 without the call to FP2B and, FPLP, an implementation of algorithm 1. First column of table 3 gives program's names, column 2 gives the number of variables of the initial problem and column 3 gives the amount of temporary variables used to decompose complex expressions in elementary operations. Column 4 gives the number of binary variables used by FPLP. For each filtering algorithm, table 3 gives the amount of milliseconds required to filter the constraints (columns $t(ms)$). For all filtering algorithm but FP2B, table 3 gives also the percentage of reduction compared to the reduction obtained by FP2B (columns %(FP2B)). The time out (TO) was set to 2 minutes.

The results from table 3 show that FPLP achieves much better domain reductions than 2B-consistency and 3B-consistency filtering algorithms. FPLP requires more times than FP2B but the latter achieves a very weaker pruning on theses benchmarks. This is exemplified by the two `Absorb1` and `SqrtV1` benches. Here, FP2B suffers from the multiple occurrences of the variables. FPLP also consistently outperforms FP3B : it almost always provides much smaller domains and it requires much less time.

A comparison of FPLP with and without a call to FP2B shows that a cooperation between these two filtering methods can significantly decrease the computation time but does not change the filtering capabilities.

6 Conclusion

In this paper, we have introduced a new filtering algorithm for handling constraints over FP numbers. This algorithm benefits from the linearizations of the relaxations over $\mathcal{R}$ of the initial constraints over $\mathcal{F}$ to reduce the domains of the variables with a MILP solver. Experiments show that FPLP drastically improves

[4] See http://www.cprover.org/cdfpl/.
[5] See http://archi.snu.ac.kr/realtime/
[6] See http://www.lix.polytechnique.fr/~goubault/.

the filtering process, especially when combined with a FP2B filtering process. MILP benefits from a more global view of the constraint system than local consistencies, and thus provides an effective way to handle multiple occurrences of variables.

Additional experiments are required to better understand the interactions between the two algorithms and to improve their performances.

References

1. F.A. Al-Khayyal and J.E. Falk. Jointly constrained biconvex programming. *Mathematics of Operations Research*, pages 8:2:273–286, 1983.
2. Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using smt solvers instead of sat solvers. *Int. J. Softw. Tools Technol. Transf.*, 11:69–83, January 2009.
3. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, 1999. Springer-Verlag.
4. Glencora Borradaile and Pascal Van Hentenryck. Safe and tight linear estimators for global optimization. *Mathematical Programming*, 2005.
5. Bernard Botella, Arnaud Gotlieb, and Claude Michel. Symbolic execution of floating-point computations. *Softw. Test., Verif. Reliab.*, 16(2):97–121, 2006.
6. Angelo Brillout, Daniel Kroening, and Thomas Wahl. Mixed abstractions for floating-point arithmetic. In *Proceedings of FMCAD 2009*, pages 69–76. IEEE, 2009.
7. Liqian Chen, Antoine Miné, and Patrick Cousot. A sound floating-point polyhedra abstract domain. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 3–18, Berlin, Heidelberg, 2008. Springer-Verlag.
8. Hélène Collavizza, Michel Rueher, and Pascal Hentenryck. CPBPV: a constraint-programming framework for bounded program verification. *Constraints*, 15(2):238–264, 2010.
9. Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. Smt-based bounded model checking for embedded ansi-c software. *IEEE Transactions on Software Engineering*, May 2011.
10. Patrick Cousot, Radhia Cousot, Jerome Feret, Antoine Miné, Laurent Mauborgne, David Monniaux, and Xavier Rival. Varieties of static analyzers: A comparison with astree. In *TASE '07*, pages 3–20. IEEE, 2007.
11. Malay K Ganai and Aarti Gupta. Accelerating high-level bounded model checking. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, ICCAD '06, pages 794–801, New York, NY, USA, 2006. ACM.
12. K. Ghorbal, E. Goubault, and S. Putot. A logical product approach to zonotope intersection. In *Computer Aided Verification*, pages 212–226. Springer, 2010.
13. Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. In *ISSTA*, pages 53–62, 1998.
14. Arnaud Gotlieb, Bernard Botella, and Michel Rueher. A clp framework for computing structural test data. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic*, volume 1861 of *Lecture Notes in Computer Science*, pages 399–413. Springer, 2000.

15. Yahia Lebbah, Claude Michel, Michel Rueher, David Daney, and Jean-Pierre Merlet. Efficient and safe global constraints for handling numerical constraint systems. *SIAM J. Numer. Anal*, 42:2076–2097, 2005.
16. Olivier Lhomme. Consistency techniques for numeric csps. In *IJCAI*, pages 232–238, 1993.
17. Bruno Marre and Claude Michel. Improving the floating point addition and subtraction constraints. In David Cohen, editor, *CP*, volume 6308 of *Lecture Notes in Computer Science*, pages 360–367. Springer, 2010.
18. G.P. McCormick. Computability of global solutions to factorable nonconvex programs – part i – convex underestimating problems. *Mathematical Programming*, 10:147–175, 1976.
19. C. Michel, Y. Lebbah, and M. Rueher. Safe embedding of the simplex algorithm in a CSP framework. In *Proc. of CPAIOR 2003, CRT, Université de Montréal*, pages 210–220, 2003.
20. Claude Michel. Exact projection functions for floating point number constraints. In *AMAI*, 2002.
21. Claude Michel, Michel Rueher, and Yahia Lebbah. Solving constraints over floating-point numbers. In Toby Walsh, editor, *CP*, volume 2239 of *Lecture Notes in Computer Science*, pages 524–538. Springer, 2001.
22. Antoine Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, December 2004.
23. A. Neumaier and O. Shcherbina. Safe bounds in linear and mixed-integer programming. *Math. Programming A.*, page 99:283 296, 2004.
24. Hong S. Ryoo and Nikolaos V. Sahinidis. A branch-and-reduce approach to global optimization. *Journal of Global Optimization*, pages 107–138, 1996.