Searching input values hitting suspicious Intervals in programs with floating-point operations

Hélène Collavizza, Claude Michel, Michel Rueher

▶ **To cite this version:**

Hélène Collavizza, Claude Michel, Michel Rueher. Searching input values hitting suspicious Intervals in programs with floating-point operations. 2015. hal-01224009v1

HAL Id: hal-01224009

https://hal.science/hal-01224009v1

Preprint submitted on 3 Nov 2015 (v1), last revised 4 Aug 2016 (v2)

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

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

Searching input values hitting suspicious Intervals in programs with floating-point operations*

Hélène Collavizza, Claude Michel, and Michel Rueher

University of Nice–Sophia Antipolis, I3S/CNRS
BP 121, 06903 Sophia Antipolis Cedex, France
firstname.lastname@i3s.unice.fr

Abstract. Programs with floating-point computations are often derived from mathematical models or designed with the semantics of the real numbers in mind. However, for a given input, the computed path with floating-point numbers may differ from the path corresponding to the same computation with real numbers. A common practice when validating such programs consists in estimating the accuracy of floating-point computations with respect to the same sequence of operations in an idealized semantics of real numbers. However, state-of-the-art tools compute an over-approximation of the error introduced by floating-point operations. As a consequence, totally inappropriate behaviors of a program may be dreaded but the developer does not know whether these behaviors will actually occur, or not. In this paper, we introduce a new constraint-based approach that searches for test cases in the part of the over-approximation where errors due to floating-point arithmetic would lead to inappropriate behaviors.

1 Introduction

In numerous applications, programs with floating-point computations are derived from mathematical models over the real numbers. However, computations on floating-point numbers are different from calculations in an idealized semantics[1] of real numbers [9]. For some values of the input variables, the result of a sequence of operations over the floating-point numbers can be significantly different from the result of the corresponding mathematical operations over the real numbers. As a consequence, the computed path with floating-point numbers may differ from the path corresponding to the same computation with real numbers. This can entail wrong and dangerous behaviors of critical systems. That's why identifying these values is a crucial issue for programs controlling critical systems.

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

[1] That's to say, computations as close as possible to the mathematical semantics of the real numbers; for instance, computations with arbitrary precision or computer algebra systems.

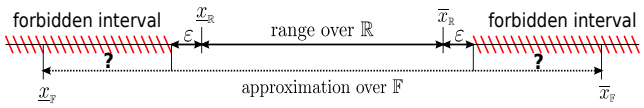

Fig. 1. Suspicious Intervals in Floating-Point Number Programs

Abstract interpretation based error analysis [4, 7] of finite precision implementations compute an over-approximation of the errors due to floating-point operations[2]. The point is that state-of-the-art tools [4, 7, 10] compute an over-approximation of the error introduced by floating-point operations. In [16, 17], we have introduced a hybrid approach combining abstract interpretation and constraint programming techniques that reduces the number of false alarms. However, the remaining false alarms are very embarrassing: inappropriate behaviors of a program may be dreaded but we can not know whether the predicted unstable behaviors will occur with actual data. This problem is depicted in Fig. 1 where:

- $[\underline{x}_{\mathbb{R}}, \overline{x}_{\mathbb{R}}]$ stands for the domain of variable x over $\mathbb{R}$, the set of real numbers;
- $[\underline{x}_{\mathbb{F}}, \overline{x}_{\mathbb{F}}]$ stands for the domain of variable x in the over-approximation computed over $\mathbb{F}$, the set of floating-point numbers.

In practice, the range of a sequence of operations over the real numbers can be determined, whether by calculation or from physical limits. A tolerance ε around this range is usually accepted to take into account approximation errors, e.g. measurement, statistical, or even floating-point arithmetic errors. In other words, this tolerance – specified by the user– defines an acceptable loss of accuracy between the value computed over the floating-point numbers and the value calculated over the real numbers. However, values outside the interval $[\underline{x}_{\mathbb{R}} - \varepsilon, \overline{x}_{\mathbb{R}} + \varepsilon]$ can lead a program to misbehave, e.g. take a wrong branch in the control flow. The values of the approximation over $\mathbb{F}$ that intersect with the forbidden interval are what we call a *suspicious* interval.

The problem we address in this paper consists in verifying whether a program can actually produce values inside the suspicious intervals $[\underline{x}_{\mathbb{F}}, \underline{x}_{\mathbb{R}} - \varepsilon)$ and $(\overline{x}_{\mathbb{R}} + \varepsilon, \overline{x}_{\mathbb{F}}]$. To handle this problem, we introduce a new constraint-based approach that searches for test cases that hit the suspicious intervals in programs with floating-point computations. Broadly speaking, our framework reduces this test case generation problem to a constraint-solving problem over the floating-point numbers where the domain of a critical decision variable has been shrunk to a suspicious interval. If no test case can be generated, the suspicious interval can be discarded.

A constraint solver –based on filtering techniques designed to handle constraints over floating-point numbers– is used to search values for the input data. Preliminary results of experiments on small programs with classical floating-point errors are very encouraging.

[2] Note that only very recent approaches [10] compute sound error bounds in presence of unstable tests.

```
1 /* Pre−condition :  a ≥ b and a ≥ c */
2 float heron(float a, float b, float c) {
3   float s, squared_area;
4   squared_area = 0.0f;
5   if (a <= b + c) {
6     s = (a + b + c) / 2.0f;
7     squared_area = s*(s-a)*(s-b)*(s-c);
8   }
9   return sqrt(squared_area);
10 }
```

Fig. 2. Heron

Before going into the details, we illustrate our approach on a small example. Assume we want to compute the area of a triangle from the lengths of its sides a, b, and c with Heron's formula: $\sqrt{s*(s-a)*(s-b)*(s-c)}$ where $s = (a + b + c)/2$. The C program in Fig. 2 implements this formula, when a is the longest side of the triangle.

The test of line 5 ensures that the given lengths form a valid triangle.

Now, suppose that the input domains are `a` $\in [5, 10]$ and `b`, `c` $\in [0, 5]$. Over the real numbers, `s` is greater than any of the sides of the triangle and `squared_area` cannot be negative. Moreover, `squared_area` cannot be greater than 156.25 over the real numbers since the triangle area is maximized for a right triangle with $b = c = 5$ and $a = 5\sqrt{2}$. However, these properties may not hold over the floating-point numbers because absorption and cancellation phenomena can occur[3].

Tools performing value analysis over the floating-point numbers [7, 16] approximate the domain of `squared_area` to the interval $[-1262.21, 979.01]$. Since this domain is an over-approximation, we do not know whether input values leading to `squared_area` < 0 or `squared_area` > 156.25 actually exist. Note that input domains –here a ∈ [5,10] and b, c ∈ [0,5]– are usually provided by the user.

Assume the value of the tolerance[4] ε is 10^{-5}, the suspicious intervals for `squared_area` are $[-1262.21, -10^{-5})$ and $(156.25001, 979.01]$. CPBPV_FP, the system we developed, generated test cases for both intervals:

- `a` $= 5.517474$, `b` $= 4.7105823$, `c` $= 0.8068917$, and `squared_area` equals $-1.0000001 \cdot 10^{-5}$;
- `a` $= 7.072597$, `b` $=$ `c` $= 5$, and `squared_area` equals 156.25003.

CPBPV_FP could also prove the absence of test cases for a tolerance $\varepsilon = 10^{-3}$ with `squared_area` $> 156.25 + \varepsilon$.

In order to limit the loss of accuracy due to cancellation [9], ligne 7 of Heron's program can be rewritten as folows:

[3] Let's remind that absorption in an addition occurs when adding two numbers of very different orders of magnitude, and the result is the value of the biggest number, i.e., when $x + y$ with $y \neq 0$ yields x. Cancellation occurs in $s - a$ when s is so close to a that the subtraction cancels most of the significant digits of the result.

[4] Note that even this small tolerance may lead to an exception in statement 9.

```
squared_area = ((a+(b+c))*(c-(a-b))*(c+(a-b))*(a+(b-c)))/16.0f;
```

However, there are still some problems with this optimized program. Indeed, CPBPV_FP found the test case `a` $= 7.0755463$, `b` $= 4.350216$, `c` $= 2.72533$, and `squared_area` equals $-1.0000001 \cdot 10^{-5}$ for interval $[-1262.21, -10^{-5})$ of `squared_area`. There are no more problems in the interval $(156.25001, 979.01]$ as it was proven by CPBPV_FP.

2 Framework for generating test cases

This section details the framework we designed to generate test cases reaching suspicious intervals for a variable x in a program P with floating-point computations.

The kernel of our framework is FPCS [15, 14, 2, 13], a constraint solver over floating-point constraints; that's to say a symbolic execution approach for floating-point problems which combines interval propagation with explicit search for satisfiable floating-point assignments. FPCS is used inside the CPBPV bounded model checking framework [6]. We call CPBPV_FP the adaptation of CPBPV for generating test cases that hit the suspicious intervals in programs with floating-point computations.

The inputs of CPBPV_FP are: P, an annotated program; a critical test ct for variable x; $[\underline{x}_{\mathbb{F}}, \underline{x}_{\mathbb{R}} - \varepsilon)$ or $(\overline{x}_{\mathbb{R}} + \varepsilon, \overline{x}_{\mathbb{F}}]$, a suspicious interval for x. Annotations of P specify the range of the input variables of P as well as the suspicious interval for x. The latter assertion is just posted before the critical test ct.

To compute the suspicious interval for x, we approximate the domain of x over the real numbers by $[\underline{x}_{\mathbb{R}}, \overline{x}_{\mathbb{R}}]$, and over the floating-point numbers by $[\underline{x}_{\mathbb{F}}, \overline{x}_{\mathbb{F}}]$. These approximations are computed with RAICP [16], a hybrid system that combines abstract interpretation and constraint programming techniques in a single static and automatic analysis. The current implementation of RAICP is based upon FLUCTUAT [7], REALPAVER [11] and FPCS. So, the suspicious intervals for x are $[\underline{x}_{\mathbb{F}}, \underline{x}_{\mathbb{R}} - \varepsilon)$ and $(\overline{x}_{\mathbb{R}} + \varepsilon, \overline{x}_{\mathbb{F}}]$, where ε is a tolerance specified by the user.

CPBPV_FP performs first some pre-processing: P is transformed into DSA-like form[5]. If the program contains loops, CPBPV_FP unfolds loops k times where k is a user specified constant. Loops are handled in CPBPV and RAICP with standard unfolding and abstraction techniques. So, there are no more loops in the program when we start the constraint generation process. Standard slicing operations are also performed to reduce the size of the control flow graph.

In a second step, CPBPV_FP searches for executable paths reaching ct. For each of these paths, the collected constraints are sent to FPCS, which solves the corresponding constraint systems over the floating point numbers. FPCS returns either a satisfiable instantiation of the input variables of P or $\emptyset$.

[5] DSA stands for Dynamic Single Assignment. In DSA-like form, all variables are assigned exactly once in each execution path. In bounded model checking k is usually incremented until a counterexample is found or until the number of time units is large enough for the application.

As said before, FPCS [15, 14, 2, 13] is a constraint solver designed to solve a set of constraints over floating-point numbers without losing any solution. It uses $2B$-consistency along with projection functions adapted to floating-point arithmetic [14, 2] to filter constraints over the floating-point numbers. FPCS provides also stronger consistencies like kB-consistencies, which allow better filtering results. FPCS allows one to reason correctly over the floating-point numbers with respect to the floating-point arithmetic.

The search of solutions in constraint systems over floating numbers is more tricky than the standard bisection-based search in constraint systems over intervals of real numbers. Thus, we have also implemented different strategies combining selection of specific points and pruning. Details on theses strategies are given in the experiments section.

CPBPV_FP ends up with one of the following results:

- a test case proving that P can produce a suspicious value for x;
- a proof that no test case reaching the suspicious interval can be generated: this is the case if the loops in P cannot be unfolded beyond the bound k (See [6] for details on bounded unfoldings) ;
- an inconclusive answer: no test case could be generated but the loops in P could be unfolded beyond the bound k. In other words, the process is incomplete and we cannot conclude whether P may produce a suspicious value.

3 Preliminary experiments

We experimented with CPBPV_FP on six small programs with cancellation and absorption phenomena, two very common pitfalls of floating-point arithmetic. The benchmarks are listed in the two first columns of table 1.

The first two benchmarks concern the `heron` program and the `optimized_heron` program with the suspicious intervals described in the section 1.

Program `slope` (see Fig. 3) approximates the derivative of the square function $f(x) = x^2$ at a given point x_0. More precisely, it computes the slope of a nearby secant line with a finite difference quotient: $f'(x_0) \approx \frac{f(x_0+h)-f(x_0-h)}{2h}$. Over the real numbers, the smaller h is, the more accurate the formula is. For this function, the derivative is given by $f'(x) = 2x$ which yields exactly 26 for $x = 13$. Over the floats, FLUCTUAT [7] approximates the return value of the slope program to the interval $[0, 25943]$ when $h \in [10^{-6}, 10^{-3}]$ and $x_0 = 13$.

Program `polynomial` in Fig. 4 illustrates an absorption phenomeno. It computes the polynomial $(a^2+b+10^{-5})*c$. For input domains $a \in [10^3, 10^4]$, $b \in [0, 1]$ and $c \in [10^3, 10^4]$, the minimum value of the polynomial over the real numbers is equal to 1000000000.01.

`simple_interpolator` and `simple square` are two benchmarks extracted from [10]. The first benchmark computes an interpolator, affine by sub-intervals while the second is a rewrite of a square root function used in an industrial context.

```
float slope(float x0, float h) {
  float x1 = x0 + h; float x2 = x0 - h;
  float fx1 = x1*x1; float fx2 = x2*x2;
  float res = (fx1 - fx2) / (2.0*h);
  return res;
}
```

Fig. 3. Approximation of the derivative of x^2 by a slope

```
float polynomial(float a, float b, float c) {
  float poly = (a*a + b + 1e-5f) * c;
  return poly;
}
```

Fig. 4. Computation of polynomial $(a^2 + b + 10^{-5}) * c$

All experiments were done on an Intel Core 2 Duo at 2.8 GHz with 4 GB of memory running 64-bit Linux. We assume C programs handling IEEE 754 compliant floating-point arithmetic, intended to be compiled with GCC without any optimization option and run on an x86_64 architecture managed by a 64-bit Linux operating system. Rounding mode was to the nearest, i.e., where ties round to the nearest even digit in the required position.

3.1 Strategies and solvers

We run CPBPV_FP with the following search strategies for the FPCS solver:

- `std`: standard prune & bisection-based search used in constraint-systems over intervals : we split the domain of the selected variable in two domains of equal size;
- `fpc`: we split the domain of the selected variable in five intervals:
  - 3 degenerated intervals containing only a single floating point number: the smallest float l, the largest float r, and the mid-point m;
  - open interval (l, m);
  - open interval (m, r);
- `fp3s`: we select 3 degenerated intervals containing only a single floating point number: the smallest float l, the largest float r, and the mid-point m.

For all these strategies, we select first the variables with the largest domain and we perform after a 3B−consistency filtering step before starting the splitting process.

We compared CPBPV_FP with CBMC [5] and CDFL [8], two state-of-the-art software bounded model checkers based on SAT solvers that are able to deal with floating-point computations. We also run a simple generate & test strategy: the program is run with randomly generated input values and we test whether the result is inside the suspicious interval. The process is stopped as soon as a test case hitting the suspicious interval is found.

Name	Condition	CDFL	CBMC	std	fpc	fpc3s	s?
`heron`	$area < 10^{-5}$	3.874s	0.280s	> 180	0.705	0.022 (n)	y
	$area > 156.25 + 10^{-5}$	> 180s	34.512s	22.323	7.804	0.083 (n)	y
`optimized_heron`	$area < 10^{-5}$	7.618s	0.932s	> 180	0.148	0.022 (n)	y
	$area > 156.25 + 10^{-5}$	> 180s	> 180s	8.988	30.477	0.101 (n)	n
`slope` with $h \in [10^{-6}, 10^{-3}]$	$dh < 26.0 - 1.0$	2.014s	1.548s	0.021	0.012	0.012 (y)	y
	$dh > 26.0 + 1.0$	1.599s	0.653s	0.055	0.011	0.011 (y)	y
	$dh < 26.0 - 10.0$	0.715s	1.108s	0.006	0.006	0.007 (n)	n
	$dh > 26.0 + 10.0$	1.025s	1.080s	0.006	0.006	0.006 (n)	n
`slope` with $h \in [10^{-9}, 10^{-6}]$	$dh < 26.0 - 1.0$	0.299s	0.241s	0.013	0.007	0.007 (y)	y
	$dh > 26.0 + 1.0$	0.333s	0.246s	0.015	0.007	0.007 (y)	y
	$dh < 26.0 - 10.0$	0.291s	0.224s	0.013	0.007	0.007 (y)	y
	$dh > 26.0 + 10.0$	0.342s	0.436s	0.016	0.007	0.007 (y)	y
`polynomial`	$r < 10^9 + 0.0099999904 - 10^{-3}$	0.170s	0.295s	0.022	0.006	0.006 (y)	y
`simple_interpolator`	$res < -10^{-5}$	0.296s	0.264s	0.018	0.012	0.012 (y)	y
`simple_square`	$S > 1.453125$	$--$	1.079s	0.012	0.012	0.012 (n)	n

Table 1. Time required by the different solvers and strategies to handle the different benchmarks

3.2 Results

Table 1 reports the results for the other strategies and solvers. Since strategy `fpc3s` is incomplete we indicate whether a test case was found. Column `s?` specifies whether a test case actually exists. Note that the computation times of CBMC and CDFL include the pre-processing time for generating the constraint systems; the pre-processing time required by CPBPV are around 0.6s but CPBPV is a non-optimised system written in `java`.

4 Discussion

The generate & test strategy behaves quite well on programs with only one input variable when a test case exists but it is unable to find any test case for programs with more than one input variable. More precisely, it found a test case in less than 0.008s for the 6 suspicious intervals of program `slope` where a test case exists as well as for program `simple_interpolator`.

Strategy `fpc` is definitely the most efficient and most robust on all these benchmarks. Note that CBMC and CDFL could handle neither the initial nor the optimized version of program `heron` in a timeout of 20 minutes whereas FPCS found solutions in reasonable time.

These preliminary results are very encouraging: they show that our approach is effective for generating test cases for suspicious values outside the range of acceptable values on small programs with classical floating-point errors. More importantly, a strong point of CP is definitely its refutation capabilities.

Of course, experiments on more significant benchmarks and on real applications are still necessary to evaluate the full capabilities and limits of CPBPV_FP.

4.1 Related work

The goals of software bounded model checkers based on SAT solvers are close to our approach. The point is that SAT solvers tend to be inefficient on these problems due to the size of the domains of floating-point variables and the cost of bit-vector operations [8]. CDFL [8] tries to address this issue by embedding an abstract domain in the conflict driven clause learning algorithm of a SAT solver. Note that the constraint programming techniques used in our approach are better suited to generate several test cases than these SAT-based approaches. The advantage of CP is that it provides a uniform framework for representing and handling integers, real numbers and floats. SAT solvers often use bitwise representations of numerical operations, which may be very expensive (e.g., thousands of variables for one equation in CDFL).

4.2 Further work

We have recently designed a new constraint solver FPLP [1] which relies on relaxations over the real numbers of a problem over the floating-point numbers. Safe bounds of the domains are computed with a mixed integer linear programming solver (MILP) on safe linearizations of these relaxations. FPLP has a more global view of the problem than FPCS, and thus, should be able to provide better bounds. So, both solvers could be combined to improve the filtering process and get tighter domains for the variables of the problem.

A new abstract-interpretation based robustness analysis of finite precision implementations has recently been proposed [10] for sound rounding error propagation in a given path in presence of unstable tests.

Brain and al [12, 3] have recently introduced a bit-precise decision procedure for the theory of floating-point arithmetic. The core of their approach is a generalisation of the conflict-driven clause-learning algorithm used in modern SAT solvers. Their technique is significantly faster than a bit-vector encoding approach.

A close connection between our floating-point solvers and the two above mentioned approaches is certainly worth exploring.

References

1. Mohammed Said Belaid, Claude Michel, and Michel Rueher. Boosting local consistency algorithms over floating-point numbers. In Michela Milano, editor, *18th International Conference on Principles and Practice of Constraint Programming (CP 2012)*, volume 7514 of *Lecture Notes in Computer Science*, pages 127–140. Springer, 2012.
2. Bernard Botella, Arnaud Gotlieb, and Claude Michel. Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability*, 16(2):97–121, 2006.

3. Martin Brain, Vijay D'Silva, Alberto Griggio, Leopold Haller, and Daniel Kroening. Interpolation-based verification of floating-point programs with abstract cdcl. In *Static Analysis - 20th International Symposium SAS*, volume 7935 of *LNCS*. Springer, 2013.
4. 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.
5. Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176, 2004.
6. Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. A constraint-programming framework for bounded program verification. *Constraints Journal*, 15(2):238–264, 2010.
7. David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *FMICS*, volume 5825 of *LNCS*, pages 53–69. Springer, 2009.
8. Vijay D'Silva, Leopold Haller, Daniel Kroening, and Michael Tautschnig. Numeric bounds analysis with conflict-driven learning. In *Proc. TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 48–63. Springer, 2012.
9. David Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.
10. Eric Goubault and Sylvie Putot. Robustness analysis of finite precision implementations. In *Programming Languages and Systems - 11th Asian Symposium, APLAS*, volume 8301 of *LNCS*, pages 50–57. Springer, 2013.
11. Laurent Granvilliers and Frédéric Benhamou. Algorithm 852: RealPaver: an interval solver using constraint satisfaction techniques. *ACM Transactions on Mathematical Software*, 32(1):138–156, 2006.
12. Leopold Haller, Alberto Griggio, Martin Brain, and Daniel Kroening. Deciding floating-point logic with systematic abstraction. In *Formal Methods in Computer-Aided Design, FMCAD*, pages 131–140. IEEE, 2012.
13. Bruno Marre and Claude Michel. Improving the floating point addition and subtraction constraints. In *CP*, volume 6308 of *LNCS*, pages 360–367. Springer, 2010.
14. Claude Michel. Exact projection functions for floating-point number constraints. In *7th International Symposium on Artificial Intelligence and Mathematics*, 2002.
15. Claude Michel, Michel Rueher, and Yahia Lebbah. Solving constraints over floating-point numbers. In *CP*, volume 2239 of *LNCS*, pages 524–538. Springer Verlag, 2001.
16. Olivier Ponsini, Claude Michel, and Michel Rueher. Refining abstract interpretation based value analysis with constraint programming techniques. In Michela Milano, editor, *18th International Conference on Principles and Practice of Constraint Programming (CP 2012)*, volume 7514 of *Lecture Notes in Computer Science*, pages 593–607. Springer, 2012.
17. Olivier Ponsini, Claude Michel, and Michel Rueher. Verifying floating-point programs with constraint programming and abstract interpretation techniques. *Automated Software Engineering*, pages 1–27, 2014.