Generating Test Cases inside Suspicious Intervals for Floating-Point Number Programs[*]

Hélène Collavizza
University of Nice–Sophia Antipolis, I3S/CNRS
BP 121, 06903 Sophia Antipolis Cedex, France
helen@polytech.unice.fr

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

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

Michel Rueher
University of Nice–Sophia Antipolis, I3S/CNRS
BP 121, 06903 Sophia Antipolis Cedex, France
Michel.Rueher@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. State-of-the-art tools compute an over-approximation of the error introduced by floating-point operations with respect to the same sequence of operations in an idealized semantics of real numbers. Thus, totally inappropriate behaviors of a program may be dreaded but the developer does not know whether these behaviors will actually occur, or not. We introduce here a new constraint-based approach that searches for input values hitting the part of the over-approximation where errors due to floating-point arithmetic would lead to inappropriate behaviors. Preliminary results of experiments on small programs with classical floating-point errors are very encouraging.

[*]This work was partially supported by ANR VACSIM (ANR-11-INSE-0004), ANR AEOLUS (ANR-10-SEGI-0013), and OSEO ISI PAJERO projects.
Part of this work was done while Michel Rueher was visiting professor at NII (National Institute of Informatics).

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Model checking; D.2.5 [**Testing and Debugging**]: Debugging aids,Testing tools

General Terms

Verification

Keywords

Floating-point computations, constraint systems, satisfiable floating-point assignments, unstable programs, test case generation.

1. INTRODUCTION

In numerous applications, programs with floating-point computations are derived from mathematical models over the real numbers. However, for some values of the input variables, the result of a sequence of operations over the floating-point numbers can be significantly different from calculations in an idealized semantics of the real numbers. As a consequence, computed paths with floating-point numbers may differ from the paths corresponding to the same computation with real numbers. Identifying these values is a crucial issue for programs controlling critical systems.

Abstract interpretation based error analysis [3, 6, 9] of finite precision implementations compute an over-approximation of the errors due to floating-point operations. Inappropriate behaviors of a program may thus be dreaded but we cannot 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; and $[\underline{x}_{\mathbb{F}}, \overline{x}_{\mathbb{F}}]$ 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 often 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 an approximation over $\mathbb{F}$ that intersect with a 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 generates test cases hitting the suspicious intervals in programs with floating-point computations. 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. The generated test cases not only demonstrate that wrong paths can actually be executed but they provide also valuable information for debugging. If no test case can be generated, the suspicious interval can be discarded.

2. FRAMEWORK FOR TEST CASES GENERATION

The kernel of our framework is FPCS [14, 13, 1, 12], 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 [5] bounded model checking framework. CPBPV_FP stands for the adaptation of CPBPV for generating test cases hitting the suspicious intervals in programs with floating-point computations.

The inputs of CPBPV_FP are :

- P, an annotated program;
- x, a variable used in a critical test; and
- $[\underline{x_F}, \overline{x_F}]$, 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. Usually, these annotations are posted just before a critical test using variable x.

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 $]\underline{x}_{\mathbb{F}}, \overline{x}_{\mathbb{F}}]$. 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.

These approximations are mainly computed with rAiCp [15], a hybrid system that combines abstract interpretation and constraint programming techniques in a single static and automatic analysis. rAiCp is substantially more precise than state-of-the-art AI analyzers and CP solvers used separately. Of course, computer algebra systems or interval solvers can also help for computing these approximations. Safe approximations computed with different tools can even be intersected.

CPBPV_FP performs first some pre-processing: P is transformed into DSA-like form.[1] Loops are handled in CPBPV with standard unfolding and abstraction techniques: if the program contains loops, CPBPV_FP unfolds loops k times.[2] 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 suspicious interval for x. For each of these paths, the collected constraints are sent to FPCS, which solves the corresponding constraint systems over the floating point numbers and returns either a satisfiable instantiation of the input variables of P or an empty set. As said before, FPCS [14, 13, 1, 12] is a constraint solver designed to solve a set of constraints over floating-point numbers without losing any solution. It uses $2B$-consistency[3] along with projection functions adapted to floating-point arithmetic [13, 1] to filter constraints over the floating-point numbers. FPCS provides also stronger consistencies like $3B$-consistencies[4], which allow better filtering results. FPCS allows one to reason correctly over the floating-point numbers with respect to the floating-point arithmetic.

CPBPV_FP ends up with one of the following results:

- A test case showing 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 [5] 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 two cases of floating-point arithmetic pitfalls: programs with cancellation and absorption phenomena.[5]

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

[1]DSA stands for Dynamic Single Assignment. In DSA-like form, all variables are assigned exactly once in each execution path.

[2]k is usually incremented until a counterexample is found or until the number of time units is large enough for the application.

[3]$2B$-consistency [11] states a local property on the bounds of the domains of a variable at a single constraint level. Informally, 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 holds when x is set to the upper or lower bound of its domain.

[4]$3B$-consistency [11] 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.

[5]Absorption in an addition occurs when adding two numbers of very different order 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 s and a.

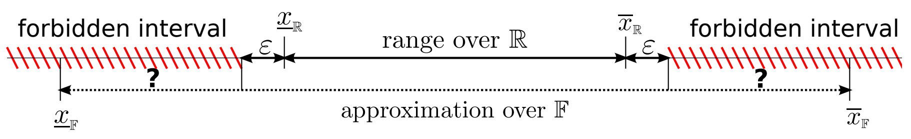
forbidden interval
$\underline{x}_{\mathbb{R}}$
ε
range over $\mathbb{R}$
$\overline{x}_{\mathbb{R}}$
ε
forbidden interval
?
?
$\underline{x}_{\mathbb{F}}$
approximation over $\mathbb{F}$
$\overline{x}_{\mathbb{F}}$

Figure 1: Can the program hit a forbidden zone over the floating-point numbers?

```
1  /* Pre−condition :  a ≥ b and a ≥ c */
2  float heron(float a, float b, float c) {
3    float s, squared_area;
4
5    squared_area = 0.0f;
6    if (a <= b + c) {
7      s = (a + b + c) / 2.0f;
8      squared_area = s*(s−a)*(s−b)*(s−c);
9    }
10   return sqrt(squared_area);
11 }
```

Figure 2: Program Heron

```
/* Pre−condition :  a ≥ b and b ≥ c */
float optimized_heron(float a, float b, float c) {
  float squared_area;

  squared_area = 0.0f;
  if (a <= b + c) {
    squared_area = ((a+(b+c))*(c−(a−b))
                      *(c+(a−b))*(a+(b−c)))/16.0f;
  }
  return sqrt(squared_area);
}
```

Figure 3: Optimized Heron

C programs handling IEEE 754 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 set to the nearest, i.e., where ties round to the nearest even digit in the required position.

We first describe the programs, and then we compare the performance with existing tools.

3.1 Program `Heron`

The program `Heron` in Fig. 2 computes the area of a triangle from the lengths of its sides a, b, and c; a being the longest side of the triangle. It use Heron's formula:

$$\sqrt{s*(s-a)*(s-b)*(s-c)}$$

where

$$s=(a+b+c)/2.$$

The test of line 6 ensures that the given lengths form a valid triangle.

Suppose that the input domains are:

- `a` $\in [5, 10]$;
- `b`, `c` $\in [0, 5]$.

```
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;
}
```

Figure 4: Program `slope`

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

Figure 5: Programs `polynomial`

Over the real numbers, `s` is greater or equal than any of the sides of the triangle and `squared_area` can neither be negative nor greater than 156.25 since the triangle area is maximized for a right triangle with $b = c = 5$ and $a = 5\sqrt{2}$. Value analysis tools FLUCTUAT [6] and rAiCp[15] 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.

With a tolerance ε of 10^{-5}, the suspicious intervals for `squared_area` are $[-1262.21, -10^{-5}]$ and $[156.25001, 979.01]$. CPBPV_FP managed to generate test cases for both intervals:

- `a` $= 5.517474$, `b` $= 4.7105823$, `c` $= 0.8068917$, and `squared_area` $= -1.0000001 \cdot 10^{-5}$ for suspicious interval $[-1262.21, -10^{-5}]$;
- `a` $= 7.072597$, `b` $=$ `c` $= 5$, and `squared_area` $= 156.25003$ for suspicious interval $[156.25001, 979.01]$.

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

To limit the loss of accuracy due to cancellation [8], line 8 of this program can be rewritten in the following way :

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

On the optimized version of `Heron` (Fig. 3), CPBPV_FP still found a test case

- `a` $= 7.0755463$,
- `b` $= 4.350216$, `c` $= 2.72533$,
- `squared_area` equals $-1.0000001 \cdot 10^{-5}$.

Name	Condition	FPCS	CDFL	CBMC	CPBPV_FP	sol?
`slope` with $h \in [10^{-6}, 10^{-3}]$	$res < 26.0_f - 1.0_f$	0.020s	2.014s	1.548s	0.624s	yes
	$res > 26.0_f + 1.0_f$	0.022s	1.599s	0.653s	0.603s	yes
	$res < 26.0_f - 10.0_f$	0.007s	0.715s	1.108s	0.588s	no
	$res > 26.0_f + 10.0_f$	0.007s	1.025s	1.080s	0.593s	no
`slope` with $h \in [10^{-9}, 10^{-6}]$	$res < 26.0_f - 1.0_f$	0.010s	0.299s	0.241s	0.593s	yes
	$res > 26.0_f + 1.0_f$	0.009s	0.333s	0.246s	0.608s	yes
	$res < 26.0_f - 10.0_f$	0.011s	0.291s	0.224s	0.582s	yes
	$res > 26.0_f + 10.0_f$	0.003s	0.342s	0.436s	0.594s	yes
`heron`	$squared\_aera < 10_f^{-5}$	0.655s	3.874s	0.280s	1.109s	yes
	$squared\_area > 156.25_f + 10_f^{-5}$	1.412s	> 1200s	34.512s	2.294s	yes
`optimized_heron`	$squared\_area < 10_f^{-5}$	0.262s	7.618s	0.932s	0.982s	yes
	$squared\_area > 156.25_f + 10_f^{-5}$	37.352s	> 1200s	> 1200s	95.890s	no
`polynomial`	$r < 1000000000.01_f - 10_f^{-3}$	0.006s	0.170s	0.295s	0.605s	yes
`simple_interpolator`	$res < 10_f^{-5}$	0.017s	0.296s	0.264s	0.613s	yes
`simple_square`	$S > 1.453125$	0.011s	$--$	1.079s	0.608s	no

Table 1: Time required by each solver to solve each problem

when `squared_area` is set to the interval $[-1262.21, -10^{-5}]$. CPBPV_FP did also prove that no test case exists with `squared_area` $\in [156.25001, 979.01]$.

3.2 Program `slope`

This program approximates the derivative of a function at a given point x_0 by computing 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. The program `slope` in Fig. 4 implements this formula with the square function $f(x) = x^2$.

For function $f(x) = x^2$ the derivative is given by

$$f'(x) = 2x$$

which yields exactly 26 for $x = 13$. Over the floats, Fluctuat and rAiCp approximates the return value of the slope program to the interval $[0, 25943]$ when $h \in [10^{-6}, 10^{-3}]$ and $x_0 = 13$.

CPBPV_FP found test cases for the different suspicious intervals with $\epsilon = 1$:

- `h` $= 7.934571 \cdot 10^{-6}$ and `res` $= 24.999998$ when `res` is restricted to interval $[0, 25]$
- `h` $= 2.5324175 \cdot 10^{-6}$ and `res` $= 30.126923$ when `res` is restricted to interval $]27, 25943]$.

CPBPV_FP proved also that no test case exists with $\epsilon = 10$.

The situation gets even worse if we try to increase the accuracy of the formula by decreasing the value of h. For an $\epsilon = 10$ and $h \in [10^{-6}, 10^{-3}]$, CPBPV_FP proved that no test case exists whereas for $h \in [10^{-9}, 10^{-6}]$. CPBPV_FP generated test cases for the different suspicious intervals.

The bad accuracy of program `slope` comes from a catastrophic cancellation phenomenon: `fx1` and `fx2` are very close numbers and the subtraction cancels the most significant digits leaving only the digits coming from the rounding of the previous operations.

3.3 Program `polynomial`

The program `polynomial` (Fig. 5) computes the polynomial

$$(a^2 + b + 10^{-5}) * c$$

and illustrates an absorption phenomenon.

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. CPBPV_FP shows that even with a tolerance of $\varepsilon = 10^{-3}$, there are input values for which the program computes a result less than $1000000000.01 - \varepsilon$.

3.4 Computation times

Solving times are given in Table 1.
Programs `simple_interpolator` and `simple_square` are two benchmarks extracted from [9]: the first benchmark computes an interpolator, affine by sub-intervals; the second is a rewrite of a square root function used in an industrial context. First column gives the name of the program while the second column gives the condition that is checked. Columns 3 to 6 give the time required to solve the problem by the solver which name the column. Column 7 tells whether the problem has a solution or not. Column FPCS specifies only how much time is spent in the constraint solver. Note that values in column CPBPV_FP include the corresponding time of Column FPCS.
CBMC [4] and CDFL [7] are two state of art software bounded model checkers based on SAT solvers that are able to deal with floating-point computations.

On `slope` and `polynomial`, the performance of CBMC and CDFL are very similar to the one of CPBPV_FP: all these systems required little time for finding test cases for these two programs.

Arithmetic expressions are a bit more complex in program `heron` than in the other benchmarks. CPBPV_FP found nevertheless solutions in reasonable time. But CBMC and CDFL could handle neither the initial nor the optimized version of program `heron` within a timeout of 20 minutes.

4. DISCUSSION

These preliminary results of experiments are very encouraging: they show that our approach is effective for generating test cases for suspicious values outside the range of acceptable values on on small programs with classical floating-point errors, and thus, to determine whether critical computations are unstable or not.

Performances of FPCS are especially encouraging. The advantage of CP is that it provides an efficient framework for representing and handling constraints over floating-point numbers. SAT solvers often use bitwise representations of numerical operations, which may be very expensive (e.g., thousands of variables for one equation in CDFL). Another strong point of CP is its refutation capabilities on constraint systems over the floating point numbers. Of course, experiments on more significant benchmarks and on real applications are still necessary to evaluate the full capabilities and limits of CPBPV_FP.

A new abstract-interpretation based robustness analysis of finite precision implementations has recently been proposed [9] for sound rounding error propagation in a given path in presence of unstable tests. Brain et al [10, 2] have recently introduced a bit-precise decision procedure for the theory of point arithmetic. The core of their approach is a generalisation of the conflict-driven clause-learning algorithm used in modern SAT solver. 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.

5. REFERENCES

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

[2] M. Brain, V. D'Silva, A. Griggio, L. Haller, and D. Kroening. Interpolation-based verification of floating-point programs with abstract cdcl. In *Static Analysis - 20th International Symposium SAS*, volume 7935 of *LNCS*. Springer, 2013.

[3] L. Chen, A. Miné, and P. 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.

[4] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176, 2004.

[5] H. Collavizza, M. Rueher, and P. V. Hentenryck. A constraint-programming framework for bounded program verification. *Constraints Journal*, 15(2):238–264, 2010.

[6] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *FMICS*, volume 5825 of *LNCS*, pages 53–69. Springer, 2009.

[7] V. D'Silva, L. Haller, D. Kroening, and M. Tautschnig. Numeric bounds analysis with conflict-driven learning. In *Proc. TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 48–63. Springer, 2012.

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

[9] E. Goubault and S. 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.

[10] L. Haller, A. Griggio, M. Brain, and D. Kroening. Deciding floating-point logic with systematic abstraction. In *Formal Methods in Computer-Aided Design, FMCAD*, pages 131–140. IEEE, 2012.

[11] O. Lhomme. Consistency techniques for numeric CSPs. In *13th International Joint Conference on Artificial Intelligence*, pages 232–238, 1993.

[12] B. Marre and C. Michel. Improving the floating point addition and subtraction constraints. In *CP*, volume 6308 of *LNCS*, pages 360–367. Springer, 2010.

[13] C. Michel. Exact projection functions for floating-point number constraints. In *7th International Symposium on Artificial Intelligence and Mathematics*, 2002.

[14] C. Michel, M. Rueher, and Y. Lebbah. Solving constraints over floating-point numbers. In *CP*, volume 2239 of *LNCS*, pages 524–538. Springer Verlag, 2001.

[15] O. Ponsini, C. Michel, and M. Rueher. Refining abstract interpretation based value analysis with constraint programming techniques. In M. 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.