Hardware-independent proofs of numerical programs

Sylvie Boldo, Thi Minh Tuyen Nguyen

Hardware-independent proofs of numerical programs *

Sylvie Boldo Thi Minh Tuyen Nguyen
INRIA Saclay – Île-de-France, F-91893 Orsay cedex, France
LRI, Univ. Paris-Sud, F-91405 Orsay cedex, France

Abstract

On recent architectures, a numerical program may give different answers depending on the execution hardware and the compilation. Our goal is to formally prove properties about numerical programs that are true for multiple architectures and compilers. We propose an approach that states the rounding error of each floating-point computation whatever the environment. This approach is implemented in the Frama-C platform for static analysis of C code. Small case studies using this approach are entirely and automatically proved.

1 Introduction

Floating-point computations often appear in current critical systems from domains such as physics, aerospace system, energy, etc. For such systems, hardwares and softwares play an important role.

All current microprocessor architectures support IEEE-754 floating-point arithmetic [1]. However, there exist some architecture-dependent issues. For example, the x87 floating-point unit uses the 80-bit internal floating-point registers on the Intel platform. The fused multiply-add (FMA) instruction, supported by the PowerPC and the Intel Itanium architectures, computes $xy \pm z$ with a single rounding. These issues can introduce subtle inconsistencies between program executions. This means that the floating-point computations of a program running on different architectures may be different [2].

Static analysis is an approach for checking a program without running it. Deductive verification techniques which perform static analysis of code, rely on the ability of theorem provers to check validity of formulas in first-order logic or even more expressive logics. They usually come with expressive specification languages such as JML [3, 4] for Java, ACSL [5] for C, Spec# [6] for C#, etc. to specify the requirements. For automatic analysis of floating-point codes, a successful approach is abstract interpretation based static analysis, that includes Astrée [7, 8] and Fluctuat [9].

Floating-point arithmetic has been formalized since 1989 in order to formally prove hardware components or algorithms [10, 11, 12]. There exist less works on specifying and proving behavioral properties of floating-point programs in deductive verification systems. A work on floating-point in JML for Java is presented in 2006 by Leavens [13]. Another proposal has been made in 2007 by Boldo and Filliâtre [14]. Ayad and Marché extended this to increase genericity and handle exceptional behaviors [15].

However, these works only follow the strict IEEE-754 standard, with neither FMA, nor extended registers. Correctly defining the semantics of the common implementations of floating-point is tricky, because semantics may change according to arguments of compilers and processors. As a result, formal verification of such programs is a challenge. The purpose of this paper is to present an approach to prove numerical programs with few restrictions on the compiler and the processor.

More precisely, we require the compiler to preserve the order of operations of the C language and we only consider rounding-to-nearest mode, double precision numbers and computations. Our approach is implemented in the Frama-C platform[1] associated with Why [16] for static analysis of C code.

This paper is organized as follows. Section 2 presents some basic knowledge needed about floating-point arithmetic, including the x87 unit and the FMA. Section 3 presents a bound on the rounding error

*This work was supported by the Hisseo project, funded by Digiteo (`http://hisseo.saclay.inria.fr/`).

[1] `http://frama-c.cea.fr/`

of a computation in all possible cases (extended registers, FMA). Two small case studies are presented in Section 4. These examples illustrate our approach and show the difference of the results between the usual (but maybe incorrect) model and our approach.

2 Floating-point Arithmetic

2.1 The IEEE-754 floating-point standard

The IEEE-754 standard [1] for floating-point arithmetic was developed to define formats and behaviors for floating-point numbers and computations. A floating-point number x in a format (p, e_{min}, e_{max}), where e_{min} and e_{max} are the minimal and maximal unbiased exponents and p is the precision, is represented by the triplet (s, m, e) so that

$$x = (-1)^s \times 2^e \times m \tag{1}$$

where $s \in \{0,1\}$ is the sign of x, e is any integer so that $e_{min} \leq e \leq e_{max}$, m $(0 \leq m < 2)$ is the significand (in p bits) of the representation.

We only consider the binary 64-bit format (usually *double* in C or Java language), that satisfies the format (1) with $(53, -1022, 1023)$, as it concentrates all the problems. Our ideas could be re-used in other formats.

When approximating a real number x by its rounding $\circ(x)$, a rounding error happens. We here consider only round-to-nearest mode, that includes both the default rounding mode (round-to-nearest, ties to even) and the new round-to-nearest, ties away from zero, of the revision of the IEEE-754 standard. In radix 2 and round-to-nearest mode, a bound on the error is known [17].

If $|\circ(x)| \geq 2^{e_{min}}$, x is called normal number. In other words, it is said to be in the normal range. We then bound the relative error:

$$\left|\frac{x - \circ(x)}{x}\right| \leq 2^{-p}. \tag{2}$$

For smaller $|\circ(x)|$, the value of the relative error becomes large (up to 0.5). In that case, x is in the subnormal range and we prefer a bound based on the absolute error:

$$|x - \circ(x)| \leq 2^{e_{min}-p}. \tag{3}$$

2.2 Floating-point computations depend on the architecture

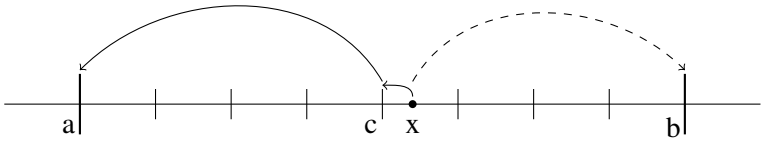
With the same program containing floating-point computations, the result may be different depending on the compiler and the processor. We present in this section some architecture-dependent issues resulting in such problems.

A first cause is the fact that some processors (IBM PowerPC or Intel/HP Itanium) have a *fused multiply-add* (FMA) instruction which computes $(x \times y) \pm z$ as if with unbounded range and precision, and rounds only once to the destination format. This operation can speed up and improve the accuracy of dot product, matrix multiplication and polynomial evaluation, but few processors now support it. But how should $a \times b + c \times d$ be computed? When a FMA is available, the compiler may choose either $\circ(a \times b + \circ(c \times d))$, or $\circ(\circ(a \times b) + c \times d)$, or $\circ(\circ(a \times b) + \circ(c \times d))$ which may give different results.

Another well-known cause of discrepancy happens in the IA32 architecture (Intel 386, 486, Pentium etc.) [2]. The IA32 processors feature a floating-point unit called "x87". This unit has 80-bit registers in "double extended" format (64-bit significand and 15-bit exponent), often associated to the *long double* C type. When using the x87 mode, the intermediate calculations are computed and stored in the x87 registers (80 bits). The final result is rounded to the destination format. Extended registers may also lead to double rounding, where floating-point results are rounded twice. For instance, the operations are

computed in the *long double* type of x87 floating-point registers, then rounded to IEEE double precision type for storage in memory. Double rounding may yield different result from direct rounding to the destination type.

An example is given in the following figure: we assume x is near the midpoint c of two consecutive floating-point numbers a and b in the destination format. Using round-to-nearest, with single rounding, x is rounded to b. However, with double rounding, it may firstly be rounded towards the middle c and then be rounded to a (if a is even). The results obtained in the two cases are different.



This is illustrated by the program of Figure 1. In this example, $y = 2^{-53} + 2^{-64}$ and x are exactly representable in double precision. The values 1 and $1 + 2^{-52}$ are two consecutive floating-point numbers. With strict IEEE-754 double precision computations for `double` type, the result obtained is $z = 1 + 2^{-52}$. Otherwise, on IA32, if the computations on `double` is performed in the `long double` type inside x87 unit, then converted to double precision, $z = 1.0$.

```
int main(){
  double x = 1.0;
  double y = 0x1p-53 + 0x1p-64; // y = 2^-53+2^-64
  double z = x + y;
  printf("z=%a\n",z);
}
```

Figure 1: A simple program giving different answers depending on the architecture.

Another example which gives inconsistencies in result between x87 and SSE[2] is presented in Figure 2. This example will be presented and reused in Section 4. In this example, we have a function `int sign(double x)` which returns a value which is either -1 if $x < 0$, or 1 if $x \geq 0$. The function `int eps_line(double sx, double sy, double vx, double vy)` then makes a direction decision depending on the sign of a few floating-point computations. We execute this program on SSE unit and obtain that `Result = 1`. When it is performed on IA32 inside x87 unit, the result is `Result = -1`.

```
int sign(double x) {
  if (x >= 0) return 1;
  else return -1;
}
int eps_line(double sx,double sy,double vx,double vy){
  return sign(sx*vx+sy*vy)*sign(sx*vy-sy*vx);
}
int main(){
  double sx = -0x1.0000000000001p0;  // sx = -1-2^-52
  double vx = -1.0;
  double sy = 1.0;
  double vy =  0x1.fffffffffffffp-1; // vy = 1-2^-53
  int r = eps_line(sx,sy,vx,vy);
  printf("Result = %d\n",r);
}
```

Figure 2: A more complex program giving different answers depending on the architecture.

3 Hardware-independent bounds for floating-point computations

As the result of floating-point computations may depend on the compiler and the architecture, static analysis is the perfect tool, as it will verify the program without running it, therefore without enforcing the architecture or the compiler. As we want both correct and interesting properties on a floating-point computation without knowing which rounding will be in fact executed, the chosen approach is to consider only the rounding error. This will be insufficient in some cases, but we believe this can give useful and sufficient results in most cases.

3.1 Rounding error in 64-bit rounding, 80-bit rounding and double rounding

We know that the choice between 64-bit, 80-bit and double rounding is the main reason that causes the discrepancies of result. We prove a rounding error bound that is valid whatever the hardware, and the chosen rounding. We denote by $\circ_{64}$ the round-to-nearest in the `double` 64-bit type. We denote by $\circ_{80}$ the round-to-nearest to the extended 80-bit registers.

Theorem 1. *For a real number x, let $\square(x)$ be either $\circ_{64}(x)$, or $\circ_{80}(x)$, or the double rounding $\circ_{64}(\circ_{80}(x))$.*

$$\textit{We have either } \left(|x| \geq 2^{-1022} \textit{ and } \left| \frac{x-\square(x)}{x} \right| \leq 2050 \times 2^{-64} \textit{ and } |\square(x)| \geq 2^{-1022} \right)$$
$$\textit{or } \left(|x| \leq 2^{-1022} \textit{ and } |x-\square(x)| \leq 2049 \times 2^{-1086} \textit{ and } |\square(x)| \leq 2^{-1022} \right).$$

This theorem is the basis of our approach to correctly prove numerical programs whatever the hardware. These bounds are tight as they are reached in all cases where $\square$ is the double rounding. They are a little bigger than the ones for 64-bit rounding (2050 and 2049 instead of 2048) for both cases. These bounds are therefore both correct, very tight, and just above the 64-bit's.

As 2^{-1022} is a floating-point number, we have $\square(2^{-1022}) = 2^{-1022}$. As all rounding modes are monotone[2], $\square(x)$ is also monotone. Then $|x| \geq 2^{-1022}$ implies $|\square(x)| \geq 2^{-1022}$ and vice versa.

Now let us prove the bounds of Theorem 1 on the rounding error for all possible values of $\square$.

3.1.1 Case 1, $\square(x) = \circ_{64}(x)$: Rounding error in 64-bit rounding

Here, we have $p = 53$ and $e_{min} = -1022$. Therefore the results of Section 2.1 give the following error bounds that are smaller than the desired ones. Therefore Theorem 1 holds in 64-bit rounding.

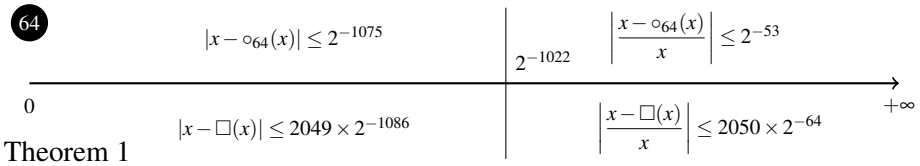


Figure 3: Rounding error in 64-bit rounding vs. Theorem 1

3.1.2 Case 2, $\square(x) = \circ_{80}$: Rounding error in 80-bit rounding

We consider here the 80-bit registers used in x87. They have a 64-bit significand and a 15-bit exponent. Thus, $p = 64$ and the smallest positive number in normal range is 2^{-16382}.

The error bounds of Section 2.1 are much smaller in this case than Theorem 1's except in the case where $|x|$ is between 2^{-16382} and 2^{-1022}. There, we have $|x-\circ(x)| \leq 2^{-64} \times |x| \leq 2^{-64} \times 2^{-1022} = 2^{-1086}$.

[2] A monotone function f is a function such that, for all x and y, $x \leq y$ implies $f(x) \leq f(y)$.

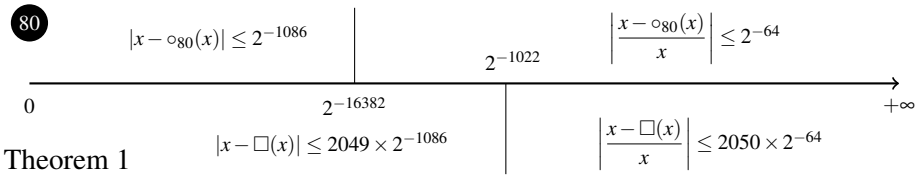


Figure 4: Rounding error in 80-bit rounding vs. Theorem 1

So, all bounds are much smaller than that of Theorem 1 so Theorem 1 holds in 80-bit rounding.

3.1.3 Case 3, $\square(x) = \circ_{64}(\circ_{80}(x))$: Rounding error in double rounding

The bounds here will be that of Theorem 1. We split in two cases depending on the value of $|x|$.

Normal range. We first assume that $|x| \ge 2^{-1022}$. We bound the relative error by some computations and the previous formulas:

$$\left|\frac{x-\circ_{64}(\circ_{80}(x))}{x}\right| \le \left|\frac{x-\circ_{80}(x)}{x}\right| + \left|\frac{\circ_{80}(x)-\circ_{64}(\circ_{80}(x))}{x}\right| \le 2^{-64} + \left|\frac{\circ_{80}(x)-\circ_{64}(\circ_{80}(x))}{\circ_{80}(x)} \times \frac{\circ_{80}(x)}{x}\right|$$
$$\le 2^{-64} + 2^{-53} \times (2^{-64}+1) \le 2050 \times 2^{-64}$$

Subnormal range. We now assume that $|x| \le 2^{-1022}$. The absolute error to bound is $|x-\circ_{64}(\circ_{80}(x))|$. We have two cases depending on whether x is in the 80-bit normal or subnormal range.

If x is in the 80-bit subnormal range, then $|x| < 2^{-16382}$ and
$|x-\circ_{64}(\circ_{80}(x))| \le |x-\circ_{80}(x)| + |\circ_{80}(x)-\circ_{64}(\circ_{80}(x))| \le 2^{-1086} + 2^{-1075} \le 2049 \times 2^{-1086}$.

If x is in the 80-bit normal range, then $2^{-16382} \le |x| < 2^{-1022}$ and
$|x-\circ_{64}(\circ_{80}(x))| \le |x-\circ_{80}(x)| + |\circ_{80}(x)-\circ_{64}(\circ_{80}(x))| \le 2^{-64} \times |x| + 2^{-1075} \le 2049 \times 2^{-1086}$.

Then Theorem 1 holds in double rounding. In conclusion, Theorem 1 is proved for all 3 roundings.

We use the Coq library developed with the Gappa tool with the help of the Gappa tactic [18] to prove the correctness of Theorem 1. The corresponding theorem and proof (228 lines) in Coq is available at `http://www.lri.fr/~nguyen/research/rnd_64_80_post.html`. The formal proof exactly corresponds to the one described in the preceding Section. It is not very difficult, but needs many computations and a very large number of subcases. The formal proof gives a very strong guarantee on this result, allowing its use in the Frama-C platform.

3.2 Hardware and compiler-independent proofs of numerical programs

3.2.1 Rounding error in presence of FMA

Theorem 1 gives rounding error formulas for various roundings denoted by $\square$ (64-bit, 80-bit and double rounding). Now, let us consider the FMA that computes $x \times y + z$ with one single rounding. The question is whether a FMA was used in a computation. We therefore need an error bound that covers all the possible cases.

The idea is very simple: we consider a FMA as a *rounded* multiplication followed by a rounded addition. And we only have to consider another possible "rounding" that is the identity: $\square(x) = x$.

This specific "rounding" magically solves the FMA problem: the result of a FMA is $\square_1(x \times y + z)$, that may be considered as $\square_1(\square_2(x \times y) + z)$ with $\square_2$ being the identity. So we handle in the same way all operations even in presence of FMA or not, by considering one rounding for each basic operation (addition, multiplication...). Of course, this "rounding" easily verifies the formulas of Theorem 1.

3.2.2 Proofs of programs

What is the use of this odd rounding? The idea is that each *basic* operation (addition, subtraction, multiplication, division and square root) will be considered as rounded with a $\square$ that may be one of the four possible roundings. Let us go back to the computation of `a*b+c*d`: it becomes $\square(\square(a\times b)+\square(c\times d))$ with each $\square$ being one of the 4 roundings. It gives us 64 possibilities. In fact, only 45 possibilities are allowed (for example, the addition cannot be exact). But *all* the real possibilities are *included* in all the considered possibilities. And all considered possibilities have a rounding error bounded by Theorem 1.

So, by considering the identity as a rounding like the others, we handle all the possible uses of the FMA in the same way as we handle multiple roundings. Note that absolute value and negation may produce a rounding if we put a 80-bit number into a 64-bit number.

The idea now is to do forward analysis of the rounding errors, that is to say propagate the errors and bound them at each step of the computation. Therefore, we have put the formulas of Theorem 1 as postconditions of each floating-point operation in the Frama-C platform to look into the rounding error of the whole program. Based on the work of [15], we create a new "pragma" called `multirounding` to implements this. Ordinarily, the pragma directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself; here, it lets Frama-C know that floating-point computations may be done with extended registers and/or FMA.

In our pragma, each floating-point number is represented by two values, an exact one (a real value, as if no rounding occurred) and a rounded one (the true floating-point value). At each computation, we are only able to bound the difference between these two values, without knowing the true rounded value.

Theorem 2. *Let $\odot$ be an operation among addition, subtraction, multiplication, division, square root, negation and absolute value. Let $x=\odot(y,z)$ be the exact result of this operation (without rounding). Then, whatever the architecture and the compiler, the computed result $\tilde{x}$ is such that*

If $|x|\geq 2^{-1022}$, then $\tilde{x}\in \left[x-2050\times 2^{-64}\times|x|\ ,\ x+2050\times 2^{-64}\times|x|\right] \ \setminus\ \left]-2^{-1022}\ ,\ 2^{-1022}\right[$.

If $|x|\leq 2^{-1022}$, then $\tilde{x}\in \left[x-2049\times 2^{-1086}\ ,\ x+2049\times 2^{-1086}\right]\cap\left[-2^{-1022}\ ,\ 2^{-1022}\right]$.

This is straightforward as the formulas of Theorem 1 subsume all possible roundings (64 or 80-bit) and operations (using FMA or not), whatever the architecture and the compiler choices.

Theorem 3. *If we define each result of an operation by the formulas of Theorem 2, and if we are able to deduce from these intervals an interval $\mathscr{I}$ for the final result, then the really computed final result is in $\mathscr{I}$ whatever the architecture and the compiler that preserves the order of operations.*

This is proved by using Theorem 2 and minding the FMA and the order of the operations.

The next question is the convenience of this approach. We have a collection of inequalities that might be useless. They are indeed useful and practical as we rely on the Gappa tool [19, 20] that is intended to help verifying and formally proving properties on numerical programs. Formulas of Theorem 1 have been chosen so that Gappa may take advantage of them and give an adequate final rounding error.

We may have chosen other formulas for Theorem 1 (such as $|\square(x)-x|\leq 2050\times 2^{-64}|x|+2049\times 2^{-1086}$ or $|\square(x)-x|\leq\max\left(2050\times 2^{-64}|x|,2049\times 2^{-1086}\right)$ but those are not as conveniently handled by Gappa as the chosen ones.

4 Case Studies

4.1 Double rounding example

The first example is very easy. It concerns the program of Figure 1. In this program, the result may either be 1 or $1+2^{-52}$, depending on the arguments of compiler we use. We add to the original program an

assertion written in ACSL [5]. By using Frama-C/Why and thanks to the Gappa prover, we automatically prove that in every architecture or compiler, the result of this program is always in $[1, 1+2^{-52}]$.

4.2 Avionics example

We now present a more complex case study containing floating-point computations to illustrate our approach. This example is part of KB3D [21][3], an aircraft conflict detection and resolution program. The aim is to make a decision corresponding to value -1 and 1 to decide if the plane will go to its left or its right. Note that KB3D is formally proved correct using PVS and assuming the calculations are exact [21]. However, in practice, when the value of the computation is small, the result may be inconsistent or incorrect. The original code is in Figure 2 and may give various answers depending on the architecture/compilation. To prove the correctness of this program which is independent to the architecture/compiler, we need to modify this program to know whether the answer is correct or not.

The modified program (See Figure 5) provides an answer that may be 1, -1 or 0. The idea is that, if the result is nonzero, then it is correct (meaning the same as if the computations were done on real numbers). If the result is 0, it means that the result may be under the influence of the rounding errors and the program is unable to give a certified answer.

In the original program, the inconsistency of the result is derived from the function `int sign(double x)`. To use this function only at the specification level, we define a logic function `logic integer l_sign (real x)` with the same meaning. Then we define another function `int sign (double x,`

```
#pragma JessieFloatModel(multirounding)
#pragma JessieIntegerModel(math)

//@ logic integer l_sign(real x) = (x >= 0.0) ? 1 : -1;

/*@ requires e1<= x-\exact(x) <= e2;
  @ ensures  \abs(\result) <= 1 &&
  @          (\result != 0 ==> \result == l_sign(\exact(x)));
  @*/
int sign(double x, double e1, double e2) {
  if (x > e2)  return  1;
  if (x < e1)  return -1;
  return 0;
}

/*@ requires
  @   sx == \exact(sx)  && sy == \exact(sy) &&
  @   vx == \exact(vx)  && vy == \exact(vy) &&
  @   \abs(sx) <= 100.0 && \abs(sy) <= 100.0 &&
  @   \abs(vx) <= 1.0   && \abs(vy) <= 1.0;
  @ ensures
  @    \result != 0
  @      ==> \result == l_sign(\exact(sx)*\exact(vx)+\exact(sy)*\exact(vy))
  @            * l_sign(\exact(sx)*\exact(vy)-\exact(sy)*\exact(vx));
  @*/
int eps_line(double sx, double sy,double vx, double vy){
  int s1,s2;
  s1=sign(sx*vx+sy*vy, -0x1.90641p-45, 0x1.90641p-45);
  s2=sign(sx*vy-sy*vx, -0x1.90641p-45, 0x1.90641p-45);
  return s1*s2;
}
```

Figure 5: Avionics program

[3] See also `http://research.nianet.org/fm-at-nia/KB3D/`.

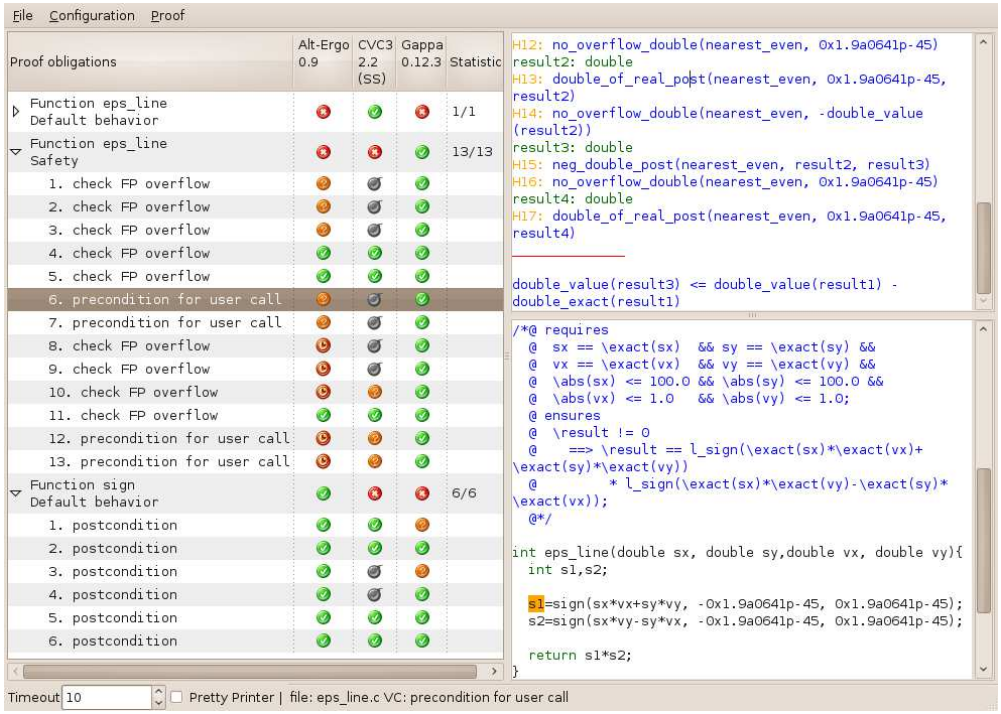


Figure 6: Result of Figure 5 program

`double e1, double e2)` that gives the sign of x provided we know its rounding error is between $e1$ and $e2$. In the other cases, the result is zero.

The function `int eps_line (double sx, double sy, double vx, double vy)` of Figure 5 then does the same computations as the one of Figure 2, but the result may be different. More precisely, if the modified function gives a nonzero answer, it is the correct one (it gives the correct sign). But it may answer zero (contrary to the original program) when it is unable to give a certified answer. As in interval arithmetic, the program does not lie, but it may not answer.

About the other assertions, the given value of sx, $vx\ldots$ are reasonable for the position and the speed of the plane. The assertions about $s1$ and $s2$ are here to help the automatic provers.

The most interesting parts are the value chosen for $e1$ and $e2$: they need to bound the rounding error of the computation $sx*vx+sy*vy$ (and its counterpart). For this, we will rely on the Gappa tool. In particular, it will solve all the required proofs that no overflow occur.

In the usual formalization where all computations directly round to 64 bits, the values $e2 = -e1 = 0x1p-45$ are correct (it has been proved using the Gappa tool). With our approach and a generic rounding, we have proved that the values $e2 = -e1 = 0x1.90641p-45$ are correct. This means that the rounding error of $sx*vx+sy*vy$ will always be smaller than this value whatever the architecture and the compiler choices. This means that, even if a FMA is used or if extended registers are used somewhere, this function *does not lie*.

The analysis of this program (obtained from the verification condition viewer gWhy [16]) is given in Figure 6. By using different automatic theorem prover: Alt-Ergo [22], CVC3 [23], Gappa, we successfully prove all proof obligations in this program.

Nearly all the proof obligations are quick to prove. The proof that the values $e1$ and $e2$ bound the rounding error is slightly longer (about 10 seconds). This is due to the fact that, for each operation, we have to split into 2 cases: normal and subnormal and this increases the number of proof obligations to solve (exponential in the number of computations).

5 Conclusions and further work

We have proposed an approach to give correct rounding errors whatever the architecture and the choices of the compiler (preserving the order). This is implemented in the Frama-C framework from the Beryllium release for all basic operations: addition, subtraction, multiplication, division, square root, negation, absolute value and we have proved its correctness in Coq.

As we use the same conditions for all basic operations, it is both simple and efficient. Moreover, it handles both rounding according to 64-bit rounding in IEEE-754 double precision, 80-bit rounding in x87, double rounding in IA-32 architecture, and FMA in Itanium and PowerPC processors.

Nevertheless, the time to run a program needs to be taken into account. With a program containing few floating-point operations, it works well. However, it will be a little slower with programs containing a large number of floating-point operations because of the disjunction in Theorem 1 . The scalability could be achieved by several means: either modifying Gappa so that it may handle other kinds of formulas as good as those of Theorem 1, or replacing the formulas in Theorem 1 by other ones which do not contain disjunctions.

Another drawback is that we may only prove rounding errors. There is no way to prove, for example, that a computation is correct (even if it would be correct in all possible roundings). This means that some subtle floating-point properties may be lost but bounding the final rounding error is usually what is wanted by engineers and this does not appear to be a big flaw.

Note that we only consider double precision numbers as they are the most used. This is easily applied to single precision computations the same way (with single rounding, 80-bit rounding or double rounding). The idea would be to give similar formulas with the same case splitting and to provide the basic operations with those post-conditions.

We only handle rounding-to-nearest (ties to even and ties away from zero). The reason is that directed roundings do not suffer from these problems: double rounding gives the correct answer and if some intermediate computations are done in 80-bit precision, the final result is more accurate, but still correct as it is always rounded in the correct direction. Only rounding to nearest causes discrepancies.

This work is at the boundary between software and hardware for floating-point programs and this aspect of formal verification is very important. Moreover, this work deals both with normal and subnormal numbers, the latter ones being usually dismissed.

Another interesting point is that our error bounds may be used by other tools. As shown here, considering a slightly bigger error bound for each operation suffices to give a correct final error. This means that if Fluctuat for example would use them, it would also consider all possible cases of hardware and of compilation (preserving the order).

The next step would be to allow the compiler to do anything, including re-organizing the operations. This is a challenge as it may give very different results. For example, if $|e| \ll |x|$, then $(e+x)-x$ gives zero while $e+(x-x)$ gives e. Nevertheless, some ideas could probably be reused to give a loose bound on the rounding error.

Acknowledgments

We are grateful to Claude Marché for his suggestions, his help in creating a new pragma in Frama-C/Why and his constructive remarks. We thank Guillaume Melquiond for his help in the use of Gappa tool. We are grateful to César Muñoz for providing the case study and explanations. We also thank Pascal Cuoq for his helpful suggestions about the FMA.

References

[1] Microprocessor Standards Subcommittee: IEEE Standard for Floating-Point Arithmetic. IEEE Std. 754-2008 (August 2008) 1–58

[2] Monniaux, D.: The pitfalls of verifying floating-point computations. TOPLAS **30**(3) (May 2008) 12

[3] : JML-Java Modeling Language `www.jmlspecs.org`.

[4] Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (STTT) **7**(3) (June 2005) 212–232

[5] Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. (2008) `http://frama-c.cea.fr/acsl.html`.

[6] Barnett, M., Leino, K.R.M., Rustan, K., Leino, M., Schulte, W.: The Spec# Programming System: An Overview, Springer (2004) 49–69

[7] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In: ESOP. Number 3444 in Lecture Notes in Computer Science (2005) 21–30

[8] Monniaux, D.: Analyse statique : de la théorie à la pratique. Habilitation to direct research, Université Joseph Fourier, Grenoble, France (June 2009)

[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. Volume 5825 of LNCS., Springer (2009) 53–69

[10] Carreño, V.A., Miner, P.S.: Specification of the IEEE-854 floating-point standard in HOL and PVS. In: HOL95: 8th International Workshop on Higher-Order Logic Theorem Proving and Its Applications, Aspen Grove, UT (September 1995)

[11] Russinoff, D.M.: A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. LMS Journal of Computation and Mathematics **1** (1998) 148–200

[12] Harrison, J.: Formal verification of floating point trigonometric functions. In: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design, Austin, Texas (2000) 217–233

[13] Leavens, G.T.: Not a number of floating point problems. Journal of Object Technology **5**(2) (2006) 75–83

[14] Boldo, S., Filliâtre, J.C.: Formal Verification of Floating-Point Programs. In: 18th IEEE International Symposium on Computer Arithmetic, Montpellier, France (June 2007) 187–194

[15] Ayad, A., Marché, C.: Multi-prover verification of floating-point programs. In Giesl, J., Hähnle, R., eds.: Fifth International Joint Conference on Automated Reasoning, Edinburgh, Scotland, Springer (July 2010)

[16] Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Computer Aided Verification (CAV). Volume 4590 of LNCS., Springer (July 2007) 173–177

[17] Goldberg, D.: What every computer scientist should know about floating point arithmetic. ACM Computing Surveys **23**(1) (1991) 5–47

[18] Boldo, S., Filliâtre, J.C., Melquiond, G.: Combining Coq and Gappa for Certifying Floating-Point Programs. In: 16th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning. Volume 5625 of Lecture Notes in Artificial Intelligence., Grand Bend, Canada, Springer (July 2009) 59–74

[19] Daumas, M., Melquiond, G., Muñoz, C.: Guaranteed proofs using interval arithmetic. In Montuschi, P., Schwarz, E., eds.: 17th IEEE Symposium on Computer Arithmetic, Cape Cod, MA, USA (2005) 188–195

[20] Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. Transactions on Mathematical Software **37**(1) (2009)

[21] Dowek, G., Muñoz, C., Carreño, V.: Provably safe coordinated strategy for distributed conflict resolution. In: Proceedings of the AIAA Guidance Navigation, and Control Conference and Exhibit 2005, AIAA-2005-6047, San Francisco, California (2005)

[22] Conchon, S., Contejean, E., Kanig, J.: CC(X): Efficiently Combining Equality and Solvable Theories without Canonizers. In: SMT 2007: 5th International Workshop on Satisfiability Modulo. (2007)

[23] Barrett, C., Tinelli, C.: CVC3. In: Proceedings of the 19^{th} International Conference on Computer Aided Verification (CAV '07). Volume 4590 of LNCS., Springer-Verlag (July 2007) 298–302 Berlin, Germany.