Formal verification of numerical programs: from C annotated programs to mechanical proofs

Sylvie Boldo, Claude Marché

Math.Comput.Sci. (2011) 5:377–393
DOI 10.1007/s11786-011-0099-9

Mathematics in Computer Science

Formal Verification of Numerical Programs: From C Annotated Programs to Mechanical Proofs

Sylvie Boldo · Claude Marché

Received: 16 November 2010 / Revised: 28 April 2011 / Accepted: 31 May 2011 / Published online: 12 November 2011

Abstract Numerical programs may require a high level of guarantee. This can be achieved by applying formal methods, such as machine-checked proofs. But these tools handle mathematical theorems while we are interested in C code, in which numerical computations are performed using floating-point arithmetic, whereas proof tools typically handle exact real arithmetic. To achieve this high level of confidence on C programs, we use a chain of tools: Frama-C, its Jessie plugin, Why and provers among Coq, Gappa, Alt-Ergo, CVC3 and Z3. This approach requires the C program to be annotated: each function must be precisely specified, and we prove the correctness of the program by proving both that it meets its specifications and that no runtime error may occur. The purpose of this paper is to illustrate, on various examples, the features of this approach.

Keywords Floating-point arithmetic · C program · Formal specification · Automated reasoning

Mathematics Subject Classification (2010) 68N30 · 65Y04

1 Introduction

Given a program using floating-point arithmetic, it is pretty hard to know the final rounding error of the result. We are interested in verifying numerical programs with a very high level of guarantee by using formal methods: the expected functional behavior of a given program is expressed by formal specifications, and we prove that the code satisfies these specifications.

This work was partly funded by the *F∮st* (ANR-08-BLAN-0246, http://fost.saclay.inria.fr/) and *U3CAT* (ANR-08-SEGI-021, http://frama-c.com/u3cat/) projects of the French national research organization (ANR), and by the *Hisseo* project (http://hisseo.saclay.inria.fr/) of the Digiteo cluster and the *Domaine d'Intérêt Majeur "Logiciel et Systèmes Complexes"* of the Île-de-France regional council.

S. Boldo (✉) · C. Marché (✉)
INRIA Saclay-Île-de-France, ProVal, 91893 Orsay, France
e-mail: Sylvie.Boldo@inria.fr

C. Marché
e-mail: Claude.Marche@inria.fr

S. Boldo · C. Marché
LRI, Univ. Paris-Sud, CNRS, 91405 Orsay, France

The model we use as basis for our method is the fact that each floating-point result is a correct rounding of the exact real value for all basic operations (addition, subtraction, multiplication, division and square root). This property is defined in the IEEE-754 standard [27] and all modern processors comply with it. However, even if each computation is correct, i.e. the best possible, there is no guarantee that the final result after many such computations is still accurate.

Static analysis is an approach for checking a program without running it. Deductive verification techniques 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 specification languages—such as SPARK [16] for Ada, JML [15] for Java, ACSL [3] for C, Spec# [2] for C#—which are expressive enough to specify complex functional properties on the behavior of programs.

For automatic analysis of floating-point code, there exist several methods for bounding the final error of a program, including interval arithmetic, forward and backward analysis [26,39]. Another approach is abstract-interpretation-based static analysis, used e.g. by Astrée [18,32] and Fluctuat [21,24].

Floating-point arithmetic has been formalized using deductive formal methods since 1989: in PVS [17], in ACL2 [36], in HOL-light [25]. These approaches consider either hardware components or abstract algorithms, and never C source code. Since 2001, a high-level formalization of floating-point numbers [5,19] is available for the Coq proof assistant [4]. In this paper we build upon the latter to prove source code written in C when needed.

There exist very few works on specifying and proving functional properties directly on floating-point source code, using deductive verification systems. An early work on floating-point support in JML for Java is presented in 2006 by Leavens [29], where mostly runtime assertion checking is considered. Annotations involve Java boolean expressions, which can themselves involve floating-point computations, meaning that annotations can generate rounding errors and overflow, which leads to many issues and traps [29]. This must be avoided when using a theorem proving approach.

The first proposal amenable to formal proof is made in 2007 by Boldo and Filliâtre [11], where verifications conditions in Coq are generated from floating-point C programs. Ayad and Marché [1] extended this approach to increase genericity, handle exceptional behaviors and support many different provers. The latter is implemented in the Frama-C/Jessie/Why tool chain and we base this paper on it.

It should be noticed that all these approaches assume that floating-point computations strictly follow the IEEE standard. However some processors and/or compilers may invalidate this assumption, e.g. the double roundings induced by extended 80-bits numbers available on INTEL x86 processors. Nevertheless, the approach we use here has been recently extended to handle multiple architectures and compilers [13,14].

2 The Basics of the Verification Process

Our case studies are conducted using the Frama-C framework for static analysis of C source code[1] and its associated Jessie plugin [23,33] that uses the deductive verification platform Why [22]. In that setting, the expected behavioral properties of the input C code must be stated using formal specifications, expressed in the ACSL [3] annotation language.

2.1 The Tool Chain

The tool chain is described in Fig. 1. The annotated C code is given to the Frama-C kernel which performs the syntactic analysis and type checking of both the code and its annotations, and then normalizes the code [35]. Frama-C calls the Jessie plugin which transforms the code into the input language of Why. The Why verification condition generator then produces a set of proof obligations (the verification conditions, abbreviated as VCs), that are a set of logic formulas. Soundness of the code with respect to its annotations is ensured by proving these formulas valid.

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

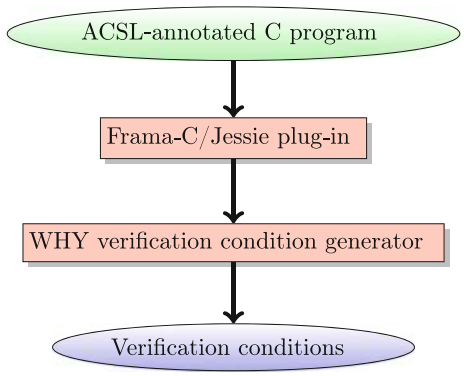

Fig. 1 Chain of tools: from the C program to the proof obligations

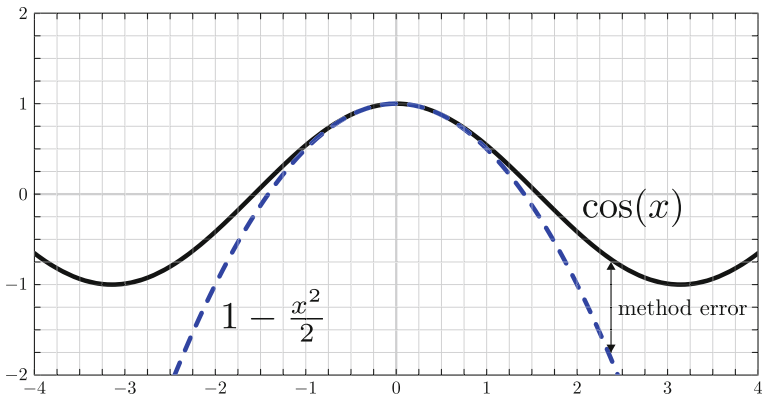

Fig. 2 Cosine function and its Taylor approximation at order 2

Additionally, other VCs are generated to guarantee the absence of run-time errors in the program: typically division by zero, dereferencing of the null pointer (or more generally pointer to non properly allocated memory blocks). In the context of floating-point programs, VCs are generated to ensure the absence of overflow and undefined result: it is verified that no infinite values and no NaN will ever be generated by the computations. However, in some specific case such values may be intended to appear, such a case is discussed in Sect. 3.2

2.2 Basics of ACSL

We illustrate the main basic features of ACSL on the following toy example, supposed to compute an approximation of the cosine function for an argument close to zero. This example comes from [12] where it is fully proved within Coq only. We use single precision numbers to have significantly higher rounding errors to specify.

```
float my_cosine(float x) {
  return 1.0f - x * x * 0.5f;
}
```

The first step is to decide what property we want to formally specify. Figure 2 displays the graph of the cosine function, in plain line, and the graph of the polynomial function $1-x^2/2$ in dashed line. The difference is classically called the *method error*.

Figure 3 shows a zoom of the previous graphs on the interval $[1/32 - 2^{-18}, 1/32]$. The additional thin line is the graph of the result of the C function `my_cosine` computed with single-precision floating-point numbers. Each

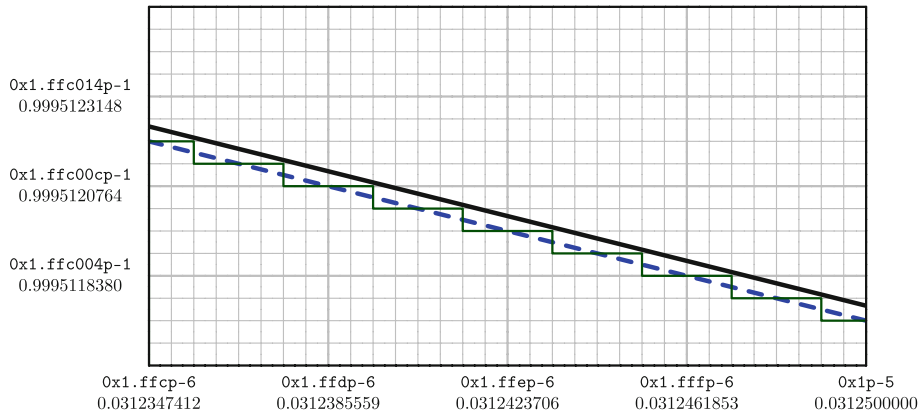

Fig. 3 Zoom of Fig. 2 around $x = 1/32$

step in the y-axis corresponds to a representable number[2] (2^{-24} between each). The difference between the dashed and the thin curves is classically the *rounding error*. Given that picture, we can conjecture that both the method error and the rounding error are bounded by 2^{-24}. (Indeed we made the choice of zooming around $1/32$ because they are both of the same and worst magnitude there.)

These conjectures can be described formally by annotations as follows.

```
/*@ requires \abs(x) <= 0x1p-5;
  @ ensures \abs(\result - \cos(x)) <= 0x1p-23;
  @*/
float my_cosine(float x) {
  //@ assert \abs(1.0 - x*x*0.5 - \cos(x)) <= 0x1p-24;
  return 1.0f - x * x * 0.5f;
}
```

The *precondition*, introduced by **requires**, states that we expect argument x in the interval $[-1/32; 1/32]$. The *postcondition*, introduced by **ensures**, states that the distance between the value returned by the function, denoted by keyword **result**, and the model of the program, which is here the true mathematical cosine function denoted by **cos** in ACSL, is not greater than 2^{-23}. It is important to notice that in annotations the operators like + or $*$ denote operations on real numbers and not on floating-point numbers. In particular, there is no rounding error and no overflow in annotations, unlike in the early Leavens' proposal [29]. The C variables of type `float`, like `x` and `\result` in this example, are interpreted as the real number they represent. Thus, the last annotation, given as an assertion inside the code, is a way to make explicit the reasoning we made above, making the total error the sum of the method error and the rounding error: it states that the method error is less than 2^{-24}. Again, it is thanks to the choice of having exact operations in the annotations that we are able to state a property of the method error.

2.3 Back-end Provers

The way the provers are called on the VCs is schematized in Fig. 4. Why implements translators that are able to print the VCs in the expected input syntax of provers. A first class are the SMT-style provers like Alt-Ergo, Z3 and CVC3: they are able to handle first-order formulas with built-in integer and real arithmetic, and non-interpreted predicate and function symbols defined by axioms. For those provers we pass some axiomatization of floating-point computations, in which the most important ingredient is the axiomatization of the rounding function [1] which, given a format (e.g. single or double precision), a rounding mode and a real number, returns its approximation in that format.

[2] We use the C99 notation for hexadecimal floating-point literals: `0x`hh`.`hh`p`dd, where h are hexadecimal digits and dd is in decimal, denotes number $hh.hh \times 2^{dd}$, e.g. `0x1.Fp-4` is $(1 + 15/16) \times 2^{-4}$.

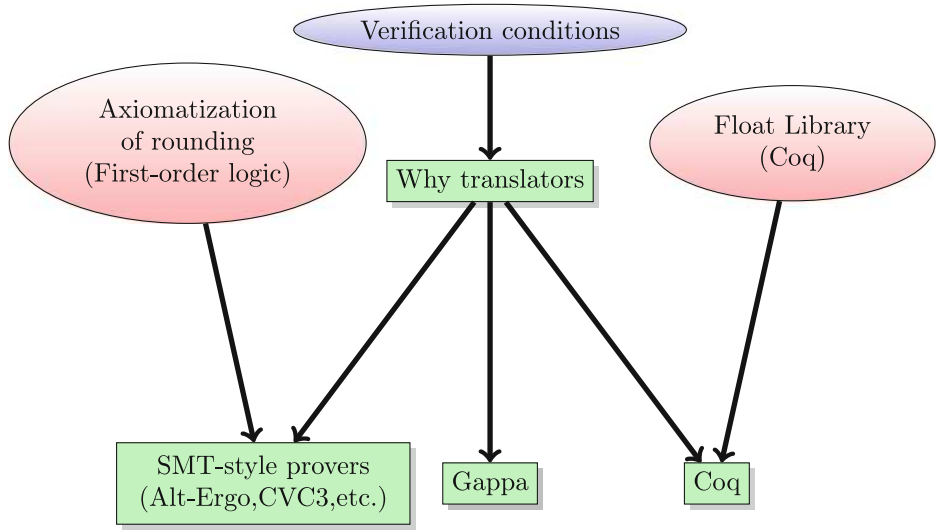

Fig. 4 Back-end provers

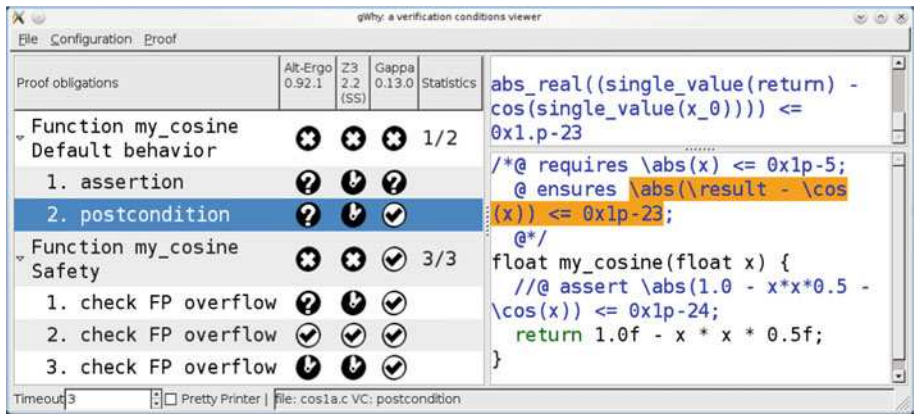

Fig. 5 Cosine example: VCs displayed in the Why GUI

The Gappa prover [30] is specialized for reasoning on floating point computations. The rounding function above is built-in, and so is integer and real arithmetic, but it does not support first-order quantification. It will be the prover of choice for VCs where floating-point rounding is involved (including proving absence of overflow), because other provers will only reason from a partial axiomatization of the rounding function, whereas they will deal better with quantified formulas.

Running the cosine annotated code into our proof chain generates 5 VCs: 3 of them amount to prove that there are no overflow when the subtraction and the two multiplications are computed. Another is the assertion itself, and the last is the postcondition. These VCs can be displayed using the Why GUI: a screenshot of it is shown on Fig. 5. Each line of the table on the left of the window corresponds to a VC, whereas the columns correspond to some external theorem provers, namely here Alt-Ergo, Z3, and Gappa. Each prover can be called on each VC, and the results are displayed on that table: a check sign means that the prover proved the formula valid, whereas other icons (with black background) mean either the answer "unknown" (i.e. the prover did not conclude to its validity), a time out, etc. For that particular example, we see that the first two provers proved only the second of the overflow VCs, whereas the Gappa prover proved all VCs except the one coming from the assertion.

Finally, what can be done if a VC cannot be proved by any prover, like the assertion in the example above? The Why platform offers the possibility to call an interactive proof assistant instead. Here we use Coq, which comes

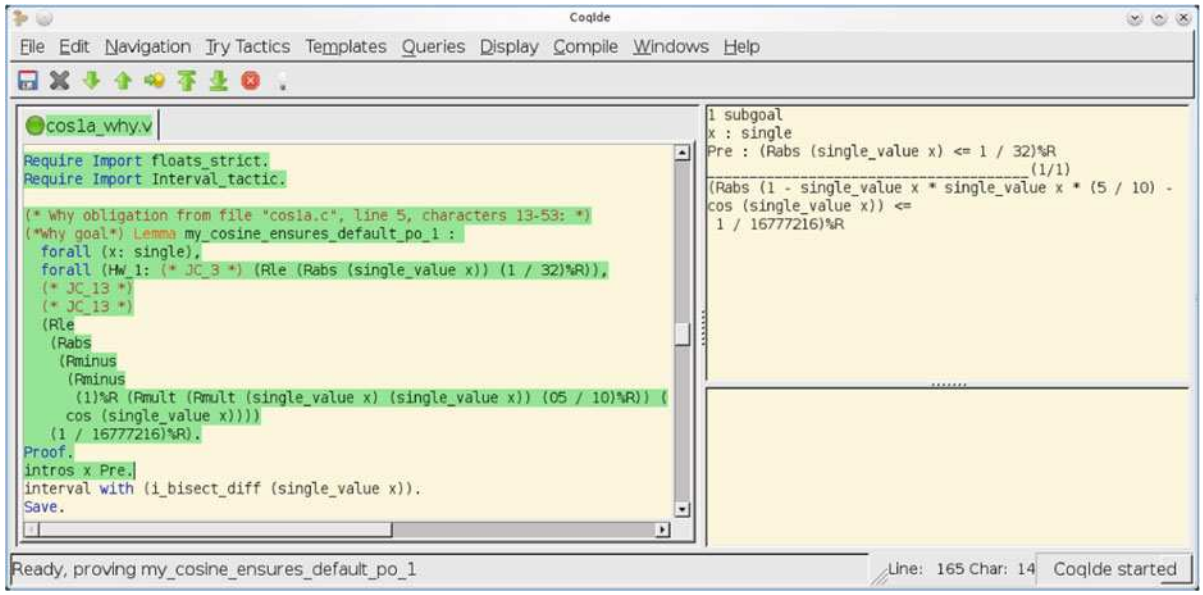

Fig. 6 Last unproved VC, proved within Coq IDE

with a large library on floating-point numbers[3] [5,19]. The Why translator for Coq uses that library for exporting the VCs. Calling Coq on the assertion VC of our cosine example results in the lemma displayed in the Coq IDE shown in Fig. 6. The proof is performed in two lines, by using the `interval` tactics [31] specialized for proving bounds on real expressions. It bissects the interval where `(single_value x)` (denoting the real represented by the float `x`) lies, and computes the interval of expressions in the goal, until it is proved. (Gappa cannot do it directly because it does not support the cosine function.)

Last but not least, the axiomatization used for SMT solvers has been realized with Coq [1,11], thus providing a high level of trust in our multi-provers back-end.

2.4 Exact Values

ACSL provides floating-point annotations based on [11] that allow to specify the exact computations of numerical programs. More precisely, each floating-point expression e has a "ghost" value denoted `\exact`(e) which does not suffer from rounding. This real value is then computed with the same operations as the float value except that the ghost operation is exact. The construct `\round_error`(e) is then used for $|e - $ `\exact`$(e)|$.

These features can be used to specify differently the same cosine example. Here is a possibility:

```
/*@ requires \abs(\exact(x)) <= 0x1p-5;
  @ requires \round_error(x) <= 0x1p-20;
  @ ensures \abs(\exact(\result) - \cos(\exact(x))) <= 0x1p-24;
  @ ensures \round_error(\result) <= \round_error(x) + 0x2.0081p-25;
  @*/
float my_cosine2(float x) {
  float r = 1.0f - x * x * 0.5f;
  //@ assert \abs(\exact(r) - \cos(\exact(x))) <= 0x1p-24;
  return r;
}
```

This is read as follows: the precondition assumes that the exact value of x is in $[-1/32; 1/32]$, with a rounding error less than 2^{-20}. In the code, the method error is again stated as an assertion, this time using the `\exact` construct. Then, the postcondition says that, as before, the exact value of the result is close to the cosine for not more than

[3] Available at http://lipforge.ens-lyon.fr/www/pff/.

2^{-24}, and the rounding error on the result is bounded by the original rounding error on x plus `0x2.0081p-25` (i.e just a bit above 2^{-24}). One may wonder how this bound was determined: it is indeed possible to ask the Gappa tool itself the most accurate bound[4]. This bound depends on the bound on the rounding error on x as input. Here is a small table to give in idea on how these bounds are related: the first column is the bound on the rounding error on x, whereas the second column is the bound on the rounding error of the result, minus the one on x.

error(x)	error(result) − error(x)	
2^{-28}	`0x1.01800001p-25`	$\simeq 2^{-24.9916}$
2^{-26}	`0x1.0480001p-25`	$\simeq 2^{-24.9749}$
2^{-24}	`0x1.108001p-25`	$\simeq 2^{-24.9099}$
2^{-22}	`0x1.40801p-25`	$\simeq 2^{-24.6758}$
2^{-20}	`0x2.0081p-25`	$\simeq 2^{-23.9986}$
2^{-18}	`0x5.009p-25`	$\simeq 2^{-22.6774}$
2^{-16}	`0x2.203p-22`	$\simeq 2^{-20.9120}$
2^{-14}	`0x8.221p-22`	$\simeq 2^{-18.9762}$

The VCs for proving these annotations in this variant are not very different from the previous one: 3 VCs for overflow, proved by Gappa, the assertion proved within Coq by the interval tactic, and 2 VCs for the 2 postconditions, both proved by Gappa. The first postcondition on the exact value of the result is also proved by SMT solvers: this is expected since it is a simple consequence of the assertion.

3 Examples

The full code of all these examples (and more) and their proofs are available either on http://www.lri.fr/~sboldo/research.html, on the Hisseo web page http://hisseo.saclay.inria.fr/gallery.html, or the ProVal gallery http://proval.lri.fr/gallery/index.en.html.

3.1 Clock Drift

The purpose of this example is to illustrate the handling of loops via loop invariants. We consider the following C code which increments a counter by steps of 0.1.

```
float clock_single(int n) {
  float t = 0.0f;
  int i;
  for(i=0;i<n;i++) t = t + 0.1f;
  return t;
}
```

This is indeed a classical example taken to illustrate rounding errors, coming from the fact that 0.1 is not exactly representable. It is also related to a known bug which arose in the critical software of a patriot missile battery.[5]

In a mathematical model of that program, $t = i \times 0.1$ is true at each iteration of the loop. But the rounding of 0.1 in single precision is `0x0.199999Ap0` $\simeq 0.1 + 1.5e-9$. Naively, one could expect $i \times 0.1 \leq t \leq i \times (0.1 + 1.5e-9)$ to be a loop invariant in this C program. However, the behavior of floating-point computations is much more complicated, because when t gets higher, the rounding error when computing $t + 0.1f$ can become either positive or

[4] Although this is not yet well instrumented in Frama-C/Jessie: given the corresponding VC expressed in Gappa, one has to replace the expected bound by a "?" and run Gappa on it directly.

[5] An internal clock was incremented by steps of 0.1 s, and because of rounding errors occuring over time, the drift between that internal clock and the real time prevented the anti-missile to launch, thus causing the death of several soldiers during the Gulf War in 1991. See http://autarkaw.wordpress.com/2008/06/02/round-off-errors-and-the-patriot-missile/ and http://en.wikipedia.org/wiki/MIM-104_Patriot#Failure_at_Dhahran.

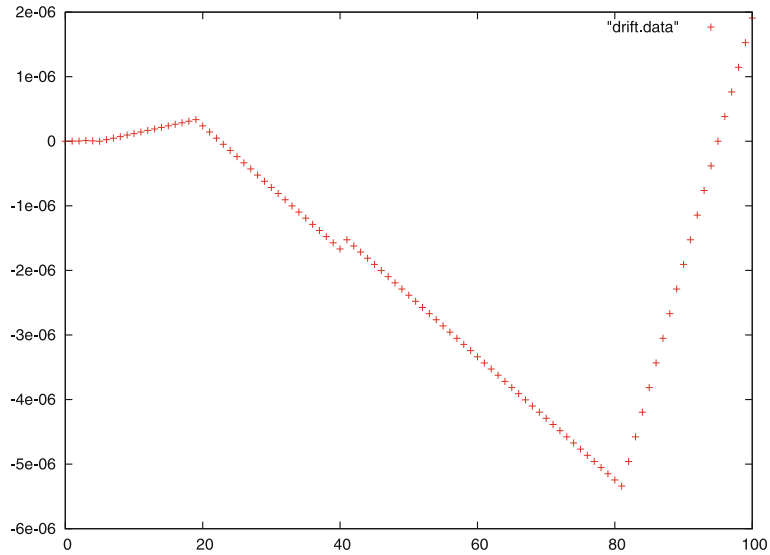

Fig. 7 $t - 0.1 \times i$ for the first 100 iterations

negative, and gets higher in absolute value: Fig. 7 shows the values of $t - i \times 0.1$ for the first 100 iterations of the loop.

What we can prove on such a program is then an invariant of the form $|t - i \times 0.1| \leq i \times C$ for some constant C to determine. The problem is that the constant C

depends on how many iterations we are going to make. Following the analogy with the clock of the missile battery, we assume that it will not run for more than one day[6], so we are going to bound the number n of iterations by one million.

The fully annotated code is as follows:

```
#define NMAX 1000000
#define NMAXR 1000000.0

/*@ lemma real_of_int_inf_NMAX:
  @   \forall integer i; i <= NMAX ==> i <= NMAXR;   */

//@ lemma real_of_int_succ: \forall integer n; n+1 == n + 1.0;

#define A 1.49012e-09
#define B 0x1p-8
#define C (B + A)

/*@ requires 0 <= n <= NMAX;
  @ ensures \abs(\result - n*0.1) <= n * C;   */
float clock_single(int n) {
  float t = 0.0f;
  int i;

  /*@ loop invariant 0 <= i <= n;
    @ loop invariant \abs(t - i * 0.1) <= i * C ;
    @ loop variant n-i;
    @*/
  for(i=0;i<n;i++) {
  L: //@ assert 0.0 <= t <= NMAXR*(0.1+C)  ;
     t = t + 0.1f;
     //@ assert \abs(t - (\at(t,L) + (float)0.1)) <=  B;
  }
  return t;
}
```

[6] As for the analogy with the missile battery bug: indeed the clock had run for 4 days, although it was not supposed to.

The lemmas are given to help the automatic provers: the lemma `real_of_int_inf_NMAX` is needed by Gappa to prove the assertion on the rounding error. The lemma `real_of_int_succ` helps the SMT solvers to prove preservation of the loop invariant. `A` is a bound of $(float)0.1 - 0.1$, whereas B is a bound of `round_error`$(t + (float)0.1)$ for $0 \leq t \leq 0.1 \times NMAX$.

Running this code into our tool chain results in 17 VCs. The first two correspond to the two lemmas: the first one is proved by Gappa, whereas the second is proved by all SMT solvers and Gappa. 5 VCs come from runtime error checks: 1 for the overflow in $t + 0.1$, proved by both SMT solvers and Gappa, and 4 others from integer overflow checks and loop termination checks, proved by SMT solvers. The remaining 10 VCs come from the behavioral properties of the C function: 3 to check that loop invariants initially hold (proved by SMT solvers and Gappa), 2 for the first assertion (proved by SMT solvers), 1 for the second assertion (proved by Gappa only), 3 to check that the loop invariants are preserved by any loop iteration (proved by SMT solvers but not Gappa), and finally the postcondition, proved by SMT solvers. In all and thanks to the lemmas, everything is proved with automatic provers only.

The constant B was determined using Gappa, similarly as for the cosine example. Here are other values of B in function of NMAX, together with the corresponding approximate value of NMAX×C, that is the bound we obtain on the error at the end of iteration.

NMAX	10^3	10^4	10^5	10^6	10^7
B	2^{-18}	2^{-15}	2^{-11}	2^{-8}	2^{-4}
NMAX×C	0.004	0.3	48.8	3,906	625,000

In other words, it is proved that the drift after a bit more than one day is not larger than 4,000 s. This is a lot, but it is of course only an upper bound, the real error is much smaller because rounding errors compensate. Such compensation phenomena are discussed in other examples below.[7]

3.2 Playing with Special Values: Infinities and NaNs

The purpose of this example is to illustrate the support of special values. In most cases, one wants to guarantee that no overflow occur, but if overflows are intended, then our tool chain can support it, by declaring a specific pragma in the code. Additionally, ACSL proposes a set of constructs to handle special values.

The following code illustrates that.

```
#pragma JessieFloatModel(full)

/*@ predicate sorted{L}(double *t, integer a, integer b) =
  @    \forall integer i,j; a <= i <= j <= b ==> \le_float(t[i],t[j]);
  @*/

/*@ requires n >= 0 && \valid_range(t,0,n-1);
  @ requires ! \is_NaN(v);
  @ requires \forall integer i; 0 <= i <= n-1 ==> ! \is_NaN(t[i]);
  @ ensures -1 <= \result < n;
  @ behavior success:
  @   ensures \result >= 0 ==> \eq_float(t[\result],v);
  @ behavior failure:
  @   assumes sorted(t,0,n-1);
  @   ensures \result == -1 ==>
  @     \forall integer k; 0 <= k < n ==> \ne_float(t[k],v);
  @*/
int binary_search(double t[], int n, double v) {
  int l = 0, u = n-1;
  /*@ loop invariant
    @   0 <= l && u <= n-1;
    @ for failure:
```

[7] To conclude the story of the missile battery bug: indeed fixed-point computations where in use there, which make the rounding errors larger because they do not compensate. The lesson to learn is that it would be much better to use a representable number for the tick interval: 1/8 or 1/16 for example.

```
    @   loop invariant
    @     \forall integer k;
    @      0 <= k < l ==> \lt_float(t[k],v);
    @   loop invariant
    @     \forall integer k;
    @      u < k <= n-1 ==> \lt_float(v,t[k]);
    @ loop variant u-l;
    @*/
  while (l <= u ) {
    int m = l + (u - l) / 2;
    if (t[m] < v) l = m + 1;
    else if (t[m] > v) u = m - 1;
    else return m;
  }
  return -1;
}
```

It is a classical binary search of a value in an array sorted in increasing order. Here, we want an array of floats, and we want to allow infinite values to occur, but no NaNs. The pragma switches the Jessie plugin in the mode where overflows are allowed. The predicate `sorted`(t, a, b) means that array t is sorted between indices a and b included. Since it contains floats and possibly infinities, we cannot use real number comparison <= but must use the predicate `le_float` extended to special values, specified by the IEEE-754 standard.

The function searches a given value v in an array t, between indices 0 and $n - 1$. The first precondition tells that array t is properly allocated for these indices. The second requires that v is not NaN, and the third requires that the array contains no NaNs either. Notice that since we want a sorted array, NaNs must be excluded. The first postcondition tells that the result value is between -1 and $n - 1$. When the result is non-negative, it should be an index of a cell of t containing v, this is the postcondition of the behavior `success`. When the result is -1, one expects that v does not occur in t, this is the postcondition of behavior `failure`. The soundness of this behavior is valid only under the assumption that the array is sorted.

The loop invariants are needed to prove the failure behavior: they state that at each loop iteration, if the value v occurs somewhere then it must be between the indices l and u. More precisely, values before index l must be lower whereas values after u must be higher.

On this code, our tool chain produces 37 VCs. 19 VCs comes from runtime error checks, and are all proved by both CVC3 and Z3. 8 VCs comes from the loop invariants and the global post-condition, all are proved with all SMT provers. The VCs corresponding to behaviors are all proved by CVC3. Notice that Gappa is not useful here, because this program does not involve computations, only comparisons. Moreover, the handling of special values is implemented in the axiomatic part of Fig. 4.

We refer to [1] for more details on the handling of special values, and also other specificities such as changing the default rounding mode.

3.3 Sterbenz Subtraction

One of the most basic property of floating-point arithmetic is the following one discovered by Kahan and Sterbenz [37] in the 1970s. If the inputs of a subtraction are near enough one to another, then the subtraction is exact.

This is specified as a function that returns the subtraction between its two floating-point inputs. The precondition is that x and y are such that $\frac{y}{2} \leq x \leq 2y$. Note that the division and multiplication inside the annotations are exact. The postcondition is the fact that the result is equal to the mathematical subtraction of the inputs and therefore does not suffer from rounding.

```
/*@ requires y / 2.0 <= x <= 2.0 * y;
  @ ensures  \result == x - y;
  @*/
float Sterbenz(float x, float y) {
  return x - y;
}
```

There are two VCs that were proved with the Coq interactive prover. The first one is the postcondition: the fact that the result is exactly $x - y$. This was rather easy (7 lines long) as it is only the application of a more general theorem of the library (any precision, any radix, any rounding). The other proof is the fact that the computation $x - y$ cannot overflow. The reason is that x and y are nonnegative (as $\frac{y}{2} \leq x \leq 2y$) and therefore $|x - y| \leq \max(|x|, |y|)$ cannot be greater than the overflow threshold. This was easy but a little tedious and 20 lines long.

Using Automatic Provers One can reduce the amount of proof to do within Coq by inserting appropriate assertions in the code, to state that x and y are non-negative. The code is as follows:

```
/*@ requires  y / 2.0 <= x <= 2.0 * y;
  @ ensures   \result == x - y;
  @*/
float Sterbenz(float x, float y) {
  //@ assert 0.0 <= y;
  //@ assert 0.0 <= x;
  return x - y;
}
```

Now, Alt-Ergo is able to automatically prove the two assertions, Gappa is able to automatically prove the absence of overflow (and the second assertion too), so the only VC remaining to prove within Coq is the post-condition, proved in 7 lines.

3.4 Veltkamp/Dekker Algorithm

This example is also from the 1970s but is much more complex. Here, the floating-point properties have been thoroughly used in order to get an exact result. This function indeed computes the exact error of the multiplication [20,38] with only floating-point operations (and no FMA). This requires 16 operations and each of them has a specific role to play to get the exact error. This algorithm was already proved with a generic radix and precise underflow restrictions and overflow restrictions so that no infinity will be created [6]. The question was how to specify it as a program to get the full benefit of the proof.

```
/*@ requires xy == \round_double(\NearestEven,x*y) &&
  @          \abs(x) <= 0x1.p995 &&
  @          \abs(y) <= 0x1.p995 &&
  @          \abs(x*y) <=  0x1.p1021;
  @ ensures  ((x*y == 0 || 0x1.p-969 <= \abs(x*y))
  @                 ==> x*y == xy+\result);
  @*/
double Dekker(double x, double y, double xy) {

  double C,px,qx,hx,py,qy,hy,tx,ty,r2;
  C=0x8000001p0;
  /*@ assert C == 0x1p27+1; */

  px=x*C;
  qx=x-px;
  hx=px+qx;
  tx=x-hx;

  py=y*C;
  qy=y-py;
  hy=py+qy;
  ty=y-hy;

  r2=-xy+hx*hy;
  r2+=hx*ty;
  r2+=hy*tx;
  r2+=tx*ty;
  return r2;
}
```

We have two floating-point numbers x and y. The rounding of $x \times y$ in rounding to nearest, xy, is provided to the function: this is the first part of the pre-condition above. We then have several pre-conditions to prevent overflow: $|x|$, $|y|$ and $|x \times y|$ must not be too big to guarantee that no computation inside the function will overflow. The postcondition is more interesting: we do not require underflow conditions (as it will not mean any failure, only a less precise result [6]), but we assume that $|x \times y|$ is either 0 or greater than 2^{-969}. This is enough to guarantee that underflow will not endanger the result. Note that it does not mean there is no subnormal number: for example x might be subnormal if y is big enough and the result will still be correct. What is guaranteed by this function is that it computes the error of the multiplication, that is to say the floating-point number r_2 such that $x \times y = xy + r_2$ where all computations are mathematical exact ones (as in annotations). We have the result mathematically equal to $x \times y - xy$ under the given assumptions.

Even with the previous results of [6], the proofs were still quite long. The reasons are the following ones:

- the previous proofs handled underflows while here, we assume one degenerate case does not happen (while x or y may be subnormal). So the splitting of cases is different and that makes the reusing of proofs difficult.
- the overflow proofs were done by hand in [6] and were very tedious. We then used the Gappa tactic [12] to take advantage of the Gappa automations inside Coq. This was especially interesting here as the use of Gappa requires high-level knowledge only given by another part of the Coq proof: for example the fact that hx is the head of x, that is to say the rounding to nearest on 27 bits of x. The Gappa tactic has saved a lot of lines of Coq and has enhanced the result: the overflow requirements are now weaker.

Given the results of [6], the additional proofs and the full better overflow proof are less than 900 lines long. This also includes the simplest overflow proofs directly done by Gappa which are proved in one line. Note that the file has 20 theorems to prove, is more than 2700 lines long and takes nearly 15 min to be compiled on a 2×3GHz processor.

3.5 Kahan Algorithm for an Accurate Discriminant

The next function illustrates two different features. The first one is the function call: we will use the preceding Dekker function. The second one is the fact that a floating-point test may be wrong.

This function computes an accurate discriminant, meaning $b^2 - a \times c$ using Kahan's algorithm [28]. The result is proved correct within 2 ulps in the original paper and then formally proved in [10]. The ulp is the unit in the last place of a floating-point number, meaning the value of the last bit of its mantissa. For example in double precision, $\mathrm{ulp}(1) = 2^{-52}$.

Unfortunately, both the pen-and-paper proof and the initial formal proofs were assuming that the test was giving the correct answer, meaning that $p + q \leq 3|p - q| \iff \circ(p + q) \leq \circ(3 \times |\circ(p - q)|)$, where $\circ$ is the rounding to nearest. But this is incorrect as we may find a and b such that this does not hold. The solution was to look deeper into these few cases. More precisely, they correspond to particular values of p and q where $p \approx 2q$ or vice versa. A special proof is done in these few degenerate cases as the values are in a small subset and the algorithm is stable enough to be correct. In all cases, we guarantee the result still holds [8].

Note that the computation of $3 \times |p - q|$ may overflow and therefore requires tighter bounds on b and $a \times c$ than expected.

To specify the postcondition, we have the same kind of requirements: underflow and overflow requirements, that subsume Dekker's ones. The interesting part is that ulp is not part of our basic axiomatic of floating-point arithmetic (it could have been, but we chose to limit the number of keywords). We then define an axiomatic that defines it as we wish: it is a power of the radix such that, for any normal floating-point number f, we have $2^{52}\mathrm{ulp}(f) \leq |f| < 2^{53}\mathrm{ulp}(f)$, and for any subnormal floating-point number f, we have $\mathrm{ulp}(f) = 2^{-1074}$.

```
/*@ axiomatic FP_ulp {
  @ logic real ulp(double f);
  @
  @ axiom ulp_normal1:   \forall double f; 0x1p-1022 <= \abs(f)
  @       ==> \abs(f) < 0x1.p53 * ulp(f);
```

```
  @ axiom ulp_normal2:   \forall double f; 0x1p-1022 <= \abs(f)
  @       ==> ulp(f) <= 0x1.p-52 * \abs(f);
  @ axiom ulp_subnormal: \forall double f; \abs(f) <  0x1p-1022
  @       ==> ulp(f) == 0x1.p-1074;
  @ axiom ulp_pow :      \forall double f;
  @       \exists integer i; ulp(f) == \pow(2.,i);
  @ } */


/*@ requires
  @     (b==0.   || 0x1.p-916 <= \abs(b*b))
  @  && (a*c==0. || 0x1.p-916 <= \abs(a*c))
  @  && \abs(b) <= 0x1.p510 && \abs(a) <= 0x1.p995 && \abs(c) <= 0x1.p995
  @  && \abs(a*c) <= 0x1.p1021;
  @ ensures \result==0.  || \abs(\result-(b*b-a*c)) <= 2.*ulp(\result);
  @ */
double discriminant(double a, double b, double c) {
  double p,q,d,dp,dq;
  p=b*b;
  q=a*c;

  if (p+q <= 3*fabs(p-q))
    d=p-q;
  else {
    dp=Dekker(b,b,p);
    dq=Dekker(a,c,q);
    d=(p-q)+(dp-dq);
  }
  return d;
}
```

The generated Coq file contains first some declarations generated from the axiomatic `FP_ulp`: we fill the definition of ulp with the ulp defined in the support library, and prove the 4 theorems corresponding to the 4 axioms, by 90 lines of proof script.

The remaining of the Coq file contains 17 verification conditions. As the algorithm's proof was already done, these are quite short to do: 150 lines to prove the postcondition (that requires the check of all the overflow/underflow assumptions), 140 lines for the safety proofs (Dekker preconditions and overflows).

3.6 Wave Equation Resolution Scheme

To finish the examples, here is a numerical analysis program. This function is a finite difference numerical scheme for the resolution of the one-dimensional acoustic wave equation. Given a rope attached at its two ends, we create a wave by applying a force (initializations). The rope then undulates, its behavior being modelled by some mathematical equations that can be discretized and computed. The mathematical model is the search for a function u from $\mathbb{R}^2$ to $\mathbb{R}$, solution of the differential equation

$$\frac{\partial^2 u(x,t)}{\partial t^2} - c^2 \frac{\partial^2 u(x,t)}{\partial x^2} = 0$$

knowing initial values of u and its derivative for $t = 0$. The value $u(x,t)$ gives the position of the rope at the abscissa x and the time t. It is discretized both in space and time with steps $(\Delta x, \Delta t)$. The result is a matrix p where $p[i][k] = p_i^k$ is the position of the rope at the abscissa $i \times \Delta x$ and the time $k \times \Delta t$. The matrix p is computed by the following piece of code. For simplicity, annotated initialization functions are omitted.

```
/*@ axiomatic dirichlet {
  @  predicate analytic_error{L}
  @           (double **p, integer ni, integer i, integer k, double a)
  @           reads p[..][..];  } */


/*@ requires ni >= 2 && nk >= 2
  @          && 1 != 0
  @          && dx > 0. && dt > 0.  && v > 0.
  @          && \exact(dx) > 0. && \exact(dt) > 0.
  @          && \exact(v)==v
```

```
  @           && \abs(\exact(dx) - dx) / dx <= 0x1.p-53
  @           && \abs(\exact(dt) - dt) / dt <= 0x1.p-51
  @           && 3./5. <= \exact(dt)/\exact(dx) * \exact(v) <= 1-0x1.p-50
  @           && 0x1.p-1000 <= v <= 0x1.p1000
  @           && ni <= 0x1.p64 && nk <= 4194304
  @           && \exact(dx) <= 1;
  @
  @ ensures \forall integer i; \forall integer k;
  @     0 <= i <= ni ==> 0 <= k <= nk ==>
  @     \round_error(\result[i][k]) <= 78./2*0x1.p-52*(k+1)*(k+2); */
double **forward_prop(int ni, int nk, double dx, double dt, double v,
                      double xs, double l) {
  double **p;
  int i, k;
  double a1, a, dp;

  a1 =  dt/dx*v;
  a  = a1*a1;
  /*@ assert 1./4 <= a <= 1 && 0 < \exact(a) <= 1 &&
    @         \round_error(a) <= 0x1.p-49; */

  p = array2d_alloc(ni+1, nk+1);

  p[0][0]=0.;
  /*@ loop invariant 1 <= i <= ni && analytic_error(p,ni,i-1,0,a);
    @ loop variant ni-i; */
  for (i=1; i<ni; i++) {
    p[i][0] = p_zero(xs, l, i*dx);
  }
  p[ni][0] =0.;

  p[0][1] = 0.;
  /*@ loop invariant 1 <= i <= ni && analytic_error(p,ni,i-1,1,a);
    @ loop variant ni-i; */
  for (i=1; i<ni; i++) {
    dp = p[i+1][0] - 2.*p[i][0] + p[i-1][0];
    p[i][1] = p[i][0] + 0.5*a*dp;
  }
  p[ni][1] = 0.;

  /* propagation = time loop */
  /*@ loop invariant 1 <= k <= nk && analytic_error(p,ni,ni,k,a);
    @ loop variant nk-k; */
  for (k=1; k<nk; k++) {
    p[0][k+1] = 0.;

    /* time iteration = space loop */
    /*@ loop invariant 1 <= i <= ni && analytic_error(p,ni,i-1,k+1,a);
      @ loop variant ni-i; */
    for (i=1; i<ni; i++) {
      dp = p[i+1][k] - 2.*p[i][k] + p[i-1][k];
      p[i][k+1] = 2.*p[i][k] - p[i][k-1] + a*dp;
    }
    p[ni][k+1] = 0.;
  }
  return p;
}
```

There are two very different parts to prove the correctness of this program. The first one is the bound on the rounding error. This was done in [7] and the corresponding annotated C file is given above. The annotations are particularly decisive here as this proof requires a complex predicate defined in Coq. More precisely, we proved the that $p_i^k - \texttt{\textbackslash exact}(p_i^k) = \sum_{l=0}^{k} \sum_{j=-l}^{l} \alpha_j^l \ \varepsilon_{i+j}^{k-l}$, where (α) is a constant sequence depending on the wave equation constants and ε is the rounding error made by one iteration of the loop, assuming the inputs are correct. The loop invariant contains both bounds on the ε_i^k and the fact that the rounding error is exactly defined by the previous formula. This allows us to exhibit the compensation of rounding errors and then have a very good bound, proportional to k^2 for the rounding error of the computed values: as we need to compute k^2 values to obtain p_i^k, to obtain an error proportional to k^2 is a first-rate bound.

What is proved is a bound on the rounding error assuming the scheme does not diverge. There are 117 verification conditions and the proofs are about 4,000 lines long. The Coq compilation takes about 1 h and 10 min. Among

the 117 verification conditions, there are 84 safety proof obligations that have to do with memory access, variant decrease, overflow, and precondition for function call.

The second part is the proof of the method error. This is a mathematical proof done in [9]: it proves that the scheme does not diverge and that the program computes something near the exact solution of the partial differential equation, assuming the rope is infinite. The mathematical method error proof (with an axiom about the finite support of the solution, see [9] for more details) is also about 4,000 lines long. The proved bound is the one of the mathematical textbooks, proportional to $\Delta x^2 + \Delta t^2$.

It remains to adapt this last proof for a finite rope and to link the two proofs to try to get rid of axioms. This linking part will notice that the mathematical discretization of the differential equation described in [9] is exactly the one of the previous program.

An important question is the re-usability of this proof. The method that expresses exactly $p_i^k - \texttt{\textbackslash exact}(p_i^k)$ as a mathematical formula is powerful, but not applicable to any program. We think this technique can be applied to several cases of numerical scheme, as used numerical schemes have many mathematical properties, such as stability, that could be used to bound rounding errors. All the correct and practical definitions of the mathematics needed by the method error proof could be reused (about half of the developments), but the proof itself (the other half) heavily depends on the wave equation and the chosen scheme.

4 Conclusion

We have shown that a very high guarantee on numerical programs is achievable. We have a usable chain from the annotated program to the mathematical theorems that is able to secure a program both from its rounding errors and from its other possible failures (pointer dereferencing, out-of-bound array accesses, overflows…).

A positive point in this approach is that we build upon a highly expressive specification language, which allows to specify arbitrarily complex intended behaviors. Then, thanks to the multiple provers back-end, simple VCs such as the ones coming from overflow checks can be discharged with a fair amount of automation. But also, for complex properties we always have the possibility to switch to Coq and perform complex computer-assisted proofs. Thus, a given specified program can be proved in its deeper details using Coq and its correctness therefore ascertained.

A never-ending perspective is to find cunning techniques to better bound the rounding errors, especially when they compensate. A technique that states the analytical error has been developed in [7] for the example of Sect. 3.6 where usual methods gave an error proportional to 2^k that was cut down to k^2. This technique of the analytical error and precise floating-point error cancellation coming with its formal proof is new. The reason is that it requires very generic specifications as the loop invariant needs to be logically defined: it states there exists a function that has such and such property. And ACSL allows us to express such a high-level property on a C program. We then use Coq as a back-end to formally check the specifications. This genericity is an advantage compared to automatic methods that cannot express our loop invariant.

A fair question to ask is how much of expertise is needed for someone new to this approach to prove its own program from scratch. Important issues need to be addressed to spread the use of deductive verification of floating-point programs.

- First, the appropriate annotations may be hard to find. Unlike tools specialized to some specific property, deductive verification approach offers a very expressive specification language allowing the user to potentially formalize any desired property. Assuming that the programmer has in mind a clear idea of the expected property, a mandatory step is to formalize it in the specification language. This is something that can be learned similarly as programming can be learned. Performing a proof that the program verifies this specification can be difficult too, and is also something that can be learned. One should not try to make a computer-assisted proof before writing a pen and paper proof first, from which the adequate lemmas and/or loop invariants can be discovered. One could not expect a systematic recipe for such a task, however any tool instrumentation could be used, such as first displaying graphs to provide intuitions, or using a tool to compute appropriate bounds like we did with Gappa in some examples. Then turning pen and paper proofs into formal ones allows to uncover the gaps.

- If the program is large, then it is likely that the number of annotations needed will be large too. Known tools that scale up on large programs typically use abstract interpretation techniques with well-chosen domains [18,21]. These do not need manual annotations but are able to verify only a restricted kind of properties such as: safety (no runtime error), bounds on rounding errors, stability of conditional branching.
  Our approach aims more at proving complex properties on small programs. Typical applications can be for example the so-called SCADE operators, which are small hand-written subroutines inserted in automatically generated large C codes. In the context of numerical analysis, there are also quite short subroutines that can be proved [7]. This emphasizes an important feature of our approach: its modularity. A function is annotated and proved independently from the rest of the program. Modularity allows to cope with libraries of formally verified programs, that can be reused. We believe this is the right path to spread the use of deductive verification.
  A promising future work is to make various kind of static analyses to cooperate. For example, the technique of *slicing*[8] reduces a program with respect to a given property to check, such that the property holds on the full program if it holds on the reduced program. There are also promising techniques combining abstract interpretation and weakest preconditions calculus for automatically generating and propagating annotations [34]. The plug-in architecture of Frama-C allows and even encourages such cooperations.
- The annotations may need to be crafted in a particular way to maximize automation. Clearly this requires an advanced understanding of the respective power of various provers. In some examples of this paper, we had to insert assertions to play the role of cuts in the VCs, so that one part is solved by Gappa and others by SMT solvers. An interesting perspective is to turn the Gappa underlying techniques into a decision procedure for a theory of floating-point rounding in a suitable form for SMT solvers. Generally speaking, writing the annotations and performing the proof is an interactive process where some feedback from failing proof must be taken into account to adjust the annotations. Sometimes, trying to complete the proof of a VC in Coq helps to understand why automatic provers fail and gives an idea of an appropriate cut. The final annotated code, as we gave it here for each example, is only the result of such an iterative process and does not show its intermediate steps.

Acknowledgments We would like to thank the other people who contributed to the design of the tools and some of the examples: Ali Ayad, for contributions to the floating-point support in Frama-C/Jessie, and examples on approximations on transcendental functions and the clock drift; François Clément, Jean-Christophe Filliâtre, Micaela Mayero and Guillaume Melquiond, for their contribution to the wave equation example; and more generally Guillaume Melquiond for his support on the use of Gappa and the Coq `interval` tactic.

References

1. Ayad, A., Marché, C.: Multi-prover verification of floating-point programs. In: Giesl, J., Hähnle, R. (eds.) Fifth International Joint Conference on Automated Reasoning. Lecture Notes in Artificial Intelligence, Springer, Edinburgh (2010)
2. Barnett M., Leino K.R.M., Schulte W.: The spec# programming system: an overview. In: Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04). Lecture Notes in Computer Science, vol. 3362, pp. 49-69. Springer, Berlin (2004)
3. 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
4. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer, Berlin (2004)
5. Boldo, S.: Preuves formelles en arithmétiques à virgule flottante. PhD thesis, École Normale Supérieure de Lyon (2004)
6. Boldo, S.: Pitfalls of a full floating-point proof: example on the formal proof of the Veltkamp/Dekker algorithms. In: Furbach, U., Shankar, N. (eds.) Third International Joint Conference on Automated Reasoning. Lecture Notes in Computer Science, Seattle, USA, vol. 4130, pp. 52–66. Springer, Berlin (2006)
7. Boldo, S.: Floats & Ropes: a case study for formal numerical program verification. In: 36th International Colloquium on Automata, Languages and Programming, Rhodos, Greece. Lecture Notes in Computer Science—ARCoSS, vol. 5556, pp. 91–102. Springer, Berlin (2009)
8. Boldo, S.: Kahan's algorithm for a correct discriminant computation at last formally proven. IEEE Trans. Comput. **58**(2), 220–225 (2009)

[8] http://frama-c.com/slicing.html.

9. Boldo, S., Clément, F., Filliâtre, J.-C., Mayero, M., Melquiond, G., Weis, P.: Formal proof of a wave equation resolution scheme: the method error. In: Proceedings of the First Interactive Theorem Proving Conference, Edinburgh, Scotland. LNCS, Springer, Berlin (2010)
10. Boldo, S., Daumas, M., Kahan, W., Melquiond, G.: Proof and certification for an accurate discriminant. In: 12th IMACS-GAMM International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics, Duisburg, Germany (2006)
11. Boldo, S., Filliâtre, J.-C.: Formal verification of floating-point programs. In: 18th IEEE International Symposium on Computer Arithmetic, Montpellier, France, pp. 187–194 (2007)
12. 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, Grand Bend, Canada. Lecture Notes in Artificial Intelligence, vol. 5625, pp. 59–74. Springer, Berlin (2009)
13. Boldo, S., Nguyen, T.M.T.: Hardware-independent proofs of numerical programs. In: Muñoz, C. (ed.) Proceedings of the Second NASA Formal Methods Symposium. number NASA/CP-2010-216215 in NASA Conference Publication, Washington D.C., USA, pp. 14–23 (2010)
14. Boldo, S., Nguyen, T.M.T.: Proofs of numerical programs when the compiler optimizes. Innov. Syst. Softw. Eng. **7**, 1–10 (2011)
15. 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. Int. J. Softw. Tools Technol. Transf. (STTT) **7**(3), 212–232 (2005)
16. Carré, B., Garnsworthy, J.: SPARK—an annotated Ada subset for safety-critical programming. In: Proceedings of the Conference on TRI-ADA'90, TRI-Ada'90, pp. 392–402. ACM Press, New York (1990)
17. 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 (1995)
18. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In: ESOP. Lecture Notes in Computer Science, vol. 3444, pp. 21–30 (2005)
19. Daumas M., Rideau L., Théry L.: A generic library of floating-point numbers and its application to exact computing. In: 14th International Conference on Theorem Proving in Higher Order Logics, Edinburgh, Scotland, pp. 169–184 (2001)
20. Dekker, T.J.: A floating point technique for extending the available precision. Numerische Mathematik **18**(3), 224–242 (1971)
21. 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, Berlin (2009)
22. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) 19th International Conference on Computer Aided Verification. Lecture Notes in Computer Science, vol. 4590, pp. 173–177. Springer, Berlin (2007)
23. Gerlach, J., Burghardt, J.: An experience report on the verification of algorithms in the c++ standard library using frama-c. In: Beckert, B., Marché, C. (eds.) Formal Verification of Object-Oriented Software, Papers Presented at the International Conference, Karlsruhe Reports in Informatics, pp. 191–204. Paris, France, June (2010). http://digbib.ubka.uni-karlsruhe.de/volltexte/1000019083
24. Goubault, E., Putot, S.: Static analysis of numerical algorithms. In: Yi, K. (ed.) SAS. LNCS, vol. 4134, pp. 18–34. Springer, Berlin (2006)
25. 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, pp. 217–233 (2000)
26. Higham, N.J.: Accuracy and stability of numerical algorithms, 2nd edn, xxx+680 pp. SIAM, Philadelphia (2002). http://www.maths.manchester.ac.uk/~higham/asna/
27. IEEE.: IEEE Standard for Floating-Point Arithmetic. IEEE Std. 754-2008 (2008)
28. Kahan, W.: On the cost of Floating-Point Computation Without Extra-Precise Arithmetic. World-Wide Web document, Nov. (2004)
29. Leavens, G.T.: Not a number of floating point problems. J. Object Technol. **5**(2), 75–83 (2006)
30. Melquiond, G.: Floating-point arithmetic in the Coq system. In: Proceedings of the 8th Conference on Real Numbers and Computers, pp. 93–102. Santiago de Compostela, Spain (2008)
31. Melquiond, G.: Proving bounds on real-valued functions with computations. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Proceedings of the 4th International Joint Conference on Automated Reasoning, Sydney, Australia. Lecture Notes in Artificial Intelligence, vol. 5195, pp. 2–17 (2008)
32. Monniaux, D.: Analyse statique : de la théorie à la pratique. Habilitation to direct research Université Joseph Fourier, Grenoble, France (2009)
33. Moy, Y., Marché, C.: Jessie Plugin Tutorial, Beryllium version. INRIA (2009). http://www.frama-c.cea.fr/jessie.html
34. Moy, Y., Marché, C.: Modular inference of subprogram contracts for safety checking. J. Symb. Comput. **45**, 1184–1211 (2010)
35. Necula, G.C., McPeak, S., Rahul, S.P., Weime, W.: Cil: Intermediate language and tools for analysis and transformation of c programs. In: Conference on Compiler Construction (CC'02) (2002)
36. 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 J. Comput. Math. **1**, 148–200 (1998)
37. Sterbenz, P.H.: Floating Point Computation. Prentice Hall, Upper Saddle River (1974)
38. Veltkamp, G.W.: Algolprocedures voor het berekenen van een inwendig product in dubbele precisie. RC-Informatie 22, Technische Hogeschool Eindhoven (1968)
39. Wilkinson, J.H.: Rounding Errors in Algebraic Processes. Prentice-Hall, Upper Saddle River, NJ 07458, USA (1963)