Automating the Verification of Floating-Point Programs

Clément Fumex, Claude Marché, Yannick Moy

Automating the Verification of Floating-Point Programs*

Clément Fumex[1,2,3], Claude Marché[1,2], and Yannick Moy[3]

[1] Inria, Université Paris-Saclay, F-91120 Palaiseau
[2] LRI, CNRS & Univ. Paris-Sud, F-91405 Orsay
[3] AdaCore, F-75009 Paris

Abstract. In the context of deductive program verification, handling floating-point computations is challenging. The level of proof success and proof automation highly depends on the way the floating-point operations are interpreted in the logic supported by back-end provers. We address this challenge by combining multiple techniques to separately prove different parts of the desired properties. We use abstract interpretation to compute numerical bounds of expressions, and we use multiple automated provers, relying on different strategies for representing floating-point computations. One of these strategies is based on the native support for floating-point arithmetic recently added in the SMT-LIB standard. Our approach is implemented in the Why3 environment and its front-end SPARK 2014 for the development of safety-critical Ada programs. It is validated experimentally on several examples originating from industrial use of SPARK 2014.

1 Introduction

Numerical programs appear in many critical software systems, for example to compute trajectories, to control movements, to detect objects. As most processors are now equipped with a floating-point (FP for short) unit, many numerical programs are implemented in FP arithmetic, to benefit from the additional precision of FP numbers around the origin compared to fixed-point numbers, and from the speed of FP computations performed in hardware.

Safety conditions for critical software systems require strong guarantees on the functional behavior of the computations performed. Automatically verifying that these guarantees are fulfilled is desirable. Among other verification approaches, deductive program verification is the one that offers the largest expressive power for the properties to verify: complex functional specification can be stated using expressive formal specification languages. Verification then relies on the abilities of automated theorem provers to check that a code satisfies a given formal specification.

* Work partly supported by the Joint Laboratory ProofInUse (ANR-13-LAB3-0007, http://www.spark-2014.org/proofinuse) and by the SOPRANO project (ANR-14-CE28-0020, http://soprano-project.fr/) of the French national research organization

For some time, FP arithmetic was not well-supported by automated provers and thus deductive verification of FP programs was relying on interactive proof assistants, requiring a lot of expertise [6]. In recent years, FP arithmetic started to be supported natively by the automated solvers of the SMT (*Satisfiability Modulo Theory*) family. A theory reflecting the IEEE-754 standard for FP arithmetic [24] was added in the SMT-LIB standard in 2010 [33]. This theory is now supported by at least the solvers Z3 [32] and MathSAT5 [8].

Our initial goal is to build upon the recent support for FP arithmetic in SMT-LIB to propose an environment for deductive verification of numerical programs with a higher level of automation. We implemented this approach in the program verifier Why3 and in its front-end SPARK for verifying Ada programs. Indeed a support for FP arithmetic in Why3 already existed before [1,6], but it was based on an axiomatization of FP operations in terms of their interpretations into operations on real numbers. It was suitable for using as back provers either the Gappa solver dedicated to reason about FP rounding [14], for the simplest verification conditions, or the Coq proof assistant for the rest. To achieve a higher level of automation, in particular on the examples we considered coming from users of SPARK, we identified the need for combining several theorem provers, and even more, a need for combining deductive verification with an abstract interpretation based analysis (namely the CodePeer tool for Ada). The main goal of the new support we designed is thus to exploit the SMT solvers with native support for FP arithmetic (Z3), while maintaining the ability to use solvers that do not offer native support (CVC4, Alt-Ergo, Gappa, Coq). Our new approach can prove automatically FP properties that were beyond the reach of the previous approach. For example, we were previously unable to prove automatically the assertion in the following toy Ada code, because the rounding error on `X + 2.0` was over-approximated.

```
procedure Range_Add (X : Float_32; Res : out Float_32) is
begin
   pragma Assume (X in 10.0 .. 1000.0);
   Res := X + 2.0;
   pragma Assert (Res >= 12.0);
end Range_Add;
```

We first give a quick introduction to auto-active verification with the environments SPARK and Why3 in Section 2. In Section 3, we present a new Why3 theory used in the verification condition generation process so as to exploit the support of FP arithmetic in SMT-LIB, while still keeping use of other provers, thanks to an axiomatization. We evaluate experimentally our approach in Section 4, showing our results on 22 examples extracted from industrial programs. We also present a case study for the computation of safe bounds for the trajectory of a device from an embedded safety-critical software, where the combination of techniques is achieved through the insertion of ghost code. We refer to our extended research report [19] for more details on our approach, including the full axiomatization and the complete source code of our examples. Section 5 draws conclusions and discusses some related work and future work.

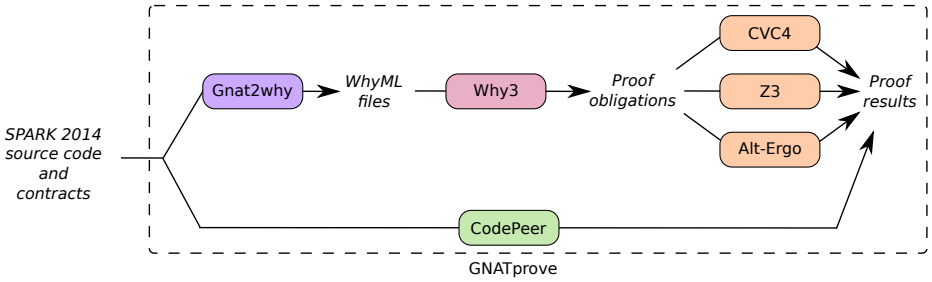

Fig. 1. Deductive verification in SPARK 2014

2 Quick Introduction to SPARK and Why3

Why3 is an environment for deductive program verification, providing a rich language for specification and programming, called WhyML. WhyML is used as an intermediate language for verification of C, Java or Ada programs [17,25], and is also intended to be comfortable as a primary programming language. Why3 generates proof obligations, called verification conditions (VCs for short), from program annotations using a weakest-precondition calculus. It relies on external provers, both automated and interactive, in order to discharge the auxiliary lemmas and VCs. As most of the provers do not support some of the language features (typically pattern matching, polymorphic types, or recursion), Why3 applies a series of encoding transformations to eliminate unsupported constructions before dispatching a VC. Why3 comes with a rich standard library providing general-purpose theories useful for specifying programs [3]. This naturally includes integer and real arithmetic. In this work, we added a new theory of floating-point arithmetic, that we present in Section 3. Why3 provides a mechanism called *realization* that allows a user to construct a model for her axiomatizations, using a proof assistant. This feature can be used to guarantee that an axiomatization is consistent, or even that it is a faithful abstraction of an existing model. In Section 3.6, we use this feature to ensure that our own axiomatization of FP arithmetic is faithful to the IEEE-754 standard [24].

SPARK is an environment for the verification of Ada programs used in critical software development [10,29]. The SPARK language subset and toolset for static verification has been applied for many years in on-board aircraft systems, control systems, cryptographic systems, and rail systems. As displayed in Figure 1, to formally prove a SPARK program, the tool GNATprove uses WhyML as an intermediate language. The SPARK program is translated into an equivalent WhyML program which can then be verified using the Why3 tool. Since version SPARK 17, GNATprove also includes the static analyzer CodePeer as prover. CodePeer [2] is a tool developed at AdaCore to detect errors in Ada programs, based on modular abstract interpretation. The benefit of this integration for our work is that CodePeer computes precise bounds on FP computations.

Automatic verification in both Why3 and GNATprove relies on the ability to interact with users through assertions, and more generally verification-only code also called *ghost code* [25]. This type of verification is called *auto-active verification*, to characterise *tools where user input is supplied before VC generation [and] therefore lie between automatic and interactive verification* (hence the name auto-active) [26]. We use ghost code in Section 4.2 to prove a case study involving FP computations.

3 VC Generation for Floating-Point Computations

We now describe what we designed for the support for FP types in Why3 and then how we use this support to enhance SPARK's already existing support for FP types. In Section 3.1 we present the signature, in other words the user interface, of our Why3 formalization of the IEEE-754 standard. That signature is generic, parameterized by the size of FP numbers. In Section 3.2 we show how we build specific instances for 32 and 64 bits formats, and how we can write literal constants in these formats. Then in Section 3.3 we show how the FP computations in Ada programs are translated by GNATprove into Why3 intermediate code based on our formalization, thus producing verification conditions involving the symbols of our signature. Section 3.4 explains how we map our signature to the SMT-LIB FP theory, so as to exploit SMT solvers with native support for that theory. Section 3.5 presents an axiomatization for our Why3 theory of FP, to be used by provers that do not support FP natively. Finally Section 3.6 explains how we ensure that our theory and axiomatization is conformant to the IEEE standard.

The IEEE-754 standard [24] defines an expected behavior of FP computations. In any binary formats, an interpretation of a bit sequence under the form of a sign, a mantissa and an exponent is given, so that the set of FP numbers denotes a finite subset of real numbers.

sign s	biased exponent e	mantissa m

The number of bits of e is denoted eb. The *significand* is the mantissa plus a *hidden bit* which is 1 for the so-called *normal* numbers and 0 for *subnormal* ones. The number of bits of the significand, that is also the number of bits of the mantissa plus 1, is denoted sb. The numbers eb and sb characterize the format, for the standard binary format on 32 bits and 64 bits we respectively have $eb = 8, sb = 24$ and $eb = 11, sb = 53$. Let us call *bias* the number $2^{eb-1} - 1$. The interpretation of the sequence of bits above is then as follows.

- if $0 < e < 2^{eb} - 1$, it represents the real number $(-1)^s \cdot \overline{1.m} \cdot 2^{e-bias}$ (normal numbers)
- if $e = 0$, it represents ± 0 if $m = 0$ (positive and negative zeros), $(-1)^s \cdot \overline{0.m} \cdot 2^{-bias+1}$ otherwise (subnormal numbers)
- if $e = 2^{eb} - 1$, $\pm\infty$ if $m = 0$ (positive and negative infinities) *Not-a-Number* otherwise, abbreviated as NaN.

For each of the basic arithmetic operations (add, sub, mul, div, and also sqrt, fused-multiply-add, etc.) the standard requires that it acts as if it first computes a true real number, and then *rounds* it to a number representable in the chosen format, according to some *rounding mode*. The standard defines five rounding modes: if a real number x lies between two consecutive representable FP numbers x_1 and x_2, then the rounding of x is as follows. With directed modes toward $+\infty$ (resp. $-\infty$) it is x_2 (resp. x_1). With directed mode toward 0, it is x_1 if $x > 0$ and x_2 if $x < 0$. With directed modes to nearest, it is the closest to x among x_1 and x_2, and if x is exactly the middle of $[x_1, x_2]$ then in the case 'ties to away' it is x_2 if $x > 0$ and x_1 if $x < 0$; whereas in the case 'ties to even' the one with even mantissa is chosen.

As seen above, the standard defines three special values: $-\infty, +\infty$ and NaN. It also distinguishes between positive zero (+0) and negative zero (-0). These numbers should be treated both in the input and the output of the arithmetic operations as usual, e.g. $(+\infty) + (+\infty) = (+\infty)$, $(+\infty) + (-\infty) = \text{NaN}$, $1/(-\infty) = -0$, $\pm 0 / \pm 0 = \text{NaN}$, etc.

3.1 Signature for a Generic Theory of IEEE FP Arithmetic

We first present the signature of our theory. Such a signature presents the elements that some user should use: type names, constants, logic symbols for functions and predicates. These elements are only declared with their proper profiles, but no definition nor axiomatization are given.

One of our goals was to make this signature as close as possible to the SMT-LIB theory. One of the difficulty is that one wants to describe a generic signature, in the sense that it should be parameterized by the number of bits *eb* and *sb*. In SMT-LIB, this is done using the ad-hoc built-in construct with underscore character to handle parametric sorts, e.g. `(_ FloatingPoint 8 24)` denotes the sort of IEEE 32-bits binary floating-point numbers. In Why3 there is no such ad-hoc construct, but instead it is possible to define a parametric theory that can be *cloned* later on, for particular instances of the parameters.

Our generic signature for floats thus starts as follows.

```
theory GenericFloat
  constant eb : int  (* number of bits of the exponent *)
  constant sb : int  (* number of bits of the significand *)
  axiom eb_gt_1: 1 < eb
  axiom sb_gt_1: 1 < sb
  type t     (* abstract type of floats *)
```

The abstract type t denotes the sort of FP numbers for the given `eb` and `sb`, both assumed greater than 1.

The next part of the signature provides the rounding modes and the arithmetic operations.

```
type mode = RNE | RNA | RTP | RTN | RTZ
function add mode t t : t    (* add *)
function sub mode t t : t    (* sub *)
```

```
function mul mode t t : t    (* mul *)
function div mode t t : t    (* div *)
```

It continues with the comparison operators

```
predicate le t t
predicate lt t t
predicate ge (x:t) (y:t) = le y x
predicate gt (x:t) (y:t) = lt y x
predicate eq t t                          (* different from = *)
```

It includes predicates for classification of numbers.

```
predicate is_infinite  t
predicate is_nan       t
predicate is_finite    t
```

Finally, it includes rounding and conversion functions.

```
function roundToIntegral mode t : t    (* rounding to an integer *)
function to_real   t     : real   (* conversion to a real number *)
function of_int mode int : t       (* conversion from an integer *)
function to_int mode t   : int       (* conversion to an integer *)
```

Conversions from and to integers need a rounding mode in both ways, since not all integers are representable. `of_int` may even return an infinite value. The results of `to_int` and `to_real` are unspecified if the argument is infinite or NaN. See [19] for additional elements in the signature.

3.2 Theory Clones and FP Literals

The theory above is generic. So far, it does not allow the construction of FP literals. Indeed it is possible only on *clones* of our theory for the binary formats of standard sizes 32 and 64 bits. For the purpose of allowing literal constants, we implemented a new feature in Why3 itself: the possibility to declare a type denoting FP values. Cloning the generic theory for 32-bit format is then done by the Why3 code below.

```
theory Float32

  type t = < float 8 24 >

  clone export GenericFloat with
    type t = t, constant eb = t'eb, constant sb = t'sb,
    function to_real = t'real, predicate is_finite = t'isFinite

end
```

The new declaration feature is the line of the form `type t = < float` *eb sb* `>` above. It introduces a new type identifier t that represents the FP values for the given sizes for *eb* and *sb*. It also introduces the functions `t'eb`, `t'sb`, `t'isFinite` and `t'real`, that are used in the cloning substitution above. The main purpose

this new built-in Why3's declaration is the handling of literals: what is normally a real literal in Why3 can be cast to the type t, so that one may write float literals in decimal (e.g. `(1.0:t)`, `(17.25:t)`), possibly with exponent (e.g. `(6.0e23:t)`), possibly also in hexadecimal[4] (e.g `(0x1.8p-4:t)` that represents $\frac{3}{32}$). A very important design choice is that only the real numbers that are representable in the target type can be cast. Casting a literal constant that is not representable is *rejected* by Why3's typing engine, e.g. `(0.1:t)` will raise a typing error[5]. See below for implications of this choice.

Finally, each clone contains declarations of conversions with bitvectors of the appropriate sizes. We reuse here existing Why3 theories for bitvectors [18].

```
function to_bv mode t : BV32.t
function of_bv mode BV32.t : t
```

3.3 Interpreting FP Computations in Ada

The Ada standard does not impose the rounding mode used for FP computations, except for conversion between floats and integers where it is nearest, ties to away. To avoid non-determinism, the SPARK fragment of Ada imposes the rounding mode nearest, ties to even, for arithmetic operations. Moreover, in SPARK, overflow is forbidden, so special values for infinities and NaN are not allowed to appear. Thus encoding an FP operation from an Ada program amounts to generating the corresponding operation in Why3 with the proper rounding mode, and to insert a check for the absence of overflow. Roughly speaking, a piece of Ada code like:

```
procedure P (X : in out Float_32) is
begin
   X := X + 2.0;
end P;
```

is translated into WhyML intermediate code as follows:

```
let p (x : ref Float32.t) : unit
  requires { is_finite x }
= let tmp = Float32.add RNE x (2.0:Float32.t) in
  assert { is_finite tmp };
  x := tmp
```

The additional assertion will lead to a VC to check the absence of overflow.

All operations are handled in a similar way. Signed integers in Ada are interpreted as integers in Why3, and Ada unsigned integers are interpreted as bitvectors [18]. The conversions are translated using the functions `to_int`, `of_int`, `to_bv` and `of_bv` with the mode `RNA`.

[4] C99 notation for hexadecimal FP literals: `0x`$hh.hh$`p`dd, where h are hexadecimal digits and dd is in decimal, denotes number $hh.hh \times 2^{dd}$

[5] For that purpose, we had to implement in the typing engine a specific code that checks that a literal is representable and compute its mantissa and exponent. It is worth to note that implementing such a code is significantly easier than a code that would compute a correct rounding for any literals.

A remark about literals: in the translation from SPARK to Why3, the FP literals of the Ada source are first interpreted by the GNAT compiler. For example, if one uses the constant 0.1 in Ada for a 32-bit float, it is rounded to the closest representable value `0x1.99999Ap-4`. Thus the rounded value that may be used for such constants to produce a binary code is the very same value that is passed to Why3. Thus, not only are we sure that the generated Why3 literals are representable, but we are also sure that we use the same value as the executable.

3.4 Proving VCs Using Native Support for SMT-LIB FP

To attempt to discharge generated VCs using an SMT solver like Z3 that provides native support for floats, we simply have to map the symbols of our theory to the ones of SMT-LIB. The mapping is quite straightforward since most symbols have an SMT counterpart, with the same name, prefixed by '`fp.`'. An exception is the predicate `is_finite` which does not exists in SMT-LIB, we encode it as `(not (or (fp.isInfinite` x`) (fp.isNaN` x`)))`. Other notable exceptions are the conversions with integers, `of_int` and `to_int`, which do not exist in SMT-LIB. The two functions are left uninterpreted, and such conversions are dealt with the axiomatization below.

Regarding the literals, we use the mechanism that we implemented in Why3 together with our extension of float type declarations, that allows Why3 to print literals under the SMT-LIB bitvector form `(fp` s e m`)`, for example the constant `0x1.99999Ap-4` in 32 bits is written as

```
(fp #b0 #b01111011 #b10011001100110011001101)
```

3.5 Axiomatization for Provers without Native Support

Because the VCs generated from Ada programs mix FP computations with other data-types (integers, bitvectors, arrays...) but also quantified hypotheses, we cannot hope the solver to be complete. With an axiomatization, we can hope that a prover may discharge goals including elements outside the SMT-LIB FP theory (such as conversions with integers) and we can even call a prover without native support of FP arithmetic. We provide an axiomatization for the operators introduced in our theory, thus we rely on the generic handling of first-order axioms the prover may have. Our axiomatization is naturally incomplete: our intent is to provide the axioms that are useful in practice to discharge VCs. Notice that when using a solver with native FP support, the driver mechanism of Why3 removes all these axioms, so as to avoid the prover getting lost with the extra logic context.

Handling of literals. We need a mechanism to interpret our built-in Why3 support for literals for provers without native support. This is done using a Why3 transformation that replaces each literal of the proof task by an extra constant with an axiom that specifies its value. That is, if some FP literal $(v : t)$ appears in the proof task, it is replaced by a fresh constant l of type t, declared with the axiom (`t'isFinite` l) $\wedge$ (`t'real` $l = v$).

Overflow checks. We start by introducing a few useful constants derived from `eb` and `sb`:

```
constant pow2sb   : int = pow2 sb       (* 2^{sb} *)
constant emax     : int = pow2 (eb - 1)    (* 2^{eb-1} *)
constant max_int  : int = pow2 emax - pow2 (emax - sb)
constant max_real : real = FromInt.from_int max_int
```

The constants `max_real` and `max_int` both represent the exact value of the biggest finite float, as a real and an integer respectively. In order to speak about overflows and finiteness, we introduce the following predicates.

```
predicate in_range (x:real) = - max_real <= x <= max_real
predicate no_overflow (m:mode) (x:real) = in_range (round m x)
axiom is_finite: forall x:t. is_finite x → in_range (to_real x)
axiom Bounded_real_no_overflow :
  forall m:mode, x:real. in_range x → no_overflow m x
```

The predicate `in_range` specifies the range of floats. The predicate `no_overflow` composes `round` and `in_range` to check for overflows. We stress that the two axioms specify that to be finite implies that the projection is in the float range, which in turns implies that there is no overflow. However we do not specify that no overflows implies finiteness in order to force the provers to reason on reals as well as avoid circularity in their proof attempts.

Rounding and arithmetic. The rounding function, that is used to specify the FP operations, plays a central role in our axiomatization. This rounding function operates on real numbers, as defined in IEEE-754. It takes a rounding mode m, a real value x and returns the value of the FP number nearest to x up to m.

```
function round mode real : real
axiom Round_monotonic :
  forall m:mode, x y:real. x <= y → round m x <= round m y
axiom Round_idempotent :
  forall m1 m2:mode, x:real. round m1 (round m2 x) = round m2 x
axiom Round_to_real :
  forall m:mode, x:t. is_finite x → round m (to_real x) = to_real x
```

Those axioms are completed with axioms in the clones for 32 and 64 bits, giving quite precise bounds on the error made by rounding [31]. Here are the ones for 32 bits:

```
lemma round_bound_ne :
  forall x:real [round RNE x]. no_overflow RNE x →
    x - 0x1p-24 * Abs.abs(x) - 0x1p-150 <= round RNE x
      <= x + 0x1p-24 * Abs.abs(x) + 0x1p-150
lemma round_bound :
  forall m:mode, x:real [round m x]. no_overflow m x →
    x - 0x1p-23 * Abs.abs(x) - 0x1p-149 <= round m x
      <= x + 0x1p-23 * Abs.abs(x) + 0x1p-149
```

For 64 bits, the constants `0x1p-24`, `0x1p-150`, `0x1p-23`, `0x1p-149` are replaced by `0x1p-53`, `0x1p-1075`, `0x1p-52` and `0x1p-1074`.

Of course the axioms are not completely specifying the rounding function but only give bounds. A complete specification is not an objective. Indeed, giving a complete semantic with axioms would only lose the provers, the function `round` is too complex for that. Furthermore we don't want solvers to "reason" about the function `round` itself but rather up to it. Hence we only provide some properties to help provers move the predicate around. This is at the cost of losing precision in computations, in particular proof of equality and proofs dealing with precise ranges are hard, or even impossible.

The axiomatization of arithmetic operators derives from the rounding function. We only present the float addition, the other arithmetic operators are in the same line. It is axiomatized through the projection to reals. The main axiom `add_finite` specifies the non overflowing addition of two finite floats.

```
axiom add_finite: forall m:mode, x y:t [add m x y].
  is_finite x → is_finite y →
  no_overflow m (to_real x + to_real y) →
  is_finite (add m x y) ∧
  to_real (add m x y) = round m (to_real x + to_real y)
lemma add_finite_rev: forall m:mode, x y:t [add m x y].
  is_finite (add m x y) → is_finite x ∧ is_finite y
lemma add_finite_rev_n: forall m:mode, x y:t [add m x y].
  (m = RNE ∨ m = RNA) →  is_finite (add m x y) →
  no_overflow m (to_real x + to_real y) ∧
  to_real (add m x y) = round m (to_real x + to_real y)
```

The two lemmas specify what we can deduce from the finiteness of an addition, `add_finite_rev` for the general case and `add_finite_rev_n` for the rounding modes `RNE` and `RNA`. The two lemmas are important when the finiteness of an addition appears in the context of a VC without any other fact about how it was proven (e.g. it was proven in another VC, or provided as an hypothesis). As mentioned, a second axiom is provided in the theory to specify all other cases dealing with special values (overflows, addition with a `NaN`, etc.). All other arithmetic operators, namely subtraction, multiplication, division, negation as well as absolute value, square root and fused multiply-add are specified in the same way. See [19] for all the axioms.

Conversions with integers. Since these conversions are not supported by SMT-LIB, we handle them specially with a set of axioms that are useful to discharge goals coming from our Ada examples. Here is an excerpt of these axioms concerning the addition of FP numbers that come from integer conversions.

```
predicate in_int_range (i:int) = - max_int <= i <= max_int
predicate in_safe_int_range (i: int) = - pow2sb <= i <= pow2sb
axiom of_int_add_exact: forall m n, i j.
  in_int_range i → in_int_range j → in_safe_int_range (i + j) →
    eq (of_int m (i + j)) (add n (of_int m i) (of_int m j))
```

The predicate `in_int_range` specifies the range of all finite integers in the float format, note that all are not representable though. The predicate `in_safe_int_range`, building on `pow2sb`, specifies the range in which every integer is representable in the float format. The axiom then expresses the necessary conditions to deduce that some addition is indeed exact. We have similar axioms for other operations. See [19] for a more detailed description of this part of the axiomatization.

3.6 Consistency and Faithfulness

Our FP theory, with all its axioms, is proven conformant to the IEEE standard by realizing a model of it in Coq, using the existing library Flocq [7]. While part of the realization was simply to reuse results already proved in Flocq, we did provide significant proof efforts, in particular to deal with the relation between integers and floats which is absent from this library (and might end up contribute to it). The faithfulness of the axiomatization with regard to IEEE standard is then enforced by modeling the theory's operators with Flocq's corresponding IEEE operators.

For SMT solvers with native support, we need to ensure that the axiomatization is coherent with the SMT-LIB theory of floats. This is the case if the implementation of SMT-LIB FP in a given solver is itself consistent with the IEEE standard, which is supposed to be the case.

4 Experiments

The tables from Figures 2 and 4 summarize the proof results with provers from the current SPARK toolset: SMT solvers CVC4, Alt-Ergo, Z3 and static analyzer CodePeer. We add in these figures two provers: *AE_fpa*, the prototype of Alt-Ergo with FP support [12], and *COLIBRI*, a prover based on constraint solving techniques [28,11]. Dark cells correspond to unproved VCs with a given prover. Light gray and half gray cells correspond to proved VCs with a given prover, and the running time of the prover is given in seconds (round to nearest, away from zero).

4.1 Small Representative Examples

We start with 22 simple examples representative of the problems encountered with proof of industrial programs using FP arithmetic. Each example consists in a few lines of code with a final assertion. Although each assertion should be provable, none of the assertions were provable with the version of SPARK released in 2014, when we established this list. These examples show a variety of FP computations that occur in practice, combining linear and non-linear arithmetic, conversions between integers and FP, conversions between single and double precision FP. The first 11 examples correspond to reduced examples from

Example	CVC4	Alt-Ergo	Z3	CodePeer	AE_fpa	COLIBRI
Range_Add			0	1	0	0
Range_Mult			0			0
Range_Add_Mult						0
Int_To_Float_Simple	25	0	0	1	0	
Float_To_Long_Float	0	0	0	1	0	0
Incr_By_Const			12			
Polynomial			2			0
Float_Different			0	1		0
Float_Greater			0	1		0
Diffs			1			
Half_Bound		7	0	1	4	0

Example	CVC4	Alt-Ergo	Z3	CodePeer	AE_fpa	COLIBRI
User_Rule_2			0	1	1	0
User_Rule_3			2		0	0
User_Rule_4			2		0	0
User_Rule_6			2	1	2	0
User_Rule_7					1	0
User_Rule_9						0
User_Rule_10						0
User_Rule_11			1	1	2	0
User_Rule_13			20			0
User_Rule_14			1			0
User_Rule_15						0

Fig. 2. Proof times in seconds for the reduced examples (timeout = 30 seconds)

actual programs. The last 11 examples correspond to so-called *user rules*, i.e. axioms that were manually added to the proof context in the SPARK technology prior to SPARK 2014. Out of 22 examples, 20 directly come from industrial needs. See [19] for the complete Ada source code for these small examples. Table 2 summarizes the proof results. It can be noted that all examples are now proved by the combination of provers.

4.2 A Case Study

We now present a simple case study representative of production code from an embedded safety-critical software, on which we have applied the combination of techniques presented previously. The complete Ada source code for this case study is available online at http://toccata.lri.fr/gallery/trajectory_computation.en.html. This program computes the speed of a device submitted to gravitational acceleration and drag from the atmosphere around it. The formula to compute the new speed $S(N+1)$ from the speed $S(N)$ at the previous step, after a given increment of time is:

$$S(N+1) = S(N) + \delta \tag{1}$$

where $\delta = \mathit{drag} + \mathit{factor} \times G \times \mathit{framelength}$, where *factor* is a value between -1 and 1 reflecting the importance of Archimedes' principle on the system, G is the gravitational acceleration constant, and *framelength* is a constant that defines the time in seconds between two steps in the computations.

Because of the types and the values of constants involved, both CodePeer and Z3 can prove that there is no possible overflow. To go beyond absence of overflows, we aim at proving safe bounds for the speed computed by the program at each step, based on the extreme values allowed for *drag* and the initial speed.

```
Bound : constant Float64 :=
  Drag_T'Last + Float64'Ceiling (G * Frame_Length)
with Ghost;

function Low_Bound (N : Frame) return Float64 is
  (Float64 (N) * (- Bound))
with Ghost;

function High_Bound (N : Frame) return Float64 is
  (Float64 (N) * Bound)
with Ghost;

function Invariant (N : Frame; Speed : Float64) return
    Boolean is
  (Speed in Low_Bound (N) .. High_Bound (N))
with Ghost;

procedure Compute_Speed (N         : Frame;
                         Factor    : Ratio_T;
                         Old_Speed : Float64;
                         New_Speed : out Float64)
with Global => null,
     Pre    => N < Frame'Last
               and then Invariant (N, Old_Speed),
     Post   => Invariant (N + 1, New_Speed);
```

Fig. 3. Formal Specification of `Compute_Speed`

Reasoning in real numbers, we'd like to state that (showing only the upper bound here, the lower bound is similar):

$$\delta \leq maxdrag + factor \times G \times framelength$$

and, starting from an initial speed $S(0)$ of value zero, by summation over N steps, that

$$S(N) \leq N \times (maxdrag + factor \times G \times framelength)$$

Naturally, these bounds in real numbers do not necessarily hold for the FP computations in the program. But they do hold when considering the ceiling value of the FP computations above, using $\oplus$ for FP addition and $\otimes$ for FP multiplication:

$$\delta \leq \lceil maxdrag \oplus factor \otimes G \otimes framelength \rceil \tag{2}$$

$$S(N) \leq N \otimes \lceil maxdrag \oplus factor \otimes G \otimes framelength \rceil \tag{3}$$

Indeed, given the magnitude of the integers involved (both operands and results of arithmetic operations), they are in a safe range where all integers are

VC	CVC4	Alt-Ergo	Z3	CodePeer	AE_fpa	COLIBRI
`Delta_Speed in -Bound .. Bound`				1	3	0
`In_Bounds (High_Bound(N))`				1	1	
`In_Bounds (Low_Bound(N))`			0	1	2	
`Float64(N_Bv) * Bound + Bound`			42			0
`= (Float64(N_Bv) + 1.0) * Bound`						
`Float64(N) * Bound + Bound`		44		1	25	0
`= (Float64(N) + 1.0) * Bound`						
`Float64(N) * (-Bound) - Bound`				1		0
`= (Float64(N) + 1.0) * (-Bound)`						
`T(1) = 1.0`	0	0		1	0	0
`Float64(N) + 1.0 = Float64(N + 1)`	0	1			1	0
`New_Speed >= Float64 (N) * (-Bound) Bound`	27					0
`New_Speed >= Float64 (N + 1) * (-Bound)`			1			0
`New_Speed <= Float64 (N) * Bound + Bound`	26					0
`New_Speed <= Float64 (N + 1) * Bound`			1			0
Postcondition	20	0			1	

Fig. 4. Proof times in seconds for the case study (timeout = 60 seconds). Underlined cells correspond to provers actually used by GNATprove when using switches `--codepeer=on --level=2`

represented exactly as FP numbers, and arithmetic operations like additions and multiplications are thus exact as well on such integers. This is expressed in axioms such as `of_int_add_exact` of our axiomatization relating floats and integers, as presented in Section 3.5. Hence, for the integer value Q of ceiling in the equation above, we have that $(N \otimes Q) \oplus Q = (N \oplus 1) \otimes Q$ which allows to prove the bounds on $S(N)$ by induction on N, from equations (1), (2) and (3). We follow this strategy in proving a corresponding contract on procedure `Compute_Speed` which computes the value of $S(N)$, shown in Figure 3. The ghost function `Invariant` expresses a bound on the N^{th} term of the series $S(N)$.

The automatic proof of `Compute_Speed` in GNATprove requires the collaboration of static analyzer CodePeer and SMT solvers Alt-Ergo, CVC4 and Z3. The way that we make this collaboration work is that we state intermediate assertions that are proved by either CodePeer or an SMT solver, and which are used by all to prove subsequent properties. Table 4 gives the list of intermediate assertions that we wrote to fully prove the case study, and summarizes the proof results. It can be noted that all VCs in the case study are proved by the combination of provers.

We start by bounding all quantities involved using a ghost function `In_Bounds`. CodePeer is used to prove automatically three assertions bounding the values of `Delta_Speed`, `High_Bound(N)` and `Low_Bound(N)`. Then, Z3 is used to prove the distribution of addition over multiplication that is required

to prove `Invariant(N+1)` from `Invariant(N)`, using a value `N_BV` which is the conversion of `N` into a modular type. As modular types are converted into bitvectors for Z3, the assertion mixing integers and floats can be interpreted fully in bitvectors by Z3, which allows to prove it. Then, CodePeer is used to prove equivalent assertions on signed integers. CodePeer is also used to prove that the conversion of integer 1 into float is 1.0, using an expression function `T` returning its input, which prevents the analyzer frontend from simplifying this assertion to `True`. Then, CVC4 is used to prove that adding 1 to N can be done with the same result in integers and in floats, using the previously proved assertion and the axiom `of_int_add_exact` presented in Section 3.5. Finally, a combination of CVC4 and Z3 is used to prove bounds on the value of `New_Speed`. With these assertions, the postcondition of `Compute_Speed` is proved by CVC4.

5 Conclusions and Perspectives

Our approach for automated verification of floating-point programs relies on a generic theory, written in Why3's specification language, to model FP arithmetic. This theory is faithful to the IEEE standard. Its genericity allows to map it both to the Flocq library of Coq and to the FP theory of SMT-LIB. This theory is used to encode FP computations in the VC generation process performed by Why3. The resulting VCs can be dispatched either to the CodePeer analyzer that performs interval analysis, or to SMT solvers, with or without a native support for FP theory. The versatility of the different targets for discharging VCs permit a high degree of automation of the verification process.

Related Work. Since the mid 1990s, FP arithmetic has been formalized in interactive deductive verification systems: in PVS [9], in ACL2 [34], in HOL-light [22], and in Coq [15]. These formalizations allowed one to represent abstraction of hardware components or algorithms, and prove soundness properties. Representative case studies were the formal verification of FP multiplication, division and square root instructions of the AMD-K7 microprocessor in ACL2 [34], and the development of certified algorithms for computing elementary functions in HOL-light [22,21]. See also [23] for a survey of these approaches.

In 2007, Boldo and Filliâtre proposed an approach for proving properties related to FP computations in concrete C, using the Caduceus tool and Coq for the proofs [5]. The support for FP in Caduceus was somehow ported to the Frama-C environment [13] and its Jessie plug-in, aiming at using automated solvers instead of Coq, for a higher degree of automation [1]. Several case studies using Frama-C/Jessie, with various degree of complexity were designed by different authors [6,4,20,27]. In these various case studies, proofs using Coq or PVS were still needed to discharge the most complex VCs. Yet, a significant improvement in the degree of automation was obtain thanks to the use of the automated solver Gappa dedicated to reasoning on FP rounding [14].

Regarding the use of abstract interpretation to verifying FP programs, this indeed obtained very good successes in industrial contexts. In 2004, Miné used

relational abstract domains to detect FP run-time errors [30], an approach that was implemented in the Astrée tool and successfully applied to the verification of absence of run-time errors in the control-command software of the Airbus A380. Another tool based on abstract interpretation is Fluctuat [16], which is not limited to the verification of absence of runtime errors, but is also able to compare between executions of the same code in finite precision and in infinite precision, giving some bounds on the difference between the two.

Previous support for floats in GNATprove translated every FP value in SPARK into a real value in Why3 and relied on the support for real arithmetic in provers, plus explicit use of rounding after each arithmetic operation. A limitation (documented in previous versions of SPARK) was that the mapping from FP values to the real line, even when excluding infinities and NaN, is not injective: both FP values $+0$ and -0 are translated into the real number zero. Thus, the translation was not sound when programs in the input language may distinguish values -0 and $+0$, as the representation in real numbers cannot distinguish them anymore. Contrary to the axiomatization presented in Section 3, the previous axiomatization of rounding was not realized. Consistency and conformance of the set of axioms was ensured by review only.

In our own approach, we combine different techniques from abstract interpretation (interval analysis) and theorem proving (recent support of FP in SMT solvers), to achieve verification not only of runtime errors but also functional properties given by the user, with a high degree of automation. Our approach indeed follows the same path we followed for improving the support for bit-level computations [18], where in that case we tried to exploit native support for bitvectors in SMT solvers.

Future Work. The new combined approach we designed is successful on the typical examples with FP computations coming from current industrial use of SPARK. Yet, we noticed that this new technique is not as good as the former one used in Frama-C and Why3 [6] for proving very advanced functional behaviors of programs, relating the concrete computations with some purely mathematical computations on real numbers [6,4,27]. A short term perspective is to better unify the two approaches. Notice that the authors of the CVC4 SMT solver are currently working on a native support for FP arithmetic, it will be worth to experiment our approach with this new prover when it is available. As shown by our case study, handling conversion between integers and FP numbers remains quite challenging, a better support by back-end provers is desirable.

These future work, together with further improvements in the prototype back-end experimental solvers *COLIBRI* and *AE_fpa* we mentioned quickly in the experimental results of Section 4.2, are central in the on-going project SOPRANO.

Acknowledgements. We would like to thank Guillaume Melquiond for his help with the design of the new Why3 theory for FP arithmetic and with the realization in Coq using Flocq. We also thank Florian Schanda for providing the

case study used in this article, Mohamed Iguernlala and Bruno Marre for fruitful exchanges on the use of *AE_fpa* and *COLIBRI* as well as the anonymous reviewers for their comments.

References

1. Ayad, A., Marché, C.: Multi-prover verification of floating-point programs. In: Int. Joint Conference on Automated Reasoning. LNAI, vol. 6173, pp. 127–141 (2010)
2. Baird, S., Charlet, A., Moy, Y., Taft, T.S.: CodePeer – beyond bug-finding with static analysis. In: Static Analysis of Software: the Abstract Interpretation. Wiley (2013)
3. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Let's verify this with Why3. Int. Journal on Software Tools for Technology Transfer (STTT) 17(6), 709–727 (2015), see also http://toccata.lri.fr/gallery/fm2012comp.en.html
4. Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P.: Wave equation numerical resolution: a comprehensive mechanized proof of a C program. Journal of Automated Reasoning 50(4), 423–456 (2013)
5. Boldo, S., Filliâtre, J.C.: Formal Verification of Floating-Point Programs. In: IEEE Int. Symp. on Computer Arithmetic. pp. 187–194 (2007)
6. Boldo, S., Marché, C.: Formal verification of numerical programs: from C annotated programs to mechanical proofs. Mathematics in Computer Science 5, 377–393 (2011)
7. Boldo, S., Melquiond, G.: Flocq: A unified library for proving floating-point algorithms in Coq. In: 20th IEEE Symposium on Computer Arithmetic. pp. 243–252 (2011)
8. Brain, M., D'silva, V., Griggio, A., Haller, L., Kroening, D.: Deciding floating-point logic with abstract conflict driven clause learning. Formal Methods in System Design 45(2), 213–245 (2014)
9. Carreño, V., Miner, P.S.: Specification of the IEEE-854 floating-point standard in HOL and PVS. In: Int. Workshop on Higher-Order Logic Theorem Proving and Its Applications (1995)
10. Chapman, R., Schanda, F.: Are we there yet? 20 years of industrial theorem proving with SPARK. In: Interactive Theorem Proving. LNCS, vol. 8558, pp. 17–26 (2014)
11. Chihani, Z., Marre, B., Bobot, F., Bardin, S.: Sharpening constraint programming approaches for bit-vector theory. In: CPAIOR (2017)
12. Conchon, S., Iguernlala, M., Ji, K., Melquiond, G., Fumex, C.: A three-tier strategy for reasoning about floating-point numbers in SMT. In: Computer Aided Verification (2017)
13. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. In: Int. Conf. on Software Engineering and Formal Methods. pp. 233–247. No. 7504 in LNCS (2012)
14. Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. Transactions on Mathematical Software 37(1), 1–20 (2010)
15. Daumas, M., Rideau, L., Théry, L.: A generic library of floating-point numbers and its application to exact computing. In: Theorem Proving in Higher Order Logics. pp. 169–184 (2001)
16. 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 (2009)

17. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Computer Aided Verification. LNCS, vol. 4590, pp. 173–177 (2007)
18. Fumex, C., Dross, C., Gerlach, J., Marché, C.: Specification and proof of high-level functional properties of bit-level programs. In: 8th NASA Formal Methods Symposium. LNCS, vol. 9690 (2016)
19. Fumex, C., Marché, C., Moy, Y.: Automated verification of floating-point computations in Ada programs. Research Report RR-9060, Inria (2017)
20. Goodloe, A., Muñoz, C.A., Kirchner, F., Correnson, L.: Verification of numerical programs: From real numbers to floating point numbers. In: NASA Formal Methods. LNCS, vol. 7871, pp. 441–446 (2013)
21. Harrison, J.: Floating point verification in HOL Light: The exponential function. Formal Methods in System Design 16(3), 271–305 (2000)
22. Harrison, J.: Formal verification of floating point trigonometric functions. In: Formal Methods in Computer-Aided Design. LNCS, vol. 1954, pp. 217–233 (2000)
23. Harrison, J.: Floating-point verification. In: Int. Symp. of Formal Methods Europe. pp. 529–532 (2005)
24. IEEE standard for floating-point arithmetic (2008), https://dx.doi.org/10.1109/IEEESTD.2008.4610935
25. Kosmatov, N., Marché, C., Moy, Y., Signoles, J.: Static versus dynamic verification in Why3, Frama-C and SPARK 2014. In: Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation. pp. 461–478. LNCS (2016)
26. Leino, K.R.M., Moskal, M.: Usable auto-active verification. In: Usable Verification Workshop (2010)
27. Marché, C.: Verification of the functional behavior of a floating-point program: an industrial case study. Science of Computer Programming 96(3), 279–296 (2014)
28. Marre, B., Michel, C.: Improving the floating point addition and subtraction constraints. In: Principles and Practice of Constraint Programming. pp. 360–367 (2010)
29. McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press (2015)
30. Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In: European Symposium on Programming. LNCS, vol. 2986, pp. 3–17 (2004)
31. Monniaux, D.: The pitfalls of verifying floating-point computations. ACM Transactions on Programming Languages and Systems 30(3), 12 (2008)
32. de Moura, L., Bjørner, N.: Z3, an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 4963, pp. 337–340 (2008)
33. Rümmer, P., Wahl, T.: An SMT-LIB theory of binary floating-point arithmetic. In: Int. Workshop on Satisfiability Modulo Theories (2010)
34. 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, 148–200 (1998)