Automated Verification of Floating-Point Computations in Ada Programs

Clément Fumex, Claude Marché, Yannick Moy

▶ **To cite this version:**

Clément Fumex, Claude Marché, Yannick Moy. Automated Verification of Floating-Point Computations in Ada Programs. [Research Report] RR-9060, Inria Saclay Ile de France. 2017, pp.53. hal-01511183

HAL Id: hal-01511183

https://inria.hal.science/hal-01511183

Submitted on 20 Apr 2017

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

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

Distributed under a Creative Commons Attribution - ShareAlike 4.0 International License

Automated Verification of Floating-Point Computations in Ada Programs

Clément Fumex, Claude Marché, Yannick Moy

RESEARCH REPORT
N° 9060
April 2017
Project-Team Toccata

ISSN 0249-6399 ISRN INRIA/RR--9060--FR+ENG

Automated Verification of Floating-Point Computations in Ada Programs

Clément Fumex*, Claude Marché*†, Yannick Moy‡

Project-Team Toccata

Research Report n° 9060 — April 2017 — 50 pages

Abstract: In critical software systems like the ones related to transport and defense, it is common to perform numerical computations implemented using floating-point arithmetic. Safety conditions for such systems typically require strong guarantees on the functional behavior of the performed computations. Automatically verifying that these guarantees are fulfilled is thus desirable.

Deductive program verification is a promising approach for verifying that a given code fulfills a functional specification, with a very high level of confidence. Yet, formally proving correct a program performing floating-point computations remains a challenge, because floating-point arithmetic is not easily handled by automated theorem provers.

We address this challenge by combining multiple techniques to separately prove parts of the desired properties. On the one hand, abstract interpretation computes numerical bounds for expressions that appear either in the original program, or in the ghost code added to instrument the program. On the other hand, we generate verification conditions for different automated provers, relying on different strategies for representing floating-point computations. Among these strategies, we try to exploit the native support for floating-point arithmetic recently added in the SMT-LIB standard.

Our approach is partly implemented in the Why3 environment for deductive program verification, and partly implemented in its front-end environment SPARK for the development of safety-critical Ada programs. We report on several examples and case studies used to validate our approach experimentally.

Key-words: Formal Specification, Deductive Verification, Formal Proof, Program Verifiers Why3 and SPARK, Floating-point computations

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

* Inria, Université Paris-Saclay, F-91120 Palaiseau
† LRI (CNRS & Univ. Paris-Sud), Université Paris-Saclay, F-91405 Orsay
‡ AdaCore, F-75009 Paris

**RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE**

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Vérification automatique de programmes Ada effectuant des calculs en virgule flottante

Résumé : Dans les logiciels critiques comme ceux liés aux transports et à la défense, il est courant d'effectuer des calculs numériques implémentés à l'aide de l'arithmétique à virgule flottante. Les exigences de sûreté de ces systèmes requièrent généralement des garanties fortes sur le comportement fonctionnel des calculs ainsi exécutés. Vérifier automatiquement que ces garanties sont remplies est donc souhaitable.

La vérification déductive des programmes est une approche prometteuse pour vérifier formellement qu'un code donné satisfait une spécification fonctionnelle, avec un haut niveau de confiance. Néanmoins, prouver formellement qu'un programme qui effectue des calculs en virgule flottante est correct reste un défi, car l'arithmétique à virgule flottante n'est pas facilement traitée par les prouveurs automatiques.

Nous abordons ce défi en combinant plusieurs techniques pour prouver séparément chaque partie des propriétés souhaitées. D'une part, l'interprétation abstraite calcule les bornes numériques pour les expressions qui apparaissent dans le programme d'origine, ou dans le code fantôme ajouté pour instrumenter le code. D'autre part, nous générons des obligations de preuve pour plusieurs prouveurs automatiques différents, s'appuyant sur différentes représentations des calculs en virgule flottante. En particulier, nous exploitons le support natif de l'arithmétique en virgule flottante récemment apparue dans le standard SMT-LIB.

Notre approche est mise en œuvre en partie dans la boîte à outils générique Why3 pour la vérification déductive, et en partie dans l'environnement SPARK pour le développement de programmes Ada critiques. Nous présentons des exemples et études de cas qui ont été utilisées pour valider notre approche expérimentalement.

Mots-clés : Spécification formelle, preuve de programmes, environnements de preuve Why3 et SPARK, calculs en virgule flottante

Contents

1 Introduction **4**

2 Quick Introduction to SPARK and Why3 **5**
2.1 Why3 in Brief 5
2.2 SPARK in Brief 6
2.3 Float Interval Analysis in CodePeer 7
2.4 Auto-active Verification 7

3 VC Generation for Floating-Point Computations **8**
3.1 Preliminary: The IEEE-754 Standard for FP Arithmetic 8
3.2 Signature for a generic theory of IEEE FP arithmetic 9
3.3 Theory Instances and FP Literals 10
3.4 Interpreting FP Computations in Ada 11
3.5 Proving VCs Using Native Support for SMT-LIB FP 11
3.6 Axiomatization for Provers Without Native Support 13
3.6.1 Handling of Literals 13
3.6.2 Extra Constants and Overflows 13
3.6.3 The Rounding Function 13
3.6.4 Float Equality, Comparisons 14
3.6.5 Arithmetic 15
3.6.6 Floats and Integers 15
3.7 Consistency and Faithfulness 18

4 Experiments **18**
4.1 Representative Small Examples 18
4.2 A Case Study 19

5 Conclusions **22**
5.1 Related Work 22
5.2 Future Work 24

A Complete Why3 Theory of FP Arithmetic **28**

B Ada Code Listings **44**

1 Introduction

Numerical programs are used in many critical software systems, for example to compute trajectories, to control movements, to detect objects, etc. 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 typically require strong guarantees on the functional behavior of the performed computations. Automatically verifying that these guarantees are fulfilled is thus 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.

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, and required a lot of expertise [8]. 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 standard for FP arithmetic [30] was added in the SMT-LIB standard in 2010 [38]. This theory is now supported by at least the solvers Z3 [18] and MathSAT5 [11].

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, 8], 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 [16], 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).

We report the results of experiments, which show that the new combination of techniques allow us to 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.

```
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;
```

The reason is that in our previous approach, the rounding error of the operation `X + 2.0` was over-approximated.

In this document we report on the design of our approach to combine different solvers, and how it affects the verification of FP programs. We first give a quick introduction to auto-active verification with the environments SPARK and Why3 in Section 2. In Section 3, we described how we modified the Verification Condition (VC for short) 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. In Section 4, we report

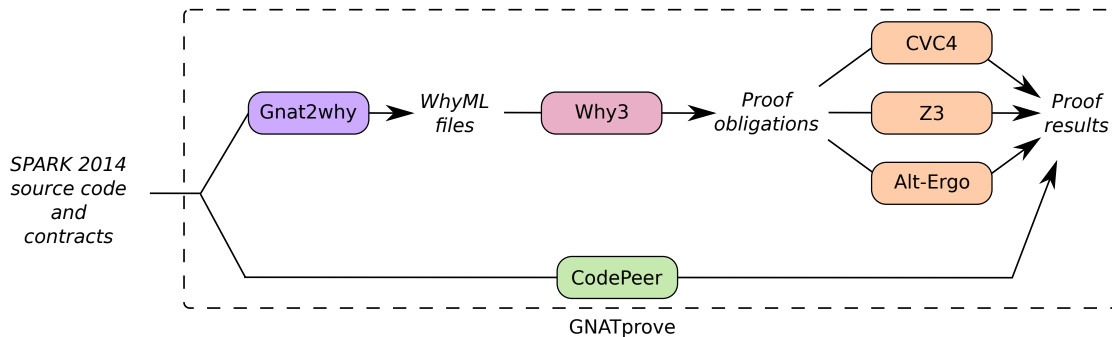

Figure 1: Deductive verification in SPARK 2014

on 22 reduced examples extracted from industrial programs which are proved with our combination of techniques, none of which could be proved previously. We also present in greater details 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. Section 5 draws conclusions and discusses some related work and future work.

2 Quick Introduction to SPARK and Why3

SPARK is an environment for the verification of Ada programs used in critical software development. As displayed in Figure 1, a given fragment of Ada code, annotated with formal specification, is compiled into a program in the intermediate language WhyML. WhyML is the input language of the Why3 tool which computes the verification conditions that ensure the conformance of the code with respect to the specifications.

2.1 Why3 in Brief

Why3 is an environment for deductive program verification, providing a rich language for specification and programming, called WhyML. It relies on external provers, both automated and interactive, in order to discharge the auxiliary lemmas and verification conditions. WhyML is used as an intermediate language for verification of C, Java or Ada programs [22, 31], and is also intended to be comfortable as a primary programming language.

The specification component of WhyML [4], used to write program annotations and background theories, is an extension of first-order logic. It features ML-style polymorphic types (prenex polymorphism), algebraic datatypes, inductive and co-inductive predicates, recursive definitions over algebraic types. Constructions like pattern matching, let-binding and conditionals can be used directly inside formulas and terms. Why3 comes with a rich standard library providing general-purpose theories useful for specifying programs [5]. This naturally includes integer and real arithmetic.

The user can formalize its own additional theories. A new type, a new function, or a new predicate can be either defined or declared abstract and axiomatized. For example, one can formalize a theory of sets of integers as follows

```
theory IntSet
  use import int.Int           (* theory requires integer arithmetic for Why3's stdlib *)
  type t                       (* abstract type of sets of integers *)
```

```
  predicate mem int t              (* (mem x s) means ''x in t'' *)
  function empty : t               (* empty set *)
  axiom mem_empty: forall x:int. not (mem x empty)
  function single int : t          (* (single x) is the singleton set {x} *)
  axiom mem_single: forall x y:int. mem x (single y) ↔ x=y
  function union t t : t           (* union of two sets *)
  axiom mem_union:
    forall x:int, s1 s2:t. mem x (union s1 s2) ↔ (mem x s1) ∨ (mem x s2)
  function incr_all t : t          (* increment by 1 all elements in a set *)
  axiom incr_spec : forall x:int, s:t. mem x (incr_all s) ↔ mem (x-1) s
end
```

The specification part of the language can serve as a common format for theorem proving problems, suitable for multiple provers. The Why3 tool generates proof obligations from lemmas and goals, then dispatches them to multiple provers, including SMT solvers Alt-Ergo, CVC4, Z3; TPTP first-order provers E, SPASS, Vampire; interactive theorem provers Coq, Isabelle and PVS. For example, one can state some lemma on sets of integers:

```
theory Test
  use IntSet
  lemma toy_test : mem 42 (incr_all (union (single 17) (single 41)))
end
```

Such a lemma is easily proved valid by SMT solvers, since they are able to handle formulas mixing integer arithmetic, equality and uninterpreted symbols axiomatized by first-order axioms.

Why3 provides a mechanism called *realization* that allow a user to construct a model for her axiomatizations, using a proof assistant. This feature can be used to guarantee that an axiomatizations is consistent, or even more that is a faithful abstraction of an existing model. We will used this feature in Section 3.7 to ensures that our own axiomatization of FP arithmetic is faithful to IEEE standard [30].

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 proof obligation. Other transformations can also be imposed by the user in order to simplify the proof search.

The programming part of WhyML is a dialect of ML with a number of restrictions to make automated proving viable. WhyML function definitions are annotated with pre- and postconditions both for normal and exceptional termination, and loops are also annotated with invariants. In order to ensure termination, recursive definitions and while-loops can be supplied with variants, *i.e.* values decreasing at each iteration according to a well-founded order. Statically-checked assertions can also be inserted at arbitrary points in the program.

The Why3 tool generates proof obligations, called verification conditions, from those annotations using a standard weakest-precondition procedure. Also, most of the elements defined in the specification part: types, functions, and predicates, can be used inside a WhyML program. We refer to Filliâtre and Paskevich [23] for more details on the programming part of WhyML. Why3's web site[1] also provides a extensive tutorial and a large collection of examples. The WhyML source code in this paper is mostly self-explanatory for those familiar with functional programming.

2.2 SPARK in Brief

SPARK, co-developed by Altran and AdaCore, is a subset of Ada targeted at formal verification [13, 35]. Its restrictions ensure that the behavior of a SPARK program is unambiguously defined (unlike Ada). It excludes constructions that cannot easily be verified by automatic tools: pointers, side effects in expressions, aliasing, goto statements, controlled types (e.g. types with finalization) and exception

[1] `http://why3.lri.fr`

handling. The SPARK language and toolset for static verification has been applied for many years in on-board aircraft systems, control systems, cryptographic systems, and rail systems.

SPARK is designed so that both the flow analysis – checking that there is no access to uninitialized variables and that global variables and subprogram parameters are accessed appropriately – and the proof of program – checking the absence of run-time errors and the conformance to the contract – can be checked. It provides dedicated features that are not part of Ada. In particular, contracts can also contain information about data dependencies, information flows, state abstraction, and data and behavior refinement that can be checked by the GNATprove tool. Essential constructs for formal verification such as loop variants and invariants have also been introduced.

As described in Figure 1, to formally prove a SPARK program, 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.

2.3 Float Interval Analysis in CodePeer

CodePeer [3] is a tool developed at AdaCore to detect errors in Ada programs. It performs a modular abstract interpretation, by analyzing first leaf functions in the call-graph and moving up the call-graph, possibly iterating on recursion. CodePeer generates a contract summarizing the effects of a function to analyze its calls, generating both necessary preconditions (to avoid errors) and necessary postconditions (which are consequences of preconditions and function implementation) on every function in the program.

CodePeer divides the set of variables in a function into multiple objects, where a variable may be split into more than one object, and objects that may refer to the same memory locations are identified as possible aliases. Then, CodePeer puts a function in Static Single Assignment (SSA) form, assigning a unique name for every program location where the value of an object may be modified. SSA variables are also created for arithmetic operations on other variables. Finally, CodePeer propagates possible values of SSA variables along the control-flow graph of the function until reaching convergence. At this point, it has computed at every program point possible values for every SSA variable. These possible values are sets of integer or float intervals, depending on the type of the SSA variable.

Since version SPARK 17, CodePeer has been integrated inside GNATprove as the first prover, before any SMT solver is called. In order to keep the modularity of GNATprove analysis, this integration forces a special mode for CodePeer analysis, where no precondition is generated. Only preconditions from users are taken into account in this special mode. One of the goals of this integration is to benefit from the capabilities of abstract interpretation techniques to compute precise bounds on FP computations, that SMT solvers cannot decide easily. CodePeer computes precise bounds on FP computations, by applying the same operations on bounds of float intervals. It always performs round to minus infinity on the low bound and round to plus infinity on the high bound of each interval, in order to safely approximate every possible rounding mode on the machine.

When a given check (like an assertion) is proved to hold by the possible values computed by CodePeer, the tool registers this fact in a file that is subsequently read by GNATprove.

2.4 Auto-active Verification

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*. 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) [32].

In both Why3 and GNATprove, auto-active verification consists in a set of specification features at the level of the source language, and a set of tool capabilities to interact with users at the level of the source code. The specification features consist in constructs to specify function contracts (preconditions and

postconditions) and data invariants, as well as specialized forms of assertions (loop invariants and loop variants, assumptions and assertions). Both Why3 and GNATprove also support ghost code[31], a feature to instrument code for verification. Ghost functions are also called lemmas when their main purpose is to support the proof of a property that is later used at the point where the function is called. See [31] for a comparison of how ghost code differs between Why3 and GNATprove. Various tool capabilities facilitate user interaction at source level: fast running time that exploits multiprocessor architectures and minimizes rework between runs, the ability to trade running time for more verification power, feedback from the toolset when verification is unsuccessful (counterexamples in particular).

Auto-active verification in the above Why3 and GNATprove has been used to fully verify algorithms, libraries and even full applications. See Why3's web page (`http://why3.lri.fr` and gallery (`http://toccata.lri.fr/gallery/why3.en.html`) for collections of examples in Why3. Examples in SPARK include allocators [20] and red-black trees [21].

3 VC Generation for Floating-Point Computations

We now describe what we designed for the support for floating point types in Why3 and then how we use this support to enhance SPARK's already existing support for floating point types.

In Section 3.1, we review the main parts of the IEEE standard for FP computations. In Section 3.2 we present the signature, in other words the user interface, of our Why3 formalization of that standard. That signature is generic, parameterized by the size of FP numbers. In Section 3.3 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 3.4 we show how the FP computations in Ada programs are translated by SPARK into Why3 intermediate code based on our formalization, thus producing verification conditions involving the symbols of our signature. Section 3.5 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 3.6 presents an axiomatization for our Why3 theory of FP, to be used by provers that do not support FP natively. Finally Section 3.7 explains how we ensure that our theory and axiomatization is conformant to the IEEE standard.

3.1 Preliminary: The IEEE-754 Standard for FP Arithmetic

The IEEE-754 standard [30] defines an expected behavior of FP computations. It describes binary and decimal formats to represent FP numbers, and specifies the elementary operations and the comparison operators on FP numbers. It explains when FP exceptions occur, and introduces special values to represent signed infinities and "Not-a-Number" for the result of undefined operations. We summarize here the essential parts we need, we refer to Golberg [25] for more details.

We target only binary formats of the IEEE standard. Generally speaking, in any of these 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 denote a finite subset of real numbers, called the set of *representable* numbers in that format. In a given format, any sequence of bits is splitted in three parts:

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 mode *Up* (resp. *Down*), it is x_2 (resp. x_1). With *ToZero*, it is x_1 if $x > 0$ and x_2 if $x < 0$. With *NearestAway* and *NearestEven*, 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 first case it is x_2 if $x > 0$ and x_1 if $x < 0$; whereas in the second case 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. $(+\infty) + (+\infty) = (+\infty)$, $(+\infty) + (-\infty) = \mathrm{NaN}$, $1/(-\infty) = -0$, $\pm 0/\pm 0 = \mathrm{NaN}$, etc.

3.2 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 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* of the exponent and of the significand. In SMT, 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 two axioms are there to impose constraints on `eb` and `sb`: they both must be greater than 1. The abstract type t denotes the sort of FP numbers for the given `eb` and `sb`.

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

```
  type mode =
  | RNE (* NearestEven *)
  | RNA (* NearestAway *)
```

```
| RTP (* Up *)
| RTN (* Down *)
| RTZ (* ToZero *)

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 *)
function fma mode t t t : t  (* x * y + z *)
function sqrt mode t   : t   (* square root *)
```

It continues with the comparison operators and a few extra operations.

```
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 = *)

function min t t : t
function max t t : t
function abs t   : t   (* absolute value *)
function neg t   : t   (* opposite *)

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

Notice the particular case of the `to_real` conversion function which returns a true real number. Its result is unspecified if the argument is infinite or NaN.

Finally, it includes predicates for classification of numbers

```
predicate is_normal    t
predicate is_subnormal t
predicate is_zero      t
predicate is_infinite  t
predicate is_nan       t
predicate is_positive  t
predicate is_negative  t
predicate is_finite    t

predicate is_plus_infinity  (x:t) = is_infinite x ∧ is_positive x
predicate is_minus_infinity (x:t) = is_infinite x ∧ is_negative x
predicate is_plus_zero      (x:t) = is_zero x ∧ is_positive x
predicate is_minus_zero     (x:t) = is_zero x ∧ is_negative x
predicate is_not_nan        (x:t) = is_finite x ∨ is_infinite x
```

3.3 Theory Instances 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 instances (*clones* in Why3) of our theory for the binary formats of standard sizes 32 and 64 bits. For that purpose of literal constants, we implemented a new feature in Why3 itself: the possibility to declare FP types.

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 form of declaration is the line `type t = < float 8 24 >` 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 the new built-in Why3's declaration `type t = < float 8 24 >` 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[2] (e.g `(1.5p-4:t)` that represents 1.5×2^{-4}). A very important point to notice is that the Why3 typing engine *rejects* FP literals that are not representable, e.g. `(0.1:t)` will raise a typing error.

3.4 Interpreting FP Computations in Ada

The Ada standard does not impose the rounding mode used for FP computations. The SPARK fragment of Ada imposes the rounding mode *NearestEven* for arithmetic operations, except for conversion between floats and integers where it is *NearestAway*. Moreover, in SPARK, special values for infinities and NaN are not allowed to appear. Thus encoding an FP operation from an Ada program amounts to generate the corresponding operation 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:

```
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.

Final notice 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 we are sure that the generated Why3 literals are representable, but we are also sure that we use the same value as the executable.

3.5 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 summarized

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

Our theory	SMT-LIB
`Float32.t`	`(_ FloatingPoint 8 24)`
`Float64.t`	`(_ FloatingPoint 11 53)`
`RNE`	`RNE`
`RNA`	`RNA`
`RTP`	`RTP`
`RTN`	`RTN`
`RTZ`	`RTZ`
`abs`	`fp.abs`
`neg`	`fp.neg`
`add`	`fp.add`
`sub`	`fp.sub`
`mul`	`fp.mul`
`div`	`fp.div`
`fma`	`fp.fma`
`sqrt`	`fp.sqrt`
`roundToIntegral`	`fp.roundToIntegral`
`to_real`	`fp.to_real`
`min`	`fp.min`
`max`	`fp.max`
`le`	`fp.leq`
`lt`	`fp.lt`
`ge`	`fp.geq`
`gt`	`fp.gt`
`eq`	`fp.eq`
`is_normal`	`fp.isNormal`
`is_subnormal`	`fp.isSubnormal`
`is_zero`	`fp.isZero`
`is_infinite`	`fp.isInfinite`
`is_nan`	`fp.isNaN`
`is_positive`	`fp.isPositive`
`is_negative`	`fp.isNegative`
`is_finite` x	`(not (or (fp.isInfinite` x`) (fp.isNaN` x`)))`

Figure 2: Conversion of our Why3's FP theory to SMT-LIB

in the table of Figure 2. It is straightforward since most of the symbols have their SMT-LIB equivalent. An exception is our symbol `is_finite` which does not exist in SMT-LIB and we need to translate into a Boolean formula.

Regarding the literals, we use a 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 1.0 in 32 bits in written in SMT-LIB format as `(fp #b0 #b01111111 #b00000000000000000000000)` and `0x1.99999Ap-4` is written as `(fp #b0 #b01111011 #b10011001100110011001101)`

3.6 Axiomatization for Provers Without Native Support

It is possible to discharge VCs using provers without native support of FP arithmetic: we can 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.

3.6.1 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, declared with the axiom $(\texttt{is_finite}\ l) \wedge (\texttt{to_real}\ l = v)$.

3.6.2 Extra Constants and Overflows

There are several constants useful to the axiomatization that we can derive from `eb` and `sb`:

```
constant pow2sb   : int = pow2 sb       (* 2^{eb} *)
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 predicates `in_range` and `no_overflow`.

```
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.

3.6.3 The Rounding Function

The central element of this axiomatization is the rounding function, that can be used to specify the FP operations. 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 floating point 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 [37, 10]. 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
end
```

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 (a special case being when dealing with integers as seen in 3.6.6).

Notice that when using a solver with native FP support, the driver mechanism of Why3 removes the round function and the axioms, so as to avoid the prover getting lost with the extra logic context.

3.6.4 Float Equality, Comparisons

We relate the float equality to the standard equality as much as possible. We also provide reflexivity (for finite floats), symmetry and transitivity as well as relate it to the equality of the operands projections to reals and integers.

```
predicate eq t t

axiom feq_eq: forall x y.
  is_finite x → is_finite y → not (is_zero x) → eq x y → x = y

axiom eq_feq: forall x y.
  is_finite x → is_finite y → x = y → eq x y

axiom eq_refl: forall x. is_finite x → eq x x

axiom eq_sym: forall x y. eq x y → eq y x

axiom eq_trans: forall x y z. eq x y → eq y z → eq x z

axiom eq_to_real_finite: forall x y.
  is_finite x ∧ is_finite y → (eq x y ↔ to_real x = to_real y)
```

Comparisons are specified on finite floats in terms of the comparisons of the real numbers represented:

```
axiom le_finite: forall x y [le x y].
  is_finite x ∧ is_finite y → (le x y ↔ to_real x ≤ to_real y)
```

and similar axioms for other comparison symbols. A few more axioms are stated for transitivity properties and the like. See Appendix A for all the axioms.

3.6.5 Arithmetic

We can now axiomatize the arithmetic operators. We only present the float addition, the other arithmetic operators are in the same line.

```
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

predicate to_nearest (m:mode) = m = RNE ∨ m = RNA

lemma add_finite_rev_n: forall m:mode, x y:t [add m x y].
  to_nearest m →
  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 function `add` is axiomatized through the projection to reals. The main axiom `add_finite` specifies the non overflowing addition of two finite floats. 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 mode `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.). A last axiom mentioning the addition is discussed in Section 3.6.6.

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 Appendix A for all the axioms.

3.6.6 Floats and Integers

One of the main drawbacks of Z3's native support for floating point is the handling of the conversion between floats and integers. We try to compensate for this in the theory. Those conversions typically arise when a loop iterates over an integer and the iterator is involved in a floating point computation but also in many other cases. Furthermore such computations might be correct (i.e. without rounding errors) but if the converted integer is considered as any other float, the axiomatization provided in 3.6.5 does not give good results because of the rounding function (see Section 4.2 for a detailed example on this difficulty and how to deal with it).

The axiomatization is based on three predicates `in_int_range`, `in_safe_int_range` and `is_int`:

```
predicate in_int_range (i:int) = - max_int ≤ i ≤ max_int

(* range in which every integer is representable *)
predicate in_safe_int_range (i: int) = - pow2sb ≤ i ≤ pow2sb

(* This predicate specify if a float is finite and is an integer *)
predicate is_int (x:t)
```

The first two are on integers while the third one is on floats. 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. It is used to specify correct integer computation in a float format as we will see in the

second paragraph below. The predicate is_int characterizes all floating point value that are integers. It is used to relate the projections to reals and to integers, and to help specify the rounding of a float into an integer.

When using a solver with native FP support the driver mechanism of Why3 removes these three predicates.

Float integers and their properties There is a collection of axioms to characterize float integers. Either by construction: converting an integer into a float, being a big enough float or applying roundToIntegral, or by propagation as shown in the following axioms.

```
function of_int (m:mode) (x:int) : t
function to_int (m:mode) (x:t) : int

(* by construction *)
axiom of_int_is_int: forall m, x.
  in_int_range x → is_int (of_int m x)

axiom big_float_is_int: forall m i.
  is_finite i →
  i .≤ neg (of_int m pow2sb) ∨ (of_int m pow2sb) .≤ i →
    is_int i

axiom roundToIntegral_is_int: forall m:mode, x:t. is_finite x →
  is_int (roundToIntegral m x)

(* by propagation *)
axiom eq_is_int: forall x y. eq x y → is_int x → is_int y

axiom add_int: forall x y m. is_int x → is_int y →
  is_finite (add m x y) → is_int (add m x y)

axiom sub_int: forall x y m. is_int x → is_int y →
  is_finite (sub m x y) → is_int (sub m x y)

axiom mul_int: forall x y m. is_int x → is_int y →
  is_finite (mul m x y) → is_int (mul m x y)

axiom fma_int: forall x y z m. is_int x → is_int y → is_int z →
  is_finite (fma m x y z) → is_int (fma m x y z)

axiom neg_int: forall x. is_int x → is_int (neg x)

axiom abs_int: forall x. is_int x → is_int (abs x)
```

And some basic properties that they all have:

```
axiom is_int_of_int: forall m m' x.
  is_int x → eq x (of_int m' (to_int m x))

axiom is_int_to_int: forall m x.
  is_int x → in_int_range (to_int m x)

axiom is_int_is_finite: forall x.
  is_int x → is_finite x

axiom int_to_real: forall m x.
  is_int x → to_real x = FromInt.from_int (to_int m x)

axiom roundToIntegral_int: forall m i.
  is_int i → roundToIntegral m i .= i
```

```
axiom neg_to_int: forall m x.
  is_int x → to_int m (neg x) = - (to_int m x)
```

Exact integers An integer in the range of a specific floating point type is either representable or rounded to a nearest float, which is necessarily an integer. This is where the constant `pow2sb` is interesting: as seen in `big_float_is_int`, any float bigger than `pow2sb` is an integer (in fact, bigger than `pow2sb/2`) and any integer whose absolute value is smaller or equal to `pow2sb` is representable as stipulated in the following axiom.

```
axiom to_int_of_int: forall m:mode, i:int.
  in_safe_int_range i →
    to_int m (of_int m i) = i
```

We can then refine the function `round` for integers:

```
(* round and integers *)
axiom Exact_rounding_for_integers:
  forall m:mode, i:int.
    in_safe_int_range i →
    round m (FromInt.from_int i) = FromInt.from_int i
```

And express properties stipulating the exactness of computation of “safe” integers represented as float with the following axioms:

```
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))

axiom of_int_sub_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)) (sub n (of_int m i) (of_int m j))

axiom of_int_mul_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)) (mul n (of_int m i) (of_int m j))
```

Rounding to an integer The operator `roundToIntegral` takes a rounding mode and a float and returns an integer float. Depending on the value of the rounding mode we find back well known rounding such as truncation, ceil and floor. We start with ceil

```
(** ceil *)
axiom ceil_le: forall x:t. is_finite x → x .≤ (roundToIntegral RTP x)

axiom ceil_lest: forall x y:t. le x y ∧ is_int y → le (roundToIntegral RTP x) y

axiom ceil_to_real: forall x:t.
  is_finite x →
    to_real (roundToIntegral RTP x) = FromInt.from_int (Truncate.ceil (to_real x))

axiom ceil_to_int: forall m:mode, x:t.
  is_finite x →
    to_int m (roundToIntegral RTP x) = Truncate.ceil (to_real x)
```

Floor is the dual operators and as such, axiomatized symmetrically. Ceil and floor are the basis used to defined the other rounding modes:

```
(** truncate *)
axiom truncate_neg: forall x:t.
  is_finite x → is_negative x → roundToIntegral RTZ x = roundToIntegral RTP x
```

```
axiom truncate_pos: forall x:t.
  is_finite x → is_positive x → roundToIntegral RTZ x = roundToIntegral RTN x

(* Rna *)
axiom RNA_down:
  forall x:t. lt (sub RNE x (roundToIntegral RTN x)) (sub RNE (roundToIntegral RTP x) x) →
    roundToIntegral RNA x = roundToIntegral RTN x

axiom RNA_up:
  forall x:t. gt (sub RNE x (roundToIntegral RTN x)) (sub RNE (roundToIntegral RTP x) x) →
    roundToIntegral RNA x = roundToIntegral RTP x

axiom RNA_down_tie:
  forall x:t. eq (sub RNE x (roundToIntegral RTN x)) (sub RNE (roundToIntegral RTP x) x) →
    is_negative x → roundToIntegral RNA x = roundToIntegral RTN x

axiom RNA_up_tie:
  forall x:t. eq (sub RNE (roundToIntegral RTP x) x) (sub RNE x (roundToIntegral RTN x)) →
  is_positive x → roundToIntegral RNA x = roundToIntegral RTP x
```

3.7 Consistency and Faithfulness

Our FP theory, with all its axioms, was proven conformant to the IEEE standard by realizing a model of it in Coq, using the existing library Flocq [9]. 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 (e.g. Z3), 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 IEEE, which is supposed to be the case.

4 Experiments

Tables 3 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 floating point support [14], and *Colibri* a prover developed by CEA [34, 2], both under active development within the research project *SOPRANO*[3] in which AdaCore and Inria are involved. Cells in dark gray correspond to unproved VCs with a given prover. Cells in light gray 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 Representative Small Examples

We start with the presentation of 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 every assertion should be provable, none of the assertion was 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

[3] http://soprano-project.fr/index.html

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
`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

Figure 3: Proof times in seconds for the reduced examples, detailed by provers (timeout = 30 seconds)

integers and FP, conversions between single and double precision FP. The first 11 examples correspond to reduced examples from 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. The complete Ada source codes of each of these small examples is given in Appendix B and online at `https://github.com/AdaCore/spark2014/tree/master/testsuite/gnatprove/tests/proofinuse__floating_point`.

Table 3 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. 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 δ is defined as follows:

$$\delta = drag + factor \times G \times framelength$$

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. The code implementing this computation in SPARK is as follows, with `Delta_Speed` and `New_Speed` are double precision floats:

```
Delta_Speed := Drag + Factor * G * Frame_Length;
New_Speed   := Old_Speed + Delta_Speed;
```

where the type of `Drag` and `Factor` are bounded by small integers, and constants `G` and `Frame_Length` are both small:

```
Frame_Length : constant := 1.0 / 60.0;
G : constant := 9.807; -- gravitational acceleration

subtype Ratio_T is Float64 range -1.0 .. 1.0;
subtype Drag_T is Float64 range -64.0 .. 64.0;
```

Thus, a simple interval analysis based on type bounds is sufficient to show that none of the subexpressions occurring in the computation of `Delta_Speed` overflows. We have implemented such a pre-analysis in GNATprove.

Because of the type of `Drag` and the values of constants `G` and `Frame_Length`, the absolute value of `Delta_Speed` is bounded by a small integer `Bound` (equal to 65 in our example). Thus, the addition of `Delta_Speed` to `Old_Speed`, whatever the value of this double precision float, cannot lead to an overflow, as for large values of `Old_Speed`, the result of the addition will be equal to `Old_Speed`. This is proved by both CodePeer and Z3.

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. Reasoning in real numbers, we'd like to state that:

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

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

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

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

$$\lfloor mindrag \oplus factor \otimes G \otimes framelength \rfloor \leq \delta \leq \lceil maxdrag \oplus factor \otimes G \otimes framelength \rceil \quad (2)$$

$$N \otimes \lfloor mindrag \oplus factor \otimes G \otimes framelength \rfloor \leq S(N) \leq N \otimes \lceil maxdrag \oplus factor \otimes G \otimes framelength \rceil \quad (3)$$

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 represented exactly as FP numbers, and arithmetic operations like additions and multiplications are thus exact as well on such integers, and commute. This is expressed in axioms `of_int_add_exact`, `of_int_sub_exact` and `of_int_mul_exact` presented in Section 3.6.6. Hence, for both integer values Q in the equation above (the floor and ceiling values), we have that:

$$(N \otimes Q) \oplus Q = (N \oplus 1) \otimes Q$$

which allows to prove the bounds on $S(N)$ by recurrence on N, from equations (1), (2) and (3). We follow this strategy in proving the following contract on procedure `Compute_Speed` which computes the value of `New_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);
```

Note the use of ghost function `Invariant` to express a bound on the Nth term of the series $S(N)$ that we mentioned previously. This function is itself defined based on ghost functions `Low_Bound` and `High_Bound` defining integral bounds for $S(N)$. The ghost functions can only be used in so-called ghost code (such as contracts) with no possible interference on operational code, and are stripped by the compiler from the final executable.

The automatic proof of `Compute_Speed` 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. We start by bounding all quantities involved using a ghost function `In_Bounds`:

```
function In_Bounds (V : Float64) return Boolean is
  (V in - Bound * Float64 (Frame'Last) .. Bound * Float64 (Frame'Last))
with Ghost;
```

CodePeer is used to prove automatically the following three assertions:

```
pragma Assert (Delta_Speed in -Bound .. Bound);
pragma Assert (In_Bounds (High_Bound(N)));
pragma Assert (In_Bounds (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 following assertion mixing integers and floats can be interpreted fully in bitvectors by Z3, which allows to prove it:

```
pragma Assert (Float64(N_Bv) * Bound + Bound = (Float64(N_Bv) + 1.0) * Bound);
```

Then, CodePeer is used to prove equivalent assertions on signed integers:

```
pragma Assert (Float64(N) * Bound + Bound = (Float64(N) + 1.0) * Bound);
pragma Assert (Float64(N) * (-Bound) - Bound = (Float64(N) + 1.0) * (-Bound));
```

Then, CodePeer is 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`:

```
pragma Assert (T(1) = 1.0);
```

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 3.6.6:

```
pragma Assert (Float64(N) + 1.0 = Float64(N + 1));
```

Finally, a combination of CVC4 and Z3 is used to prove bounds on the value of `New_Speed`:

```
pragma Assert (New_Speed >= Float64 (N) * (-Bound) - Bound);
pragma Assert (New_Speed >= Float64 (N + 1) * (-Bound));
pragma Assert (New_Speed <= Float64 (N) * Bound + Bound);
pragma Assert (New_Speed <= Float64 (N + 1) * Bound);
```

With these assertions, the postcondition of `Compute_Speed` is proved by Alt-Ergo. Table 4 summarizes the proof results. It can be noted that all VCs in the case study are proved by the combination of provers.

5 Conclusions

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.

5.1 Related Work

Since the mid 1990s, FP arithmetic has been formalized in interactive deductive verification systems: in PVS [12], in ACL2 [39], in HOL-light [28], and in Coq [17]. 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 [39], and the development of certified algorithms for computing elementary functions in HOL-light [28, 27]. See also [29] 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 [7]. The support for FP in Caduceus was somehow ported to the Frama-C environment [15] 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,

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` `= (Float64(N_Bv) + 1.0) * Bound`			42			0
`Float64(N) * Bound + Bound` `= (Float64(N) + 1.0) * Bound`		44		1	25	0
`Float64(N) * (-Bound) Bound` `= (Float64(N) + 1.0) * (-Bound)`				1		0
`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
Post-condition	20	0			1	

Figure 4: Proof times in seconds for the case study, detailed by provers (timeout = 60 seconds)

with various degree of complexity were designed by different authors [8, 6, 26, 33]. 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 [16].

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 [36], 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 [19], 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 of this approach is 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 is not fully correct when programs in the input language may distinguish values 0- and 0+, as the representation in real numbers cannot distinguish them anymore, possibly leading to unsound proofs. This was a documented limitation of the previous approach in SPARK. Rounding operation was axiomatized to allow proving bounds on FP operations [37], with further axioms dealing with monotonicity of rounding, rounding on integer values, etc. Contrary to the axiomatization presented in Section 3, the previous axiomatization of rounding was not realized. Soundness 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 [24], where in that case we tried to exploit native support for bitvectors in SMT solvers.

5.2 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 [8] for proving very advanced functional behaviors of programs, relating the concrete computations with some purely mathematical computations on real numbers. 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 on designing the new Why3 theory for FP arithmetic and on realizing it in Coq using Flocq. We also thank Mohamed Iguernlala and Bruno Marre for fruitful exchanges on the use of *AE_fpa* and *Colibri*.

References

[1] Ali Ayad and Claude Marché. Multi-prover verification of floating-point programs. In Jürgen Giesl and Reiner Hähnle, editors, *Fifth International Joint Conference on Automated Reasoning*, volume 6173 of *Lecture Notes in Artificial Intelligence*, pages 127–141, Edinburgh, Scotland, July 2010. Springer.

[2] Roberto Bagnara, Matthieu Carlier, Roberta Gori, and Arnaud Gotlieb. Exploiting binary floating-point representations for constraint propagation. *INFORMS Journal on Computing*, 28(1):31–46, 2016.

[3] S. Baird, A. Charlet, Y. Moy, and T. S. Taft. CodePeer – beyond bug-finding with static analysis. In Jean-Louis Boulanger, editor, *Static Analysis of Software: the Abstract Interpretation*. Wiley, 2013.

[4] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.

[5] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(6):709–727, 2015. See also `http://toccata.lri.fr/gallery/fm2012comp.en.html`.

[6] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Wave equation numerical resolution: a comprehensive mechanized proof of a C program. *Journal of Automated Reasoning*, 50(4):423–456, April 2013.

[7] Sylvie Boldo and Jean-Christophe Filliâtre. Formal Verification of Floating-Point Programs. In *18th IEEE International Symposium on Computer Arithmetic*, pages 187–194, Montpellier, France, June 2007.

[8] Sylvie Boldo and Claude Marché. Formal verification of numerical programs: from C annotated programs to mechanical proofs. *Mathematics in Computer Science*, 5:377–393, 2011.

[9] Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In Elisardo Antelo, David Hough, and Paolo Ienne, editors, *Proceedings of the 20th IEEE Symposium on Computer Arithmetic*, pages 243–252, Tübingen, Germany, 2011.

[10] Sylvie Boldo and Thi Minh Tuyen Nguyen. Hardware-independent proofs of numerical programs. In César Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium*, number NASA/CP-2010-216215 in NASA Conference Publication, pages 14–23, Washington D.C., USA, April 2010.

[11] Martin Brain, Vijay D'silva, Alberto Griggio, Leopold Haller, and Daniel Kroening. Deciding floating-point logic with abstract conflict driven clause learning. *Formal Methods in System Design*, 45(2):213–245, October 2014.

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

[13] Roderick Chapman and Florian Schanda. Are we there yet? 20 years of industrial theorem proving with SPARK. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 17–26. Springer, 2014.

[14] Sylvain Conchon, Mohamed Iguernlala, Kailiang Ji, Guillaume Melquiond, and Clément Fumex. A three-tier strategy for reasoning about floating-point numbers in SMT. In *Computer Aided Verification*, 2017.

[15] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, number 7504 in Lecture Notes in Computer Science, pages 233–247. Springer, 2012.

[16] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *Transactions on Mathematical Software*, 37(1):1–20, 2010.

[17] Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library of floating-point numbers and its application to exact computing. In *14th International Conference on Theorem Proving in Higher Order Logics*, pages 169–184, Edinburgh, Scotland, 2001.

[18] Leonardo de Moura and Nikolaj Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[19] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *FMICS*, volume 5825 of *LNCS*, pages 53–69. Springer, 2009.

[20] Claire Dross and Yannick Moy. Abstract software specifications and automatic proof of refinement. In *1st International Conference on Reliability, Safety and Security of Railway Systems*, 2016.

[21] Claire Dross and Yannick Moy. Auto-active proof of red-black trees in spark. In *NASA Formal Methods*, 2017.

[22] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.

[23] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[24] Clément Fumex, Claire Dross, Jens Gerlach, and Claude Marché. Specification and proof of high-level functional properties of bit-level programs. In *8th NASA Formal Methods Symposium*, volume 9690 of *Lecture Notes in Computer Science*, Minneapolis, MN, USA, June 2016. Springer.

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

[26] Alwyn Goodloe, César A. Muñoz, Florent Kirchner, and Loïc Correnson. Verification of numerical programs: From real numbers to floating point numbers. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods, 5th International Symposium*, volume 7871 of *Lecture Notes in Computer Science*, pages 441–446. Springer, 2013.

[27] John Harrison. Floating point verification in HOL Light: The exponential function. *Formal Methods in System Design*, 16(3):271–305, 2000.

[28] John Harrison. Formal verification of floating point trigonometric functions. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 217–233, Austin, Texas, 2000. Springer.

[29] John Harrison. Floating-point verification. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *International Symposium of Formal Methods Europe*, pages 529–532. Springer, 2005.

[30] IEEE standard for floating-point arithmetic, 2008. `https://dx.doi.org/10.1109/IEEESTD.2008.4610935`.

[31] Nikolai Kosmatov, Claude Marché, Yannick Moy, and Julien Signoles. Static versus dynamic verification in Why3, Frama-C and SPARK 2014. In *7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Lecture Notes in Computer Science, pages 461–478. Springer, October 2016.

[32] K. Rustan M. Leino and Michał Moskal. Usable auto-active verification. In *Usable Verification Workshop*, 2010.

[33] Claude Marché. Verification of the functional behavior of a floating-point program: an industrial case study. *Science of Computer Programming*, 96(3):279–296, March 2014.

[34] Bruno Marre and Claude Michel. Improving the floating point addition and subtraction constraints. In *Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming*, CP'10, pages 360–367. Springer-Verlag, 2010.

[35] John W. McCormick and Peter C. Chapin. *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015.

[36] Antoine Miné. Relational abstract domains for the detection of floating-point run-time errors. In David A. Schmidt, editor, *13th European Symposium on Programming (ESOP)*, volume 2986 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2004.

[37] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems*, 30(3):12, May 2008.

[38] Philipp Rümmer and Thomas Wahl. An SMT-LIB theory of binary floating-point arithmetic. In *Informal proceedings of 8th International Workshop on Satisfiability Modulo Theories (SMT) at FLoC, Edinburgh, Scotland*, 2010.

[39] David M. Russinoff. 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.

A Complete Why3 Theory of FP Arithmetic

For reference, here is a full verbatim of the Why3 theory of FP arithmetic as of today. Future updates will be found in Why3's standard library, see `http://why3.lri.fr/`.

```
(** {1 Formalization of Floating-Point Arithmetic}

  Full float theory (with infinities and NaN).

  A note on intended semantics: we use the same idea as the SMTLIB floating
  point theory, that defers any inconsistencies to the "parent" document.
  Hence, in doubt, the correct axiomatisation is one that implements
  [BTRW14] "An Automatable Formal Semantics for IEEE-754 Floating-Point
  Arithmetic", which in turn defers any inconsistencies to IEEE-754.

  This theory is split into two parts: the first part talks about IEEE
  operations and this is what you should use as a user, the second part is
  internal only and is an axiomatisation for the provers that do not
  natively support floating point. You should not use any symbols you find
  there in your verification conditions as solvers with native floating
  point support will leave them uninterpreted.
*)

(** {2 Rounding Modes} *)

theory RoundingMode
  type mode = RNE | RNA | RTP | RTN | RTZ
  (** {h <ul>
     <li>RNE : Round Nearest ties to Even
     <li>RNA : Round Nearest ties to Away
     <li>RTP : Round Towards Positive
     <li>RTN : Round Towards Negative
     <li>RTZ : Round Towards Zero
     </ul} *)

  predicate to_nearest (m:mode) = m = RNE \/ m = RNA
end

theory GenericFloat

  use import int.Int
  use import bv.Pow2int
  use real.Abs
  use real.FromInt
  use real.Truncate
  use import real.RealInfix
  use export RoundingMode

  (** {2 Part I - Public Interface}   *)

  constant eb : int
  (** the number of bits in the exponent.  *)

  constant sb : int
  (** the number of bits in the significand, including the hidden bit. *)

  axiom eb_gt_1: 1 < eb
  axiom sb_gt_1: 1 < sb

  (** {3 Sorts} *)
```

```
type t
(** abstract type to denote floating-point numbers, including the special values
    for infinities and NaNs *)

(** {3 Constructors and Constants} *)

constant zeroF     : t   (** +0.0 *)
(* exp_bias = 2^(sb - 1) - 1 *)
(* max_finite_exp = 2^sb - 2 - exp_bias = exp_bias *)
(* max_significand = (2^eb + 2^eb - 1) * 2^(1-eb) *)
(* max_value = (2^eb + 2^eb - 1) * 2^(1-eb) * 2 ^ max_finite_exp *)
(* [m;x] = ( 1 + m * 2^(1-eb) ) * 2^( x - exp_bias ) *)
(* max_value = [2^(eb-1); 2^sb - 2] *)

(** {3 Operators} *)

function add mode t t : t
function sub mode t t : t
function mul mode t t : t
function div mode t t : t
  (** The four basic operations, rounded in the given mode *)

function abs         t   : t   (** Absolute value *)
function neg         t   : t   (** Opposite *)
function fma  mode t t t : t   (** Fused multiply-add: x * y + z *)
function sqrt mode   t   : t   (** Square root *)

function (.-_) (x:t)   : t = neg x
function (.+)  (x y:t) : t = add RNE x y
function (.-)  (x y:t) : t = sub RNE x y
function (.*)  (x y:t) : t = mul RNE x y
function (./)  (x y:t) : t = div RNE x y
  (** Notations for operations in the default mode RNE *)

function roundToIntegral mode t : t
  (** Rounding to an integer *)

function min       t t : t
function max       t t : t
(** Minimum and Maximum
    Note that we have to follow IEEE-754 and SMTLIB here. Two things to
   note in particular:
   1) min(-0, 0) is either 0 or -0, there is a choice

   2) if either argument is NaN then the other argument is returned

 *)

(** {3 Comparisons} *)

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
(** equality on floats, different from = since not (eq NaN NaN) *)

predicate (.<=) (x:t) (y:t) = le x y
predicate (.<)  (x:t) (y:t) = lt x y
predicate (.>=) (x:t) (y:t) = ge x y
predicate (.>)  (x:t) (y:t) = gt x y
predicate (.=)  (x:t) (y:t) = eq x y
```

```
(** Notations *)


(** {3 Classification of numbers} *)

predicate is_normal    t
predicate is_subnormal t
predicate is_zero      t
predicate is_infinite  t
predicate is_nan       t
predicate is_positive  t
predicate is_negative  t

(** helper predicate for zeros, normals and subnormals. not defined
   so that the axiomatisation below can use it without talking about
   subnormals *)
predicate is_finite    t

predicate is_plus_infinity  (x:t) = is_infinite x /\ is_positive x
predicate is_minus_infinity (x:t) = is_infinite x /\ is_negative x
predicate is_plus_zero      (x:t) = is_zero x /\ is_positive x
predicate is_minus_zero     (x:t) = is_zero x /\ is_negative x
predicate is_not_nan        (x:t) = is_finite x \/ is_infinite x

axiom is_not_nan: forall x:t. is_not_nan x <-> not (is_nan x)

axiom is_not_finite: forall x:t.
  not (is_finite x) <-> (is_infinite x \/ is_nan x)



(** {3 Conversions from other sorts} *)

(* from bitvec binary interchange                           *)
(*   partly done with from_binary (for literals only)       *)
(* from another fp                 - see FloatConverter     *)
(* from real                                                *)
(*   partly done with (!) (for literals only)               *)
(* from unsigned integer bitvector - see Float_BV_Converter *)
(* from signed integer bitvector                            *)

(** {3 Conversions to other sorts} *)

(* to unsigned integer bitvector   - see Float_BV_Converter *)
(* to signed integer bitvector                              *)
(* to real *)
function to_real   t    : real

(** {2 Part II - Private Axiomatisation} *)

(** {3 Constructors and Constants} *)
axiom zeroF_is_positive : is_positive zeroF
axiom zeroF_is_zero     : is_zero zeroF

axiom zero_to_real : forall x [is_zero x].
  is_zero x <-> is_finite x /\ to_real x = 0.0

(** {3 Conversions from other sorts} *)

(* with mathematical int *)
(* note that these conversions do not feature in SMTLIB *)
```

```
(* intended semantics for of_int are the same as (_ to_fp eb sb) with a   *)
(* suitably sized bitvector, large enough to hold x                       *)
(* note values >= than the below should result in infinities              *)
(*    float32 : 0x1ffffff * 2^103                                         *)
(*    float64 : 0x3fffffffffffff * 2^970                                  *)
(* also note that this function never yields a subnormal, or a NaN, or -0 *)
function of_int (m:mode) (x:int) : t

(** {3 Conversions to other sorts} *)

(* Intended semantics for to_int are the same as (_ fp.to_sbv) with a    *)
(* suitably sized bitvector. Safe minimum sizes are given below:         *)
(*    float32 : 129                                                      *)
(*    float64 : 1025                                                     *)
(* In particular this function should be uninterpreted for infinities    *)
(* and NaN. Take care that no conclusion can be made on the result based *)
(* on the size of the bitvector chosen in those cases, i.e. this should  *)
(* not hold:                                                             *)
(*    to_int +INF < 2 ** 2048     // nope                                *)
(*    to_int +INF > 0             // nope                                *)
function to_int (m:mode) (x:t) : int

axiom zero_of_int : forall m. zeroF = of_int m 0

(** {3 Arithmetic} *)

(* The intended meaning for round is the rounding for floating point as  *)
(* described on p17 of IEEE-754. For results where the corresponding     *)
(* floating point result would be infinity or NaN this function should   *)
(* be uninterpreted.                                                     *)
(*                                                                       *)
(* Note that this means round (+INF) > 0 is not true.                    *)
(* Note also this means round (2*INF) > round (INF) is not true either.  *)
function round mode real : real

constant max_real : real (* defined when cloning *)
constant max_int  : int

constant emax : int = pow2 (eb - 1)

axiom max_int : max_int = pow2 emax - pow2 (emax - sb)
axiom max_real_int: max_real = FromInt.from_int max_int

predicate in_range (x:real) = -. max_real <=. x <=. max_real

predicate in_int_range (i:int) = - max_int <= i <= max_int

axiom is_finite: forall x:t. is_finite x -> in_range (to_real x)

(* used as a condition to propagate is_finite *)
predicate no_overflow (m:mode) (x:real) = in_range (round m x)

(* Axioms on round *)

axiom Bounded_real_no_overflow "W:non_conservative_extension:N" :
  forall m:mode, x:real. in_range x -> no_overflow m x

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

(** rounding up and down *)
axiom Round_down_le:
  forall x:real. round RTN x <=. x
axiom Round_up_ge:
  forall x:real. round RTP x >=. x
axiom Round_down_neg:
  forall x:real. round RTN (-.x) = -. round RTP x
axiom Round_up_neg:
  forall x:real. round RTP (-.x) = -. round RTN x

(* The biggest representable integer whose predecessor (i.e. -1) is  representable *)
constant pow2sb : int (* defined when cloning *)
axiom pow2sb: pow2sb = pow2 sb

(* range in which every integer is representable *)
predicate in_safe_int_range (i: int) = - pow2sb <= i <= pow2sb

(* round and integers *)

axiom Exact_rounding_for_integers:
  forall m:mode, i:int.
    in_safe_int_range i ->
      round m (FromInt.from_int i) = FromInt.from_int i

(** {3 Comparisons} *)

(** Comparison predicates *)

predicate same_sign (x y : t) =
  (is_positive x /\ is_positive y) \/ (is_negative x /\ is_negative y)
predicate diff_sign (x y : t) =
  (is_positive x /\ is_negative y) \/ (is_negative x /\ is_positive y)

axiom feq_eq: forall x y.
  is_finite x -> is_finite y -> not (is_zero x) -> x .= y -> x = y

axiom eq_feq: forall x y.
  is_finite x -> is_finite y -> x = y -> x .= y

axiom eq_refl: forall x. is_finite x -> x .= x

axiom eq_sym: forall x y. x .= y -> y .= x

axiom eq_trans: forall x y z. x .= y -> y .= z -> x .= z

axiom eq_zero: zeroF .= (.- zeroF)

axiom eq_to_real_finite: forall x y.
  is_finite x /\ is_finite y -> (x .= y <-> to_real x = to_real y)

axiom eq_special: forall x y. x .= y ->
     (is_not_nan x  /\ is_not_nan y
  /\ ((is_finite x /\ is_finite y)
      \/ (is_infinite x /\ is_infinite y /\ same_sign x y)))

axiom lt_finite: forall x y [lt x y].
  is_finite x /\ is_finite y -> (lt x y <-> to_real x <. to_real y)
```

```
axiom le_finite: forall x y [le x y].
  is_finite x /\ is_finite y -> (le x y <-> to_real x <=. to_real y)

lemma le_lt_trans:
  forall x y z:t. x .<= y /\ y .< z -> x .< z

lemma lt_le_trans:
  forall x y z:t. x .< y /\ y .<= z -> x .< z

lemma le_ge_asym:
  forall x y:t. x .<= y /\ x .>= y -> x .= y

lemma not_lt_ge: forall x y:t.
  not (x .< y) /\ is_not_nan x /\ is_not_nan y -> x .>= y

lemma not_gt_le: forall x y:t.
  not (x .> y) /\ is_not_nan x /\ is_not_nan y -> x .<= y

axiom le_special: forall x y [le x y]. le x y ->
    ((is_finite x /\ is_finite y)
 \/ ((is_minus_infinity x /\ is_not_nan y)
 \/  (is_not_nan x /\ is_plus_infinity y)))

axiom lt_special: forall x y [lt x y]. lt x y ->
    ((is_finite x /\ is_finite y)
 \/ ((is_minus_infinity x /\ is_not_nan y /\ not (is_minus_infinity y))
 \/  (is_not_nan x /\ not (is_plus_infinity x) /\ is_plus_infinity y)))

axiom lt_lt_finite: forall x y z. lt x y -> lt y z -> is_finite y

(*  lemmas on sign *)
axiom positive_to_real: forall x[is_positive x|to_real x >=. 0.0].
  is_finite x -> is_positive x -> to_real x >=. 0.0
axiom to_real_positive: forall x[is_positive x].
  is_finite x -> to_real x >. 0.0 -> is_positive x

axiom negative_to_real: forall x [is_negative x|to_real x <=. 0.0].
  is_finite x -> is_negative x -> to_real x <=. 0.0
axiom to_real_negative: forall x [is_negative x].
  is_finite x -> to_real x <. 0.0 -> is_negative x

axiom negative_xor_positive: forall x.
  not (is_positive x /\ is_negative x)
axiom negative_or_positive: forall x.
  is_not_nan x -> is_positive x \/ is_negative x

lemma diff_sign_trans:
  forall x y z:t. (diff_sign x y /\ diff_sign y z) -> same_sign x z

lemma diff_sign_product:
  forall x y:t.
    (is_finite x /\ is_finite y /\ to_real x *. to_real y <. 0.0) ->
      diff_sign x y

lemma same_sign_product:
  forall x y:t.
    (is_finite x /\ is_finite y /\ same_sign x y) ->
      to_real x *. to_real y >=. 0.0

predicate product_sign (z x y: t) =
  (same_sign x y -> is_positive z) /\ (diff_sign x y -> is_negative z)
```

```
(** {3 Overflow} *)

(* This predicate is used to tell what is the result of a rounding
   in case of overflow in the axioms specifying add/sub/mul and fma
   *)
predicate overflow_value (m:mode) (x:t) =
  match m with
  | RTN -> if is_positive x then is_finite x /\ to_real x = max_real
                            else is_infinite x
  | RTP -> if is_positive x then is_infinite x
                            else is_finite x /\ to_real x = -. max_real
  | RTZ -> if is_positive x then is_finite x /\ to_real x = max_real
                            else is_finite x /\ to_real x = -. max_real
  | (RNA | RNE) -> is_infinite x
  end

(* This predicate is used to tell what is the sign of zero in the
   axioms specifying add and sub *)
predicate sign_zero_result (m:mode) (x:t) =
  is_zero x ->
  match m with
  | RTN -> is_negative x
  | _   -> is_positive x
  end

(** {3 binary operations} *)

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].
  to_nearest m ->
  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)

axiom sub_finite: forall m:mode, x y:t [sub m x y].
  is_finite x -> is_finite y -> no_overflow m (to_real x -. to_real y) ->
  is_finite (sub m x y) /\
  to_real (sub m x y) = round m (to_real x -. to_real y)

lemma sub_finite_rev: forall m:mode, x y:t [sub m x y].
  is_finite (sub m x y) ->
  is_finite x /\ is_finite y

lemma sub_finite_rev_n: forall m:mode, x y:t [sub m x y].
  to_nearest m ->
  is_finite (sub m x y) ->
  no_overflow m (to_real x -. to_real y) /\
  to_real (sub m x y) = round m (to_real x -. to_real y)

axiom mul_finite: forall m:mode, x y:t [mul m x y].
  is_finite x -> is_finite y -> no_overflow m (to_real x *. to_real y) ->
  is_finite (mul m x y) /\
  to_real (mul m x y) = round m (to_real x *. to_real y)
```

```
lemma mul_finite_rev: forall m:mode, x y:t [mul m x y].
  is_finite (mul m x y) ->
  is_finite x /\ is_finite y

lemma mul_finite_rev_n: forall m:mode, x y:t [mul m x y].
  to_nearest m ->
  is_finite (mul m x y) ->
  no_overflow m (to_real x *. to_real y) /\
  to_real (mul m x y) = round m (to_real x *. to_real y)

axiom div_finite: forall m:mode, x y:t [div m x y].
  is_finite x -> is_finite y ->
  not is_zero y -> no_overflow m (to_real x /. to_real y) ->
  is_finite (div m x y) /\
  to_real (div m x y) = round m (to_real x /. to_real y)

lemma div_finite_rev: forall m:mode, x y:t [div m x y].
  is_finite (div m x y) ->
  (is_finite x /\ is_finite y /\ not is_zero y) \/
  (is_finite x /\ is_infinite y /\ to_real (div m x y) = 0.)

lemma div_finite_rev_n: forall m:mode, x y:t [div m x y].
  to_nearest m ->
  is_finite (div m x y) -> is_finite y ->
  no_overflow m (to_real x /. to_real y) /\
  to_real (div m x y) = round m (to_real x /. to_real y)

axiom neg_finite: forall x:t [neg x].
  is_finite x ->
  is_finite (neg x) /\
  to_real (neg x) = -. to_real x

lemma neg_finite_rev: forall x:t [neg x].
  is_finite (neg x) ->
  is_finite x /\
  to_real (neg x) = -. to_real x

axiom abs_finite: forall x:t [abs x].
  is_finite x ->
  is_finite (abs x) /\
  to_real (abs x) = Abs.abs (to_real x) /\
  is_positive (abs x)

lemma abs_finite_rev: forall x:t [abs x].
  is_finite (abs x) ->
  is_finite x /\
  to_real (abs x) = Abs.abs (to_real x)

axiom abs_universal : forall x:t [abs x]. not (is_negative (abs x))

axiom fma_finite: forall m:mode, x y z:t [fma m x y z].
  is_finite x -> is_finite y -> is_finite z ->
  no_overflow m (to_real x *. to_real y +. to_real z) ->
  is_finite (fma m x y z) /\
  to_real (fma m x y z) = round m (to_real x *. to_real y +. to_real z)

lemma fma_finite_rev: forall m:mode, x y z:t [fma m x y z].
  is_finite (fma m x y z) ->
  is_finite x /\ is_finite y /\ is_finite z

lemma fma_finite_rev_n: forall m:mode, x y z:t [fma m x y z].
  to_nearest m ->
```

```
    is_finite (fma m x y z) ->
    no_overflow m (to_real x *. to_real y +. to_real z) /\
    to_real (fma m x y z) = round m (to_real x *. to_real y +. to_real z)

  use real.Square

  axiom sqrt_finite: forall m:mode, x:t [sqrt m x].
    is_finite x -> to_real x >=. 0. ->
    is_finite (sqrt m x) /\
    to_real (sqrt m x) = round m (Square.sqrt (to_real x))

  lemma sqrt_finite_rev: forall m:mode, x:t [sqrt m x].
    is_finite (sqrt m x) ->
    is_finite x /\ to_real x >=. 0. /\
    to_real (sqrt m x) = round m (Square.sqrt (to_real x))

  predicate same_sign_real (x:t) (r:real) =
    (is_positive x /\ r >. 0.0) \/ (is_negative x /\ r <. 0.0)

  axiom add_special: forall m:mode, x y:t [add m x y].
    let r = add m x y in
    (is_nan x \/ is_nan y -> is_nan r)
    /\
    (is_finite x /\ is_infinite y -> is_infinite r /\ same_sign r y)
    /\
    (is_infinite x /\ is_finite y -> is_infinite r /\ same_sign r x)
    /\
    (is_infinite x /\ is_infinite y /\ same_sign x y
      -> is_infinite r /\ same_sign r x)
    /\
    (is_infinite x /\ is_infinite y /\ diff_sign x y -> is_nan r)
    /\
    (is_finite x /\ is_finite y /\ not no_overflow m (to_real x +. to_real y)
      -> same_sign_real r (to_real x +. to_real y) /\ overflow_value m r)
    /\
    (is_finite x /\ is_finite y
      -> if same_sign x y then same_sign r x else sign_zero_result m r)

  axiom sub_special: forall m:mode, x y:t [sub m x y].
    let r = sub m x y in
    (is_nan x \/ is_nan y -> is_nan r)
    /\
    (is_finite x /\ is_infinite y -> is_infinite r /\ diff_sign r y)
    /\
    (is_infinite x /\ is_finite y -> is_infinite r /\ same_sign r x)
    /\
    (is_infinite x /\ is_infinite y /\ same_sign x y -> is_nan r)
    /\
    (is_infinite x /\ is_infinite y /\ diff_sign x y
      -> is_infinite r /\ same_sign r x)
    /\
    (is_finite x /\ is_finite y /\ not no_overflow m (to_real x -. to_real y)
      -> same_sign_real r (to_real x -. to_real y) /\ overflow_value m r)
    /\
    (is_finite x /\ is_finite y
      -> if diff_sign x y then same_sign r x else sign_zero_result m r)

  axiom mul_special: forall m:mode, x y:t [mul m x y].
    let r = mul m x y in
       (is_nan x \/ is_nan y -> is_nan r)
    /\ (is_zero x /\ is_infinite y -> is_nan r)
    /\ (is_finite x /\ is_infinite y /\ not (is_zero x)
```

```
       -> is_infinite r)
   /\ (is_infinite x /\ is_zero y -> is_nan r)
   /\ (is_infinite x /\ is_finite y /\ not (is_zero  y)
       -> is_infinite r)
   /\ (is_infinite x /\ is_infinite y -> is_infinite r)
   /\ (is_finite x /\ is_finite y /\ not no_overflow m (to_real x *. to_real y)
       -> overflow_value m r)
   /\ (not is_nan r -> product_sign r x y)

 axiom div_special: forall m:mode, x y:t [div m x y].
   let r = div m x y in
      (is_nan x \/ is_nan y -> is_nan r)
   /\ (is_finite x /\ is_infinite y -> is_zero r)
   /\ (is_infinite x /\ is_finite y -> is_infinite r)
   /\ (is_infinite x /\ is_infinite y -> is_nan r)
   /\ (is_finite x /\ is_finite y /\ not (is_zero y) /\
      not no_overflow m (to_real x /. to_real y)
       -> overflow_value m r)
   /\ (is_finite x /\ is_zero y /\ not (is_zero x)
       -> is_infinite r)
   /\ (is_zero x /\ is_zero y -> is_nan r)
   /\ (not is_nan r -> product_sign r x y)

 axiom neg_special: forall x:t [neg x].
      (is_nan x -> is_nan (neg x))
   /\ (is_infinite x -> is_infinite (neg x))
   /\ (not is_nan x -> diff_sign x (neg x))

 axiom abs_special: forall x:t [abs x].
      (is_nan x -> is_nan (abs x))
   /\ (is_infinite x -> is_infinite (abs x))
   /\ (not is_nan x -> is_positive (abs x))

 axiom fma_special: forall m:mode, x y z:t [fma m x y z].
   let r = fma m x y z in
      (is_nan x \/ is_nan y \/ is_nan z -> is_nan r)
   /\ (is_zero x /\ is_infinite y -> is_nan r)
   /\ (is_infinite x /\ is_zero y -> is_nan r)
   /\ (is_finite x /\ not (is_zero x) /\ is_infinite y /\ is_finite z
      -> is_infinite r /\ product_sign r x y)
   /\ (is_finite x /\ not (is_zero x) /\ is_infinite y /\ is_infinite z
      -> (if product_sign z x y then is_infinite r /\ same_sign r z
          else is_nan r))
   /\ (is_infinite x /\ is_finite y /\ not (is_zero y) /\ is_finite z
      -> is_infinite r /\ product_sign r x y)
   /\ (is_infinite x /\ is_finite y /\ not (is_zero y) /\ is_infinite z
      -> (if product_sign z x y then is_infinite r /\ same_sign r z
          else is_nan r))
   /\ (is_infinite x /\ is_infinite y /\ is_finite z
      -> is_infinite r /\ product_sign r x y)
   /\ (is_finite x /\ is_finite y /\ is_infinite z
      -> is_infinite r /\ same_sign r z)
   /\ (is_infinite x /\ is_infinite y /\ is_infinite z
      -> (if product_sign z x y then is_infinite r /\ same_sign r z
          else is_nan r))
   /\ (is_finite x /\ is_finite y /\ is_finite z /\
      not no_overflow m (to_real x *. to_real y +. to_real z)
      -> same_sign_real r (to_real x *. to_real y +. to_real z)
      /\ overflow_value m r)
   /\ (is_finite x /\ is_finite y /\ is_finite z
      -> if product_sign z x y then same_sign r z
         else (to_real x *. to_real y +. to_real z = 0.0 ->
```

```
              if m = RTN then is_negative r else is_positive r))

axiom sqrt_special: forall m:mode, x:t [sqrt m x].
  let r = sqrt m x in
     (is_nan x -> is_nan r)
  /\ (is_plus_infinity x -> is_plus_infinity r)
  /\ (is_minus_infinity x -> is_nan r)
  /\ (is_finite x /\ to_real x <. 0.0 -> is_nan r)
  /\ (is_zero x -> same_sign r x)
  /\ (is_finite x /\ to_real x >. 0.0 -> is_positive r)

(* exact arithmetic with integers *)

axiom of_int_add_exact: forall m n, i j.
  in_safe_int_range i -> in_safe_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))

axiom of_int_sub_exact: forall m n, i j.
  in_safe_int_range i -> in_safe_int_range j ->
  in_safe_int_range (i - j) -> eq (of_int m (i - j)) (sub n (of_int m i) (of_int m j))

axiom of_int_mul_exact: forall m n, i j.
  in_safe_int_range i -> in_safe_int_range j ->
  in_safe_int_range (i * j) -> eq (of_int m (i * j)) (mul n (of_int m i) (of_int m j))

(* min and max *)

lemma Min_r : forall x y:t. y .<= x -> (min x y) .= y
lemma Min_l : forall x y:t. x .<= y -> (min x y) .= x
lemma Max_r : forall x y:t. y .<= x -> (max x y) .= x
lemma Max_l : forall x y:t. x .<= y -> (max x y) .= y

(* _____________ *)

use real.Truncate

(* This predicate specify if a float is finite and is an integer *)
predicate is_int (x:t)

(** characterizing integers *)

(* by construction *)
axiom zeroF_is_int: is_int zeroF

axiom of_int_is_int: forall m, x.
  in_int_range x -> is_int (of_int m x)

axiom big_float_is_int: forall m i.
  is_finite i ->
  i .<= neg (of_int m pow2sb) \/ (of_int m pow2sb) .<= i ->
    is_int i

axiom roundToIntegral_is_int: forall m:mode, x:t. is_finite x ->
  is_int (roundToIntegral m x)

(* by propagation *)
axiom eq_is_int: forall x y. eq x y -> is_int x -> is_int y

axiom add_int: forall x y m. is_int x -> is_int y ->
  is_finite (add m x y) -> is_int (add m x y)

axiom sub_int: forall x y m. is_int x -> is_int y ->
```

```
    is_finite (sub m x y) -> is_int (sub m x y)

  axiom mul_int: forall x y m. is_int x -> is_int y ->
    is_finite (mul m x y) -> is_int (mul m x y)

  axiom fma_int: forall x y z m. is_int x -> is_int y -> is_int z ->
    is_finite (fma m x y z) -> is_int (fma m x y z)

  axiom neg_int: forall x. is_int x -> is_int (neg x)

  axiom abs_int: forall x. is_int x -> is_int (abs x)

  (** basic properties of float integers *)

  axiom is_int_of_int: forall x m m'.
    is_int x -> eq x (of_int m' (to_int m x))

  axiom is_int_to_int: forall m x.
    is_int x -> in_int_range (to_int m x)

  axiom is_int_is_finite: forall x.
    is_int x -> is_finite x

  axiom int_to_real: forall m x.
    is_int x -> to_real x = FromInt.from_int (to_int m x)

(*  axiom int_mode: forall m1 m2 x.
    is_int x -> to_int m1 x = to_int m2 x  etc ...*)

  (** rounding ints *)

  axiom truncate_int: forall m:mode, i:t. is_int i ->
    roundToIntegral m i .= i

  (** truncate *)

  axiom truncate_neg: forall x:t.
    is_finite x -> is_negative x -> roundToIntegral RTZ x = roundToIntegral RTP x

  axiom truncate_pos: forall x:t.
    is_finite x -> is_positive x -> roundToIntegral RTZ x = roundToIntegral RTN x

  (** ceil *)

  axiom ceil_le: forall x:t. is_finite x -> x .<= (roundToIntegral RTP x)

  axiom ceil_lest: forall x y:t. x .<= y /\ is_int y -> (roundToIntegral RTP x) .<= y

  axiom ceil_to_real: forall x:t.
    is_finite x ->
      to_real (roundToIntegral RTP x) = FromInt.from_int (Truncate.ceil (to_real x))

  axiom ceil_to_int: forall m:mode, x:t.
    is_finite x ->
      to_int m (roundToIntegral RTP x) = Truncate.ceil (to_real x)

  (** floor *)

  axiom floor_le: forall x:t. is_finite x -> (roundToIntegral RTN x) .<= x

  axiom floor_lest: forall x y:t. y .<= x /\ is_int y -> y .<= (roundToIntegral RTN x)
```

```
  axiom floor_to_real: forall x:t.
    is_finite x ->
      to_real (roundToIntegral RTN x) = FromInt.from_int (Truncate.floor (to_real x))

  axiom floor_to_int: forall m:mode, x:t.
    is_finite x ->
      to_int m (roundToIntegral RTN x) = Truncate.floor (to_real x)

  (* Rna *)

  axiom RNA_down:
    forall x:t. (x .- (roundToIntegral RTN x)) .< ((roundToIntegral RTP x) .- x) ->
      roundToIntegral RNA x = roundToIntegral RTN x

  axiom RNA_up:
    forall x:t. (x .- (roundToIntegral RTN x)) .> ((roundToIntegral RTP x) .- x) ->
      roundToIntegral RNA x = roundToIntegral RTP x

  axiom RNA_down_tie:
    forall x:t. (x .- (roundToIntegral RTN x)) .= ((roundToIntegral RTP x) .- x) ->
      is_negative x -> roundToIntegral RNA x = roundToIntegral RTN x

  axiom RNA_up_tie:
    forall x:t. ((roundToIntegral RTP x) .- x) .= (x .- (roundToIntegral RTN x)) ->
    is_positive x -> roundToIntegral RNA x = roundToIntegral RTP x

  (* to_int *)
  axiom to_int_roundToIntegral: forall m:mode, x:t.
    to_int m x = to_int m (roundToIntegral m x)

  axiom to_int_monotonic: forall m:mode, x y:t.
    is_finite x -> is_finite y -> le x y -> to_int m x <= to_int m y

  axiom to_int_of_int: forall m:mode, i:int.
    in_safe_int_range i ->
      to_int m (of_int m i) = i

  axiom eq_to_int: forall m, x y. is_finite x -> x .= y ->
    to_int m x = to_int m y

  axiom neg_to_int: forall m x.
    is_int x -> to_int m (neg x) = - (to_int m x)

  axiom roundToIntegral_is_finite  : forall m:mode, x:t. is_finite x ->
    is_finite (roundToIntegral m x)
end

(** {2 Conversions to/from bitvectors} *)

theory Float_BV_Converter
  use bv.BV8
  use bv.BV16
  use bv.BV32
  use bv.BV64
  use import RoundingMode

  (* with unsigned int as bitvector *)
  type t                       (* float *)

  function of_ubv8  mode BV8.t  : t
  function of_ubv16 mode BV16.t : t
  function of_ubv32 mode BV32.t : t
```

```
  function of_ubv64 mode BV64.t : t

  function to_ubv8  mode t : BV8.t
  function to_ubv16 mode t : BV16.t
  function to_ubv32 mode t : BV32.t
  function to_ubv64 mode t : BV64.t

  use import real.RealInfix
  use real.FromInt

  predicate is_finite t
  predicate le t t
  function to_real         t : real
  function round   mode real : real

  (** of unsigned bv axioms  *)
  (* only true for big enough floats...  *)

  axiom of_ubv8_is_finite : forall m, x. is_finite (of_ubv8  m x)
  axiom of_ubv16_is_finite: forall m, x. is_finite (of_ubv16 m x)
  axiom of_ubv32_is_finite: forall m, x. is_finite (of_ubv32 m x)
  axiom of_ubv64_is_finite: forall m, x. is_finite (of_ubv64 m x)

  axiom of_ubv8_monotonic :
    forall m, x y. BV8.ule  x y -> le (of_ubv8  m x) (of_ubv8  m y)
  axiom of_ubv16_monotonic:
    forall m, x y. BV16.ule x y -> le (of_ubv16 m x) (of_ubv16 m y)
  axiom of_ubv32_monotonic:
    forall m, x y. BV32.ule x y -> le (of_ubv32 m x) (of_ubv32 m y)
  axiom of_ubv64_monotonic:
    forall m, x y. BV64.ule x y -> le (of_ubv64 m x) (of_ubv64 m y)

  axiom of_ubv8_to_real : forall m, x.
    to_real (of_ubv8  m x) = FromInt.from_int (BV8.t'int x)
  axiom of_ubv16_to_real: forall m, x.
    to_real (of_ubv16 m x) = FromInt.from_int (BV16.t'int x)
  (* of_ubv32_to_real is defined at cloning *)
  axiom of_ubv64_to_real: forall m, x.
    to_real (of_ubv64 m x) = round m (FromInt.from_int (BV64.t'int x))
end

(** {2 Standard simple precision floats (32 bits)} *)

theory Float32
  use int.Int
  use import real.Real

  type t = < float 8 24 >

  constant pow2sb : int = 16777216
  constant max_real : real = 0x1.FFFFFEp127

  clone export GenericFloat with
    type t = t,
    constant eb = t'eb,
    constant sb = t'sb,
    constant max_real = max_real,
    constant pow2sb = pow2sb,
    function to_real = t'real,
    predicate is_finite = t'isFinite,
    goal eb_gt_1,
    goal sb_gt_1,
```

```
    goal max_int,
    goal pow2sb

  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
end

(** {2 Standard double precision floats (64 bits)} *)

theory Float64
  use int.Int
  use import real.Real

  type t = < float 11 53 >

  constant pow2sb : int = 9007199254740992
  constant max_real : real = 0x1.FFFFFFFFFFFFFp1023

  clone export GenericFloat with
    type t = t,
    constant eb = t'eb,
    constant sb = t'sb,
    constant max_real = max_real,
    constant pow2sb = pow2sb,
    function to_real = t'real,
    predicate is_finite = t'isFinite,
    goal eb_gt_1,
    goal sb_gt_1,
    goal max_int,
    goal pow2sb

  lemma round_bound_ne :
    forall x:real [round RNE x].
      no_overflow RNE x ->
        x - 0x1p-53 * Abs.abs(x) - 0x1p-1075 <= round RNE x <= x + 0x1p-53 * Abs.abs(x) + 0x1p-1075

  lemma round_bound :
    forall m:mode, x:real [round m x].
      no_overflow m x ->
      x - 0x1p-52 * Abs.abs(x) - 0x1p-1074 <= round m x <= x + 0x1p-52 * Abs.abs(x) + 0x1p-1074
end

(** {2 Conversions between float formats} *)

theory FloatConverter

  use Float64
  use Float32

  use export RoundingMode

  function to_float64 mode Float32.t : Float64.t
  function to_float32 mode Float64.t : Float32.t

  lemma round_double_single :
```

```
  forall m1 m2:mode, x:real.
    Float64.round m1 (Float32.round m2 x) = Float32.round m2 x

  lemma to_float64_exact:
    forall m:mode, x:Float32.t. Float32.t'isFinite x ->
      Float64.t'isFinite (to_float64 m x)
   /\ Float64.t'real (to_float64 m x) = Float32.t'real x

  lemma to_float32_conv:
    forall m:mode, x:Float64.t. Float64.t'isFinite x ->
      Float32.no_overflow m (Float64.t'real x) ->
        Float32.t'isFinite (to_float32 m x)
     /\ Float32.t'real (to_float32 m x) = Float32.round m (Float64.t'real x)

end

theory Float32_BV_Converter
  use import Float32

  clone export Float_BV_Converter with
    type t = t,
    predicate is_finite = t'isFinite,
    predicate le = (.<=),
    function to_real = t'real,
    function round = round

  axiom of_ubv32_to_real : forall m, x.
    t'real (of_ubv32 m x) = round m (FromInt.from_int (BV32.t'int x))
end

theory Float64_BV_Converter
  use import Float64

  clone export Float_BV_Converter with
    type t = t,
    predicate is_finite = t'isFinite,
    predicate le = (.<=),
    function to_real = t'real,
    function round = round

  axiom of_ubv32_to_real : forall m, x.
    t'real (of_ubv32 m x) = FromInt.from_int (BV32.t'int x)
end
```

B Ada Code Listings

The Ada files that follow give the source code for the examples mentioned in Section 4.

```
-- types.ads

package Types with
  SPARK_Mode
is
   --  Same type definitions as in Ada.Interfaces

   type Integer_32 is range -2 ** 31 .. 2 ** 31 - 1;

   type Unsigned_32 is mod 2 ** 32;

   type Unsigned_16 is mod 2 ** 16;

   type Unsigned_8 is mod 2 ** 8;

   --  Same function as in Interfaces

   function Shift_Right (Value : Unsigned_32; Count : Natural) return Unsigned_32
      with Import,
      Convention => Intrinsic,
     Global     => null;

   function Shift_Right (Value : Unsigned_16; Count : Natural) return Unsigned_16
      with Import,
      Convention => Intrinsic,
     Global     => null;

   function Shift_Left (Value : Unsigned_32; Count : Natural) return Unsigned_32
      with Import,
      Convention => Intrinsic,
     Global     => null;

   function Shift_Left (Value : Unsigned_16; Count : Natural) return Unsigned_16
      with Import,
      Convention => Intrinsic,
      Global     => null;

   --  Subtypes of Integer_32

   subtype Natural_32 is Integer_32 range 0 .. 2 ** 31 - 1;

   subtype Positive_32 is Integer_32 range 1 .. 2 ** 31 - 1;

   --  Same as standard floating point type

   type Float_32 is new Float;
   type Float_64 is new Long_Float;

   --  Array of 10 integers
```

```
   type Index_10 is range 1 .. 10;

   type Array_10_Integer_32 is array (Index_10) of Integer_32;

   --  Uninterpreted property over integers

   function Property (X : Integer_32; Y : Index_10) return Boolean;

end Types;

-- floating_point.ads

with Types; use Types;

package Floating_Point with
  SPARK_Mode
is
   --  from MA18-004 (internal test)
   procedure Range_Add (X : Float_32; Res : out Float_32);

   --  from M809-005 (internal test)
   procedure Range_Mult (X : Float_32; Res : out Float_32);

   --  from N121-026 (industrial user)
   procedure Range_Add_Mult (X, Y, Z : Float_32; Res : out Float_32);

   --  from NC03-013 (industrial user)
   procedure Int_To_Float_Simple (X : Unsigned_16; Res : out Float_32);

   --  from NC04-023 (industrial user)
   function Float_To_Long_Float (X : Float) return Long_Float;

   C : constant := 10.0;
   type U is mod 2**32;
   subtype T is U range 0 .. 1_000_000;

   --  from O227-007 (industrial user)
   procedure Incr_By_Const (State : in out Float_32;
                            X     : T);

   --  The following User_Rule_* are user-written axioms from OldSPARK from
   --  a real world project. They are also submitted to SMTCOMP.

   subtype Squarable_Float is Float range -18446742974197923840.0 ..
                                          18446742974197923840.0;

   procedure User_Rule_2 (X, Y, Z : Float;
                          Res     : out Boolean);

   procedure User_Rule_3 (X, Y : Float;
                          Res  : out Boolean);
```

```
   procedure User_Rule_4 (X, Y : Float;
                          Res  : out Boolean);

   procedure User_Rule_6 (X, Y, Z, A : Float;
                          Res        : out Boolean);

   procedure User_Rule_7 (X, Y, Z, A : Float;
                          Res        : out Boolean);

   procedure User_Rule_9 (A, B : Float;
                          Res  : out Boolean);

   procedure User_Rule_10 (A, B : Float;
                           Res  : out Boolean);

   procedure User_Rule_11 (A, B, C, D : Float;
                           Res        : out Boolean);

   procedure User_Rule_13 (D0, D1, R : Float;
                           Res       : out Boolean);

   procedure User_Rule_14 (D0, D1, R, X : Float;
                           Res          : out Boolean);

   procedure User_Rule_15 (X, Y : Float;
                           Res  : out Boolean);

   --  from P912-012 (industrial user)
   procedure Float_Different (X, Y : Float_32; Res : out Float_32);
   procedure Float_Greater (X, Y : Float_32; Res : out Float_32);

   --  from P914-008 (industrial user)
   procedure Diffs (X, Y, Z : Float);

   --  from PB25-029 (industrial user)
   procedure Half_Bound (X : Float_32; Res : out Float_64);

end Floating_Point;

-- floating_point.adb

package body Floating_Point with SPARK_Mode
is
   --  CBMC can trivially show this is true
   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;

   --  CBMC can trivially show this is true
   procedure Range_Mult (X : Float_32; Res : out Float_32) is
```

```
begin
   pragma Assume (X in 5.0 .. 10.0);
   Res := X * 2.0 - 5.0;
   pragma Assert (Res >= X);
end Range_Mult;

--  CBMC can show this is true, but it takes a while (25 seconds)
procedure Range_Add_Mult (X, Y, Z : Float_32; Res : out Float_32) is
begin
   pragma Assume (X >= 0.0 and then X <= 180.0);
   pragma Assume (Y >= -180.0 and then Y <= 0.0);
   pragma Assume (Z >= 0.0 and then Z <= 1.0);
   pragma Assume (X + Y >= 0.0);
   Res := X + Y * Z;
   pragma Assert (Res >= 0.0 and then Res <= 360.0);
end Range_Add_Mult;

--  CBMC can trivially show this is true
procedure Int_To_Float_Simple (X : Unsigned_16; Res : out Float_32) is
   L : constant := 7.3526e6;
begin
   pragma Assume (X /= 0);
   pragma Assert (Float_32 (X) >= 0.9);  --  @ASSERT:PASS
   Res := L / Float_32 (X);              --  @OVERFLOW_CHECK:PASS
end Int_To_Float_Simple;

--  CBMC can trivially show this is true
function Float_To_Long_Float (X : Float) return Long_Float is
   Tmp : Long_Float;
begin
   pragma Assume (X >= Float'First and X <= Float'Last);
   Tmp := Long_Float (X);
   pragma Assert
        (Tmp >= Long_Float (Float'First) and
         Tmp <= Long_Float (Float'Last));
   return Tmp;
end Float_To_Long_Float;

--  CBMC can show this is true, but it takes a while (7 seconds)
procedure Incr_By_Const (State : in out Float_32;
                         X     : T)
is
begin
   pragma Assume (X < T'Last and State in 0.0 | C .. Float_32 (X) * C);
   State := State + C;
   pragma Assert (State in C .. Float_32 (X + 1) * C);
end Incr_By_Const;

procedure User_Rule_2 (X, Y, Z : Float;
                       Res     : out Boolean)
is
begin
   pragma Assume (X >= 0.0);
```

```
   pragma Assume (Y >= 0.0);
   pragma Assume (Z >= 0.0);
   pragma Assume (X <= 16777216.0);
   pragma Assume (Y <= 16777216.0);
   Res := - (X * Y) <= Z;
   pragma Assert (Res);     -- valid
end User_Rule_2;

procedure User_Rule_3 (X, Y : Float;
                       Res  : out Boolean)
is
begin
   pragma Assume (X < Y);
   pragma Assume (Y > 0.0);
   Res := X / Y <= 1.0;
   pragma Assert (Res);     -- valid
end User_Rule_3;

procedure User_Rule_4 (X, Y : Float;
                       Res  : out Boolean)
is
begin
   pragma Assume (X <= Y);
   pragma Assume (Y > 0.0);
   Res := X / Y <= 1.0;
   pragma Assert (Res);     -- valid
end User_Rule_4;

procedure User_Rule_6 (X, Y, Z, A : Float;
                       Res        : out Boolean)
is
begin
   pragma Assume (Z >= 0.0);
   pragma Assume (X >= Y);
   pragma Assume (Y >= Z);
   pragma Assume (X > Z);
   pragma Assume (A <= 0.0);
   Res := (X - Y) / (X - Z) >= A;
   pragma Assert (Res);     -- valid
end User_Rule_6;

--  User_Rule_7 (although very similar to 6) is surprisingly difficult
--  to verify.

procedure User_Rule_7 (X, Y, Z, A : Float;
                       Res        : out Boolean)
is
begin
   pragma Assume (Z >= 0.0);
   pragma Assume (X >= Y);
   pragma Assume (Y >= Z);
   pragma Assume (X > Z);
   pragma Assume (A >= 1.0);
```

```
      Res := (X - Y) / (X - Z) <= A;
      pragma Assert (Res);     -- valid
   end User_Rule_7;

   procedure User_Rule_9 (A, B : Float;
                          Res  : out Boolean)
   is
   begin
      pragma Assume (abs A < 5800.0 * abs B);
      Res := A / B <= 5800.0;
      pragma Assert (Res);     -- valid
   end User_Rule_9;

   procedure User_Rule_10 (A, B : Float;
                           Res  : out Boolean)
   is
   begin
      pragma Assume (abs A < 5800.0 * abs B);
      Res := A / B >= -5800.0;
      pragma Assert (Res);     -- valid
   end User_Rule_10;

   --  The counterexample to 11 involves infinity (a = 0, c = Float'Last)

   procedure User_Rule_11 (A, B, C, D : Float;
                           Res        : out Boolean)
   is
   begin
      pragma Assume (A >= 0.0);
      pragma Assume (B >= 0.0);
      Res := (C * C) * A + (D * D) * B >= 0.0; --@OVERFLOW_CHECK:FAIL
      pragma Assert (Res);
   end User_Rule_11;

   procedure User_Rule_13 (D0, D1, R : Float;
                           Res       : out Boolean)
   is
   begin
      pragma Assume (D1 > D0);
      pragma Assume (R in 0.0 .. 1.0);
      pragma Assume (D0 >= 0.0);
      Res := D1 - ((D1 - D0) * R) >= 0.0;
      pragma Assert (Res);     -- valid
   end User_Rule_13;

   procedure User_Rule_14 (D0, D1, R, X : Float;
                           Res          : out Boolean)
   is
   begin
      pragma Assume (D1 > D0);
      pragma Assume (R in 0.0 .. 1.0);
      pragma Assume (D0 >= 0.0);
      pragma Assume (D1 <= X);
```

```
      Res := D1 - ((D1 - D0) * R) <= X;
      pragma Assert (Res);     -- valid
   end User_Rule_14;

   procedure User_Rule_15 (X, Y : Float;
                           Res  : out Boolean)
   is
   begin
      pragma Assume (X in -7800.0 .. 7800.0);
      pragma Assume (Y in -7800.0 .. 7800.0);
      pragma Assume (X > abs Y);
      Res := X * X - Y * Y >= 0.0;
      pragma Assert (Res);     -- valid
   end User_Rule_15;

   procedure Polynomial (X : Float)
   is
   begin
      pragma Assume (X in 0.0 .. 60.0);
      pragma Assert ((((X + 2.0) * X + 3.0) + 4.0) * X + 5.0 in Float'Range);
   end Polynomial;

   procedure Float_Different (X, Y : Float_32; Res : out Float_32) is
   begin
      pragma Assume (X /= Y);
      Res := X - Y;
      pragma Assert (Res /= 0.0);
   end Float_Different;

   procedure Float_Greater (X, Y : Float_32; Res : out Float_32) is
   begin
      pragma Assume (X > Y);
      Res := X - Y;
      pragma Assert (Res > 0.0);
   end Float_Greater;

   procedure Diffs (X, Y, Z : Float) is
   begin
      pragma Assume (X - Y > 0.0);
      pragma Assume (Y - Z > 0.0);
      pragma Assert (X - Z > 0.0);
   end Diffs;

   procedure Half_Bound (X : Float_32; Res : out Float_64) is
   begin
      pragma Assume (X in -1.0 .. 1.0);
      Res := Float_64 (X * 0.5);
      pragma Assert (Res in -1.0 .. 1.0);
   end Half_Bound;

end Floating_Point;
```

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399