

# Formally Certified Round-Off Error Analysis of Floating-Point Functions

Mariano Moscato    Laura Titolo  
National Institute of Aerospace  
{mariano.moscato,laura.titolo}@nianet.org

Aaron Dutle    César A. Muñoz  
NASA Langley Research Center  
{aaron.m.dutle,cesar.a.munoz}@nasa.gov

## Abstract

Round-off errors in floating-point computations can lead to catastrophic consequences when occurring in safety-critical systems. For this reason, it is crucial to develop formal methods to ensure that the behavior of the implementation of numerical calculations respects some properties of interest with respect to the ideal real-number description. This paper presents a fully automatic analyzer for the estimation of round-off errors of floating-point valued functional expressions. This research tool computes an over-approximation of the round-off error of a given floating-point expression and also generates a formal certificate that ensures the correctness of the computed estimation. This certificate relies on a formalization of floating-point arithmetic developed in the Prototype Verification System (PVS). The proposed analysis is illustrated by means of two real case studies: a key function of the Compact Position Reporting (CPR) algorithm, which is part of communication protocol used to encode Global Positioning System (GPS) coordinates of an airplane in a compact way, and the horizontal component of an aircraft conflict detection algorithm.

## 1. Introduction

Floating-point arithmetic is an efficient solution for approximating computations on real numbers, which has been extensively used over the past several decades in a wide range of applications. One significant problem of floating-point computation is the presence of round-off errors that can make the final floating-point result very different from the one that would be obtained by using real arithmetic. This issue becomes more problematic for safety-critical applications such as aerospace and air-traffic control software,

where even a small error can lead to catastrophic consequences.

For this reason, it becomes essential to estimate the round-off error of floating-point computations in a sound way. This can be used to ensure that the floating-point implementation behaves correctly with respect to some properties of interest possessed by the ideal real number implementation. Several solutions based on different techniques have been proposed in the literature: abstract interpretation based analysis (Goubault and Putot 2006), automatic generation of verification conditions (de Dinechin et al. 2011), Taylor expansions approximation (Solovyev et al. 2015), among the many. In this paper, the static analyzer PRECISA (PVS Round-off Error Certifier v1a Static Analysis) is presented. This tool is a prototype that automatically computes an over-approximation for the round-off error of an input program, and also provides a certificate ensuring its soundness. This certificate can be automatically checked in PVS (Owre et al. 1992) without the need of any expertise in theorem proving.

PRECISA relies on the floating point formalization for PVS introduced in (Boldo and Muñoz 2006). This formalization is completed and enriched with new theories to reason about round-off errors approximations, following the IEEE-754 standard. PRECISA uses a compositional denotational semantics that symbolically computes an over-approximation for the round-off error of the input program, together with the conditions under which this approximation is correct. Associating conditions to the computed error estimation makes the analysis more precise and avoids considering computations leading to runtime errors such as division by zero or square root of a negative number. The compositionality of the proposed semantics and of the PVS floating point formalization are the basis for the modularity and scalability of PRECISA.

The contributions of this paper are the following: (1) a refinement of the PVS formalization of the floating-point IEEE-754 standard that can be used to formally check properties about several aspects of floating-point computations; (2) a technique that estimates in an automatic and certified way symbolic bounds for the round-off error of a program; (3) an extension of the branch-and-bound algorithm pre-

sented in (Narkawicz and Muñoz 2013) that computes numerical bounds for a symbolic error expression. The soundness of this extension is formally verified in PVS, ensuring the correctness of the computed results.

The paper is organized as follow. In Section 2, the new PVS formalization for floating-point round-off errors is introduced. In Section 3, the denotational semantics which collects symbolic round-off error expressions is defined for a generic declarative expression language. The PRECISA analyzer is presented in Section 4. Some experimental results and the analysis of two case studies are presented in Section 5. Related work is discussed in Section 6, and Section 7 concludes the paper.

The formal development presented in this paper is electronically available as part of the NASA PVS library (<https://github.com/nasa/pvslib>). For more information about the tool presented in this paper, the reader is referred to <https://shemesh.larc.nasa.gov/fm/PRECISA>.

## 2. Floating-Point Round-off Error Formalization

*Floating-point numbers*<sup>1</sup>, which are denoted by the symbol  $\mathbb{F}$ , are used in computer programs as a finite and discrete representation of the set of real numbers ( $\mathbb{R}$ ). In concrete implementations, such as the adherents to the IEEE-754 standard (Stevenson 1981),  $\mathbb{F}$  is defined as a finite set; whilst in more abstract developments, for example the formalization presented in (Boldo and Muñoz 2006),  $\mathbb{F}$  has an enumerable cardinality.

A conversion function  $\mathbf{R} : \mathbb{F} \mapsto \mathbb{R}$  is usually defined to refer to the real number represented by a given float. For example, in (Boldo and Muñoz 2006) a floating-point number is defined as a pair of integers  $(m, e)$ , where  $m$  is called the *significand* and  $e$  the *exponent* of the float, and the conversion function is stated as  $\mathbf{R}(m, e) = m \cdot b^e$ , where  $b \in \mathbb{N}$  is called the *base* or *radix* of the system. As this representation is redundant, notions about normality and canonicity are also defined (see (Boldo and Muñoz 2006) for details).

Not every real number can be exactly represented by a float, thus, a notion of representation error can be defined as follows. If a floating-point number  $p$  is used to represent a real number  $r$ , the difference  $|\mathbf{R}(p) - r|$  is usually called the *round-off error* (or *rounding error*) of the ideal value  $r$  with respect to  $p$ .

The floating-point number usually chosen to represent a real number  $r$ , noted in this paper as  $\mathbf{F}(r)$ , is the number for which the round-off error of  $r$  is minimal. Such a floating-point number is called the *closest* to  $r$ . When there is more than one closest representation for a real  $r$ , some tie-breaking criteria must be used. In the IEEE-754 standard (Stevenson 1981) the so-called *rounding modes* are de-

finied. Examples of them are the rounding to *even* mode, where the float with even significand is chosen, and the *away from zero* mode, where the float with the greater absolute value is chosen.

A measure of the precision of a floating-point number  $p$  as a representation of some real is given by the concept of *unit in the last place* (*ulp* in short) defined in (Boldo and Muñoz 2006) as  $\text{ulp}(p) = b^{e_p}$ , where  $e_p$  is the exponent of the canonic form of  $p$ . The ulp of a floating point can be used as a bound of the round-off error since, as shown in (Boldo and Muñoz 2006), if  $p$  is the closest representation of some real  $r$  they are apart from each other for no more than half of the ulp of  $p$ . It is also possible to define the ulp of a real number as the ulp of the canonical form of its floating-point representation, i.e.,  $\text{ulp}(r) = \text{ulp}(\mathbf{F}(r))$ . Then, the previous condition can be restated as defined in (Harrison 1999).

$$|\mathbf{R}(p) - r| \leq \frac{\text{ulp}(r)}{2}.$$

The previous formula relates an ideal value with its representation. As complex expressions can be formed by applying functions on representations, upper bounds on the round-off error of those functions based on the round-off error of their arguments is a key part of a modular approach to a rounding error calculation of arbitrary floating-point expressions.

In order to express bounds in such way, it is useful to state some relationship between a real-valued function and its floating-point counterpart. In the IEEE-754 standard (Stevenson 1981), an  $n$ -ary floating-point operation  $\tilde{f}(p_1, \dots, p_n)$  is considered *correctly rounded* if it is equivalent to perform the corresponding real-valued operation on the reals denoted by its operands  $p_1, \dots, p_n$  and then round the result to the closest floating-point number. In this work, a weaker definition is used. A floating-point function  $\tilde{f}$  is said to be *correctly rounded* if the following restriction holds<sup>2</sup>.

$$|\mathbf{R}(\tilde{f}(p_i)_{i=1}^n) - f(\mathbf{R}(p_i)_{i=1}^n)| \leq \frac{\text{ulp}(f(\mathbf{R}(p_i)_{i=1}^n))}{2}. \quad (2.1)$$

If every floating point  $p_i$  representing an ideal value  $r_i$  is assumed to carry a round-off error not bigger than some bound  $e_i$ , i.e.,  $|\mathbf{R}(p_i) - r_i| \leq e_i$ , in order to state a bound on the accumulated round-off error of  $f$ , a bounding expression  $\epsilon_f(e_i)_{i=1}^n$  fulfilling the formula stated below is needed.

$$|f(\mathbf{R}(p_i)_{i=1}^n) - f(r_i)_{i=1}^n| \leq \epsilon_f(e_i)_{i=1}^n. \quad (2.2)$$

Finally, if there exists a nonnegative real-valued expression  $v(r_i, e_i)_{i=1}^n$  such that  $|f(\mathbf{R}(p_i)_{i=1}^n)| \leq v(r_i, e_i)_{i=1}^n$  holds, the following bound for the round-off between the floating-point calculation and the real valued counterpart follows from formulas 2.1 and 2.2 and the triangle inequality.

<sup>1</sup> Also referred to simply as *floating points* or *floats*.

<sup>2</sup> In order to save space, we will note the application of a  $n$ -ary function  $f$  to the  $n$  arguments  $x_1, \dots, x_n$  as  $f(x_i)_{i=1}^n$ .

$$|\mathbf{R}(\tilde{f}(p_i)_{i=1}^n) - f(r_i)_{i=1}^n| \leq e_f(e_i)_{i=1}^n + \frac{\text{ulp}(v(r_i, e_i)_{i=1}^n)}{2}.$$

The formal development presented in this paper enriches and extends the high-level formalization of floating-point numbers introduced in (Boldo and Muñoz 2006) with the concepts about rounding error expressed above. Adhering to the original rationale, these concepts are stated in a general way. No particular format, such as IEEE single precision or IEEE double precision, nor specific rounding mode are assumed in advance. Nevertheless, explicit instantiations for those formats supporting the *even closest* rounding mode are provided in order to deal with concrete examples.

Round-off bounds for the following operations are included in the formalization. As above, every floating point  $p_i$  is assumed to be used to represent an ideal value  $r_i$  with and rounding error  $e_i$ , i.e.,  $|\mathbf{R}(p_i) - r_i| \leq e_i$ . Operations on reals are denoted as usual (+, \*,  $\sqrt{x}$ , etc.), while the floating-point operations are written with a tilde on top ( $\tilde{+}$ ,  $\tilde{*}$ ,  $\widetilde{sqr}$ t, etc.).

- Addition ( $p_1 \tilde{+} p_2$ )  
 $r_1 + r_2 + e_1 + e_2 + 1/2 \text{ulp}(|r_1 + r_2| + e_1 + e_2)$ .
- Subtraction ( $p_1 \tilde{-} p_2$ )  
 $r_1 - r_2 + e_1 + e_2 + 1/2 \text{ulp}(|r_1 - r_2| + e_1 + e_2)$ .
- Multiplication ( $p_1 \tilde{*} p_2$ )  
 $r_1 r_2 + |r_1|e_2 + |r_2|e_1 + e_1 e_2 + 1/2 \text{ulp}((|r_1| + e_1)(|r_2| + e_2))$ .
- Division ( $p_1 \tilde{/} p_2$ )  
 $r_1/r_2 + \frac{|r_1|e_2 + |r_2|e_1}{r_2 r_2 - e_2 |r_2|} + 1/2 \text{ulp}\left(\frac{|r_1|}{|r_2|} + \frac{|r_1|e_2 + |r_2|e_1}{r_2 r_2 - e_2 |r_2|}\right)$ .
- Floor ( $\widetilde{floor}(p)$ )  
 $\lfloor r \rfloor + e + 1 + 1/2 \text{ulp}(\lfloor r \rfloor + e + 1)$ .
- Square root ( $\widetilde{sqr}$ t( $p$ ))  
 $\sqrt{r} + \sqrt{e} + 1/2 \text{ulp}(\sqrt{r} + \sqrt{e})$ .

Bounds for the additive inverse and the absolute value are also included. As they can be implemented exactly, the corresponding bound only is affected by the round-off error of the argument of the operation.

- Additive inverse ( $\tilde{-}p$ ) and absolute value ( $\widetilde{abs}(p)$ )  
 $r + e$ .

The formalization of round-off error of functions is parametric on the function. This features aims to facilitate further extensions to the formalization, such as the adding of support for other operations.

### 3. Static Analysis for Round-off Errors

In this section, a denotational semantics for a generic declarative expression language is introduced. This semantics collects information about the round-off error of floating-point operations. For each possible program execution path, the semantics computes a symbolic expression representing the

round-off error of the program when taking this path, and collects the correspondent path condition together with other validity conditions that are needed to avoid erroneous behaviors. Associating these conditions to the computed error bounds makes the analysis more precise and avoids considering erroneous cases such as a division by zero or a square root of a negative number.

In the following, the sets of arithmetic and boolean expressions over reals are denoted as  $AExpr$  and  $BExpr$ , respectively. The floating point counterparts of  $AExpr$  and  $BExpr$  are denoted as  $\widetilde{AExpr}$  and  $\widetilde{BExpr}$ , respectively.

The considered expression language contains conditionals, let-expressions and function calls. Given a set of process symbols  $\Pi$ , a denumerable set of variables  $Var$ ,  $p \in \Pi$ , and  $x \in Var$ , the syntax of program expressions is given by the following grammar.

$$\begin{aligned} Expr ::= & \widetilde{AExpr} \mid \text{if } \widetilde{BExpr} \text{ then } Expr \text{ else } Expr \mid \\ & \text{let } x = \widetilde{AExpr} \text{ in } Expr \mid p(\widetilde{AExpr}, \dots, \widetilde{AExpr}). \end{aligned}$$

A program is defined as a set of *function declarations* of the form  $p(\vec{x}) = S$  (for some expression  $S$ ), where  $\vec{x}$  denotes a generic tuple of variables in  $Var$ . The set of programs is denoted as  $Prog$ .

The proposed semantics collects, for each program path, the corresponding path condition and two *symbolic* arithmetic expressions representing (1) the value of the output assuming the use of real arithmetic and (2) an upper bound for the round-off error due to floating-point operations. The previous information is stored in a *conditional error bound*, which is a tuple on the form  $(c, r, e)$  where  $c$  is the representation of a boolean condition, and  $r$  and  $e$  are symbolic arithmetic expressions in  $AExpr$ . Intuitively,  $(c, r, e)$  means that if the condition  $c$  is true, then the output of the ideal real numbers implementation of the program is  $r$  and the round-off error of the floating-point implementation is bounded by  $e$ . Roughly speaking, the semantics of a program is a set of conditional error bounds, one for each possible execution path that can be taken following one or another branch in the conditionals.

In general, the use of floating-point arithmetic makes the tests of conditionals unstable. This means that the control flow is altered and does not correspond to the control flow that would be taken by the same program with perfect real arithmetic. The presence of unstable tests can lead to unsound results, since the output of the program is not only disturbed by rounding errors, but also by the errors in the evaluation of the tests of the conditionals provoked by the rounding errors.

Consider the conditional statement *if*  $\phi$  *then*  $S_1$  *else*  $S_2$ . When  $\phi$  and its real counterpart (the one corresponding to the perfect real arithmetic flow) have the same truth value, the real and the floating-point flows coincide, thus, the analysis is sound. On the other hand, when they have different values, the error of taking the incorrect branch must be con-

sidered. In order to distinguish the previous cases, it is necessary to keep the information on the truth value of both floating point and real boolean tests. Therefore, the *condition* in a *conditional error bound* is defined as a pair  $(\eta, \tilde{\eta})$  where  $\eta \in BExpr$  and  $\tilde{\eta} \in \widetilde{BExpr}$ . Here,  $\eta$  represents the path conditions of the real flow and  $\tilde{\eta}$  the ones of the floating-point flow.

The domain of conditions is denoted as  $Cond = BExpr \times \widetilde{BExpr}$  and  $\mathbb{C} = \wp(Cond \times AExpr \times AExpr)$  denotes the domain formed by sets of conditional error bounds, which is the support domain of the presented semantics. An *environment* is defined as a function mapping a variable to a set of conditional error bounds, i.e.,  $Env = Var \rightarrow \mathbb{C}$ . The empty environment is denoted as  $\perp_{Env}$  and maps every variable to  $\emptyset$ .

The semantics of arithmetic expressions is a function  $\mathcal{A} : \widetilde{AExpr} \times Env \rightarrow \mathbb{C}$  defined as follows, where  $\sigma \in Env$ ,  $a, a_1, a_2 \in \widetilde{AExpr}$ ,  $n \in \mathbb{F}$ ,  $x \in Var$  and  $R, E : Var \rightarrow Var$  are two functions that associate to each variable a fresh variable representing the real value and the error of  $x$ , respectively.

$$\begin{aligned} \mathcal{A}[\![n]\!]_{\sigma} &:= \{(true, true, \mathbf{R}(n), 0)\} \\ \mathcal{A}[\![x]\!]_{\sigma} &:= \begin{cases} \{(true, true, R(x), E(x))\} & \text{if } \sigma(x) = \emptyset \\ \sigma(x) & \text{otherwise} \end{cases} \\ \mathcal{A}[\![a_1 \dot{+} a_2]\!]_{\sigma} &:= \bigcup \{(\eta_1 \wedge \eta_2, \tilde{\eta}_1 \wedge \tilde{\eta}_2, r_1 + r_2, \\ & e_1 + e_2 + 1/2 \text{ulp}(|r_1 + r_2| + e_1 + e_2)) \\ & \mid (\eta_1, \tilde{\eta}_1, r_1, e_1) \in \mathcal{A}[\![a_1]\!]_{\sigma}, (\eta_2, \tilde{\eta}_2, r_2, e_2) \in \mathcal{A}[\![a_2]\!]_{\sigma}\} \\ \mathcal{A}[\![a_1 \dot{-} a_2]\!]_{\sigma} &:= \bigcup \{(\eta_1 \wedge \eta_2, \tilde{\eta}_1 \wedge \tilde{\eta}_2, r_1 - r_2, \\ & e_1 + e_2 + 1/2 \text{ulp}(|r_1 - r_2| + e_1 + e_2)) \\ & \mid (\eta_1, \tilde{\eta}_1, r_1, e_1) \in \mathcal{A}[\![a_1]\!]_{\sigma}, (\eta_2, \tilde{\eta}_2, r_2, e_2) \in \mathcal{A}[\![a_2]\!]_{\sigma}\} \\ \mathcal{A}[\![a_1 \dot{*} a_2]\!]_{\sigma} &:= \bigcup \{(\eta_1 \wedge \eta_2, \tilde{\eta}_1 \wedge \tilde{\eta}_2, r_1 r_2, \\ & |r_1|e_2 + |r_2|e_1 + e_1 e_2 + 1/2 \text{ulp}((|r_1| + e_1)(|r_2| + e_2))) \\ & \mid (\eta_1, \tilde{\eta}_1, r_1, e_1) \in \mathcal{A}[\![a_1]\!]_{\sigma}, (\eta_2, \tilde{\eta}_2, r_2, e_2) \in \mathcal{A}[\![a_2]\!]_{\sigma}\} \\ \mathcal{A}[\![a_1 \dot{/} a_2]\!]_{\sigma} &:= \\ & \bigcup \{(\eta_1 \wedge \eta_2 \wedge (r_2 - e_2 > 0 \vee r_2 + e_2 < 0), \tilde{\eta}_1 \wedge \tilde{\eta}_2, r_1/r_2, \\ & \frac{|r_1|e_2 + |r_2|e_1}{r_2 r_2 - e_2 |r_2|} + 1/2 \text{ulp}\left(\frac{|r_1|}{|r_2|} + \frac{|r_1|e_2 + |r_2|e_1}{r_2 r_2 - e_2 |r_2|}\right)) \\ & \mid (\eta_1, \tilde{\eta}_1, r_1, e_1) \in \mathcal{A}[\![a_1]\!]_{\sigma}, (\eta_2, \tilde{\eta}_2, r_2, e_2) \in \mathcal{A}[\![a_2]\!]_{\sigma}\} \\ \mathcal{A}[\![\dot{-}a]\!]_{\sigma} &:= \bigcup \{(\eta, \tilde{\eta}, -r, e) \mid (\eta, \tilde{\eta}, r, e) \in \mathcal{A}[\![a]\!]_{\sigma}\} \\ \mathcal{A}[\![\widetilde{abs}(a)]\!]_{\sigma} &:= \bigcup \{(\eta, \tilde{\eta}, |r|, e) \mid (\eta, \tilde{\eta}, r, e) \in \mathcal{A}[\![a]\!]_{\sigma}\} \\ \mathcal{A}[\![\widetilde{floor}(a)]\!]_{\sigma} &:= \bigcup \{(\eta, \tilde{\eta}, \lfloor r \rfloor, e + 1 + 1/2 \text{ulp}(\lfloor r \rfloor + e + 1)) \\ & \mid (\eta, \tilde{\eta}, r, e) \in \mathcal{A}[\![a]\!]_{\sigma}\} \\ \mathcal{A}[\![\widetilde{sqrt}(a)]\!]_{\sigma} &:= \bigcup \{(\eta \wedge (r - e \geq 0), \tilde{\eta}, \sqrt{r}, \\ & \sqrt{e} + 1/2 \text{ulp}(\sqrt{r} + \sqrt{e})) \mid (\eta, v, e) \in \mathcal{A}[\![a]\!]_{\sigma}\} \end{aligned}$$

The semantics of a floating-point numerical value  $n \in \mathbb{F}$  models the fact that no rounding occurs and the associated error is 0. The semantics of a variable  $x \in Var$  is composed of two cases. If  $x$  belongs to the environment, then the variable has been previously bound to an arithmetic expression  $a$  through a let-expression. In this case, the semantics of  $x$  is exactly the semantics of  $a$ . If  $x$  is not in the environment, then  $x$  is a parameter of the function. Here, a new conditional error bound is added with two fresh variables,  $R(x)$  and  $E(x)$ , representing the real value and the error of  $x$ , respectively.

In the case of arithmetic operations, the real value is obtained by applying the operation in real arithmetic to the real values of the operands, and the new error bound is obtained as a function of the error bounds and real values of the operands. The new condition is obtain as the combination of the conditions of the operands.

For division and square root, additional constraints are needed in the conditions to exclude the cases in which the program produces runtime errors. More in detail, for the division,  $r_2 - e_2 > 0 \vee r_2 + e_2 < 0$  implies that  $r_2$  and  $\mathbf{F}(r_2)$  are different from 0. Notice that  $e_2$  is always positive and  $\mathbf{F}(r_2) \in [r_2 - e_2, r_2 + e_2]$ . For the square root,  $r - e \geq 0$  implies that  $r \geq 0$  and  $\mathbf{F}(r) \geq 0$ .

Let  $\text{MIGC} := \{p(\vec{x}) \mid p \in \Pi, \vec{x} \text{ are distinct variables}\}$ . An *interpretation* is a function  $\rho : \text{MIGC} \rightarrow \mathbb{C}$  modulo variance. The set of all interpretations is denoted as  $Int$ . The empty interpretation is denoted as  $\perp_{Int}$  and maps everything to  $\emptyset$ .

Given  $\sigma \in Env$  and  $\rho \in Int$ , the semantics of program expressions,  $\mathcal{S} : Expr \times Env \times Int \rightarrow \mathbb{C}$ , returns the set of conditional error bounds representing the maximum round-off error for each execution path, together with the corresponding condition.

$$\begin{aligned} \mathcal{S}[\![a]\!]_{\sigma}^{\rho} &:= \mathcal{A}[\![a]\!]_{\sigma} \\ \mathcal{S}[\![let\ x = a\ in\ S]\!]_{\sigma}^{\rho} &:= \mathcal{S}[\![S]\!]_{\sigma[x \mapsto \mathcal{A}[\![a]\!]_{\sigma}]}^{\rho} \\ \mathcal{S}[\![if\ \phi\ then\ S_1\ else\ S_2]\!]_{\sigma}^{\rho} &:= \\ & \mathcal{S}[\![S_1]\!]_{\sigma}^{\rho} \Downarrow_{(\mathbf{R}_B(\phi), \phi)} \cup \mathcal{S}[\![S_2]\!]_{\sigma}^{\rho} \Downarrow_{(\neg \mathbf{R}_B(\phi), \neg \phi)} \cup \\ & \bigcup \{(\eta_1 \wedge \eta_2, \tilde{\eta}_1, r_2, e_1 + |r_1 - r_2|) \Downarrow_{(\neg \mathbf{R}_B(\phi), \phi)} \\ & \mid (\eta_1, \tilde{\eta}_1, r_1, e_1) \in \mathcal{S}[\![S_1]\!]_{\sigma}^{\rho}, (\eta_2, \tilde{\eta}_2, r_2, e_2) \in \mathcal{S}[\![S_2]\!]_{\sigma}^{\rho}\} \cup \\ & \bigcup \{(\eta_1 \wedge \eta_2, \tilde{\eta}_2, r_1, e_2 + |r_1 - r_2|) \Downarrow_{(\mathbf{R}_B(\phi), \neg \phi)} \\ & \mid (\eta_1, \tilde{\eta}_1, r_1, e_1) \in \mathcal{S}[\![S_1]\!]_{\sigma}^{\rho}, (\eta_2, \tilde{\eta}_2, r_2, e_2) \in \mathcal{S}[\![S_2]\!]_{\sigma}^{\rho}\} \\ \mathcal{S}[\![p(a_1, \dots, a_n)]\!]_{\sigma}^{\rho} &:= \\ & \bigcup \{(\eta \wedge \eta_1 \wedge \dots \wedge \eta_n, \tilde{\eta} \wedge \tilde{\eta}_1 \wedge \dots \wedge \tilde{\eta}_n, r[R(x_1)]/r_1 \dots \\ & R(x_n)]/r_n, e[E(x_1)]/e_1 \dots E(x_n)]/e_n) \mid (\eta, \tilde{\eta}, r, e) \\ & \in \rho(p(x_1 \dots x_n)), \forall i = 1 \dots n. (\eta_i, \tilde{\eta}_i, r_i, e_i) \in \mathcal{A}[\![a_i]\!]_{\sigma}\} \end{aligned}$$

Intuitively, the semantics of a let-expression *let*  $x = a$  *in*  $S$  updates the current environment by associating to variable  $x$  the semantics of expression  $a$ .

The semantics of the conditional uses an auxiliary operator  $\Downarrow$  for propagating new information in the conditions. Given  $b \in BExpr$  and  $\tilde{b} \in \widetilde{BExpr}$ ,  $(\eta, \tilde{\eta}, r, e) \Downarrow_{(b, \tilde{b})} = (\eta \wedge b, \tilde{\eta} \wedge \tilde{b}, r, e)$ . The definition of  $\Downarrow$  naturally extends to sets of conditional error bounds: given  $C \subseteq \mathbb{C}$ ,  $C \Downarrow_{(b, \tilde{b})} = \bigcup_{c \in C} c \Downarrow_{(b, \tilde{b})}$ .

As already mentioned, tests in conditionals need to be treated carefully to guarantee soundness. Consider the conditional *if*  $\phi$  *then*  $S_1$  *else*  $S_2$ . Function  $\mathbf{R}_B : \widetilde{BExpr} \rightarrow BExpr$  is introduced to convert a floating-point expression to a real one, by simply replacing each operation on floating-point with the corresponding operation on reals and by applying  $\mathbf{R}$  to numerical values. The semantics of  $S_1$  and  $S_2$  are enriched with the information about the fact that real and floating-point flows match, i.e., both  $\phi$  and  $\mathbf{R}_B(\phi)$  have the same value. If real and floating point flows do not coincide, the error of taking one branch instead of the other has to be considered. For example, if  $\phi$  is satisfied but  $\mathbf{R}_B(\phi)$  is not, the *then* branch is taken in the floating point computation, but the *else* would have been taken in the real one. In this case, the error is the difference between the real value of the result of  $S_2$  and the floating point result of  $S_1$ . It is easy to show that this error is bounded by the round-off error of  $S_1$  plus the difference between the real values of  $S_1$  and  $S_2$ . The condition  $(\neg \mathbf{R}_B(\phi), \phi)$  is propagated in order to model that  $\phi$  holds but  $\mathbf{R}_B(\phi)$  does not.

The semantics of a function call, combines the conditions coming from the interpretation of the function and the ones coming from the semantics of the parameters. Variables representing real values and errors of formal parameters are replaced with the symbolic expressions coming from the semantics of the actual parameters.

The semantics of a program  $\mathcal{F} : Prog \times Env \rightarrow \mathbb{C}$  is defined as the least fixed point (lfp) of the immediate consequence operator  $\mathcal{D} : Prog \times Env \times Int \rightarrow \mathbb{C}$ , i.e.,  $\mathcal{F}[D] := \text{lfp}(\mathcal{D}[D]_{\perp Env}^{\perp Int})$ , which is defined as follows for each function symbol  $p$  defined in  $D$ :<sup>3</sup>

$$\mathcal{D}[D]_{\sigma}^{\rho}(p(x_1 \dots x_n)) := \bigcup \{ \mathcal{S}[S]_{\sigma}^{\rho} \mid p(x_1 \dots x_n) = S \in D \}.$$

The least fixed point of  $\mathcal{D}$  is guaranteed to exist by the Knaster-Tarski Fixpoint theorem. In fact, it is direct to see that  $(\mathbb{C}, \subseteq, \cup, \cap, Cond \times AExpr \times AExpr, \emptyset)$  is a complete lattice and  $\mathcal{D}$  is monotonic over  $\mathbb{C}$ , since at each iteration new conditional error bounds are added but not removed. When the program is non-recursive, this fixpoint computation converges in a finite number of steps. While this is a restrictive assumption, it is not unreasonable in avionics or embedded software, which tends to avoid recursion. However,

<sup>3</sup>It is assumed that in  $D$  there is only one declaration for each process symbol. In this case, the notation is simplified by omitting the information about the types.

in the future, the use of precise widening operators (Cousot and Cousot 1977) will be explored in order to ensure the convergence for a wider variety of programs.

In the following, two simple examples of the computation of the semantics are shown.

### EXAMPLE 3.1

Consider the following simple program.

$$g(x, y) = \text{if } x = 3 \text{ then } x \tilde{+} y \text{ else } x \tilde{*} y$$

The semantics of the program is the following, where  $r_x = R(x)$ ,  $r_y = R(y)$ ,  $e_x = E(x)$  and  $e_y = E(y)$ . The symbolic expressions  $e_+ = e_x + e_y$  and  $e_* = |r_x| * e_y + |r_y| * e_x + e_x * e_y$  indicate the propagation of the errors of  $x$  and  $y$  in the sum and multiplication.

$$\begin{aligned} \mathcal{F}[\{g(x, y)\}] = & \\ & \{ (r_x = 3, x = 3, r_x + r_y, e_+ + 1/2 \text{ulp}(|r_x + r_y| + e_+)), \\ & (r_x \neq 3, x \neq 3, r_x * r_y, e_* + 1/2 \text{ulp}(|r_x r_y| + e_*)), \\ & (r_x = 3, x \neq 3, r_x + r_y, e_* + |(r_x + r_y) - (r_x r_y)| \\ & \quad + 1/2 \text{ulp}(|r_x r_y| + e_*)), \\ & (r_x \neq 3, x = 3, r_x * r_y, e_+ + |(r_x + r_y) - (r_x r_y)| \\ & \quad + 1/2 \text{ulp}(|r_x + r_y| + e_+)) \} \end{aligned}$$

The last two elements correspond to the case in which the real flow does not correspond to the floating point-one. Here, the real and floating-point conditions do not match and the error of taking the wrong branch is taken in consideration to compute the error. For instance, consider  $x = 3$ ,  $r_x = 3.1$  and  $y = r_y = 5$ . In this case, the floating-point flow takes the *else* branch, but the real flow would take the *then* one. Thus, the error corresponding to that case will be:

$$\begin{aligned} & e_+ + |(r_x + r_y) - (r_x r_y)| + 1/2 \text{ulp}(|r_x + r_y| + e_+) \\ & = 0.1 + |8.1 - 15.5| + 1/2 \text{ulp}(8.1 + 0.1) \\ & = 7.6 + 1/2 \text{ulp}(8.2) \simeq 7.6 \end{aligned}$$

In this case, the round-off error ( $\simeq 0.1$ ) is negligible with respect to the error of taking the incorrect branch.

### EXAMPLE 3.2

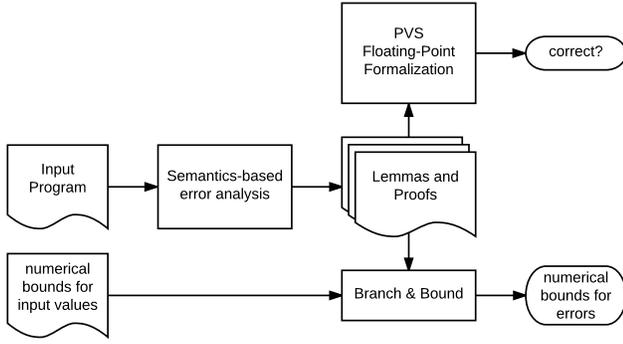
Consider the following program.

$$f(x) = \text{if } x \neq 0 \text{ then } 1 \tilde{/} x \text{ else } x$$

The semantics of the program is the following, where  $R(x) = r_x$ ,  $E(x) = e_x$ ,  $e_f = e_x / (r_x * r_x - e_x * |r_x|)$  and  $c = r_x + e_x < 0 \vee r_x - e_x > 0$ .

$$\begin{aligned} \mathcal{F}[\{f(x)\}] = & \{ (r_x = 0, x = 0, r_x, e_x), \\ & (c \wedge r_x \neq 0, x \neq 0, 1/r_x, e_f + 1/2 \text{ulp}(|1/r_x| + e_f)), \\ & (c \wedge r_x = 0, x \neq 0, r_x, e_f + 1/2 \text{ulp}(|1/r_x| + e_f) + |r_x - 1/r_x|), \end{aligned}$$

**Figure 1.** The analysis framework.



$$(c \wedge r_x \neq 0, x = 0, 1/r_x, e_x + |r_x - 1/r_x|)$$

Condition  $c$  ensures that no division by zero occurs. In this case,  $c$  contradicts the condition  $r_x = 0$  in the third element and  $x = 0$  in the fourth one. This is an expected result since in these cases, a division by zero occurs in the ideal real computation and the presented analysis cannot compute the estimation of the round-off error.

#### 4. PRECISA: An Analyzer for Estimating Certified Roundoff Errors

In this section, the prototype tool PRECISA is presented. This tool combines the semantics of Section 3 with a method to automatically generate PVS lemmas and proofs certifying that the computed bounds are correct, with respect to the IEEE-754 standard formalization defined in Section 2. In addition, a branch-and-bound technique is used to estimate a numerical enclosure for the symbolic error expressions generated.

Figure 1 depicts the functional architecture of the tool. Given an input program, the first functional block computes its semantics. For each function  $f$  in the input program, a set of conditional error bounds is generated. Each conditional error bound is then translated into a PVS lemma that states that, provided the conditions are satisfied, the floating-point value resulting from the execution of  $f$  differs from the result of  $f$  computed assuming infinite precision, by at most the round-off error approximation computed by the semantics.

The PVS translation is straightforward. The conditional error bound  $(\eta, \tilde{\eta}, r, e)$  is translated into a lemma where the hypothesis are  $\eta$  and  $\tilde{\eta}$  and the consequence states that the difference between  $r$  and the output of  $f$  using floating-point arithmetic is at most  $e$ . The following example illustrates this translation.

##### EXAMPLE 4.1

Consider the function  $g$  and its semantics defined in Example 3.1. PRECISA automatically generates the following lemmas in PVS, one for each conditional error bound in the

semantics of  $g$ .

$$\begin{aligned} &\forall r_x, r_y, e_x, e_y \in \mathbb{R}, x, y \in \mathbb{F} \\ &|x - r_x| \leq e_x \wedge |y - r_y| \leq e_y \wedge x = 3 \wedge r_x = 3 \\ &\Rightarrow |g(x, y) - (r_x + r_y)| \leq e_+ + 1/2 \text{ulp}(|r_x + r_y| + e_x + e_y) \end{aligned}$$

$$\begin{aligned} &\forall r_x, r_y, e_x, e_y \in \mathbb{R}, x, y \in \mathbb{F} \\ &|x - r_x| \leq e_x \wedge |y - r_y| \leq e_y \wedge x \neq 3 \wedge r_x \neq 3 \\ &\Rightarrow |g(x, y) - (r_x * r_y)| \leq e_* + 1/2 \text{ulp}(|r_x * r_y| + e_*) \end{aligned}$$

$$\begin{aligned} &\forall r_x, r_y, e_x, e_y \in \mathbb{R}, x, y \in \mathbb{F} \\ &|x - r_x| \leq e_x \wedge |y - r_y| \leq e_y \wedge x \neq 3 \wedge r_x = 3 \\ &\Rightarrow |g(x, y) - (r_x + r_y)| \leq e_* + |(r_x + r_y) - (r_x + r_y)| \\ &\quad + 1/2 \text{ulp}(|r_x * r_y| + e_*) \end{aligned}$$

$$\begin{aligned} &\forall r_x, r_y, e_x, e_y \in \mathbb{R}, x, y \in \mathbb{F} \\ &|x - r_x| \leq e_x \wedge |y - r_y| \leq e_y \wedge x = 3 \wedge r_x \neq 3 \\ &\Rightarrow |g(x, y) - (r_x * r_y)| \leq e_+ + |(r_x + r_y) - (r_x + r_y)| \\ &\quad + 1/2 \text{ulp}(|r_x + r_y| + e_*) \end{aligned}$$

Furthermore, for each function in the input program, PRECISA generates a lemma stating the overall round-off error of the function, independently from the chosen computational flow. In this case, the round-off error is the maximum of all the round-off errors of each flow and the hypothesis of the lemma is the disjunction of all the conditions for each conditional error bound in the semantics of the function.

This lemma guarantees the overall soundness and completeness of the method since it ensures that the round-off errors of each computational flow is correct and that all possible computational flows have been considered. If by mistake some computational flow is missing, this lemma cannot be proven. In other words, if a computational flow is missing, PVS will not be able to discharge all the possible cases for the considered function.

##### EXAMPLE 4.2

Consider the function  $g$  and its semantics defined in Example 3.1. Let  $g_{\mathbb{R}}$  be the real counterpart of  $g$ , where all the arithmetic operations are evaluated with infinite precision on real numbers. The lemma that summarizes the overall round-off error of  $g$  is the following:

$$\begin{aligned} &\forall r_x, r_y, e_x, e_y \in \mathbb{R}, x, y \in \mathbb{F} \\ &|x - r_x| \leq e_x \wedge |y - r_y| \leq e_y \\ &\Rightarrow |g(x, y) - g_{\mathbb{R}}(\mathbf{R}(x), \mathbf{R}(y))| \leq \max(e_1, e_2, e_3, e_4) \end{aligned}$$

where  $e_1 = e_+ + 1/2 \text{ulp}(|r_x + r_y| + e_x + e_y)$ ,  $e_2 = e_* + 1/2 \text{ulp}(|r_x * r_y| + e_*)$ ,  $e_3 = e_* + |(r_x + r_y) - (r_x + r_y)| + 1/2 \text{ulp}(|r_x * r_y| + e_*)$  and  $e_4 = e_+ + |(r_x + r_y) - (r_x + r_y)| + 1/2 \text{ulp}(|r_x + r_y| + e_+)$ .

Proving lemmas in PVS can be a tedious task which requires a high level of expertise to be completed properly. For this reason, along with the lemmas, the presented method also generates the corresponding proof scripts, which can be automatically verified in PVS without the need of human intervention. In particular, for each conditional error bound, which is translated to a lemma, a PVS proof script is generated using structural induction on arithmetic expressions. A proof script, when executed in PVS, yields a PVS proof. PVS proofs are internally represented as trees. Each node of this tree contains the name of the lemma to be applied and its parameters. The parameters are the real, the floating-point and the error expressions coming from the operands semantics of the arithmetic operation corresponding to the lemma. A proof subtree is generated recursively for each operand. The leaves of the tree correspond to the base cases of variables and numerical values.

The generated proof scripts are modular. This means that the proof script for a function  $f$  is generated just once, and if  $f$  is used inside another function  $g$ , the proof of  $g$  will use the proof script of  $f$  with the actual parameters, but it would not generate the whole  $f$  proof again. This feature makes proof scripts reusable, efficient, and more readable. The output of PRECISA is a PVS file containing the lemmas and their respective proof scripts. This file will be the input to the PVS theorem prover, which using the formalization described in Section 2, automatically proves the lemmas. This last steps requires no expertise or familiarity with the PVS theorem prover.

In the proposed analysis, the error expressions are kept in a symbolic form so that the method is modular and independent from the initial values of the input variables. To compute a numerical estimation of the error of a program, given some initial ranges for the input variables, an extension of the branch-and-bound algorithm defined in (Narkawicz and Muñoz 2013) is used.

This branch-and-bound algorithm relies on bounding functions for arithmetic operators. These bounding functions, which are defined using interval arithmetic, compute a correct enclosure for a real-valued arithmetic expression. The main idea is to recursively split the domain of the function into smaller subdomains and evaluate the bounds in these subdomains. The recursion stops when a precise enclosure is found, based on a given precision, or when the maximum recursion depth is reached.

In the presented approach, the real-valued arithmetic expressions are the symbolic error expressions computed by the semantics, while the initial domain is given as an input of the analysis by providing initial intervals for every input variable of the program. The algorithm of (Narkawicz and Muñoz 2013) is enhanced to support the symbolic error functions produced by the semantic analysis. The output of the algorithm is a numerical enclosure for the error expression. If the error expression is undefined for the range of

the input values, e.g., when the range of an input value includes zero and that value is used in a division, the algorithm returns an error. Similar to the original branch-and-bound algorithm, the extended branch-and-bound algorithm is specified and formally verified in PVS. This guarantees that the numerical bounds of the error expressions are sound.

## 5. Case Studies and Experimental Results

In this section, some experimental results and case studies are presented to show performances and applicability of PRECISA. The times in this section corresponds to a 2.5 GHz Intel Core i7, 16 GB of RAM, running under MacOS 10.11.3 El Capitan.

### 5.1 A simple comparison with Fluctuat

Fluctuat (Goubault and Putot 2006) is a commercial static analyzer of numerical programs based on *abstract interpretation* (Cousot and Cousot 1977). To approximate the round-off error of a program, it uses several approximations such as interval arithmetic and zonotopic abstractions (Goubault and Putot 2011) based on affine arithmetic (de Figueiredo and Stolfi 2004). In addition, to improve the precision of the computed error bounds, Fluctuat, similarly to PRECISA, uses a global optimization search that splits the initial domain in smaller domains and computes the error for each of these subdomains.

Consider the following example introduced in (Goubault and Putot 2011) where variable  $x$  has an initial value in the interval  $[0, 1]$ .

$$\begin{aligned} y(x) &= (x \tilde{-} 1) \tilde{*} (x \tilde{-} 1) \tilde{*} (x \tilde{-} 1) \tilde{*} (x \tilde{-} 1) \\ z(x) &= (x \tilde{*} x \tilde{*} x \tilde{*} x) \tilde{-} (4 \tilde{*} x \tilde{*} x \tilde{*} x) \tilde{+} (6 \tilde{*} x \tilde{*} x) \tilde{-} (4 \tilde{*} x) \tilde{+} 1 \\ t(x) &= z(x) \tilde{-} y(x) \end{aligned}$$

In (Goubault and Putot 2011), this example is analyzed with the Fluctuat tool. For the function  $t(x)$ , Fluctuat, using the interval abstraction, computes a numerical error enclosure of  $[-8, 8]$ , while using the zonotopic abstraction gets an enclosure of  $[-1.95, 1.94]$ . The much tighter enclosure  $[-2 \cdot 10^{-6}, 2 \cdot 10^{-6}]$  is obtained by using the zonotopic abstraction and the global optimization technique with 1000 domain subdivisions. In this case, according to (Goubault and Putot 2011), Fluctuat takes 78 seconds to compute the error bound in an Intel Core 2 Duo 2GHz, 4GB of RAM, running under MacOS Snow Leopard. For  $y(x)$  and  $z(x)$ , Fluctuat with zonotopic abstraction computes the error enclosures  $[-4.2 \cdot 10^{-7}, 4.2 \cdot 10^{-7}]$  and  $[-2.1 \cdot 10^{-6}, 2.1 \cdot 10^{-6}]$ , respectively.

PRECISA takes 0.01 seconds to compute the symbolic error expressions for  $y(x)$ ,  $z(x)$  and  $t(x)$ . In Table 1, the numerical enclosures for the symbolic error expressions computed by PRECISA by using the branch-and-bound algorithm are presented. The time in seconds and the number of domain splits performed by the algorithm are also reported. In this simple example, the results obtained with PRECISA

**Table 1.** Experimental results for PRECISA

fun.	round-off error enclosure	time in sec.	# splits
y(x)	$[-4.17 \cdot 10^{-7}, 4.17 \cdot 10^{-7}]$	0.078885	0
z(x)	$[-2.5 \cdot 10^{-6}, 2.5 \cdot 10^{-6}]$	0.661031	12
t(x)	$[-2.74 \cdot 10^{-6}, 2.74 \cdot 10^{-6}]$	3.361300	77

are comparable in precision with the ones obtained with Fluctuat. For the case of  $t(x)$ , PRECISA is slightly less precise than Fluctuat, where PRECISA computes the numerical enclosure in approximately 3 seconds while Fluctuat computes it in 78 seconds. This gap must be also impacted by the discrepancy in the hardware platforms used in the experiments. Furthermore, the example above should not be taken as representative of both systems. Since the authors do not have access to Fluctuat, it is, at this time, difficult to make a deeper comparison between both systems. However, this is an encouraging first result for a research tool as PRECISA. A deeper experimental evaluation is needed to assess its real potential.

## 5.2 Case study: Compact Position Reporting

The CPR (Compact Position Reporting) algorithm encodes and decodes the position of an aircraft in a compact way. CPR is used in the Automatic Dependent Surveillance-Broadcast (ADS-B) communication protocol and it was proposed to reduce to 17 the number of bits required to transmit latitude and longitude. It has been discussed within the standard organizations responsible for the ADS-B protocol, i.e., RTCA in the US and EUROCAE in Europe, that CPR is numerically unstable when implemented in single-precision floating point arithmetic, which under some circumstances could potentially lead to major numerical errors in computing the actual aircraft positions. The authors are currently conducting a formal analysis of the CPR protocol.

The CPR algorithm divides the earth surface in indexed zones. Each zone is itself divided in smaller and equally sized sections called bins. Each zone contains  $2^{17}$  bins that are enumerated from 0 to 131072. Each position on the earth surface is uniquely identified (with a precision of 5.1 meters) with a latitude zone, a latitude bin, a longitude zone and a longitude bin. The idea of the CPR encoding process is that the zone number remains constant for a long period of time, thus just the bin numbers are needed to be transmitted in the ADS-B messages. There are two kinds of encoding which are based on two different subdivisions of the earth in zones, called even and odd encoding.

In the decoding phase, if the receiver already knows the zone number and receives a bin number (encoded either in the even or in the odd way), the receiver can uniquely decode it obtaining the original coordinates of the aircraft. Otherwise, if the receiver does not know in which zone the aircraft is situated, the decoding algorithm uses a property of

the encoding algorithm that relates even and odd encoding to find the zone number. This property (roughly) states that the zone number is linearly proportional to the difference between the odd and the even bin encoding. Thus, given an even and an odd encoding for an initial coordinate, it is possible to retrieve the zone number, and then the original position.

PRECISA was used to examine the roundoff error for the following function  $lat\_zone$ , which given the odd and even encoding for a latitude coordinate, computes the zone index number. Variables  $lat_0$  and  $lat_1$  denote even and odd encodings respectively, and have initial values in  $[0, 131072]$ .

$$lat\_zone(lat_0, lat_1) = \widetilde{floor}(((59 \tilde{*} lat_0 \tilde{-} 60 \tilde{*} lat_1) \tilde{/} 131072) \tilde{+} 0.5)$$

For this function, PRECISA computes the error enclosure  $[-1.00001, 1.00001]$  in 0.36 seconds performing 19 splits on the initial domain, and assuming the original program uses single-precision numbers. The error is wide due to the use of  $\widetilde{floor}$ , which is intrinsically unstable. In fact a small perturbation on the input can provoke an error of one unit in the output. This means that, in the worst case, the CPR decoding function can return a latitude that is different from the original one by the size of one entire zone, which is approximately 600 kilometers.

## 5.3 Case study: Conflict Detection Algorithm

*CD2D* is the horizontal component of a formally verified pairwise state-based conflict detection algorithm developed at NASA (Galdino et al. 2007). The algorithm considers two aircraft, the ownship and the intruder. It uses a relative coordinate system where the intruder is the center of the system and the position and velocity of the ownship are represented by vectors  $\mathbf{s} = (s_x, s_y)$  and  $\mathbf{v} = (v_x, v_y)$ , respectively. In air traffic management, a *loss of separation* occurs when two aircraft are flying too close to each other with respect to a minimum distance. A *conflict* is defined as a predicted loss of separation within a lookahead time.

In the *CD2D* algorithm, the function  $\tau$  is used to compute the time to closest point of horizontal approach between two aircraft within a lookahead time  $T$ . It is defined as follows.

$$\tau(s_x, s_y, v_x, v_y, T) = \min(\max(0, \tilde{-}(s_x \tilde{*} v_x \tilde{+} s_y \tilde{*} v_y)), T \tilde{*} (v_x \tilde{*} v_x \tilde{+} v_y \tilde{*} v_y))$$

The initial values for the variables are set to  $s_x, s_y \in [-100, 100]$ ,  $v_x, v_y \in [-600, 600]$  and  $T = 0.083$  hours. PRECISA computes an error enclosure of  $[-7.8 \cdot 10^{-3}, 7.8 \cdot 10^{-3}]$  for  $\tau$  in 9.58 seconds by using 947 domain splits in the branch-and-bound. In (Goodloe et al. 2013), the authors manually calculated an upper bound of  $2^{-21} \simeq 4.76 \cdot 10^{-7}$  for the roundoff error of  $\tau$ , and checked it with the help of the Gappa (de Dinechin et al. 2011) tool. The numerical error enclosure computed by PRECISA is larger than the

one found in (Goodloe et al. 2013), but the latter was not computed in an automatic way and, hence, can be prone to human error.

## 6. Related work

Several different techniques can be found in the literature to estimate round-off error of floating-point computations.

Some semantics-based approaches have been proposed with this aim. In (Goubault 2001), a concrete floating-point semantics is given for basic arithmetic operations, which is then abstracted in order to give information about the possible loss of precision due to round-off errors using interval and affine arithmetic. In (Martel 2002, 2006), a family of semantics for floating-point operations is proposed. Each elementary operation is associated with a first-order term that is then combined to produce higher order error terms. In (Martel 2005), different semantics based on interval, stochastic, automatic differentiation, and error series methods are introduced and compared with the semantics in (Martel 2002, 2006).

RangeLab (Martel 2011) is a tool that determines the range of the roundoff errors for elementary arithmetic expression (sum, multiplication, division, square root). Unlike PRECISA, RangeLab is able to deal with (stable) loops by using a widening operator (Cousot and Cousot 1977). It uses interval arithmetic to approximate the error bounds. Interval arithmetic is a simple enclosure domain that performs well in several cases, but can be very imprecise, leading to excessively large over-approximations (for example, when examining subtraction). The branch-and-bound approach that is proposed in this paper is able to mitigate some of the interval analysis accuracy problems leading to more precise results. Additionally, RangeLab does not generate a certificate for the computed bounds, and it is less precise than the method proposed in this paper on the semantics of conditionals. In particular, RangeLab does not distinguish the errors that different flows produce and does not soundly handle unstable tests.

In (Ayad and Marché 2010; Boldo and Filliâtre 2007; Boldo and Marché 2011), the authors propose a methodology to formally verify C programs annotated with information about the maximum floating-point error allowed. Verification conditions (VCs) are automatically generated by using Caduceus (Filliâtre and Marché 2004) or FramaC (Kirchner et al. 2015) and then discharged either in Coq (Bertot and Castéran 2004), Gappa (de Dinechin et al. 2011) or by means of a suitable SMT solver. In (Marché 2014), the aforementioned method is applied to an industrial case study, whilst PRECISA being still a proof of concept implementation has not been applied yet to a program of comparable size. The empirical study in (Marché 2014) highlights the fact that specifying annotations and proving properties about floating-point numbers on a complex C program requires high level of expertise on floating-point com-

putations and interactive theorem proving. On the contrary, PRECISA does not require such level of expertise since it automatically provides a certificate for the soundness of the error bounds.

In (Darulova and Kuncak 2014), the developer writes the program with real numbers in mind in a functional subset of Scala, and then specifies the desired precision for the computations. The proposed tool automatically finds (if it exists) the finite-precision format needed to meet the precision requirements. Then, the input program is compiled in one that uses the computed finite precision. To do so, for each supported finite precision representation (fixed or floating-point, extended, double or single precision), a set of VCs over reals is automatically generated and proved in Z3 (de Moura and Bjørner 2008) by using a novel combination of SMT solvers with affine and interval arithmetic. The least restrictive representation that satisfies the precision specification is chosen. Several approximations on the program are applied in order to meet the limitation of Z3, which for complex functions is not able to give an answer in reasonable time.

Fluctuat (Goubault and Putot 2006) is a commercial analyzer of numerical programs based on *abstract interpretation* (Cousot and Cousot 1977). It can be considered as the state of the art tool for static analysis of floating-point error bounds. Fluctuat accepts as input a C (or ADA) program with annotations about input bound and uncertainties, and it produces bounds for the round-off error of the program expressions decomposed with respect to its provenance. PRECISA shares nice features with Fluctuat such as the sound treatment of unstable tests, the modularity of the analysis, and the automation in the generation of error bounds. There are, nevertheless, some differences between these approaches. On the one hand, Fluctuat provides support for iterative programs and uses a zonotopic abstract domain (Goubault and Putot 2011) that is based on affine arithmetic (de Figueiredo and Stolfi 2004) which is shown to be more precise than interval arithmetic. On the other hand, in Fluctuat no certificate is provided for the computed bounds, thus, trusting the results of the analysis implies trusting the implementation of the tool.

Gappa (de Dinechin et al. 2011) automatically computes enclosures for floating-point expression by using interval arithmetic, and it generates a proof of the results that can be checked in the Coq proof assistant. The main drawback of this approach is that it requires some level of expertise from the user, e.g., hints on the proof must be provided to Coq, and, furthermore, the Gappa specification corresponding to the C program to verify is not automatically generated. In the presented work, these problems are addressed by implementing a fully-automatic analysis that automatically generate bounds for the floating-point computations and the correspondent proofs that can be discharged directly in PVS without assistance from the user.

In (Ramananandro et al. 2016), the authors developed VCFloat: a framework to automatically compute round-off error terms of C expressions with the correspondent Coq correctness proof. Similar to the current approach, VCFloat uses interval arithmetic to approximate the error bounds and generates validity conditions on the expressions. In order to obtain a more precise analysis, the approach presented in this paper additionally accumulates path conditions, generating different bounds depending on the control flow. Furthermore, (Ramananandro et al. 2016) computes the ulp by using the maximum exponent allowed in the floating-point representation, while the approach presented here computes the actual exponent for the maximum absolute value in the expression bounds. In this way, tighter bounds are obtained.

The tool FPTaylor (Solovyev et al. 2015) expresses floating-point expressions by means of Symbolic Taylor Expansions and applies a branch-and-bound global optimization technique to obtain tight bounds for round-off errors. In addition, FPTaylor emits certificates for HOL Light (Harrison 2009). Unlike PRECISA, it provides support for trigonometric functions. However, FPTaylor does not handle conditionals and it is not able to deal with discontinuous functions such as absolute value or floor.

## 7. Conclusion and future work

In this paper, PRECISA, a static analyzer for estimating round-off errors of floating-point computations is defined. The presented analyzer is a prototype tool which is fully automatic and generates PVS certificates that ensure the correctness of the error bounds with respect to the floating-point IEEE-754 standard. That is, the computed estimations are always sound over-approximations of the possible round-off error that can occur in the program.

The analysis is defined in a compositional way, which is a highly desirable characteristic of an analyzer, since it enables an incremental, modular, and efficient analysis. This makes the presented tool suitable to be used also with partial programs.

PRECISA is parametric with respect to the chosen floating-point precision (single and double are implemented at the moment) and it supports all the to-the-nearest rounding modalities introduced in the IEEE-754 standard. In the future, in order to improve the precision of the round-off error approximation, PRECISA will also support mixed floating-point precision and integer numbers and operations.

An interesting advantage of the analysis presented in this work is the sound treatment of unstable tests. In the literature, the *stable test hypothesis* is widely used to deal with this problem. However, this hypothesis may yield unsound results when the real flow does not correspond to the floating-point one. To the best of the authors' knowledge, the only other technique which is sound with respect to unstable tests is the one presented in (Goubault and Putot 2013) for the Fluctuat analyzer.

In the implementation of PRECISA, the semantics-based analysis and the PVS floating-point formalization are completely independent from the numerical evaluation of the error expression. This means that different techniques can be used at this point for the evaluation, depending on the type of the considered expression and on the desired precision/efficiency trade-off.

In this work, a branch-and-bound algorithm based on interval arithmetic is proposed to evaluate the symbolic error expression. Interval arithmetic is a simple enclosure domain which performs well in several cases, but can lead to excessively large over-approximation in others (for example subtraction). Global search techniques, like branch-and-bound, help mitigate these problems, leading to tighter enclosures. Other enclosure domains have been proposed in the literature, such as affine arithmetics (de Figueiredo and Stolfi 2004), Polyhedra (Cousot and Halbwachs 1978), and Bernstein polynomials (Lorentz 1986) which, in some cases, can lead to more accurate results or can converge in a fewer number of steps compared to interval arithmetic. The authors plan to instantiate the generic branch-and-bound algorithm of (Narkawicz and Muñoz 2013) with specific functions for the aforementioned domains, in order to obtain better approximations.

In the future, the branch-and-bound algorithm will also be enhanced in order to consider the conditions generated by the semantics. The main idea behind this improvement is that the subdomains that do not meet the conditions have to be discarded and they do not have to participate in the final computation of the error enclosure. Thus, the final result is obtained by propagating the information coming from the intervals that satisfy the conditions. Only in the case in which the branch-and-bound finds an error, i.e., a division by 0, that is not explicitly avoided by the conditions will it propagate the error to the final results. The output of the algorithm will be an enclosure for the error expression and the intervals in which this enclosure holds. The use of constraint programming will also be explored to simplify the input of the branch-and-bound algorithm. By using these techniques, an improvement in the efficiency is expected, since the input of the branch-and-bound will be a simplified expression.

The main drawback of the presented approach is that it generates exceedingly large certificates for programs with nested conditionals. In fact, the number of conditional error bounds grows exponentially in these cases. Due to the unstable tests handling, four different conditional error bounds are generated for each conditional. In order to deal with this problem, abstraction on the domain  $\mathbb{C}$  should be considered in the future. In this way, the number of elements in the semantics will be reduced and consequently also the size of the generated PVS certificate.

The support of recursion and loops will also be considered by defining abstractions on the domain of conditional

error bounds and widening operators on these domains. Another interesting future direction is to use the ideas of this paper to automatically generate ACSL annotations about round-off errors for C programs. The annotated program could then be automatically verified in a tool like FramaC. Finally, PRECISA will be extended to support a larger set of functions, such as trigonometric functions, which are widely used in safety-critical systems of interest to NASA.

## References

- A. Ayad and C. Marché. Multi-prover verification of floating-point programs. In *Proceedings of the 5th International Joint Conference on Automated Reasoning, IJCAR 2010*, volume 6173 of *Lecture Notes in Computer Science*, pages 127–141. Springer, 2010. doi: 10.1007/978-3-642-14203-1\_11. URL [http://dx.doi.org/10.1007/978-3-642-14203-1\\_11](http://dx.doi.org/10.1007/978-3-642-14203-1_11).
- Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. ISBN 978-3-642-05880-6. doi: 10.1007/978-3-662-07964-5. URL <http://dx.doi.org/10.1007/978-3-662-07964-5>.
- S. Boldo and J. Filliâtre. Formal verification of floating-point programs. In *18th IEEE Symposium on Computer Arithmetic (ARITH-18 2007)*, pages 187–194. IEEE Computer Society, 2007. doi: 10.1109/ARITH.2007.20. URL <http://dx.doi.org/10.1109/ARITH.2007.20>.
- S. Boldo and C. Marché. Formal verification of numerical programs: From C annotated programs to mechanical proofs. *Mathematics in Computer Science*, 5(4):377–393, 2011. doi: 10.1007/s11786-011-0099-9. URL <http://dx.doi.org/10.1007/s11786-011-0099-9>.
- S. Boldo and C. A. Muñoz. A High-Level Formalization of Floating-Point Numbers in PVS. Technical Report CR-2006-214298, NASA, 2006.
- P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, POPL 1977*, pages 238–252. ACM, 1977. doi: 10.1145/512950.512973. URL <http://doi.acm.org/10.1145/512950.512973>.
- P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proceedings of Fifth ACM Symp. Principles of Programming Languages*, pages 84–96, 1978.
- E. Darulova and V. Kuncak. Sound compilation of reals. In *Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014*, pages 235–248. ACM, 2014. doi: 10.1145/2535838.2535874. URL <http://doi.acm.org/10.1145/2535838.2535874>.
- F. de Dinechin, C. Lauter, and G. Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Trans. Computers*, 60(2):242–253, 2011. doi: 10.1109/TC.2010.128. URL <http://dx.doi.org/10.1109/TC.2010.128>.
- L. H. de Figueiredo and J. Stolfi. Affine arithmetic: Concepts and applications. *Numerical Algorithms*, 37(1-4):147–158, 2004. doi: 10.1023/B:NUMA.0000049462.70970.b6. URL <http://dx.doi.org/10.1023/B:NUMA.0000049462.70970.b6>.
- L. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3\_24. URL [http://dx.doi.org/10.1007/978-3-540-78800-3\\_24](http://dx.doi.org/10.1007/978-3-540-78800-3_24).
- J. C. Filliâtre and C. Marché. Multi-prover verification of C programs. In *Proceedings of the 6th International Conference on Formal Engineering Methods, ICFEM 2004*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2004. doi: 10.1007/978-3-540-30482-1\_10. URL [http://dx.doi.org/10.1007/978-3-540-30482-1\\_10](http://dx.doi.org/10.1007/978-3-540-30482-1_10).
- A. Galdino, C. Muñoz, and M. Ayala. Formal verification of an optimal air traffic conflict resolution and recovery algorithm. In D. Leivant and R. de Queiroz, editors, *Proceedings of the 14th Workshop on Logic, Language, Information and Computation*, volume 4576 of *Lecture Notes in Computer Science*, pages 177–188, Rio de Janeiro, Brazil, July 2007. Springer-Verlag.
- A. Goodloe, C. A. Muñoz, F. Kirchner, and L. Correnson. Verification of numerical programs: From real numbers to floating point numbers. In *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, volume 7871 of *Lecture Notes in Computer Science*, pages 441–446. Springer, 2013. doi: 10.1007/978-3-642-38088-4\_31. URL [http://dx.doi.org/10.1007/978-3-642-38088-4\\_31](http://dx.doi.org/10.1007/978-3-642-38088-4_31).
- E. Goubault. Static analyses of the precision of floating-point operations. In *Proceedings of the 8th International Symposium on Static Analysis, SAS 2001*, volume 2126 of *Lecture Notes in Computer Science*, pages 234–259. Springer, 2001. doi: 10.1007/3-540-47764-0\_14. URL [http://dx.doi.org/10.1007/3-540-47764-0\\_14](http://dx.doi.org/10.1007/3-540-47764-0_14).
- E. Goubault and S. Putot. Static analysis of numerical algorithms. In *Proceedings of the 13th International Symposium on Static Analysis, SAS 2006*, volume 4134 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2006. doi: 10.1007/11823230\_3. URL [http://dx.doi.org/10.1007/11823230\\_3](http://dx.doi.org/10.1007/11823230_3).
- E. Goubault and S. Putot. Static analysis of finite precision computations. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2011*, volume 6538 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 2011. doi: 10.1007/978-3-642-18275-4\_17. URL [http://dx.doi.org/10.1007/978-3-642-18275-4\\_17](http://dx.doi.org/10.1007/978-3-642-18275-4_17).
- E. Goubault and S. Putot. Robustness analysis of finite precision implementations. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems APLAS 2013*, volume 8301 of *Lecture Notes in Computer Science*, pages 50–57. Springer, 2013. doi: 10.1007/978-3-319-03542-0\_4. URL [http://dx.doi.org/10.1007/978-3-319-03542-0\\_4](http://dx.doi.org/10.1007/978-3-319-03542-0_4).
- J. Harrison. A machine-checked theory of floating point arithmetic. In *Proceedings of the 12th International Conference on Theorem*

- Proving in Higher Order Logics*, TPHOLS '99, pages 113–130. Springer-Verlag, 1999. ISBN 3-540-66463-7. URL <http://dl.acm.org/citation.cfm?id=646526.694890>.
- J. Harrison. HOL light: An overview. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLS 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66. Springer, 2009. doi: 10.1007/978-3-642-03359-9\_4. URL [http://dx.doi.org/10.1007/978-3-642-03359-9\\_4](http://dx.doi.org/10.1007/978-3-642-03359-9_4).
- F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015. doi: 10.1007/s00165-014-0326-7. URL <http://dx.doi.org/10.1007/s00165-014-0326-7>.
- G. G. Lorentz. *Bernstein Polynomials*. Chelsea Publishing Company, 1986.
- C. Marché. Verification of the functional behavior of a floating-point program: An industrial case study. *Science of Computer Programming*, 96:279–296, 2014. doi: 10.1016/j.scico.2014.04.003. URL <http://dx.doi.org/10.1016/j.scico.2014.04.003>.
- M. Martel. Propagation of roundoff errors in finite precision computations: A semantics approach. In *Proceedings of the 11th European Symposium on Programming Languages and Systems, ESOP 2002*, volume 2305 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2002. doi: 10.1007/3-540-45927-8\_14. URL [http://dx.doi.org/10.1007/3-540-45927-8\\_14](http://dx.doi.org/10.1007/3-540-45927-8_14).
- M. Martel. An overview of semantics for the validation of numerical programs. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2005*, volume 3385 of *Lecture Notes in Computer Science*, pages 59–77. Springer, 2005. ISBN 3-540-24297-X. doi: 10.1007/978-3-540-30579-8\_4. URL [http://dx.doi.org/10.1007/978-3-540-30579-8\\_4](http://dx.doi.org/10.1007/978-3-540-30579-8_4).
- M. Martel. Semantics of roundoff error propagation in finite precision calculations. *Higher-Order and Symbolic Computation*, 19(1):7–30, 2006. URL <http://dx.doi.org/10.1007/s10990-006-8608-2>.
- M. Martel. RangeLab: A static-analyzer to bound the accuracy of finite-precision computations. In *13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2011*, pages 118–122. IEEE Computer Society, 2011. doi: 10.1109/SYNASC.2011.52. URL <http://dx.doi.org/10.1109/SYNASC.2011.52>.
- A. Narkawicz and C. A. Muñoz. A formally verified generic branching algorithm for global optimization. In *Revised Selected Papers of the 5th International Conference on Verified Software: Theories, Tools, Experiments, VSTTE 2013*, volume 8164 of *Lecture Notes in Computer Science*, pages 326–343. Springer, 2013. doi: 10.1007/978-3-642-54108-7\_17. URL [http://dx.doi.org/10.1007/978-3-642-54108-7\\_17](http://dx.doi.org/10.1007/978-3-642-54108-7_17).
- S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.
- T. Ramanandro, P. Mountcastle, B. Meister, and R. Lethin. A unified coq framework for verifying C programs with floating-point computations. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2016*, pages 15–26. ACM, 2016. doi: 10.1145/2854065.2854066. URL <http://doi.acm.org/10.1145/2854065.2854066>.
- A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *Proceedings of the 20th International Symposium on Formal Methods, FM 2015*, volume 9109 of *Lecture Notes in Computer Science*, pages 532–550. Springer, 2015. doi: 10.1007/978-3-319-19249-9\_33. URL [http://dx.doi.org/10.1007/978-3-319-19249-9\\_33](http://dx.doi.org/10.1007/978-3-319-19249-9_33).
- D. Stevenson. A proposed standard for binary floating-point arithmetic. *IEEE Computer*, 14(3):51–62, 1981. doi: 10.1109/C-M.1981.220377. URL <http://dx.doi.org/10.1109/C-M.1981.220377>.