# Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis

Mariano Moscato[1], Laura Titolo[1⋆], Aaron Dutle[2], and César A. Muñoz[2]

[1] National Institute of Aerospace,
{mariano.moscato,laura.titolo}@nianet.org
[2] NASA Langley Research Center,
{aaron.m.dutle,cesar.a.munoz}@nasa.gov

**Abstract.** This paper introduces a static analysis technique for computing formally verified round-off error bounds of floating-point functional expressions. The technique is based on a denotational semantics that computes a symbolic estimation of floating-point round-off errors along with a proof certificate that ensures its correctness. The symbolic estimation can be evaluated on concrete inputs using rigorous enclosure methods to produce formally verified numerical error bounds. The proposed technique is implemented in the prototype research tool PRECiSA (Program Round-off Error Certifier via Static Analysis) and used in the verification of floating-point programs of interest to NASA.

## 1 Introduction

Floating-point arithmetic is the most commonly used representation of real arithmetic in computer programs. One significant problem of floating-point arithmetic is the presence of round-off errors that can make a numerical computation significantly different from the actual real arithmetic computation. These errors are especially problematic in safety-critical applications such as aerospace and avionics software, where even small computation errors can lead to catastrophic consequences. Having a correct and externally verifiable estimation of how close a computed result is to the ideal real number computation is fundamental to the safety analysis of such systems.

This paper presents a modular static analysis technique for computing *provably sound* over-approximations of floating-point round-off errors. Given a set of functions over floating-point values, symbolic upper bounds on the round-off error of these functions are automatically computed by using a denotational semantics framework. Additionally, proof certificates assuring the correctness of such bounds are also generated. The main features of the proposed technique are: (1) automatic generation of proof certificates that provide an externally verifiable guarantee that the computed error estimations are correct; (2) modularity and reusability, due to being defined by a compositional denotational semantics

⋆ Research by the first two authors was supported by the National Aeronautics and Space Administration under NASA/NIA Cooperative Agreement NNL09AA00A.

that symbolically models the accumulation of floating point round-off errors in functional expressions; (3) correctly handling of conditional expressions, i.e., the *stable test hypothesis* is not assumed in conditional if-then-else expressions where the logical value of the condition is compromised by round-off errors; (4) extensibility, i.e., new floating-point operations can be integrated into the denotational semantic framework assuming they satisfy some basic properties; and (5) computation of accurate round-off errors via a generic branch-and-bound algorithm that supports several rigorous enclosure methods, e.g., interval arithmetic, affine arithmetic [7], and Bernstein polynomials [14].

The static analysis presented in this paper has been implemented in a prototype tool called PRECiSA (Program Round-off Error Certifier via Static Analysis). The current implementation of PRECiSA uses SRI's Prototype Verification System (PVS) [21], but the theoretical framework presented in this paper can be implemented in any modern interactive proof assistant. PRECiSA accepts as input a program composed of a set of functional floating-point expressions. The output of the tool is a PVS theory that consists of a set of lemmas stating accumulated round-off error estimations for each function in the program. These lemmas are equipped with PVS proof scripts that *automatically* discharge them. When numerical values for the input variables appearing in the program are provided, PRECiSA also generates PVS lemmas stating concrete numerical bounds on the round-off errors, along with corresponding proof scripts to discarge them without user intervention. PRECiSA is publicly available under NASA's Open Source Agreement[3] and can be used, without installation, through a web interface[4].

The paper is organized as follows. A formalization of floating-point round-off errors is presented in Section 2. This formalization enables the generation of proof certificates and the computation of provably correct bounds. In Section 3, a compositional denotational semantics modeling the accumulation of floating-point round-off errors is defined. This semantics is the core of the proposed analysis and it computes a symbolic over-approximation of the round-off error of a given function, along with a proof certificate ensuring its correctness. PRECiSA, an implementation of the proposed analysis, is presented in Section 4. This implementation is illustrated with an example taken from a verification effort at NASA. Experimental results and comparison to similar tools are shown in Section 5. Related work is discussed in Section 6.

## 2 Formalization of Floating-Point Round-off Errors

The NASA PVS Library[5] includes two formalizations of floating-point numbers: a hardware-level model of the IEEE-854 floating-point standard [16] and high-level model of the IEEE-754 standard [1]. These formalizations are related by functions that translate from one representation into the other. In the high-level

[3] https://github.com/nasa/PRECiSA.
[4] http://precisa.nianet.org.
[5] https://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library

model, a *floating-point number*, or simply a *float*, is defined as a pair of integers $(m, \mathsf{e})$, where $m$ is called the *significand* and $\mathsf{e}$ the *exponent* of the float. A conversion function $\mathbf{R} : \mathbb{F} \mapsto \mathbb{R}$ is defined to refer to the real number represented by a given float, i.e., $\mathbf{R}((m, \mathsf{e})) = m \cdot \beta^{\mathsf{e}}$, where $\beta \in \mathbb{N}$ is called the *base* or *radix* of the system. IEEE-754 *formats*, e.g., binary single and double precisions, can be defined in this formalization by instantiating specific theory parameters. As this representation is redundant, notions about normality and canonicity are also defined (see [1] for details). By abuse of notation, $\tilde{v}$ will be used to represent a floating-point number in $\mathbb{F}$ and its real value $\mathbf{R}(\tilde{v})$.

Since not every real number can be exactly represented by a float, a notion of representation error is defined as follows. Let $\tilde{v}$ be a floating-point number that represents a real number $r$, the difference $|\tilde{v} - r|$ is called the *round-off error* (or *rounding error*) of $\tilde{v}$ with respect to $r$. The *closest* floating-point to $r$, denoted $\mathbf{F}(r)$, is defined as a floating-point number for which the round-off error with respect to $r$ is minimal. In cases where this float is not unique, the IEEE-754 standard defines several *rounding modes* such as the *round-ties-to-even* mode, where the float with even significand is chosen, and the *round-ties-to-away* mode, where the float with the greater absolute value is chosen.

The *unit in the last place* (*ulp*) is a measure of the precision of a floating-point number $\tilde{v}$ as a representation of a real number. It can be defined as $\mathsf{ulp}(\tilde{v}) = \beta^{\mathsf{e}_{\tilde{v}}}$, where $\mathsf{e}_{\tilde{v}}$ is the exponent of the canonical form of $\tilde{v}$. Note that the canonical form of a given float depends on the format being used (single precision, double precision, etc.). Then, the $\mathsf{ulp}$ also depends on the format. The $\mathsf{ulp}$ of a floating point can be used as a bound of the round-off error since, as shown in [1], if $\tilde{v}$ is the closest representation of some real $r$, the two numbers are apart from each other for no more than half of the $\mathsf{ulp}$ of $\tilde{v}$. The $\mathsf{ulp}$ of a real number is defined as the $\mathsf{ulp}$ of the canonical form of its closest floating-point representation, i.e., $\mathsf{ulp}(r) = \mathsf{ulp}(\mathbf{F}(r))$. Then, the previous bound can be stated as follows [11].

$$|\tilde{v} - r| \leq \tfrac{1}{2}\,\mathsf{ulp}(r). \tag{2.1}$$

The work presented in this paper extends the high-level model with a formalization of round-off errors of floating-point expressions $\widetilde{\mathsf{op}}(\tilde{v}_1, \ldots, \tilde{v}_n)$ with respect to a real-valued expression $\mathsf{op}(r_1, \ldots, r_n)$, where $\widetilde{\mathsf{op}}$ is a floating-point operator representing a real-valued operator $\mathsf{op}$ and $\tilde{v}_i$ is a floating-point value representing a real value $r_i$, for $1 \leq i \leq n$. For that purpose, it is necessary to consider: (a) the error introduced by the application of $\widetilde{\mathsf{op}}$ versus $\mathsf{op}$ and (b) the propagation of the errors carried out by the arguments, i.e., the difference between $\tilde{v}_i$ and $r_i$, for $1 \leq i \leq n$, in the application. In the case of arithmetic operators, the IEEE-754 standard states that every operation should be performed as if it would be calculated with infinite precision and then rounded to the nearest floating-point value. Then, from Formula (2.1), the application of an $n$-ary floating-point operator $\widetilde{\mathsf{op}}$ to the floating-point values $\tilde{v}_1, \ldots, \tilde{v}_n$ must fulfill the following condition.

$$|\widetilde{\mathsf{op}}(\tilde{v}_i)_{i=1}^{n} - \mathsf{op}(\tilde{v}_i)_{i=1}^{n}| \leq \tfrac{1}{2}\,\mathsf{ulp}(\mathsf{op}(\tilde{v}_i)_{i=1}^{n}), \tag{2.2}$$

where the notation $f(x_i)_{i=1}^n$ is used to represent $f(x_1, \ldots, x_n)$.

To estimate how the errors of the arguments are propagated to the result of the application of the operator, it is necessary to bound the difference between the application of the real operator on real values and the application of the same operator on the floating-point arguments. The expression $\epsilon_{\mathsf{op}}(e_i)_{i=1}^n$ is used to represent such difference, where each $e_i$ is a bound of the round-off error carried by every floating point $\tilde{v}_i$ representing a real value $r_i$, i.e., $|\tilde{v}_i - r_i| \leq e_i$. Therefore, $\epsilon_{\mathsf{op}}(e_i)_{i=1}^n$ satisfies the following condition.

$$|\mathsf{op}(\tilde{v}_i)_{i=1}^n - \mathsf{op}(r_i)_{i=1}^n| \leq \epsilon_{\mathsf{op}}(e_i)_{i=1}^n. \tag{2.3}$$

The following bound of the round-off error between the floating-point expression and the real-valued counterpart follows from Formula (2.2), Formula (2.3), the triangle inequality, and the fact that $\mathsf{ulp}$ is monotonically increasing on non-negative inputs [1].

$$|\widetilde{\mathsf{op}}(\tilde{v}_i)_{i=1}^n - \mathsf{op}(r_i)_{i=1}^n| \leq \varepsilon_{\widetilde{\mathsf{op}}}(r_i, e_i)_{i=1}^n, \tag{2.4}$$

where $\varepsilon_{\widetilde{\mathsf{op}}}(r_i, e_i)_{i=1}^n = \epsilon_{\mathsf{op}}(e_i)_{i=1}^n + \frac{1}{2}\,\mathsf{ulp}(\upsilon(r_i, e_i)_{i=1}^n)$ and $\upsilon(r_i, e_i)_{i=1}^n$ is a real-valued expression that satisfies $|\mathsf{op}(\tilde{v}_i)_{i=1}^n| \leq \upsilon(r_i, e_i)_{i=1}^n$.

Additional restrictions on the variables in Formula (2.4) are needed when the operators are not total. For example, when dealing with the division operation, it is necessary to guarantee that the second argument of both the floating-point operator and the real-valued operator is not zero. The expressions $\eta_{\mathsf{op}}(r_i)_{i=1}^n$ and $\eta_{\widetilde{\mathsf{op}}}(\tilde{v}_i)_{i=1}^n$ will be used to represent any such conditions on the arguments of the operators.

In this work, the operators $\widetilde{\mathsf{op}}$ and $\mathsf{op}$ in Formula (2.4) are generic. They can be instantiated with any floating point operation and its real counterpart as long as Formula (2.4) holds for all $\tilde{v}_1, \ldots, \tilde{v}_n \in \mathbb{F}, r_i, \ldots, r_n \in \mathbb{R}, e_1, \ldots, e_n \in \mathbb{R}_{\geq 0}$, when $|\tilde{v}_i - r_i| \leq e_i$ with $1 \leq i \leq n$, $\eta_{\mathsf{op}}(r_i)_{i=1}^n$, and $\eta_{\widetilde{\mathsf{op}}}(\tilde{v}_i)_{i=1}^n$. Some examples of round-off error approximation functions for arithmetic operators are presented below. It is worth noting how the additional constraints are used in the division and in the square root to guarantee the validity of the output, and in the subtraction and arctangent to improve the precision of the error approximation. For example, as mentioned in [1], the floating point subtraction $\tilde{v}_1 \mathbin{\tilde{-}} \tilde{v}_2$ can be exactly computed when $\tilde{v}_2 \mathbin{\tilde{/}} 2 \leq x \leq 2 \mathbin{\tilde{*}} \tilde{v}_2$. This property is captured by the error approximation function $\varepsilon_{\tilde{-}'}$ and corresponding constraint $\eta_{\tilde{-}'}$ shown below.

- $\varepsilon_{\tilde{+}}(r_1, e_1, r_2, e_2) := e_1 + e_2 + 1/2\,\mathsf{ulp}(|r_1 + r_2| + e_1 + e_2)$.
- $\varepsilon_{\tilde{-}}(r_1, e_1, r_2, e_2) := e_1 + e_2 + 1/2\,\mathsf{ulp}(|r_1 - r_2| + e_1 + e_2)$, $\eta_{\tilde{-}}(\tilde{v}_1, \tilde{v}_2) := (\tilde{v}_2/2 > \tilde{v}_1) \vee (\tilde{v}_1 > 2\tilde{v}_2)$.
- $\varepsilon_{\tilde{-}'}(r_1, e_1, r_2, e_2) := e_1 + e_2$, $\eta_{\tilde{-}'}(\tilde{v}_1, \tilde{v}_2) := (\tilde{v}_2/2 \leq \tilde{v}_1 \leq 2\tilde{v}_2)$.
- $\varepsilon_{\tilde{*}}(r_1, e_1, r_2, e_2) := |r_1|e_2 + |r_2|e_1 + e_1 e_2 + 1/2\,\mathsf{ulp}((|r_1| + e_1)(|r_2| + e_2))$.
- $\varepsilon_{\tilde{/}}(r_1, e_1, r_2, e_2) := \frac{|r_1|e_2 + |r_2|e_1}{r_2 r_2 - e_2|r_2|} + 1/2\mathsf{ulp}\left(\frac{|r_1| + e_1}{|r_2| - e_2}\right)$, $\eta_{/}(r_1, r_2) := (r_2 \neq 0)$, and $\eta_{\tilde{/}}(\tilde{v}_1, \tilde{v}_2) := (\tilde{v}_2 \neq 0)$.
- $\varepsilon_{\tilde{-}}(r, e) := e$.
- $\varepsilon_{\widetilde{abs}}(r, e) := e$.
- $\varepsilon_{\widetilde{floor}}(r, e) := e + max(\lfloor r \rfloor - \lfloor r - e \rfloor, \lfloor r \rfloor - \lfloor r + e \rfloor) + 1/2\,\mathsf{ulp}(|\lfloor r \rfloor| + e)$.
- $\varepsilon_{\widetilde{sqrt}}(r, e) := \sqrt{e} + 1/2\,\mathsf{ulp}(\sqrt{r + e})$, $\eta_{sqrt}(r) := (r \geq 0)$, and $\eta_{\widetilde{sqrt}}(\tilde{v}) := (\tilde{v} \geq 0)$.

– $\varepsilon_{\widetilde{sin}}(r,e) := min(2,e) + 1/2\,\mathsf{ulp}(|sin(r)| + min(2,e))$.
– $\varepsilon_{\widetilde{cos}}(r,e) := min(2,e) + 1/2\,\mathsf{ulp}(|cos(r)| + min(2,e))$.
– $\varepsilon_{\widetilde{atan}}(r,e) := e + 1/2\,\mathsf{ulp}(atan(|r| + e))$, $\eta_{atan}(r,e) := (|r| \leq e)$.
– $\varepsilon_{\widetilde{atan}'}(r,e) := \frac{e}{min((r-e)^2,(r+e)^2)} + \frac{1}{2}\,\mathsf{ulp}(atan(|r| + e))$,$\eta_{atan'}(r,e) := (|r| > e)$.

The fact that the previous definitions satisfy Formula (2.4) is formally proven in PVS and the proofs are electronically available in the NASA PVS Library.

## 3 Denotational Semantics

In this section, a denotational semantics for a declarative expression language that relies on the floating-point formalization presented in Section 2 is defined. This semantics computes a symbolic expression representing the round-off error of the program and collects the information needed to provide a certificate that guarantees its soundness.

In the following, the sets of arithmetic and boolean expressions over reals are denoted as $\mathbb{A}$ and $\mathbb{B}$, respectively. The floating point counterparts of $\mathbb{A}$ and $\mathbb{B}$ are denoted as $\widetilde{\mathbb{A}}$ and $\widetilde{\mathbb{B}}$, respectively. The expression language considered in this paper contains conditionals, let expressions, and function calls. Given a set $\Omega$ of pre-defined arithmetic floating-point operations, a set $\Sigma$ of function symbols, and a denumerable set $\mathbb{V}$ of variables, $\widetilde{\mathbb{E}}$ denotes the set of program expressions, which syntax is given by the following grammar.

$$\widetilde{A} ::= \tilde{k} \mid x \mid \widetilde{\mathsf{op}}(\widetilde{A},\ldots,\widetilde{A}) \quad \widetilde{B} ::= \mathit{true} \mid \mathit{false} \mid \widetilde{B} \wedge \widetilde{B} \mid \widetilde{B} \vee \widetilde{B} \mid \neg\widetilde{B} \mid \widetilde{A} < \widetilde{A} \mid \widetilde{A} = \widetilde{A}$$
$$\widetilde{E} ::= \widetilde{A} \mid \mathit{if}\ \widetilde{B}\ \mathit{then}\ \widetilde{E}\ \mathit{else}\ \widetilde{E} \mid \mathit{let}\ x = \widetilde{A}\ \mathit{in}\ \widetilde{E} \mid \tilde{f}(\widetilde{A},\ldots,\widetilde{A})$$

where $\widetilde{A} \in \widetilde{\mathbb{A}}$, $\widetilde{B} \in \widetilde{\mathbb{B}}$, $\widetilde{E} \in \widetilde{\mathbb{E}}$, $\tilde{k} \in \mathbb{F}$, $x \in \mathbb{V}$, $\widetilde{\mathsf{op}} \in \Omega$, and $\tilde{f} \in \Sigma$.

A program is defined as a set of *function declarations* of the form $\tilde{f}(x_1,\ldots,x_n) = \widetilde{E}$, where $x_1,\ldots,x_n$ are pairwise distinct variables in $\mathbb{V}$ and all free variables appearing in $\widetilde{E}$ are in $\{x_1,\ldots,x_n\}$. The natural number $n$ is called the *arity* of $\tilde{f}$. Henceforth, it is assumed that programs are well-formed in the sense that for every function call $\tilde{f}(x_1,\ldots,x_n)$ that occurs in a program $\widetilde{P}$, a unique function $\tilde{f}$ of arity $n$ is defined in $\widetilde{P}$. The set of programs is denoted as $\widetilde{\mathbb{P}}$.

The proposed semantics collects, for each program path, the corresponding path conditions (for both the real and the floating-point flow), 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 accumulated round-off error that the result might include due to floating-point operations. Furthermore, the semantics computes a symbolic proof of the correctness of the computed round-off error. The set of symbolic proofs that can be generated by the semantics is denoted by $\Pi$. The previous information is stored in a *conditional error bound*, which is a tuple on the form $(\eta,\tilde{\eta},r,e,\pi)$ where $\eta \in \mathbb{B}$, $\tilde{\eta} \in \widetilde{\mathbb{B}}$, $r,e \in \mathbb{A}$, $\pi \in \Pi$, and such that $\eta \neq \mathit{false}$ and $\tilde{\eta} \neq \mathit{false}$. Intuitively, $(\eta,\tilde{\eta},r,e,\pi)$ means that if the conditions $\eta$ and $\tilde{\eta}$ are true, then the output of the ideal real numbers implementation of the program is $r$, and $\pi$ is a formal proof that the round-off error of the floating-point implementation is bounded by $e$.

Both real and floating-point path conditions are collected in order to detect the presence of the program flow anomaly usually referred to as *unstable test*.

**Definition 1 (Test Stability).** *Let $\mathbf{R}_{\mathbb{B}} : \widetilde{\mathbb{B}} \to \mathbb{B}$ be the function converting 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 the floating-point values.*

*A conditional expression if $\tilde{\phi}$ then $\tilde{E}_1$ else $\tilde{E}_2$ is said to be* unstable *when it exists an assignment for the variables in $\tilde{\phi}$ to $\mathbb{F}$ such that $\tilde{\phi}$ and $\mathbf{R}_{\mathbb{B}}(\tilde{\phi})$ evaluate to a different boolean value. Otherwise the conditional expression is said to be* stable.

The presence of unstable tests makes the floating-point control flow different from the real arithmetic execution flow, and leads to unsound results when rounding errors provoke the unsound evaluation of conditionals. By separately collecting the information about real and floating-point flows, it is possible to consider the additional error of taking the incorrect branch in the cases in which the flows do not match. This guarantees a sound treatment of unstable tests in the proposed semantics.

Let $\mathbf{C}$ be the set of all conditional error bounds, and $\mathbb{C} \coloneqq \wp(\mathbf{C})$ be the domain formed by sets of conditional error bounds, which is the support domain of the proposed semantics. An *environment* is defined as a function mapping a variable to a set of conditional error bounds, i.e., $Env = \mathbb{V} \to \mathbb{C}$. The empty environment is denoted as $\bot_{Env}$ and maps every variable to the empty set $\varnothing$.

The semantics of arithmetic expressions is a function $\mathcal{A} : \widetilde{\mathbb{A}} \times Env \to \mathbb{C}$ defined as follows, where $\sigma \in Env$, $x \in \mathbb{V}$, and $\phi_r, \phi_e : \mathbb{V} \to \mathbb{V}$ are two functions that associate to each variable $x$ a fresh variable representing the real value and the error of $x$, respectively. Let $\widetilde{\mathsf{op}}$ be an $n$-ary floating-point operator in $\Omega$ such that its real-valued counterpart is denoted as $\mathsf{op}$. As stated in Section 2, it is assumed that there exists a function $\varepsilon_{\widetilde{\mathsf{op}}}$ such that Formula (2.4) holds and let $\pi_{\widetilde{\mathsf{op}}}(\pi_i)_{i=1}^n$ be a proof for that statement, which is defined in function of the proofs $\pi_i$ corresponding to $\widetilde{\mathsf{op}}$ operands. Furthermore, $\pi_{cnst}$ and $\pi_{var}$ are the proofs of Formula (2.4) for the constant and variable cases, respectively, which must be provided according to the formalization of Section 2.

$$\mathcal{A}[\![\tilde{k}]\!]_\sigma \coloneqq \{(true, true, \tilde{k}, 0, \pi_{cnst})\} \qquad \mathcal{A}[\![\mathbf{F}(k)]\!]_\sigma \coloneqq \{(true, true, k, |k - \mathbf{F}(k)|, \pi_{cnst})\}$$

$$\mathcal{A}[\![x]\!]_\sigma \coloneqq \begin{cases} \{(true, true, \phi_r(x), \phi_e(x), \pi_{var}(x))\} & \text{if } \sigma(x) = \varnothing \\ \sigma(x) & \text{otherwise} \end{cases}$$

$$\begin{aligned} &\mathcal{A}[\![\widetilde{\mathsf{op}}(\widetilde{A}_i)_{i=1}^n]\!]_\sigma \coloneqq \\ &\quad \bigcup \{ (\bigwedge_{i=1}^n \eta_i \wedge \eta_{\mathsf{op}}(r_i)_{i=1}^n, \bigwedge_{i=1}^n \tilde{\eta}_i \wedge \eta_{\widetilde{\mathsf{op}}}(\widetilde{A}_i)_{i=1}^n, \mathsf{op}(r_i)_{i=1}^n, \varepsilon_{\widetilde{\mathsf{op}}}(e_i)_{i=1}^n, \pi_{\widetilde{\mathsf{op}}}(\pi_i)_{i=1}^n) \\ &\qquad \mid \forall 1 \le i \le n \colon (\eta_i, \tilde{\eta}_i, r_i, e_i, \pi_i) \in \mathcal{A}[\![\widetilde{A}_i]\!]_\sigma, \eta_{\mathsf{op}}(r_i)_{i=1}^n \in \mathbb{B}, \eta_{\widetilde{\mathsf{op}}}(\widetilde{A}_i)_{i=1}^n \in \widetilde{\mathbb{B}}, \\ &\qquad \bigwedge_{i=1}^n \eta_i \wedge \eta_{\mathsf{op}}(r_i)_{i=1}^n \neq false, \ \bigwedge_{i=1}^n \tilde{\eta}_i \wedge \eta_{\widetilde{\mathsf{op}}}(\widetilde{A}_i)_{i=1}^n \neq false \} \end{aligned}$$

No rounding error is associated to a floating-point constant $\tilde{k}$, while the error of rounding a real constant $k$ is the difference between its real value and its rounding. The semantics of a variable $x \in \mathbb{V}$ is composed of two cases. If $x$ belongs to the environment, then the variable has been previously bound to an arithmetic expression $\widetilde{A}$ through a let-expression. In this case, the semantics of $x$ is exactly the semantics of $\widetilde{A}$. 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, $\phi_r(x)$ and $\phi_e(x)$, representing the real value and the error of $x$, respectively. In the case of a floating-point arithmetic operation $\widetilde{\mathsf{op}}$, the new error bound is obtained by applying $\varepsilon_{\widetilde{\mathsf{op}}}$ to the errors and real values of the operands and the new conditions are obtained as the combination of the conditions of the operands. Predicates $\eta_{\mathsf{op}}$ and $\eta_{\widetilde{\mathsf{op}}}$ represent the additional constraints needed when $\mathsf{op}$ and $\widetilde{\mathsf{op}}$ are not total (as explained in Section 2). The proof for $\widetilde{\mathsf{op}}$ is defined by merging $\pi_{\widetilde{\mathsf{op}}}$ with the proofs of its operands.

Let $\mathbb{K} := \{\tilde{f}(x_1, \ldots, x_n) \mid \tilde{f} \in \Sigma, x_1, \ldots, x_n \in \mathbb{V}\}$ be the set of all possible function calls. An *interpretation* is a function $\rho{:}\mathbb{K} \to \mathbb{C}$ modulo variance. The set of all interpretations is denoted as *Int*. The empty interpretation is denoted as $\perp_{Int}$ and maps everything to the empty set. Given $\sigma \in Env$ and $\rho \in Int$, the semantics of program expressions, $\mathcal{E} : \widetilde{\mathbb{E}} \times Env \times Int \to \mathbb{C}$, returns the set of conditional error bounds representing an upper bound of the round-off error for each execution path, together with the corresponding conditions.

$$
\begin{aligned}
&\mathcal{E}[\![\widetilde{A}]\!]^{\rho}_{\sigma} := \mathcal{A}[\![\widetilde{A}]\!]_{\sigma}\\
&\mathcal{E}[\![\mathit{let}\ x = \widetilde{A}\ \mathit{in}\ \widetilde{E}]\!]^{\rho}_{\sigma} := \mathcal{E}[\![\widetilde{E}]\!]^{\rho}_{\sigma[x \mapsto \mathcal{A}[\![\widetilde{A}]\!]_{\sigma}]}\\
&\mathcal{E}[\![\mathit{if}\ \widetilde{B}\ \mathit{then}\ \widetilde{E}_1\ \mathit{else}\ \widetilde{E}_2]\!]^{\rho}_{\sigma} := \mathcal{E}[\![\widetilde{E}_1]\!]^{\rho}_{\sigma} \Downarrow_{(\mathbf{R}_{\mathbb{B}}(\widetilde{B}),\widetilde{B})} \cup\ \mathcal{E}[\![\widetilde{E}_2]\!]^{\rho}_{\sigma} \Downarrow_{(\neg\,\mathbf{R}_{\mathbb{B}}(\widetilde{B}),\neg\widetilde{B})} \cup\\
&\quad \bigcup\{(\eta_1 \wedge \eta_2, \tilde{\eta}_1, r_2, e_1 + |r_1 - r_2|, \pi_{un}(r_1, r_2, \pi_1)) \mid (\eta_1, \tilde{\eta}_1, r_1, e_1, \pi_1) \in \mathcal{E}[\![\widetilde{E}_1]\!]^{\rho}_{\sigma},\\
&\qquad (\eta_2, \tilde{\eta}_2, r_2, e_2, \pi_2) \in \mathcal{E}[\![\widetilde{E}_2]\!]^{\rho}_{\sigma},\ \eta_1 \wedge \eta_2 \neq \mathit{false}\} \Downarrow_{(\neg\,\mathbf{R}_{\mathbb{B}}(\widetilde{B}),\widetilde{B})} \cup\\
&\quad \bigcup\{(\eta_1 \wedge \eta_2, \tilde{\eta}_2, r_1, e_2 + |r_1 - r_2|, \pi_{un}(r_1, r_2, \pi_2))) \mid (\eta_1, \tilde{\eta}_1, r_1, e_1, \pi_1) \in \mathcal{E}[\![\widetilde{E}_1]\!]^{\rho}_{\sigma},\\
&\qquad (\eta_2, \tilde{\eta}_2, r_2, e_2, \pi_2) \in \mathcal{E}[\![\widetilde{E}_2]\!]^{\rho}_{\sigma},\ \eta_1 \wedge \eta_2 \neq \mathit{false}\} \Downarrow_{(\mathbf{R}_{\mathbb{B}}(\widetilde{B}),\neg\widetilde{B})}\\
&\mathcal{E}[\![\tilde{f}(\widetilde{A}_i)_{i=1}^{n}]\!]^{\rho}_{\sigma} := \bigcup\{(\eta \wedge \bigwedge_{i=1}^{n} \eta_i, \tilde{\eta} \wedge \bigwedge_{i=1}^{n} \tilde{\eta}_i, \bar{r}, \bar{e}, \bar{\pi}) \mid (\eta, \tilde{\eta}, r, e, \pi) \in \rho(\tilde{f}(x_1 \ldots x_n)),\\
&\quad \forall 1 \leq i \leq n{:}(\eta_i, \tilde{\eta}_i, r_i, e_i, \pi_i) \in \mathcal{A}[\![\widetilde{A}_i]\!]_{\sigma},\ \bar{r} = r[\phi_r(x_1)/r_1, \ldots, \phi_r(x_n)/r_n],\\
&\quad \bar{e} = e[\phi_e(x_1)/e_1, \ldots, \phi_e(x_n)/e_n],\ \bar{\pi} = \pi[\phi_r(x_1)/r_1, \ldots, \phi_r(x_n)/r_n, \phi_e(x_1)/e_1,\\
&\quad \ldots, \phi_e(x_n)/e_n, \pi_{var}(x_1)/\pi_1, \ldots, \pi_{var}(x_n)/\pi_n],\ \eta \wedge \bigwedge_{i=1}^{n} \eta_i \neq \mathit{false},\ \tilde{\eta} \wedge \bigwedge_{i=1}^{n} \tilde{\eta}_i \neq \mathit{false}\}
\end{aligned}
$$

Intuitively, the semantics of the expression *let* $x = \widetilde{A}$ *in* $\widetilde{E}$ updates the current environment by associating to variable $x$ the semantics of expression $\widetilde{A}$.

The semantics of the conditional uses an auxiliary operator $\Downarrow$ for propagating new information in the conditions. Given $b \in \mathbb{B}$ and $\tilde{b} \in \widetilde{\mathbb{B}}$, $(\eta, \tilde{\eta}, r, e, t) \Downarrow_{(b,\tilde{b})} = (\eta \wedge b, \tilde{\eta} \wedge \tilde{b}, r, e, t)$ if $\eta \wedge b \neq \mathit{false}$ and $\tilde{\eta} \wedge \tilde{b} \neq \mathit{false}$, otherwise it is undefined. The definition of $\Downarrow$ naturally extends to sets of conditional error bounds: given $C \subseteq \mathbb{C}$, $C \Downarrow_{(b,\tilde{b})} = \bigcup_{c \in C} c \Downarrow_{(b,\tilde{b})}$. Tests in conditionals need to be treated carefully to guarantee soundness. Consider the conditional *if* $\widetilde{B}$ *then* $\widetilde{E}_1$ *else* $\widetilde{E}_2$. The semantics of $\widetilde{E}_1$ and $\widetilde{E}_2$ are enriched with the information about the fact that real and floating-point flows match, i.e., both $\widetilde{B}$ and $\mathbf{R}_{\mathbb{B}}(\widetilde{B})$ 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 $\widetilde{B}$ is satisfied but $\mathbf{R}_{\mathbb{B}}(\widetilde{B})$ 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 $\widetilde{E}_2$ and the floating point result of $\widetilde{E}_1$. It is easy to show that this error is bounded by the round-off error of $\widetilde{E}_1$ plus the difference between the real values of $\widetilde{E}_1$ and $\widetilde{E}_2$. The condition $(\neg\,\mathbf{R}_{\mathbb{B}}(\widetilde{B}), \widetilde{B})$ is propagated in order to model that $\widetilde{B}$ holds but $\mathbf{R}_{\mathbb{B}}(\widetilde{B})$ does not. The proof $\pi_{un}$ formalizes the previous argumentation in terms of the formal development defined in Section 2.

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, and the

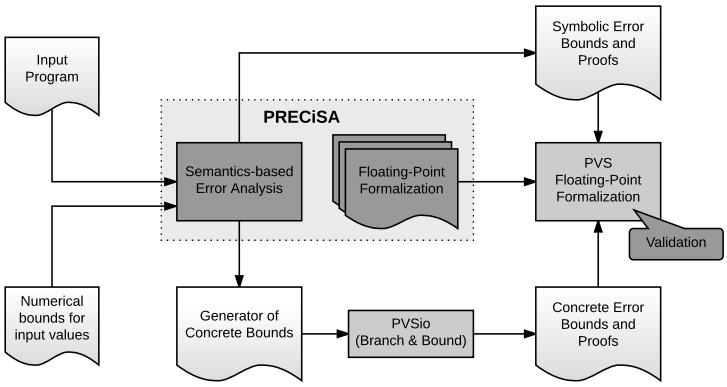


**Fig. 1.** PRECiSA architecture.

proofs for the variables representing formal parameters are replaced by the proofs for the actual parameters.

The semantics of a program is a function $\mathcal{F} : \widetilde{\mathbb{P}} \times Env \to \mathbb{C}$ defined as the least fixed point ($lfp$) of the immediate consequence operator $\mathcal{P} : \widetilde{\mathbb{P}} \times Env \times Int \to \mathbb{C}$, i.e., given $\widetilde{P} \in \widetilde{\mathbb{P}}$, $\mathcal{F}[\![\widetilde{P}]\!] := lfp(\mathcal{P}[\![\widetilde{P}]\!]^{\perp_{Int}}_{\perp_{Env}})$, which is defined as $\mathcal{P}[\![\widetilde{P}]\!]^{\rho}_{\sigma}(\tilde{f}(x_1 \dots x_n)) := \mathcal{E}[\![\widetilde{E}]\!]^{\rho}_{\sigma}$ for each function symbol $\tilde{f}$ defined in $\widetilde{P}$ such that $\tilde{f}(x_1 \dots x_n) = \widetilde{E} \in \widetilde{P}$. The least fixed point of $\mathcal{P}$ is guaranteed to exist from the Knaster-Tarski Fixpoint theorem [25]. In fact, it is easy to see that $(\mathbb{C}, \subseteq, \cup, \cap, \mathbf{C}, \varnothing)$ is a complete lattice and $\mathcal{P}$ is monotonic over $\mathbb{C}$, since at each iteration new conditional error bounds are added but not removed. When the program terminates in a finite number of steps for any possible input, this fixpoint computation converges in a finite number of steps. While this is a restrictive assumption in general, it is not unreasonable in avionics or embedded software, which tends to avoid recursion. However, in the future, the use of precise widening operators [3] on abstractions of this semantics will be explored in order to ensure the convergence for a wider variety of programs.

The semantics presented in this section allows for a static analysis that is compositional and parametric with respect to the functions used to approximate the round-off error of the arithmetic operations. Indeed, any floating-point operation $\widetilde{\mathsf{op}}$ can be supported by this analysis, as long as an approximation error function $\varepsilon_{\widetilde{\mathsf{op}}}$ satisfying Formula (2.4) is provided.

## 4 PRECiSA

PRECiSA (Program Round-off Error Certifier via Static Analysis) is a prototype implementation of the static analysis proposed in Section 3. PRECiSA accepts as input a floating-point program in the grammar defined in Section 3 and automatically generates an estimation of the floating-point round-off error together with proof certificates in PVS ensuring this estimation is correct.

Fig. 1 depicts the functional architecture of PRECiSA. Given an input program, its semantics as defined in Section 3 is computed. This semantics is instantiated with the

error approximation functions of the floating-operators from Section 2. Additionally, in order to improve the precision, PRECiSA distinguishes special cases in which the error estimation can be refined depending on the input. These cases include the subtraction $x \tilde{-} y$ when $y \tilde{/} 2 \leq x \leq 2 \tilde{*} y$, and the multiplication for a non-negative power of 2, which can be computed exactly.

For each function $\tilde{f}$ in the input program, a set of conditional error bounds is generated. Each conditional error bound, corresponding to a possible computational flow of $\tilde{f}$, is then translated into a PVS lemma stating that, provided the conditions are satisfied, the floating-point value resulting from the execution of $\tilde{f}$ on floating-point values differs from the exact real-number computation by at most the round-off error approximation computed by the semantics. The translation of a conditional error bound $(\eta, \tilde{\eta}, r, e, \pi)$ into a PVS lemma is straightforward. The hypotheses of the lemma are $\eta$ and $\tilde{\eta}$. The conclusion states that the difference between $r$ and the output of $\tilde{f}$ using floating-point arithmetic is at most $e$. Since proving lemmas in PVS can be a tedious task and it often requires a high level of expertise, PRECiSA generates the proof script corresponding to each generated PVS lemma from the symbolic proof $\pi$.

PRECiSA computes round-off errors in symbolic form so that the analysis is modular and independent from the initial values of the input variables. As explained above, PRECiSA translates this symbolic information into PVS lemmas and proofs. Additionally, given some initial ranges for the input variables, PRECiSA computes concrete numerical estimations of these symbolic error expressions. Furthermore, it also generates PVS lemmas (and proof scripts) stating the correctness of such concrete bounds, and an additional lemma assuring the overall concrete round-off error of the function, independently from the chosen computational flow.

In order to compute the concrete numerical bounds, the branch-and-bound algorithm presented in [20] has been enhanced to support the symbolic error expressions produced by PRECiSA. This branch-and-bound algorithm relies on a parametric enclosure method for computing provably correct approximations of real-valued arithmetic expressions. PRECiSA currently uses interval arithmetic, but other enclosure methods such as Bernstein polynomials and affine arithmetic can be used since they are already defined in PVS [17,19]. The algorithm recursively splits the domain of the function into smaller subdomains and computes an enclosure of the original expression 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. 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. This enhanced branch-and-bound algorithm is specified and formally verified in PVS. Hence, the numerical bounds of the error expressions are provably sound concretizations of the symbolic bounds generated using the semantics of Section 3.

As shown in Fig. 1, the current version of PRECiSA outputs two different PVS files: one containing the lemmas and proofs on the symbolic error bounds and one including the lemmas and proofs on the concrete numerical error bounds computed assuming specific initial ranges for the input variables. These files can be automatically discharged in PVS with no user intervention.

The rest of this section illustrates the use of PRECiSA in the formal analysis of the Compact Position Reporting (CPR) algorithm, which is part of the Automatic Dependent Surveillance Broadcast (ADS-B) protocol. This protocol, which is a safety-critical component of advanced air traffic operational concepts, ensures that every aircraft automatically and periodically broadcast its current position and velocity vectors to

nearby aircrafts and ground stations. The CPR algorithm is used to encode and decode the aircraft position (latitude and longitude). The standard organizations responsible for this protocol (RTCA in the US and EUROCAE in Europe) are currently studying reports of numerical stability issues in CPR. As part of the work presented in this paper, the authors have confirmed that under some circumstances, CPR may report incorrect aircraft positions that are several miles off of the actual position.

The CPR decoding function *rLat* is presented below. This function recovers the current latitude of the aircraft starting from the received encoded latitude YZ and a given reference latitude $LatS$ (in degrees). The reference latitude, in general, corresponds to a previously decoded latitude.

$$j(LatS, \mathrm{YZ}) = \widetilde{floor}((LatS \mathbin{\tilde{/}} (360 \mathbin{\tilde{/}} 59)) \mathbin{\tilde{-}} (\mathrm{YZ} \mathbin{\tilde{/}} 131072)) \mathbin{\tilde{+}} 0.5)$$
$$rLat(LatS, \mathrm{YZ}) = 360 \mathbin{\tilde{/}} 59 \mathbin{\tilde{*}} (j(LatS, \mathrm{YZ}) \mathbin{\tilde{+}} (\mathrm{YZ} \mathbin{\tilde{/}} 131072))$$

PRECiSA is able to differentiate the cases in which the accumulated error in the argument of the floating-point floor operation is large enough to make its result different from the ideal result for at least one unit. In cases where the accumulated error does not affect the result of the floor, PRECiSA computes a round-off error of $6.547117 \times 10^{-14}$ on *rLat* assuming double precision floating-point arithmetic and the following ranges for the inputs: $LatS \in [-90, 90]$ and $\mathrm{YZ} \in [0, 131071]$. The symbolic bound is generated in 0.18s and the concrete value is computed in 1.31s. For these cases, it can be proved that the latitude decoded by the double precision floating-point procedure corresponds to its ideal definition.

On the contrary, when the accumulated error affects the result of the floor, PRECiSA computes an error bound of $\approx 6.1$, which corresponds to several hundred nautical miles off with respect to the original position. The characterization of the input values to CPR that cause the floor operation to be unstable is still a matter of research.

## 5 Experimental Results

In this section, PRECiSA is compared in terms of accuracy and performance with the following floating-point analysis tools: Gappa (ver. 1.3.1) [6], Fluctuat (ver. 3.1376) [8], FPTaylor (ver. 0.9) [24] and Real2Float [15] (see Section 6 for a description of each tool). This comparison was performed using benchmarks taken from the FPTaylor repository. The selected benchmarks involve nonlinear expressions and polynomial approximations of functions, taken from well-known equations used in physics, control theory, and biological modeling. The experimental environment consisted of a 2.5 GHz Intel Core i7-4710MQ with 24 GB of RAM, running under Ubuntu 16.04 LTS. The benchmarks presented in this section and the corresponding PVS certificates are available as part of the PRECiSA distribution.

Table 1 shows the the round-off error bounds computed by the aforementioned tools. Since FPTaylor offers two different modes for the analysis, only the best estimation obtained with either mode is reported in the table. Gappa and Fluctuat allow the user to manually provide hints to obtain tighter error bounds. However, for the sake of uniformity in the comparison, the table only shows error estimations that are fully automatically computed. For the same reason, for all examples and tools, input variables and constants are assumed to be real numbers. This means that they carry a round-off error that has to be taken into consideration in the analysis.

| | Gappa | Fluctuat | Real2Float | FPTaylor | PRECiSA |
|---|---|---|---|---|---|
| carbonGas | 2.61e-08 | 4.51e-08 | 2.21e-08 | *8.06e-09* | **7.32e-09** |
| verhulst | 4.18e-16 | 5.51e-16 | 4.66e-16 | **2.47e-16** | *2.91e-16* |
| predPrey | 2.04e-16 | 2.49e-16 | 2.51e-16 | **1.59e-16** | *1.77e-16* |
| rigidBody1 | **2.95e-13** | *3.22e-13* | 5.33e-13 | **2.95e-13** | **2.95e-13** |
| rigidBody2 | *3.61e-11* | 3.65e-11 | 6.48e-11 | *3.61e-11* | **3.60e-11** |
| doppler1 | 2.02e-13 | 3.90e-13 | 7.65e-12 | **1.58e-13** | *1.99e-13* |
| doppler2 | 3.92e-13 | 9.75e-13 | 1.57e-11 | **2.89e-13** | *3.83e-13* |
| doppler3 | 1.08e-13 | 1.57e-13 | 8.59e-12 | **6.62e-14** | *1.05e-13* |
| turbine1 | 8.40e-14 | 9.20e-14 | 2.46e-11 | **1.67e-14** | *2.33e-14* |
| turbine2 | 1.28e-13 | 1.29e-13 | 2.07e-12 | **1.95e-14** | *3.07e-14* |
| turbine3 | 3.99e+01 | 6.99e-14 | 1.70e-11 | **9.64e-15** | *1.72e-14* |
| sqroot | 5.71e-16 | 6.83E-16 | 1.28e-15 | *5.02e-16* | **4.29e-16** |
| sine | 1.13e-15 | 7.97E-16 | 6.03e-16 | **4.43e-16** | *5.96e-16* |
| sineOrder3 | *8.89e-16* | 1.15E-15 | 1.19e-15 | **5.94e-16** | 1.11e-15 |

**Table 1.** Experimental results for absolute round-off error bounds (**bold** indicates the best approximation, *italic* indicates the second best.)

It can be seen in Table 1 that FPTaylor and PRECiSA produce more tight results than the other approaches. This is probably because both tools use accurate symbolic error expressions and optimization techniques to compute the numerical error bounds.

The times for the computation of the bounds in Table 1 are shown in Table 2.[6] It can be noticed that Gappa and Fluctuat are the fastest approaches. However, Gappa sometimes produces too coarse over-estimates (see for example turbine3 in Table 1) presumably because it uses interval arithmetic to compute the bounds. Unlike the other tools considered here, Fluctuat does not produce certificates for the soundness of its results.

PRECiSA, FPTaylor, and Real2Float show similar performance in half of the cases. However, in the other half, PRECiSA takes much longer in computing the bounds. This difference in the performance may be due to the fact that the calculation of the concrete bounds is performed inside the theorem prover. Conversely, the rest of the tools use specific developments that allow them to perform more efficiently. A possible enhancement for PRECiSA is to use a more performant tool to compute the bounds such as the Kodiak solver [23], a C++ implementation of the same branch and bound algorithm used by PRECiSA.

## 6 Related Work

Diverse techniques to estimate round-off error of floating-point computations can be found in the literature. Fluctuat [8] is a commercial analyzer that accepts as input a C (or ADA) program with annotations about input ranges and uncertainties, and produces bounds for the round-off error of the program expressions decomposed with

[6] Times for PRECiSA do not include type-checking of the PVS formalization, which takes approximately 4 min. However, this type-checking only occurs once at the beginning of the same PVS session used to compute all the bounds in Table 1.

| | Gappa | Fluctuat | Real2Float | FPTaylor | PRECiSA |
|---|---|---|---|---|---|
| carbonGas | 0.152 | 0.025 | 0.815 | 1.209 | 3.830 |
| verhulst | 0.034 | 0.043 | 0.465 | 0.812 | 0.789 |
| predPrey | 0.052 | 0.031 | 0.735 | 0.916 | 0.477 |
| rigidBody1 | 0.086 | 0.029 | 0.494 | 0.877 | 0.653 |
| rigidBody2 | 0.112 | 0.024 | 0.287 | 1.115 | 0.565 |
| doppler1 | 0.057 | 0.025 | 5.998 | 3.026 | 107.696 |
| doppler2 | 0.069 | 0.029 | 5.993 | 3.008 | 26.520 |
| doppler3 | 0.063 | 0.029 | 5.970 | 21.927 | 45.875 |
| turbine1 | 0.165 | 0.028 | 67.960 | 2.906 | 110.272 |
| turbine2 | 0.100 | 0.026 | 3.972 | 1.939 | 7.145 |
| turbine3 | 0.130 | 0.026 | 67.460 | 3.430 | 351.022 |
| sqroot | 0.281 | 0.024 | 0.712 | 1.157 | 0.343 |
| sine | 0.145 | 0.025 | 0.948 | 1.296 | 6.023 |
| sineOrder3 | 0.114 | 0.026 | 0.304 | 0.847 | 1.616 |

**Table 2.** Times in seconds for the generation of round-off error bounds and certificates.

respect to its provenance. Fluctuat provides support for iterative programs and unstable tests. It uses a zonotopic abstract domain [9] that is based on affine arithmetic. The prototype implementation presented in this paper is not competitive with Fluctuat in terms of speed. However, PRECiSA, which is publicly available under NASA's Open Source Agreement, provides a formal proof certificate of the correctness of the computed error estimation. The experimental evaluation shows that, for the considered benchmarks, both Fluctuat and PRECiSA provide similar results in terms of accuracy.

The tool FPTaylor [24] uses symbolic Taylor expansions to approximate floating-point expressions and applies a global optimization technique to obtain tight bounds for round-off errors. In addition, FPTaylor emits certificates for HOL Light [12], similarly to PRECiSA. Because of the technique used by FPTaylor, it is restricted to smooth functions. Therefore, it is not able to deal with non-derivable functions such as absolute value or floor, which are used, for example, in the CPR algorithm considered in Section 4. Unlike PRECiSA, which targets programs with conditional and function calls, FPTaylor is designed to analyze arithmetic expressions.

VCFloat [22] is a tool that automatically computes round-off error terms for numerical C expressions along with their correctness proof in Coq. This tool uses interval arithmetic to approximate the error bounds and generates validity conditions on the expressions. VCFloat computes the ulp by using the maximum exponent allowed in the floating-point representation, while PRECiSA computes the actual exponent for the maximum absolute value in the expression bounds, leading to more accurate estimations.

Real2Float[15] computes certified bounds for round-off errors by using an optimization technique employing semidefinite programming and sum of square certificates. Real2Float handles the ulp in the same way as VCFloat, which can result in coarser error approximations.

Gappa [6] computes enclosures for floating-point expressions via interval arithmetic. This enclosure method enables a quick computation of the bounds, but sometimes it can result in pessimistic error estimations. This tool also generates a proof of the results that can be checked in the Coq proof assistant. In Gappa, the bound computation, the

certification construction, and their verification may require hints from the user. Thus, some level of expertise is required, unlike PRECiSA which is fully automatic.

Rosa [4] automatically compiles an ideal real number program to a floating-point one with the aim of minimizing the overall round-off error. In the same line, FPTuner [2] implements a rigorous approach to precision allocation supporting also mixed-precision.

## 7 Conclusion

In this paper, a static analysis technique for estimating floating-point round-off errors is presented. The analysis enables the automatic generation of formal proof certificates of the correctness of such estimations. The analysis enjoys several useful features. It is defined in a compositional way, which allows for an incremental, modular, and efficient treatment of the program being analyzed. It is fully automatic, thus no human intervention is required to generate and formally verify the error estimations. The technique supports the generation of formal certificates that can be checked by an external tool. The proposed static analysis is sound with respect to unstable conditions. 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 techniques that are sound with respect to unstable tests are the one presented in [10] for the Fluctuat analyzer and Rosa [4]. The proposed analysis is parametric with respect to floating-point precision and rounding mode. Finally, it can be extended with any floating-point operator provided the existence of a round-off error estimation that satisfies some basic properties.

The proposed technique is implemented in the prototype tool PRECiSA. PRECiSA is fully automatic and generates PVS certificates that guarantee the correctness of the error bounds with respect to the floating-point IEEE-754 standard. Furthermore, given concrete ranges for the input variables of a program, the numerical estimations computed by PRECiSA are provably sound over-approximations of the possible round-off error that can occur in the program. The current implementation of PRECiSA supports single and double-precision floating-point formats and provides all the to-the-nearest rounding modalities introduced in the IEEE-754 standard. 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 for the concrete bound estimation depending on the expression type and on the desired precision/efficiency trade-off. Currently, PRECiSA uses a branch-and-bound algorithm based on interval arithmetic. Preliminary experimental results are encouraging for the applicability of PRECiSA in the formal verification of software of interest to NASA.

The floating-point round-off error formalization presented in this paper is available as part of the NASA PVS Library (`https://github.com/nasa/pvslib`). It consists of more than 150 PVS theories and several new proof strategies. Although the framework can be implemented in any modern proof assistant, the choice of PVS for this research project is convenient for the following reasons. First, PVS is used in the verification and validation of algorithms and concepts developed under NASA's Safe Autonomous Systems Operations (SASO) Project such as separation assurance algorithms for unmanned aircraft systems [18]. These algorithms, which involve critical numerical computations, are used as test cases for the framework and tool proposed here. Second, the NASA PVS Library includes independently developed hardware-level [16] and

high-level [1] formalizations of floating-point arithmetic, which are proved to be equivalent. The latter formalization is used and extended in this paper. Third, the NASA PVS Library also includes several formalizations of enclosure methods such as interval arithmetic [5], Bernstein polynomial basis [19], and affine arithmetic [17], which can be easily integrated in PRECiSA for computing concrete bounds of round-off errors. Finally, because of the automation support provided by PVS, no expertise in theorem proving is actually required to use the formalization presented in this paper.

The main drawback of the proposed approach is that it can generate large certificates for programs with nested conditionals. In fact, the number of conditional error bounds may grow exponentially in some cases due to the unstable tests handling (four different conditional error bounds may be generated for each conditional). In order to deal with this problem, an abstract semantics collapsing conditional error bounds produced after a given depth is being defined and will be integrated into PRECiSA in the near future. In this way, the number of elements in the semantics is reduced and consequently also the size of the generated proof certificate. Alternatively, the stable test hypothesis can be optionally enabled by the user in order to reduce the number of generated lemmas as done in most tools, although this may come at the cost of soundness. 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 future direction is the automatic generation of ACSL annotations related to round-off errors of C programs. The annotated program could then be automatically verified in a tool like Frama-C [13].

## References


1. S. Boldo and C. Muñoz. A high-level formalization of floating-point numbers in PVS. Technical Report CR-2006-214298, NASA, 2006.
2. W. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić. Rigorous floating-point mixed-precision tuning. In *Proceedings of POPL 2017*, pages 300–315. ACM, 2017.
3. 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 POPL 1977*, pages 238–252. ACM, 1977.
4. E. Darulova and V. Kuncak. Sound compilation of reals. In *Proceedings of POPL 2014*, pages 235–248. ACM, 2014.
5. M. Daumas, D. R. Lester, and C. Muñoz. Verified real number calculations: A library for interval arithmetic. *IEEE Trans. on Computers*, 58(2):226–237, 2009.
6. F. de Dinechin, C. Lauter, and G. Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Trans. on Computers*, 60(2):242–253, 2011.
7. L. H. de Figueiredo and J. Stolfi. Affine arithmetic: Concepts and applications. *Numerical Algorithms*, 37(1-4):147–158, 2004.
8. E. Goubault and S. Putot. Static analysis of numerical algorithms. In *Proceedings of SAS 2006*, volume 4134 of *LNCS*, pages 18–34. Springer, 2006.
9. E. Goubault and S. Putot. Static analysis of finite precision computations. In *Proceedings of VMCAI 2011*, volume 6538 of *LNCS*, pages 232–247. Springer, 2011.
10. E. Goubault and S. Putot. Robustness analysis of finite precision implementations. In *Proceedings of APLAS 2013*, volume 8301 of *LNCS*, pages 50–57. Springer, 2013.

11. J. Harrison. A machine-checked theory of floating point arithmetic. In *Proceedings of TPHOLs '99*, pages 113–130. Springer, 1999.
12. J. Harrison. HOL light: An overview. In *Proceedings of TPHOLs 2009*, volume 5674 of *LNCS*, pages 60–66. Springer, 2009.
13. F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Comp.*, 27(3):573–609, 2015.
14. G. G. Lorentz. *Bernstein Polynomials*. Chelsea Publishing Company, 1986.
15. V. Magron, G. Constantinides, and A. Donaldson. Certified roundoff error bounds using semidefinite programming. *CoRR*, abs/1507.03331, 2015.
16. Paul Miner. Defining the IEEE-854 floating-point standard in PVS. Technical Report TM-1995-110167, NASA, 1995.
17. M. M. Moscato, C. Muñoz, and A. P. Smith. Affine arithmetic and applications to real-number proving. In *Proceedings of ITP 2015*, volume 9236 of *LNCS*, pages 294–309. Springer, 2015.
18. C. Muñoz, A. Dutle, A. Narkawicz, and J. Upchurch. Unmanned Aircraft Systems in the National Airspace System: A formal methods perspective. *ACM SIGLOG News*, 3(3):67–76, 2016.
19. C. Muñoz and A. Narkawicz. Formalization of a representation of Bernstein polynomials and applications to global optimization. *Journal of Automated Reasoning*, 51(2):151–196, 2013.
20. A. Narkawicz and C. Muñoz. A formally verified generic branching algorithm for global optimization. In *Revised Selected Papers of VSTTE 2013*, volume 8164 of *LNCS*, pages 326–343. Springer, 2013.
21. S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proceedings of CADE 1992*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer, 1992.
22. T. Ramananandro, P. Mountcastle, B. Meister, and R. Lethin. A unified coq framework for verifying C programs with floating-point computations. In *Proceedings of CPP 2016*, pages 15–26. ACM, 2016.
23. A. Smith, C. Muñoz, A. Narkawicz, and M. Markevicius. A rigorous generic branch and bound solver for nonlinear problems. In *Proceedings of SYNASC 2015*. IEEE Computer Society Conference Publishing Services, September 2015.
24. A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. In *Proceedings of FM 2015*, volume 9109 of *LNCS*, pages 532–550. Springer, 2015.
25. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. In *Pacific Journal of Mathematics*, pages 285–309, 1955.