Automatic Generation of Guard-Stable Floating-Point Code

Laura Titolo[1], Mariano Moscato[1], Marco A. Feliu[1], and César A. Muñoz[2]

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

Abstract. In floating-point programs, guard instability occurs when the control flow of a conditional statement diverges from its ideal execution under real arithmetic. This phenomenon is caused by the presence of round-off errors in floating-point computations. Writing programs that correctly handle guard instability often requires expertise on finite precision arithmetic. This paper presents a fully automatic toolchain that generates and formally verifies a guard-stable floating-point C program from its functional specification in real arithmetic. The generated program is instrumented to soundly detect when unstable guards may occur and, in these cases, to issue a warning. The proposed approach combines the PRECiSA floating-point static analyzer, the Frama-C software verification suite, and the PVS theorem prover.

1 Introduction

The development of floating-point software is particularly challenging due to the presence of round-off errors, which originate from the difference between real numbers and their finite precision representation. Since round-off errors accumulate during numerical computations, they may significantly affect the evaluation of both arithmetic and Boolean expressions. In particular, *unstable guards*[3] occur when the guard of a conditional statement contains a floating-point expression whose round-off error makes the actual Boolean value of the guard differ from the value that would be obtained assuming real arithmetic. The presence of unstable guards amplifies the divergence between the output of a floating-point program and its ideal evaluation in real arithmetic. This divergence may lead to catastrophic consequences in safety-critical applications. Understanding how round-off errors and unstable guards affect the result and execution flow of floating-point programs requires a deep comprehension of floating-point arithmetic.

---

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

[3] In the literature [15,31], unstable guards are often referred to as unstable tests.

This paper presents a *fully automatic* integrated toolchain that generates and verifies guard-stable floating-point C code from its formal functional specification in real arithmetic. This toolchain consists of:

- PRECiSA [19,29], a static analyzer for floating-point programs,[4]
- Frama-C [17], a collaborative tool suite for the analysis of C code, and
- the Prototype Verification System (PVS) [23], an interactive theorem prover.

The input of the toolchain is a PVS specification of a numerical algorithm in real arithmetic, the desired floating-point format (single or double precision), and initial ranges for the input variables. A formally verified program transformation is introduced to implement the real-valued specification using floating-point arithmetic in the chosen floating-point format. This transformation is an extended and improved version of the one presented in [31]. Numerically unstable guards are replaced with more restrictive ones that preserve the control flow of the real-valued original specification. These new guards take into consideration the round-off error that may occur when the expressions of the original program are evaluated in floating-point arithmetic. In addition, the transformation instruments the program to emit a warning when the floating-point flow may diverge with respect to the real number specification. This program transformation is designed to limit the overhead introduced by the new guards. Symbolic error expressions are used to avoid concrete numerical assumptions on the input variables. This symbolic approach is highly modular since the transformation is still correct even if the input ranges are modified.

The static analyzer PRECiSA is extended with a module implementing the proposed program transformation and with another module that generates the corresponding C code. This C code includes ANSI/ISO C Specification Language (ACSL) [1] annotations stating the relationship between the floating-point C implementation and its functional specification in real arithmetic. To this end, the round-off errors that occur in conditional guards and the overall round-off error of each function in the program are estimated by PRECiSA. PVS proof certificates are generated stating the correctness of these estimations. The correctness property of the C program states that if the program terminates without a warning, it follows the same computational path as the real-valued specification, i.e., all unstable guards are detected.

The Frama-C/WeakestPrecondition (WP) plug-in is used to generate verification conditions in the language of PVS and it is customized to automatically integrate the PVS certificates generated by PRECiSA into the proof of such verification conditions. While PVS is an interactive theorem prover, these verification conditions are automatically proved by ad-hoc strategies developed as part of this work. Therefore, neither expertise in theorem proving nor in floating-point arithmetic is required from the user to verify the correctness of the generated C program. The proposed approach is applied to a fragment of the Detect and AvoID Alerting Logic for Unmanned Systems (DAIDALUS) software library developed by NASA [21]. DAIDALUS is the reference implementation

[4] The PRECiSA distribution is available at `https://github.com/nasa/PRECiSA`.

of detect-and-avoid for unmanned aircraft systems in the standard document RTCA DO-365 [25].

The remainder of the paper is organized as follows. Section 2 provides a brief overview of floating-point numbers, unstable guards, and the tool PRECiSA. The proposed program transformation to detect guard instability is introduced in Section 3. Section 4 describes the integrated toolchain to automatically generate and verify a probably correct floating-point C program from a PVS real-valued specification. Section 5 discusses related work and Section 6 concludes the paper.

2 Preliminaries

Floating-point numbers [16] (or *floats*) are finite precision representations of real numbers widely used in computer programs. In this work, $\mathbb{F}$ denotes the set of floating-point numbers. The expression $\mathrm{R}(\tilde{v})$ denotes the conversion of the float $\tilde{v}$ to reals, while the expression $F(r)$ denotes the floating-point number representing r, i.e., the rounding of r.

Definition 1 (Round-off error). *Let $\tilde{v} \in \mathbb{F}$ be a floating-point number that represents a real number $r \in \mathbb{R}$, the difference $|\mathrm{R}(\tilde{v}) - r|$ is called the* round-off error *(or* rounding error*) of $\tilde{v}$ with respect to r.*

When a real number r is rounded to the closest float, the round-off error is bounded by half *unit in the last place* of r, $ulp(r)$, which represents the difference between the two closest consecutive floating-point numbers $\tilde{v}_1$ and $\tilde{v}_2$ such that $\tilde{v}_1 \leq r \leq \tilde{v}_2$ and $\tilde{v}_1 \neq \tilde{v}_2$. Round-off errors accumulate through the computation of mathematical operators. The IEEE-754 standard [16] states that every basic operation should be performed as if it would be calculated with infinite precision and then rounded to the nearest floating-point value. Therefore, an initial error that seems negligible may become significantly larger when combined and propagated inside nested mathematical expressions.

Let $\widetilde{\mathbb{V}}$ be a finite set of variables representing floating-point values and $\mathbb{V}$ a finite set of variables representing real values such that $\widetilde{\mathbb{V}} \cap \mathbb{V} = \varnothing$. It is assumed that there is a function $\chi_r : \widetilde{\mathbb{V}} \to \mathbb{V}$ that associates to each floating-point variable $\tilde{x}$ a variable $x \in \mathbb{V}$ representing the real value of $\tilde{x}$. The set of arithmetic expressions over floating-point (respectively real) numbers is denoted as $\widetilde{\mathbb{A}}$ (respectively $\mathbb{A}$). The function $R_{\mathbb{A}} : \widetilde{\mathbb{A}} \to \mathbb{A}$ converts an arithmetic expression on floating-point numbers to the corresponding one on real numbers. This function is defined by replacing each floating-point operation with the corresponding one on real numbers and by applying R and χ_r to floating-point values and variables, respectively. Conversely, the function $F_{\mathbb{A}} : \mathbb{A} \to \widetilde{\mathbb{A}}$ applies the rounding F to constants and variables and replaces each real-valued operator with the corresponding floating-point one.

The function $R_{\mathbb{B}} : \widetilde{\mathbb{B}} \to \mathbb{B}$ is defined as the natural extension of $R_{\mathbb{A}}$ to Boolean expressions. Given a variable assignment $\sigma : \mathbb{V} \to \mathbb{R}$, $eval_{\mathbb{B}}(\sigma, B) \in \{true, false\}$ denotes the evaluation of the real Boolean expression B. Similarly, given $\widetilde{\sigma} : \widetilde{\mathbb{V}} \to \mathbb{F}$, $\widetilde{eval}_{\widetilde{\mathbb{B}}}(\widetilde{\sigma}, \widetilde{B}) \in \{true, false\}$ denotes the evaluation of the floating-point

Boolean expression $\widetilde{B}$. Boolean expressions are also affected by rounding errors. When $\tilde{\phi} \in \widetilde{B}$ evaluates differently in real and floating-point arithmetic, $\tilde{\phi}$ is said to be *unstable*.

Definition 2 (Unstable Guard). *A guard $\tilde{\phi} \in \widetilde{B}$ is said to be unstable if there exist two assignments $\tilde{\sigma} : \{\tilde{x}_1, \ldots, \tilde{x}_n\} \to \mathbb{F}$ and $\sigma : \{\chi_r(\tilde{x}_1), \ldots, \chi_r(\tilde{x}_n)\} \to \mathbb{R}$ such that for all $i \in \{1, \ldots, n\}$, $\sigma(\chi_r(\tilde{x}_i)) = R(\tilde{\sigma}(\tilde{x}_i))$ and $eval_{\mathbb{B}}(\sigma, R_{\mathbb{B}}(\tilde{\phi})) \neq \widetilde{eval}_{\widetilde{\mathbb{B}}}(\tilde{\sigma}, \tilde{\phi})$. Otherwise, the guard is said to be* stable.

The evaluation of a conditional statement *if* $\tilde{\phi}$ *then* $\widetilde{ae}_1$ *else* $\widetilde{ae}_2$ is said to follow an *unstable path* when $\tilde{\phi}$ is unstable. When the flows coincide, the evaluation is said to follow a *stable path*. The presence of unstable guards amplifies the effect of round-off errors in numerical programs since the computational flow of a floating-point program may significantly diverge from the ideal execution of its representation in real arithmetic. Therefore, for establishing the correctness of a numerical program, it is essential to correctly estimate the round-off error associated with both stable and unstable paths.

PRECiSA [29] is a static analyzer for floating-point programs. PRECiSA accepts as input a floating-point program and automatically generates a sound over-approximation of the floating-point round-off error and a proof certificate in PVS ensuring its correctness. Given a program to analyze, for every possible combination of real and floating-point execution paths, PRECiSA computes a *conditional error bound* of the form $\langle \eta, \tilde{\eta} \rangle_t \twoheadrightarrow (r, \tilde{v}, e)$, where $\eta \in \mathbb{B}$ is a symbolic path condition over the reals, $\tilde{\eta} \in \widetilde{\mathbb{B}}$ is a symbolic path condition over the floats, $r, e \in \mathbb{A}$ are symbolic arithmetic expressions over the reals, and $\tilde{v} \in \widetilde{\mathbb{A}}$ is a symbolic expression over the floats. Intuitively, $\langle \eta, \tilde{\eta} \rangle_t \twoheadrightarrow (r, \tilde{v}, e)$ indicates that if both conditions η and $\tilde{\eta}$ are satisfied, the output of the ideal real-valued program is r, the output of the floating-point implementation is $\tilde{v}$, and the round-off error is at most e, i.e., $|r - \tilde{v}| \leq e$. The flag t is used to indicate, by construction, whether a conditional error bound corresponds to an unstable path, when $t = \mathbf{u}$, or to a stable path, when $t = \mathbf{s}$. PRECiSA initially computes round-off error estimations in a symbolic form so that the analysis is modular. Given the initial ranges for the input variables, PRECiSA uses the Kodiak global optimizer [22,27] to maximize the symbolic error expression and obtain a concrete numerical enclosure for the error.

3 A Program Transformation to Detect Unstable Guards

This section presents a program transformation that converts a real-valued specification into a floating-point program instrumented to detect unstable guards. This program transformation significantly extends the expressivity of the input language of the transformation originally presented in [31]. In particular, it provides support for function calls, for-loops, predicates, and arithmetic expressions with inline conditionals. In addition, it improves the accuracy of the method to detect guard instability.

Let $\widetilde{\Omega}$ be a set of pre-defined floating-point operations, Σ a set of function symbols, Π a set of predicate symbols such that $\Sigma \cap \Pi = \emptyset$, and $\widetilde{\mathbb{V}}$ a finite set of variables representing floating-point values, respectively. The syntax of *floating-point program expressions* in $\widetilde{\mathbb{S}}$ is given by the following grammar.

$$\widetilde{A} \in \widetilde{\mathbb{A}} ::= \widetilde{d} \mid \tilde{x} \mid \widetilde{\odot}(\widetilde{A},\dots,\widetilde{A}) \mid \tilde{f}(\widetilde{A},\dots,\widetilde{A}) \mid \widetilde{B}?\,\widetilde{A} : \widetilde{A}$$

$$\widetilde{B} \in \widetilde{\mathbb{B}} ::= true \mid \widetilde{B} \wedge \widetilde{B} \mid \neg \widetilde{B} \mid \widetilde{A} < \widetilde{A} \mid \widetilde{A} \leq \widetilde{A} \mid \tilde{p}(\widetilde{A},\dots,\widetilde{A})$$

$$\widetilde{S} \in \widetilde{\mathbb{S}} ::= \widetilde{A} \mid let\ \tilde{x} = \widetilde{A}\ in\ \widetilde{S} \mid for(i_0, i_n, acc_0, \lambda(i, acc).\widetilde{S}) \mid if\ \widetilde{B}\ then\ \widetilde{S}\ else\ \widetilde{S}$$
$$\mid if\ \widetilde{B}\ then\ \widetilde{S}[elsif\ \widetilde{B}\ then\ \widetilde{S}]_{j=1}^{m}\ else\ \widetilde{S} \mid \omega$$

where $\widetilde{d} \in \mathbb{F}$, $\tilde{x} \in \widetilde{\mathbb{V}}$, $\tilde{f} \in \Sigma$, $\tilde{p} \in \Pi$, $\widetilde{\odot} \in \widetilde{\Omega}$, $i_0, i_n, acc_0 \in \widetilde{\mathbb{A}}$, and $i, acc \in \widetilde{\mathbb{V}}$.

The expression $\tilde{\phi}?\,\widetilde{A}_{then} : \widetilde{A}_{else}$ denotes an inline conditional statement that can be used as a parameter in an arithmetic operator or in a function call. The conjunction $\wedge$, negation $\neg$, and *true* have the usual classical logic meaning. The disjunction $\vee$ operator, the relations $>$, $\geq$, and the constant *false* can be derived. The notation $[elsif\ \widetilde{B}\ then\ \widetilde{S}]_{j=1}^{m}$ denotes a list of m conditional *elsif* branches. Bounded recursion is added to the language as syntactic sugar using the *for* construct. The expression $for(i_0, i_n, acc_0, \lambda(i, acc).body)$ emulates a for-loop where $i \in \widetilde{\mathbb{V}}$ is the control variable that ranges from i_0 to i_n, acc is the variable where the result is accumulated with initial value acc_0, and *body* is the body of the loop. For instance, $for(1, 10, 0, \lambda(i, acc).i + acc)$ represents the value $f(1,0)$, where f is the recursive function $f(i, acc) \equiv if\ i > 10\ then\ acc\ else\ f(i+1, acc+i)$. The body of the for-loop is restricted to be of type integer. Therefore, it does not accumulate round-off errors. The transformation of more generic for-loops requires the computation of the round-off error of a recursive function, which is an open problem beyond the scope of this paper. The symbol ω denotes a warning exceptional statement.

A *floating-point program* $\widetilde{P}$ is defined as a set of *function declarations* of the form $\tilde{f}(\tilde{x}_1,\dots,\tilde{x}_n) = \widetilde{S}$, where $\tilde{x}_1,\dots,\tilde{x}_n$ are pairwise distinct variables in $\widetilde{\mathbb{V}}$ and all free variables appearing in $\widetilde{S}$ are in $\{\tilde{x}_1,\dots,\tilde{x}_n\}$. The natural number n is called the *arity* of $\tilde{f}$. Henceforth, it is assumed that programs are well-formed in the sense that, in a program $\widetilde{P}$, for every function call $\tilde{f}(\widetilde{A}_1,\dots,\widetilde{A}_n)$ that occurs in the body of the declaration of a function $\tilde{g}$, a unique function $\tilde{f}$ of arity n is defined in $\widetilde{P}$ before $\tilde{g}$. Hence, the only recursion allowed is the one provided by the for-loop construct. The set of floating-point programs is denoted by $\widetilde{\mathbb{P}}$.

A *real-valued program* has the same structure of a floating-point program where floating-point expressions are replaced with real-valued ones. A real-valued program does not contain any ω statements. The set of real-valued programs is denoted by $\mathbb{P}$. The function $F_{\mathbb{P}} : \mathbb{P} \to \widetilde{\mathbb{P}}$ converts a real program P into a floating-point one by applying $F_{\mathbb{A}}$ to the arithmetic expressions occurring in P.

The input of the transformation is a real-valued program P. The straightforward floating-point implementation of P is initially computed as $\widetilde{P} := F_{\mathbb{P}}(P)$. Subsequently, the Boolean expressions in the guards of $\widetilde{P}$ are replaced with more restrictive ones that take into consideration the symbolic round-off error. This is done by means of two Boolean abstractions $\beta^+, \beta^- : \widetilde{\mathbb{B}} \to \widetilde{\mathbb{B}}$ defined as follows.

Definition 3. *Let $\epsilon_{var} : \widetilde{\mathbb{A}} \to \widetilde{\mathbb{V}}$ be a function that associates to an expression $\widetilde{ae} \in \widetilde{\mathbb{A}}$ a variable that represents its round-off error, i.e., $|\widetilde{ae} - R_{\mathbb{A}}(\widetilde{ae})| \leq \epsilon_{var}(\widetilde{ae})$. The functions $\beta^+, \beta^- : \widetilde{\mathbb{B}} \to \widetilde{\mathbb{B}}$ are defined as follows, where $\diamond \in \{\leq, <\}$.*

$$\beta^+(\widetilde{ae} \diamond 0) := \begin{cases} \widetilde{ae} \diamond 0 & \textit{if } |\widetilde{ae} - R_{\mathbb{A}}(\widetilde{ae})| \leq 0 \\ \widetilde{ae} \diamond -\epsilon_{var}(\widetilde{ae}) & \textit{otherwise} \end{cases}$$

$$\beta^-(\widetilde{ae} \diamond 0) := \begin{cases} \neg(\widetilde{ae} \diamond 0) & \textit{if } |\widetilde{ae} - R_{\mathbb{A}}(\widetilde{ae})| \leq 0 \\ \neg(\widetilde{ae} \diamond \epsilon_{var}(\widetilde{ae})) & \textit{otherwise} \end{cases}$$

$$\beta^+(\tilde{\phi}_1 \wedge \tilde{\phi}_2) := \beta^+(\tilde{\phi}_1) \wedge \beta^+(\tilde{\phi}_2) \qquad \beta^-(\tilde{\phi}_1 \wedge \tilde{\phi}_2) := \beta^-(\tilde{\phi}_1) \vee \beta^-(\tilde{\phi}_2)$$

$$\beta^+(\neg\tilde{\phi}) := \beta^-(\tilde{\phi}) \qquad \beta^-(\neg\tilde{\phi}) := \beta^+(\tilde{\phi})$$

Let $\epsilon^{\beta}_{var} : \widetilde{\mathbb{B}} \to \wp(\widetilde{\mathbb{V}})$ denote a function computing the error variables introduced by applying β^+ and β^- to a Boolean expression. Given $\tilde{\phi}, \tilde{\phi}_1, \tilde{\phi}_2 \in \widetilde{\mathbb{B}}$, $\epsilon^{\beta}_{var}(\widetilde{ae} \diamond 0) := \{\epsilon_{var}(\widetilde{ae})\}$, where $\diamond \in \{\leq, <\}$, $\epsilon^{\beta}_{var}(\tilde{\phi}_1 \wedge \tilde{\phi}_2) := \epsilon^{\beta}_{var}(\tilde{\phi}_1) \cup \epsilon^{\beta}_{var}(\tilde{\phi}_2)$, and $\epsilon^{\beta}_{var}(\neg\tilde{\phi}) := \epsilon^{\beta}_{var}(\tilde{\phi})$. For each predicate $\tilde{p}(\tilde{x}_1,\ldots,\tilde{x}_n) = \tilde{\phi}$ such that $\epsilon^{\beta}_{var}(\tilde{\phi}) = \{e_1, \ldots, e_m\}$, $\tilde{\phi} \neq \beta^+(\tilde{\phi})$, and $\neg\tilde{\phi} \neq \beta^-(\tilde{\phi})$, two new predicates are introduced:

$$\tilde{p}^+(\tilde{x}_1, \ldots, \tilde{x}_n, e_1, \ldots, e_m) = \beta^+(\tilde{\phi}) \qquad \tilde{p}^-(\tilde{x}_1, \ldots, \tilde{x}_n, e_1, \ldots, e_m) = \beta^-(\tilde{\phi})$$

Thus, the Boolean abstractions for a predicate call are defined as follows:

$$\beta^+(\tilde{p}(\widetilde{ae}_1, \ldots, \widetilde{ae}_n)) := \tilde{p}^+(\widetilde{ae}_1, \ldots, \widetilde{ae}_n, e_1, \ldots, e_m)$$
$$\beta^-(\tilde{p}(\widetilde{ae}_1, \ldots, \widetilde{ae}_n)) := \tilde{p}^-(\widetilde{ae}_1, \ldots, \widetilde{ae}_n, e_1, \ldots, e_m).$$

Generic inequalities of the form $a < b$ are handled by replacing them with their equivalent sign-test form $a - b < 0$. The following lemma states that β^+ and β^- correctly approximate a floating-point Boolean expression and its negation, respectively.

Lemma 1. *Given $\tilde{\phi} \in \widetilde{\mathbb{B}}$, let $fv(\tilde{\phi})$ be the set of free variables in $\tilde{\phi}$. For all $\sigma : fv(\phi) \to \mathbb{R}$, $\tilde{\sigma} : fv(\tilde{\phi}) \to \mathbb{F}$, and $\tilde{x} \in fv(\tilde{\phi})$ such that $F(\sigma(\chi_r(\tilde{x}))) = \tilde{\sigma}(\tilde{x})$, β^+ and β^- satisfy the following properties.*

1. $\widetilde{eval}_{\widetilde{\mathbb{B}}}(\tilde{\sigma}, \beta^+(\tilde{\phi})) \Rightarrow \widetilde{eval}_{\widetilde{\mathbb{B}}}(\tilde{\sigma}, \tilde{\phi}) \wedge eval_{\mathbb{B}}(\sigma, R_{\mathbb{B}}(\tilde{\phi}))$.
2. $\widetilde{eval}_{\widetilde{\mathbb{B}}}(\tilde{\sigma}, \beta^-(\tilde{\phi})) \Rightarrow \widetilde{eval}_{\widetilde{\mathbb{B}}}(\tilde{\sigma}, \neg\tilde{\phi}) \wedge eval_{\mathbb{B}}(\sigma, \neg R_{\mathbb{B}}(\tilde{\phi}))$.

Given a program expression $\tilde{S}$, the function $\tau : \widetilde{\mathbb{S}} \to \widetilde{\mathbb{S}} \times \wp(\widetilde{\mathbb{V}})$, defined in Fig. 1, returns a pair formed by the instrumented version of $\tilde{S}$ and the set of new error variables introduced by β^+ and β^-. The functions $\tau_{\mathbb{S}} : \widetilde{\mathbb{S}} \to \widetilde{\mathbb{S}}$ and $\tau_{\mathbb{V}} : \widetilde{\mathbb{S}} \to \wp(\widetilde{\mathbb{V}})$ return the first and the second projection of the result of τ respectively.

In the case of the conditional (Eq. (3.2)), when the round-off error is null and it does not affect the evaluation of the Boolean expression, i.e., $\tilde{\phi} = \beta^+(\tilde{\phi})$ and $\neg\tilde{\phi} = \beta^-(\tilde{\phi})$, the transformation function τ is recursively applied to the subprograms $\widetilde{S}_1$ and $\widetilde{S}_2$. Otherwise, the test on $\tilde{\phi}$ is replaced by two more restrictive tests on $\beta^+(\tilde{\phi})$ and $\beta^-(\tilde{\phi})$. The *then* branch is taken when $\beta^+(\tilde{\phi})$ is satisfied. By

$$\tau(\widetilde{d}) = \langle \widetilde{d}, \varnothing \rangle \qquad \tau(\tilde{x}) = \langle \tilde{x}, \varnothing \rangle \qquad \tau(\widetilde{\odot}(\widetilde{A}_i)_{i=1}^n) = \langle \widetilde{\odot}(\tau_{\mathbb{S}}(\widetilde{A}_i))_{i=1}^n, \bigcup_{i=1}^n \tau_{\mathbb{V}}(\widetilde{A}_i) \rangle \tag{3.1}$$

$$\tau(\mathit{if}\ \tilde{\phi}\ \mathit{then}\ \widetilde{S}_1\ \mathit{else}\ \widetilde{S}_2) = \begin{cases} \langle \mathit{if}\ \tilde{\phi}\ \mathit{then}\ \tau_{\mathbb{S}}(\widetilde{S}_1)\ \mathit{else}\ \tau_{\mathbb{S}}(\widetilde{S}_2), & \text{if } \tilde{\phi} = \beta^+(\tilde{\phi}) \text{ and} \\ \quad \tau_{\mathbb{V}}(\widetilde{S}_1) \cup \tau_{\mathbb{V}}(\widetilde{S}_2) \rangle & \quad \neg\tilde{\phi} = \beta^-(\tilde{\phi}) \\ \langle\ \mathit{if}\ \beta^+(\tilde{\phi})\ \mathit{then}\ \tau_{\mathbb{S}}(\widetilde{S}_1) & \\ \ \ \mathit{elsif}\ \beta^-(\tilde{\phi})\ \mathit{then}\ \tau_{\mathbb{S}}(\widetilde{S}_2) & \text{if } \tilde{\phi} \neq \beta^+(\tilde{\phi}) \text{ or} \\ \ \ \mathit{else}\ \omega, & \quad \neg\tilde{\phi} \neq \beta^-(\tilde{\phi}) \\ \quad \tau_{\mathbb{V}}(\widetilde{S}_1) \cup \tau_{\mathbb{V}}(\widetilde{S}_2) \cup \epsilon_{var}^{\beta}(\tilde{\phi}) \rangle & \end{cases} \tag{3.2}$$

$$\tau(\mathit{if}\ \tilde{\phi}_1\ \mathit{then}\ \widetilde{S}_1\ [\mathit{elsif}\ \tilde{\phi}_i\ \mathit{then}\ \widetilde{S}_i]_{i=2}^{n-1}\ \mathit{else}\ \widetilde{S}_n) = \tag{3.3}$$

$$\begin{cases} \langle\ \mathit{if}\ \tilde{\phi}_1\ \mathit{then}\ \tau_{\mathbb{S}}(\widetilde{S}_1) & \\ \ [\mathit{elsif}\ \tilde{\phi}_i\ \mathit{then}\ \tau_{\mathbb{S}}(\widetilde{S}_i)]_{i=2}^{n-1}\ \mathit{else}\ \tau_{\mathbb{S}}(\widetilde{S}_n), & \text{if } \forall 1 \leq i \leq n,\ \tilde{\phi}_i = \beta^+(\tilde{\phi}_i) \\ \quad \bigcup_{i=1}^n \tau_{\mathbb{V}}(S_i) \rangle & \quad \text{and } \neg\tilde{\phi}_i = \beta^-(\tilde{\phi}_i) \\ \langle\ \mathit{if}\ \beta^+(\tilde{\phi}_1)\ \mathit{then}\ \tau_{\mathbb{S}}(\widetilde{S}_1) & \\ \ [\mathit{elsif}\ \beta^+(\tilde{\phi}_i) \wedge \bigwedge_{j=1}^{i-1} \beta^-(\tilde{\phi}_j)\ \mathit{then}\ \tau_{\mathbb{S}}(\widetilde{S}_i)]_{i=2}^{n-1} & \\ \ \ \mathit{elsif}\ \bigwedge_{j=1}^{n-1} \beta^-(\tilde{\phi}_j)\ \mathit{then}\ \tau_{\mathbb{S}}(\widetilde{S}_n) & \text{otherwise} \\ \ \ \mathit{else}\ \omega, & \\ \quad \bigcup_{i=1}^n (\tau_{\mathbb{V}}(\widetilde{S}_i) \cup \epsilon_{var}^{\beta}(\tilde{\phi}_i)) \rangle & \end{cases} \tag{3.4}$$

$$\tau(\mathit{let}\ \tilde{x} = \widetilde{A}\ \mathit{in}\ \widetilde{S}) = \langle \mathit{let}\ \tilde{x} = \tau_{\mathbb{S}}(\widetilde{A})\ \mathit{in}\ \tau_{\mathbb{S}}(\widetilde{S}),\ \tau_{\mathbb{V}}(\widetilde{S}) \rangle \tag{3.5}$$

$$\tau(\mathit{for}(i_0, i_n, acc_0, \lambda(i, acc).\widetilde{S})) = \langle \mathit{for}(i_0, i_n, acc_0, \lambda(i, acc).\ \tau_{\mathbb{S}}(\widetilde{S})),\ \tau_{\mathbb{V}}(\widetilde{S}) \rangle \tag{3.6}$$

$$\tau(\tilde{g}(\widetilde{A}_1, \dots, \widetilde{A}_n)) = \langle \tilde{g}^{\tau}(\tau_{\mathbb{S}}(\widetilde{A}_1), \dots, \tau_{\mathbb{S}}(\widetilde{A}_n), e'_1, \dots, e'_m), \bigcup_{i=1}^n \{e'_i\} \rangle, \tag{3.7}$$

where $\tilde{g}^{\tau}(\tilde{x}_1, \dots, \tilde{x}_n, e_1, \dots, e_m) \in \bar{\tau}(P)$
and $\forall i = 1 \dots m$, if $e_i = \epsilon_{var}(\widetilde{ae}_i)$, then $e'_i = \epsilon_{var}(\widetilde{ae}_i[\tilde{x}_j \leftarrow \tau_{\mathbb{S}}(\widetilde{A}_j)]_{j=1}^n)$.

Fig. 1. Program transformation rules.

Property 1 in Lemma 1, this means that in the original program both $\tilde{\phi}$ and $R(\tilde{\phi})$ hold and, thus, the *then* branch is taken in both real and floating-point control flows. The *else* branch of the transformed program is taken when $\beta^-(\tilde{\phi})$ holds. This means, by Property 2 in Lemma 1, that in the original program the else branch is taken in both real and floating-point control flows. When neither $\beta^+(\tilde{\phi})$ nor $\beta^-(\tilde{\phi})$ is satisfied a warning ω is issued indicating that floating-point and real flows may diverge. The function ϵ_{var}^{β} is applied to $\tilde{\phi}$ to collect the new error variables introduced by the application of β^+ and β^-. The inline version of the conditional is transformed in the same way.

For the n-ary conditional (Eq. (3.4)), in the case the round-off error does not affect the evaluation of any of the Boolean expression, the transformation

function τ is applied recursively to the subprograms $\widetilde{S}_1,\dots,\widetilde{S}_2$. Otherwise, the guard $\tilde{\phi}_i$ of the i-th branch is replaced by the conjunction of $\beta^+(\tilde{\phi}_i)$ and $\beta^-(\tilde{\phi}_j)$ for all the previous branches $j < i$. By Lemma 1, it follows that the transformed program takes the i-th branch only when the same branch is taken in both real and floating-point control flows of the original program. A warning is issued by the transformed program when real and floating-point control flows of the original program differ. The new variables introduced by the application of β^+ and β^- in each branch are computed by the ϵ^{β}_{var} function.

In the case of the function call (Eq. (3.7)), new error variables $e'_1,\dots,e'_m$ are introduced to model the instantiated error parameters where the formal parameters $\tilde{x}_1,\dots,\tilde{x}_n$ are replaced by the actual parameters $\widetilde{A}_1,\dots,\widetilde{A}_n$. These new variables are added to the set $\tau_{\mathbb{V}}(\widetilde{S})$. Thus, when $\tilde{g}^{\tau}(\tilde{x}_1,\dots,\tilde{x}_n,e_1,\dots,e_m) \in \bar{\tau}(P)$ and for all $i = 1\dots m$, if $e_i = \epsilon_{var}(\widetilde{ae}_i)$, then $e'_i = \epsilon_{var}(\widetilde{ae}_i[\tilde{x}_j \leftarrow \tau_{\mathbb{S}}(\widetilde{A}_j)]^n_{j=1})$.

The function $\bar{\tau}$ transforms a real-valued program P into a floating-point program that is instrumented to detect unstable guards. It is defined as follows.

Definition 4 (Program Transformation). *Let $P \in \mathbb{P}$ be a real-valued program, the transformation $\bar{\tau} : \mathbb{P} \to \widetilde{\mathbb{P}}$ is defined as*

$$\bar{\tau}(P) = \bigcup\{\tilde{f}^{\tau}(\tilde{x}_1,\dots,\tilde{x}_n,e_1,\dots,e_k) = \mathit{if} \bigvee_{\tilde{g}^{\tau}(\bar{y})\in fc(\widetilde{S}')}(\tilde{g}^{\tau}(\bar{y}) = \omega)\ \mathit{then}\ \omega\ \mathit{else}\ \widetilde{S}' \mid \tilde{f}(\tilde{x}_1,\dots,\tilde{x}_n) = \widetilde{S} \in F_{\mathbb{P}}(P), \langle \widetilde{S}', \{e_1,\dots,e_k\}\rangle = \tau(\widetilde{S})\},$$

where $fc(\widetilde{S})$ returns all the function calls ocurring in $\widetilde{S}$. The new parameters $e_1,\dots,e_k$ are called symbolic error parameters*.*

A check on each function call $\tilde{g}^{\tau}(\bar{y})$ occurring in the body of $\tilde{f}$ is performed. If the returned value is warning, this is propagated as the result of $\tilde{f}$. The expression $\widetilde{S}'$ is the instrumented body of $\tilde{f}$ obtained by applying the transformation τ. Each function declaration is equipped with an additional set of arguments $e_1,\dots,e_k$ which correspond to the symbolic error parameters introduced by the application of β^+ and β^- in the body of the function. Therefore, there is one new argument for each floating-point arithmetic expression occurring in the guard of a conditional. It can be argued that it would be sufficient to add, for each argument in the original function declaration, a variable representing its rounding error. In this case, the Boolean approximations β^+ and β^- could be implemented by using the symbolic error expression computed by PRECiSA. This approach has two main problems. First, such symbolic error expressions, being real-valued, cannot be evaluated precisely in a floating-point program. A trivial floating-point implementation would be affected by rounding error, thus compromising the soundness of the transformation. Second, correctly estimating the round-off error by using uniquely floating-point-operators is likely to produce a huge symbolic expression. This will lead to unintelligible code and, possibly, in a loss of performances since a complex arithmetic expression needs to be evaluated at runtime. In addition, the round-off error of computing the error expression itself needs to be considered. This may lead to an excessively coarse over-estimation resulting in a large number of false warnings. The choice

of using symbolic error parameters to model the round-off error of arithmetic expressions avoids the aforementioned problems. This solution provides a good level of modularity since the symbolic expression is independent of the variables' initial ranges. Furthermore, this approach preserves the program structure of the original program.

The following theorem states the correctness of the program transformation $\bar{\tau}$. The straightforward floating-point implementation of the original program $F_{\mathbb{P}}(P)$ and the transformed program $\bar{\tau}(P)$ return the same output if and only if the transformed program does not emit a warning.

Theorem 1. *Given $P \in \mathbb{P}$, for all $\tilde{f}(\tilde{x}_1,\dots,\tilde{x}_n) = \widetilde{S} \in F_{\mathbb{P}}(P)$, let $\tilde{f}^\tau(\tilde{x}_1,\dots,\tilde{x}_n, e_1,\dots,e_m) \in \bar{\tau}(P)$ be its transformed version. It holds that*

$$\tilde{f}^\tau(\tilde{x}_1,\dots,\tilde{x}_n,e_1,\dots,e_m) \neq \omega \iff \tilde{f}(\tilde{x}_1,\dots,\tilde{x}_n) = \tilde{f}^\tau(\tilde{x}_1,\dots,\tilde{x}_n,e_1,\dots,e_m).$$

The proposed program transformation (including Lemma 1 and Theorem 1) has been formally specified and verified in PVS. [5]

The intended semantics of the floating-point transformed program $\bar{\tau}(P)$ is the real-valued semantics of the original program P, i.e., the real-valued semantics of the transformed program $R_{\mathbb{P}}(\bar{\tau}(P))$ is not relevant for the notion of correctness considered in this work. Therefore, even if the transformed program presents unstable guards with respect to $R_{\mathbb{P}}(\bar{\tau}(P))$, Theorem 2 ensures that its floating-point control flow preserves the control flow of the original specification P on real arithmetic. The difference between the output of the real number specification P and the one of the transformed floating-point implementation $\bar{\tau}(P)$ is bounded by the error occurring in $F_{\mathbb{P}}(P)$ taking into consideration only the stable cases ($t = \mathbf{s}$), as stated in the following theorem. In the following, $\mathcal{P}[[\tilde{P}]](\tilde{f})$ denotes the set of conditional error bounds computed by PRECiSA for the function $\tilde{f}$ defined in the program $\tilde{P}$.

Theorem 2 (Program Transformation Correctness). *Given $P \in \mathbb{P}$, for all $f(x_1,\dots,x_n) = S \in P$, let $\tilde{f}^\tau(\tilde{x}_1,\dots,\tilde{x}_n,e_1,\dots,e_m) \in \bar{\tau}(P)$ be its transformed floating-point version. Let $\sigma : \{x_1 \dots x_n\} \to \mathbb{R}$, and $\tilde{\sigma} : \{\tilde{x}_1 \dots \tilde{x}_n\} \to \mathbb{F}$, such that for all $i \in \{1,\dots,n\}$, $R(\tilde{\sigma}(\tilde{x}_i)) = \sigma(x_i)$, it holds that*

$$\tilde{f}^\tau(\tilde{x}_1,\dots,\tilde{x}_n,e_1,\dots,e_m) \neq \omega \iff |f(x_1,\dots,x_n) - \tilde{f}^\tau(\tilde{x}_1,\dots,\tilde{x}_n,e_1,\dots,e_m)| \leq e_{\tilde{f}}$$

where $\tilde{f}^\tau(\tilde{x}_1,\dots,\tilde{x}_n,e_1,\dots,e_m) \in \bar{\tau}(P)$ and $e_{\tilde{f}} = max\{e \mid \langle \eta, \tilde{\eta} \rangle_t \twoheadrightarrow (r, \tilde{v}, e) \in \mathcal{P}[[F_{\mathbb{P}}(P)]](\tilde{f}), t = \mathbf{s}\}$.

Therefore, all the unstable cases of the original program are detected in the transformed program and they no longer influence the overall round-off error.

Example 1. Consider the following fragment of DAIDALUS[6], a software library that implements a detect-and-avoid logic for unmanned aircraft systems (UAS).

[5] This formalization is available at `https://shemesh.larc.nasa.gov/fm/PRECiSA`.
[6] DAIDALUS is available from `https://shemesh.larc.nasa.gov/fm/DAIDALUS/`.

A detect-and-avoid logic ensures that UAS remain well clear, e.g., safely separated, from traffic aircraft. The real-valued program $WCV \in \mathbb{P}$ consists of six functions. The function *wcv* determines if two aircrafts (ownship and intruder), whose relative vertical position and velocity are given by (s_x, s_y, s_z) and (v_x, v_y, v_z), respectively, are in loss of horizontal (*hwcv*) and vertical (*vwcv*) well clear. The function *tcoa* computes the time to co-altitude of two vertically converging aircraft. When the aircraft are vertically diverging, the function returns 0. The function *tcpa* computes the time to (horizontal) closest point of approach. The function *taumod* is an estimation of *tcpa* that is less demanding on sensor and surveillance technology. The constants DTHR, TTHR, ZTHR and TCOA are distance and time thresholds used in the definition of the DAIDALUS well-clear logic.

$$
\begin{aligned}
&tcoa(s_z, v_z) = if\ s_z v_z < 0\ then\ -(s_z / v_z)\ else\ 0\\
&tcpa(s_x, s_y, v_x, v_y) = if\ v_x \neq 0 \wedge v_y \neq 0\ then\ -(s_x v_x + s_y v_y)/(v_x^2 + v_y^2)\ else\ 0\\
&taumod(s_x, s_y, v_x, v_y) = \ if\ s_x v_x + s_y v_y < 0\\
&\qquad then\ (\mathrm{DTHR}^2 - s_x^2)/(s_x v_x + s_y v_y)\\
&\qquad else\ -1\\
&vwcv(s_z, v_z) = |s_z| \leq \mathrm{ZTHR} \vee (tcoa(s_z, v_z) \geq 0 \wedge tcoa(s_z, v_z) \leq \mathrm{TCOA})\\
&hwcv(s_x, s_y, v_x, v_y) = let\ t = tcpa(s_x, s_y, v_x, v_y), tm = taumod(s_x, v_x, s_y, v_y)\ in\\
&\quad s_x v_x + s_y v_y <= \mathrm{DTHR}^2\\
&\quad \vee ((s_x + t v_x)^2 + (sy + t v_y)^2 <= \mathrm{DTHR}^2 \wedge 0 <= tm \wedge tm <= \mathrm{TTHR})\\
&wcv(s_x, s_y, s_z, v_x, v_y, v_z) = hwcv(s_x, s_y, v_x, v_y) \wedge vwcv(s_z, v_z)
\end{aligned}
$$

The program $\bar{\tau}(WCV)$ is obtained by using the transformation in Fig. 1. The floating-point parameters are the rounding of the real ones, e.g., $s_x = \chi_r(\tilde{s_x})$. The floating-point rounding of each constant is denoted with a tilde. All inequalities occurring in WCV have been rearranged to be in the form of a sign-test in the transformed program. Error variables are introduced by β^+ and β^- as parameters for each floating-point expression occurring in the guards. In addition, the error parameters of the function calls are propagated to the caller. The meaning of each error variable is shown as a comment in gray.

$$
\begin{aligned}
&\widetilde{tcoa}^{\tau}(\tilde{s}_z, \tilde{v}_z, e_{tcoa}) = \ if\ \tilde{s}_z \tilde{v}_z < -e_{tcoa}\ then\ -(\tilde{s} \tilde{/} \tilde{v}) \qquad \% |(\tilde{s}_z \tilde{v}_z) - (s_z v_z)| \leq e_{tcoa}\\
&\qquad elsif\ \tilde{s}\tilde{v} \geq e_{tcoa}\ then\ 0\ \ else\ \omega\\
&\widetilde{tcpa}^{\tau}(\tilde{s}_x, \tilde{s}_y, \tilde{v}_x, \tilde{v}_y, e_x, e_y) = \qquad \% |\tilde{v}_x - v_x| \leq e_x,\ |\tilde{v}_y - v_y| \leq e_y\\
&\quad if\ (\tilde{v}_x < -e_x \vee \tilde{v}_x > e_x) \wedge (\tilde{v}_y < -e_y \vee \tilde{v}_y > e_y)\ \ then\ -(\tilde{s}_x \tilde{v}_x \tilde{+} \tilde{s}_y \tilde{v}_y) \tilde{/} (\tilde{v}_x^2 \tilde{+} \tilde{v}_y^2)\\
&\quad elsif\ (\tilde{v}_x \geq e_x \wedge \tilde{v}_x \leq -e_x) \vee (\tilde{v}_y \geq e_y \wedge \tilde{v}_y \leq -e_y)\ \ then\ 0\ \ else\ \omega\\
&\widetilde{taumod}^{\tau}(\tilde{s}_x, \tilde{s}_y, \tilde{v}_x, \tilde{v}_y, e_{tau}) = \qquad \% |(\tilde{s}_x \tilde{v}_x + \tilde{s}_y \tilde{v}_y) - (s_x v_x + s_y v_y)| \leq e_{tau}\\
&\quad if\ \tilde{s}_x \tilde{v}_x + \tilde{s}_y \tilde{v}_y < -e_{tau}\ \ then\ (\widetilde{\mathrm{DTHR}}^2 \tilde{-} \tilde{s}_x^2) \tilde{/} (s_x v_x \tilde{+} s_y v_y)\\
&\quad elsif\ \tilde{s}_x \tilde{v}_x + \tilde{s}_y \tilde{v}_y \geq e_{tau}\ \ then\ -1\ \ else\ \omega\\
&\widetilde{vwcv}^{+}(\tilde{s}_z, \tilde{v}_z, e_{tcoa}, e_1^v, e_2^v, e_3^v) = if\ \widetilde{tcoa}^{\tau}(\tilde{s}_z, \tilde{v}_z, e_{tcoa}) = \omega\ then\ \omega\ \ else\\
&\quad |\tilde{s}_z| \tilde{-} \widetilde{\mathrm{ZTHR}} \leq -e_1^v \qquad \% ||\tilde{s}_z| \tilde{-} \widetilde{\mathrm{ZTHR}}) - (|s_z| - \mathrm{ZTHR})| \leq e_1^v
\end{aligned}
$$

$$\vee\, (\widetilde{tcoa}^{\tau}(\tilde{s}_z, \tilde{v}_z, e_{tcoa}) \geq e_2^v \quad \%|\widetilde{tcoa}^{\tau}(\tilde{s}_z, \tilde{v}_z, e_{tcoa}) - tcoa(s_z, v_z)| \leq e_2^v$$
$$\wedge\, \widetilde{tcoa}^{\tau}(\tilde{s}_z, \tilde{v}_z, e_{tcoa}) \tilde{-} \widetilde{\mathrm{TCOA}} \leq -e_3^v)$$
$$\%|\widetilde{tcoa}^{\tau}((\tilde{s}_z, \tilde{v}_z, e_{tcoa}) \tilde{-} \widetilde{\mathrm{TCOA}}) - (tcoa(s_z, v_z) - \mathrm{TCOA})| \leq e_3^v$$

$$\widetilde{vwcv}^{-}(\tilde{s}_z, \tilde{v}_z, e_{tcoa}, e_1^v, e_2^v, e_3^v) = \mathit{if}\ \widetilde{tcoa}^{\tau}(\tilde{s}_z, \tilde{v}_z, e_{tcoa}) = \omega\ \mathit{then}\ \omega\ \ \mathit{else}$$
$$|\tilde{s}_z| - \widetilde{\mathrm{ZTHR}} > e_1^v \wedge (\widetilde{tcoa}^{\tau}(\tilde{s}_z, \tilde{v}_z, e_{tcoa}) < -e_2^v \vee \widetilde{tcoa}^{\tau}(\tilde{s}_z, \tilde{v}_z, e_{tcoa}) \tilde{-} \widetilde{\mathrm{TCOA}} > e_3^v)$$

$$\widetilde{hwcv}^{+}(\tilde{s}_x, \tilde{v}_x, \tilde{s}_y, \tilde{v}_y, e_x, e_y, e_{tau}, e_1^h, e_2^h, e_3^h, e_4^h) =$$
$$\mathit{let}\ t = \widetilde{tcpa}^{\tau}(\tilde{s}_x, \tilde{s}_y, \tilde{v}_x, \tilde{v}_y, e_x, e_y), tm = \widetilde{taumod}^{\tau}(\tilde{s}_x, \tilde{s}_y, \tilde{v}_x, \tilde{v}_y, e_{tau})\ \ \mathit{in}$$
$$\mathit{if}\ t = \omega \vee tm = \omega\ \mathit{then}\ \omega\ \mathit{else}$$
$$\tilde{s}_x\tilde{v}_x \tilde{+} \tilde{s}_y\tilde{v}_y \tilde{-} \widetilde{\mathrm{DTHR}}^2 \leq -e_1^h\ \%|(\tilde{s}_x\tilde{v}_x \tilde{+} \tilde{s}_y\tilde{v}_y - \widetilde{\mathrm{DTHR}}^2) - (s_x v_x + s_y v_y - \mathrm{DTHR}^2)| \leq e_1^h$$
$$\vee\, ((\tilde{s}_x \tilde{+} t\tilde{v}_x)^2 \tilde{+} (\tilde{s}_y \tilde{+} t\tilde{v}_y)^2 \tilde{-} \widetilde{\mathrm{DTHR}}^2 \leq -e_2^h$$
$$\%|((\tilde{s}_x \tilde{+} t\tilde{v}_x)^2 \tilde{+} (\tilde{s}_y \tilde{+} t\tilde{v}_y)^2 \tilde{-} \widetilde{\mathrm{DTHR}}^2) - ((s_x + tv_x)^2 + (s_y + tv_y)^2 - \mathrm{DTHR}^2)| \leq e_2^h$$
$$\wedge\, tm \geq e_3^h \qquad \%|tm - taumod(s_x, s_y, v_x, v_y)| \leq e_3^h$$
$$\wedge\, tm \tilde{-} \widetilde{\mathrm{TTHR}} \leq -e_4^h) \qquad \%|(tm \tilde{-} \widetilde{\mathrm{TTHR}}) - (taumod(s_x, s_y, v_x, v_y) - \mathrm{TTHR})| \leq e_4^h$$

$$\widetilde{hwcv}^{-}(\tilde{s}_x, \tilde{v}_x, \tilde{s}_y, \tilde{v}_y, e_x, e_y, e_{tau}, e_1^h, e_2^h, e_3^h, e_4^h) =$$
$$\mathit{let}\ t = \widetilde{tcpa}^{\tau}(\tilde{s}_x, \tilde{s}_y, \tilde{v}_x, \tilde{v}_y, e_x, e_y), tm = \widetilde{taumod}^{\tau}(\tilde{s}_x, \tilde{s}_y, \tilde{v}_x, \tilde{v}_y, e_{tau})\ \ \mathit{in}$$
$$\mathit{if}\ t = \omega \vee tm = \omega\ \mathit{then}\ \omega\ \mathit{else}\ \ (\tilde{s}_x\tilde{v}_x \tilde{+} \tilde{s}_y\tilde{v}_y \tilde{-} \widetilde{\mathrm{DTHR}}^2 > e_1^h$$
$$\wedge\, ((\tilde{s}_x \tilde{+} t\tilde{v}_x)^2 \tilde{+} (\tilde{s}_y \tilde{+} t\tilde{v}_y)^2 \tilde{-} \widetilde{\mathrm{DTHR}}^2 > e_2^h \vee tm \geq e_3^h \vee tm \tilde{-} \widetilde{\mathrm{TTHR}} > e_4^h))$$

$$\widetilde{wcv}^{+}(\tilde{s}_x, \tilde{v}_x, \tilde{s}_y, \tilde{v}_y, \tilde{s}_z, \tilde{v}_z, e_{tcoa}, e_x, e_y, e_{tau}, e_1^h, e_2^h, e_3^h, e_4^h, e_1^v, e_2^v, e_3^v) =$$
$$\mathit{let}\ \ hv = \widetilde{hwcv}^{+}(\tilde{s}_x, \tilde{v}_x, \tilde{s}_y, \tilde{v}_y, e_x, e_y, e_{tau}, e_1^h, e_2^h, e_3^h, e_4^h),$$
$$vv = \widetilde{vwcv}^{+}(\tilde{s}_z, \tilde{v}_z, e_{tcoa}, e_1^v, e_2^v, e_3^v)\ \ \mathit{in}$$
$$\mathit{if}\ \ hv = \omega \vee vv = \omega\ \mathit{then}\ \omega\ \mathit{else}\ hv \wedge vv$$

$$\widetilde{wcv}^{-}(\tilde{s}_x, \tilde{v}_x, \tilde{s}_y, \tilde{v}_y, \tilde{s}_z, \tilde{v}_z, e_{tcoa}, e_x, e_y, e_{tau}, e_1^h, e_2^h, e_3^h, e_4^h, e_1^v, e_2^v, e_3^v) =$$
$$\mathit{let}\ \ hv = \widetilde{hwcv}^{-}(\tilde{s}_x, \tilde{v}_x, \tilde{s}_y, \tilde{v}_y, e_x, e_y, e_{tau}, e_1^h, e_2^h, e_3^h, e_4^h),$$
$$vv = \widetilde{vwcv}^{-}(\tilde{s}_z, \tilde{v}_z, e_{tcoa}, e_1^v, e_2^v, e_3^v)\ \ \mathit{in}$$
$$\mathit{if}\ \ hv = \omega \vee vv = \omega\ \mathit{then}\ \omega\ \mathit{else}\ hv \vee vv$$

4 Automatic Generation and Verification of Guard-Stable C Code

The toolchain presented in this section relies on several tools: the static analyzer PRECiSA, the global optimizer Kodiak [27][7], the static analyzer Frama-C, and the interactive prover PVS. The input to the toolchain is a real-valued program expressed in the PVS specification language, the desired floating-point precision (single and double precision are supported), and initial ranges for the input

[7] Kodiak is available from `https://shemesh.larc.nasa.gov/fm/Kodiak/`.

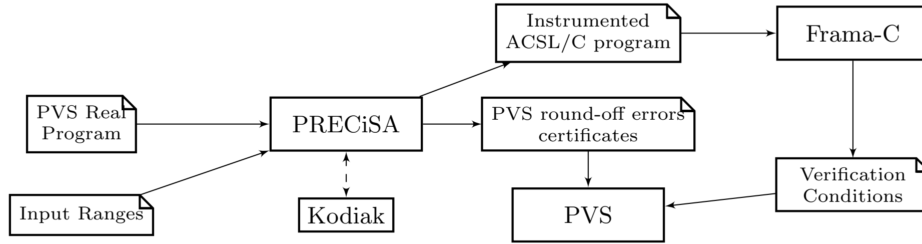

Fig. 2. Toolchain for automatically generate and verify guard-stable C code.

variables. The output is an annotated C program that is guaranteed to emit a warning when real and floating-point paths diverge in the original program and PVS certificates that ensure its correctness. An overview of the approach is depicted in Fig. 2.

In this work, PRECiSA is extended to implement the transformation defined in Section 3 and to generate the corresponding C code. Given a real-valued program P and a desired floating-point format (single or double precision), PRECiSA applies the transformation presented in Section 3. The transformed program is then converted into C syntax and ANSI/ISO C Specification Language (ACSL) annotations are generated. ACSL [1] is a behavioral specification language for C programs centered on the notion of function contract. It is used to state pre- and post-conditions, assertions, and invariants.

For each function $\tilde{f}^{\tau}$ in the transformed program, a C procedure is automatically generated. In addition, each function f in the original specification is expressed as a logic axiomatic definition in ACSL syntax. This definition can be seen as a predicate modeling the real-valued expected behavior of the function. The floating-point version $\tilde{f}$ of f is also expressed as an ACSL definition.

An ACSL predicate called *f_stable_paths* is introduced to model under which conditions real and floating-point flows coincide. ACSL preconditions are added to relate each C floating-point expression with its logic real-valued counterpart through the error variable representing its round-off error. As mentioned in Section 3, a new error variable $e := \epsilon_{var}(\widetilde{ae})$ is introduced for each floating-point arithmetic expression $\widetilde{ae}$ occurring in the conditional guards. For each new error variable, a precondition stating that $|\widetilde{ae} - R_{\mathbb{A}}(\widetilde{ae})| \leq e$ is added. A post-condition is introduced for each function stating that, when the transformed function $\tilde{f}^{\tau}$ does not emit a warning, the predicate *f_stable_paths* holds and the difference between $\tilde{f}^{\tau}$ and its real-number specification f is at most the round-off error computed for the stable paths of $\tilde{f}$. For the functions containing for-loops, a recursive real-valued version is generated as a logic axiomatic function in ACSL. An invariant is also computed in order to relate the result of each iteration of the for-loop with the corresponding call of the recursive real-valued function.

Example 2. Consider the real-valued specification *tcoa* and the instrumented function $\widetilde{tcoa}^{\tau}$ defined in Example 1.The pseudo-code of the annotated C code generated by PRECiSA is shown below, the pseudo-code of the ACSL annotation

are printed in gray.

$$
\begin{array}{l}
/{*}@\ \textit{logic auxiliary functions:}\\
\quad \textit{real tcoa}(\textit{real } s_z, \textit{real } v_z) = s_z * v_z < 0\ ?\ -(s_z/v_z) : 0\\
\quad \textit{double fp_tcoa}(\textit{double } \tilde{s}_z, \textit{double } \tilde{v}_z) = \tilde{s}_z \tilde{*} \tilde{v}_z < 0\ ?\ \tilde{-}(\tilde{s}_z \tilde{/} \tilde{v}_z) : 0\\
\quad \textit{predicate tcoa_stable_paths}(\textit{real } s_z, \textit{real} v_z, \textit{double } \tilde{s}_z, \textit{double } \tilde{v}_z) =\\
\qquad (v_z \neq 0 \wedge s_z * v_z < 0 \wedge \tilde{v}_z \neq 0 \wedge \tilde{s}_z \tilde{*} \tilde{v}_z < 0) \vee (s_z * v_z \geq 0 \wedge \tilde{s}_z \tilde{*} \tilde{v}_z \geq 0)\\
\quad \textbf{requires} : 0 \leq e\\
\quad \textbf{ensures} : \textit{result} \neq \omega \implies (\textit{result} = \textit{fp_tcoa}(\tilde{s}_z, \tilde{v}_z)\\
\quad \wedge \forall s_z, v_z (|(\tilde{s}_z \tilde{*} \tilde{v}_z) - (s_z * v_z)| \leq e \implies \textit{tcoa_stable_paths}(s_z, v_z, \tilde{s}_z, \tilde{v}_z)) {*}/\\
\textit{double tau_tcoa } (\textit{double } \tilde{s}_z, \textit{double } \tilde{v}_z, \textit{double } e)\{\\
\quad \textit{if } (\tilde{s}_z \tilde{*} \tilde{v}_z < -e)\{\textit{return } \tilde{-}(\tilde{s}_z \tilde{/} \tilde{v}_z);\}\\
\quad \textit{else } \{ \textit{ if } (\tilde{s}_z \tilde{*} \tilde{v}_z \geq e)\{\textit{return } 0;\}\\
\qquad \textit{else } \{\textit{return } \omega;\}\}\}
\end{array}
$$

As already mentioned, PRECiSA handles programs with symbolic parameters and generates a symbolic expression modeling an over-estimation of the round-off error that may occur. Given input ranges for the variables, a numerical evaluation of the symbolic expressions is performed in PRECiSA with the help of Kodiak, a rigorous global optimizer for real-valued expressions. Kodiak performs a branch-and-bound search that computes a sound enclosure for a symbolic error expression using either interval arithmetic or Bernstein polynomial basis. Therefore, it is possible to instantiate the error variables in the transformed program with numerical values representing a provably correct round-off error over-estimation.

Example 3. The following function instantiates the symbolic function shown in Example 2 assuming that $1 \leq s_z \leq 1000$ and $1 \leq v_z \leq 1000$.

$$
\begin{array}{l}
/{*}@\textbf{ensures} : \forall s_z, v_z (1 \leq s_z \leq 1000 \wedge 1 \leq v_z \leq 1000 \wedge \textit{result} \neq \omega \wedge\\
\qquad |\tilde{s}_z - s_z| \leq \textit{ulp}(s_z)/2 \wedge |\tilde{v}_z - v_z| \leq \textit{ulp}(v_z)/2)\\
\qquad \implies |\textit{result} - \textit{tcoa}(s_z, v_z)| \leq 2.78e-12\ {*}/\\
\textit{double tau_tcoa_num}(\textit{double } \tilde{s}_z, \textit{double } \tilde{v}_z)\{\\
\quad \textit{return tau_tcoa } (\tilde{s}_z, \tilde{v}_z, 1.72e-10)\}
\end{array}
$$

Besides the transformed C program, PRECiSA generates PVS theorems that act as formal certificates of the soundness of the computed estimations with respect to the floating-point IEEE-754 standard [16]. These theorems are automatically discharged in PVS by proof strategies that recursively inspect the round-off error expression and apply the corresponding lemmas included in the PVS floating-point round-off error formalization [7]. The instrumented C code for the program *WCV* defined in Example 1 and the corresponding PVS certificates are generated

by PRECiSA[8] in 7.12 seconds. The C code consists of approximately 500 lines of code including all the ACSL annotations.

The tool suite Frama-C [17] is used to compute a set of verification conditions (VCs) stating the relationship between the transformed floating-point program and the original real-valued specification. Frama-C includes several static analyzers for the C language that support ACSL annotations. The Frama-C WP plug-in implements the weakest precondition calculus for ACSL annotations through C programs. For each annotation, Frama-C computes a set of verification conditions in the form of mathematical first-order logic formulas. These verification conditions can be proved by a combination of external automated theorem provers, proof assistants, and SMT solvers.

The WP plug-in has been customized to support the PVS certificates generated by PRECiSA in the proof of correctness of the C program. PRECiSA also provides a collection of PVS proof strategies that automatically discharge the VCs generated by Frama-C. To prove the VCs for a particular function f, it is necessary to use not only properties about floating-point numbers but also the contracts of the functions that are called in the body of f. These proofs are quite tedious and error-prone since several renaming and reformulation steps are applied by Frama-C to the annotated C code. The PVS strategies follow the syntactic structure of the input functions to determine which properties and contracts are needed to prove each of the VCs generated by Frama-C. Therefore, no expertise in floating-point arithmetic or in PVS is required to verify the correctness of the generated C code.

Example 4. Consider again the pseudo-code for *tcoa* depicted in Example 2. The verification conditions computed by Frama-C for the functions *tau_tcoa* and *tau_tcoa_num* are the following.

$$\begin{aligned}
\varphi_{tau_tcoa} &= \forall e, s_z, v_z, e_s, e_v \in \mathbb{R}, \tilde{s}, \tilde{v} \in \mathbb{F} \\
&(\mathit{result} \neq \omega \wedge e \geq 0 \wedge |\tilde{v}_z - v_z| \leq e_v \wedge |\tilde{s}_z - s_z| \leq e_s \wedge |(\tilde{s}_z \tilde{*} \tilde{v}_z) - (v_z * s_z)| \leq e \\
&\Rightarrow |\mathit{result} - \mathit{tcoa}(s_z, v_z)| \leq \epsilon_{\tilde{f}}(s_z, e_s, v_z, e_v)).
\end{aligned}$$

$$\begin{aligned}
\varphi_{tau_tcoa_num} &= \forall s_z, v_z \in \mathbb{R}, \tilde{s}_z, \tilde{v}_z \in \mathbb{F}, (\mathit{result} \neq \omega \wedge 1 \leq \tilde{s}_z \leq 1000 \wedge 1 \leq \tilde{v}_z \leq 1000 \\
&\wedge |s_z - \tilde{s}_z| \leq \tfrac{1}{2}\, ulp(s_z) \wedge |v_z - \tilde{v}_z| \leq \tfrac{1}{2}\, ulp(v_z) \wedge |(\tilde{s}_z \tilde{*} \tilde{v}_z) - (v_z * s_z)| \leq \mathit{1.72e\text{-}10}) \\
&\Rightarrow |\mathit{result} - \mathit{tcoa}(s_z, v_z)| \leq \mathit{2.78e\text{-}12}
\end{aligned}$$

The expression $\epsilon_{\tilde{f}}(s_z, e_s, v_z, e_v)$ denotes the symbolic error bound computed by PRECiSA, the variable e denotes the round-off error of the expression $\tilde{s}_z \tilde{*} \tilde{v}_z$, which is introduced when the Boolean approximations β^+ and β^- are applied. The proof of these verification conditions follows from the fact that when *result* is not a warning ω, it is equal to $\widetilde{tcoa}(\tilde{s}_z, \tilde{v}_z)$ and from the numerical certificates output by PRECiSA stating that $|\widetilde{tcoa}(\tilde{s}_z, \tilde{v}_z) - tcoa(s_z, v_z)| \leq \epsilon_{\tilde{f}}(s_z, e_s, v_z, e_v)) =$ *2.78e-12*.

[8] This example is available at `https://shemesh.larc.nasa.gov/fm/PRECiSA`.

5 Related Work

Several tools are available for analyzing numerical aspects of C programs. In this work, the Frama-C [17] analyzer is used. Support for floating-point round-off error analysis in Frama-C is provided by the integration with the tool Gappa [12]. However, the applicability of Gappa is limited to straight-line programs without conditionals. Gappa's ability to verify more complex programs requires adding additional ACSL intermediate assertions and providing hints through annotation that may be unfeasible to automatically generate. The interactive theorem prover Coq can also be used to prove verification conditions on floating-point numbers thanks to the formalization defined in [6]. Nevertheless, Coq [2] tactics need to be implemented to automatize the verification process. Several approaches have been proposed for the verification of numerical C code by using Frama-C in combination with Gappa and/or Coq [4,5,3,13,18,30]. In [20], a preliminary version of the technique presented in this paper is used to verify a specific case study of a point-in-polygon containment algorithm. In contrast to the present work, the aforementioned techniques are not fully automatic and they require the user intervention in both the specification and verification processes.

Besides Frama-C, other tools are available to formally verify and analyze numerical properties of C code. Fluctuat [14] is a commercial static analyzer that, given a C program with annotations about input bounds and uncertainties on its arguments, produces an estimation of the round-off error of the program. Fluctuat detects the presence of possible unstable guards in the analyzed program, as explained in [15], but does not instrument the program to emit a warning in these cases. The static analyzer Astrée [9] detects the presence of run-time exceptions such as division by zero and under and over-flows by means of sound floating-point abstract domains. In contrast to the approach presented here, neither Fluctuat nor Astrée emit proof certificates that can be externally checked by an external prover.

Precision allocation (or tuning) tools, such as FPTuner [8], Precimonius [26], and Rosa [11], aim at selecting the lowest floating-point precision for the program variables that is enough to achieve the desired accuracy. Rosa soundly deals with unstable guards and with bounded loops when the variables appearing in the loop are restricted to a finite domain. In contrast with the approach presented in this paper, Rosa does not instrument the program to emit a warning when an unstable guard may occur. This means that the target precision may be difficult to reach without additional optimization rewritings and a program transformation as the one presented in this work. Program optimization tools aim at improving the accuracy of floating-point programs by rewriting arithmetic expressions in equivalent ones with a lower accumulated round-off error. Examples of these tools are Herbie [24], AutoRNP [32], Salsa [10], and CoHD [28].

6 Conclusion

Unstable guards, which occur when rounding errors affect the evaluation of conditional statements, are hard to detect and fix without the expert use of

specialized tools. This paper presents a toolchain that automatically generates and verifies floating-point C code that soundly detects the presence of unstable guards with respect to an ideal real number specification.

The proposed toolchain relies on different formal tools and formal techniques that have been integrated to make the generation and verification processes fully automatic. As part of the proposed approach, the program transformation originally proposed in [31] has been enhanced and improved. The floating-point static analyzer PRECiSA [19,29] has been extended with two modules. One module implements the transformation defined in Section 3. The other module generates the corresponding C/ACSL code. Thus, given a PVS program specification written in real arithmetic and the desired precision, PRECiSA automatically generates a guard-stable floating-point version in C syntax enriched with ACSL annotations. Additionally, PVS proof certificates are automatically generated by PRECiSA to ensure the correctness of the round-off error overestimations used in the program transformation.

The absence of unstable guards in the resulting floating-point implementation and the soundness of the computed round-off errors are automatically verified using a combination of Frama-C, PRECiSA, and PVS. The Frama-C/WP [17] plug-in customization developed in this work enabled a seamless integration between the proof obligations generated by Frama-C and the PVS certificates generated by PRECiSA. Having externally checkable certificates increases the level of confidence in the proposed approach. In addition, no theorem proving expertise is required from the user since proof strategies, which have been implemented as part of this work, automatically discharge the verification conditions generated by Frama-C. To the best of authors' knowledge, this is the first automatic technique that is able to generate a formally-verified floating-point program instrumented to detect unstable guards from a real-valued specification. The proposed program transformation is designed to correctly detect any divergence of flow with respect to the original program. However, due to the error over-estimation used in the Boolean approximation functions, false warnings may arise. The number of false warnings depends on the accuracy of the round-off error approximation computed by PRECiSA, which has been shown in [29] to be the most precise round-off error estimator handling programs with let-in, conditionals, and function calls.

An interesting future direction is the integration of the proposed approach with numerical optimization tools such as Salsa [10] and Herbie [24]. This integration will improve the accuracy of the mathematical expressions used inside a program and, at the same time, prevent unstable guards that may cause unexpected behaviors. The proposed approach could also be combined with tuning precision techniques [8,11]. Since the program transformation lowers the overall round-off error, this would likely increase the chance of finding a precision allocation meeting the target accuracy. Finally, the authors plan to enhance the approach to support floating-point special values and exceptions such as under- and over-flows and division by zero.

References

1. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, version 1.12 (2016)
2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004)
3. Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P.: Wave equation numerical resolution: A comprehensive mechanized proof of a C program. Journal of Automatic Reasoning **50**(4), 423–456 (2013)
4. Boldo, S., Filliâtre, J.C.: Formal verification of floating-point programs. In: Proceedings of ARITH18 2007. pp. 187–194. IEEE Computer Society (2007)
5. Boldo, S., Marché, C.: Formal verification of numerical programs: From C annotated programs to mechanical proofs. Mathematics in Computer Science **5**(4), 377–393 (2011)
6. Boldo, S., Melquiond, G.: Flocq: A unified library for proving floating-point algorithms in Coq. In: 20th IEEE Symposium on Computer Arithmetic, ARITH 2011. pp. 243–252. IEEE Computer Society (2011)
7. Boldo, S., Muñoz, C.: A high-level formalization of floating-point numbers in PVS. Tech. Rep. CR-2006-214298, NASA (2006)
8. Chiang, W., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., Rakamarić, Z.: Rigorous floating-point mixed-precision tuning. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017. pp. 300–315. ACM (2017)
9. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival: The ASTREÉ Analyzer. In: Proceedings of the 14th European Symposium on Programming (ESOP 2005). Lecture Notes in Computer Science, vol. 3444, pp. 21–30. Springer (2005)
10. Damouche, N., Martel, M.: Salsa: An Automatic Tool to Improve the Numerical Accuracy of Programs. 6th Workshop on Automated Formal Methods, AFM 2017 (2017)
11. Darulova, E., Kuncak, V.: Sound compilation of reals. In: Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 235–248. ACM (2014)
12. de Dinechin, F., Lauter, C., Melquiond, G.: Certifying the floating-point implementation of an elementary function using Gappa. IEEE Trans. on Computers **60**(2), 242–253 (2011)
13. Goodloe, A., Muñoz, C., Kirchner, F., Correnson, L.: Verification of numerical programs: From real numbers to floating point numbers. In: Proceedings of NFM 2013. Lecture Notes in Computer Science, vol. 7871, pp. 441–446. Springer (2013)
14. Goubault, E., Putot, S.: Static analysis of numerical algorithms. In: Proceedings of SAS 2006. Lecture Notes in Computer Science, vol. 4134, pp. 18–34. Springer (2006)
15. Goubault, E., Putot, S.: Robustness analysis of finite precision implementations. In: Proceedings of APLAS 2013. Lecture Notes in Computer Science, vol. 8301, pp. 50–57. Springer (2013)
16. IEEE: IEEE standard for binary floating-point arithmetic. Tech. rep., Institute of Electrical and Electronics Engineers (2008)
17. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Aspects of Computing **27**(3), 573–609 (2015)

18. Marché, C.: Verification of the functional behavior of a floating-point program: An industrial case study. Science of Computer Programming **96**, 279–296 (2014)
19. Moscato, M., Titolo, L., Dutle, A., Muñoz, C.: Automatic estimation of verified floating-point round-off errors via static analysis. In: Proceedings of the 36th International Conference on Computer Safety, Reliablilty, and Security, SAFECOMP 2017. Springer (2017)
20. Moscato, M., Titolo, L., Feliú, M., Muñoz, C.: Provably correct floating-point implementation of a point-in-polygon algorithm. In: Proceedings of the 23nd International Symposium on Formal Methods (FM 2019) (2019)
21. Muñoz, C., Narkawicz, A., Hagen, G., Upchurch, J., Dutle, A., Consiglio, M.: DAIDALUS: Detect and Avoid Alerting Logic for Unmanned Systems. In: Proceedings of the 34th Digital Avionics Systems Conference (DASC 2015). Prague, Czech Republic (September 2015)
22. Narkawicz, A., Muñoz, C.: A formally verified generic branching algorithm for global optimization. In: Proceedings of the 5th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE). pp. 326–343. Springer (2013)
23. Owre, S., Rushby, J., Shankar, N.: PVS: A prototype verification system. In: Proceedings of the 11th International Conference on Automated Deduction (CADE). pp. 748–752. Springer (1992)
24. Panchekha, P., Sanchez-Stern, A., Wilcox, J., Z., T.: Automatically improving accuracy for floating point expressions. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015. pp. 1–11. ACM (2015)
25. RTCA SC-228: DO-365, Minimum Operational Performance Standards for Detect and Avoid (DAA) Systems (May 2017)
26. Rubio-González, C., Nguyen, C., Nguyen, H., Demmel, J., Kahan, W., Sen, K., Bailey, D., Iancu, C., Hough, D.: Precimonious: tuning assistant for floating-point precision. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13. p. 27. ACM (2013)
27. Smith, A.P., Muñoz, C., Narkawicz, A.J., Markevicius, M.: A rigorous generic branch and bound solver for nonlinear problems. In: Proceedings of the 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2015. pp. 71–78 (2015)
28. Thévenoux, L., Langlois, P., Martel, M.: Automatic source-to-source error compensation of floating-point programs. In: 18th IEEE International Conference on Computational Science and Engineering, CSE 2015. pp. 9–16. IEEE Computer Society (2015)
29. Titolo, L., Feliú, M., Moscato, M., Muñoz, C.: An abstract interpretation framework for the round-off error analysis of floating-point programs. In: Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI). pp. 516–537. Springer (2018)
30. Titolo, L., Moscato, M., Muñoz, C., Dutle, A., Bobot, F.: A formally verified floating-point implementation of the compact position reporting algorithm. In: Proceedings of the 22nd International Symposium on Formal Methods (FM 2018). Lecture Notes in Computer Science, vol. 10951, pp. 364–381. Springer (2018)
31. Titolo, L., Muñoz, C., Feliú, M., Moscato, M.: Eliminating unstable tests in floating-point programs. In: Proceedings of the 28th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2018). pp. 169–183. Springer (2018)
32. Yi, X., Chen, L., Mao, X., Ji, T.: Efficient automated repair of high floating-point errors in numerical libraries. PACMPL **3**(POPL), 56:1–56:29 (2019)