Provably Correct Floating-Point Implementation of a Point-in-Polygon Algorithm

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

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

Abstract. The problem of determining whether or not a point lies inside a given polygon occurs in many applications. In air traffic management concepts, a correct solution to the point-in-polygon problem is critical to geofencing systems for Unmanned Aerial Vehicles and in weather avoidance applications. Many mathematical methods can be used to solve the point-in-polygon problem. Unfortunately, a straightforward floating-point implementation of these methods can lead to incorrect results due to round-off errors. In particular, these errors may cause the control flow of the program to diverge with respect to the ideal real-number algorithm. This divergence potentially results in an incorrect point-in-polygon determination even when the point is far from the edges of the polygon. This paper presents a provably correct implementation of a point-in-polygon method that is based on the computation of the winding number. This implementation is mechanically generated from a source-to-source transformation of the ideal real-number specification of the algorithm. The correctness of this implementation is formally verified within the Frama-C analyzer, where the proof obligations are discharged using the Prototype Verification System (PVS).

1 Introduction

PolyCARP (Algorithms for Computations with Polygons) [25,27] is a NASA developed open source software library for geo-containment applications based on polygons.[3] One of the main applications of PolyCARP is to provide geofencing capabilities to unmanned aerial vehicles (UAV), i.e., detecting whether a UAV is inside or outside a given geographical region, which is modeled using a 2D polygon with a minimum and a maximum altitude. Another application is detecting if an aircraft's trajectory encounters weather cells, which are modeled as moving polygons.

PolyCARP implements point-in-polygon methods, i.e., methods for checking whether or not a point lies inside a polygon, that are based on the *winding number* computation. The winding number of a point p with respect to a polygon

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

[3] https://shemesh.larc.nasa.gov/fm/PolyCARP.

is the number of times any point traveling counterclockwise along the perimeter of the polygon winds around p. Properties of these methods have been formally verified in the Prototype Verification System (PVS) [28]. A correct implementation of these methods is essential to safety-critical geo-containment applications that rely on PolyCARP.

When an algorithm involving real numbers is implemented using floating-point numbers, round-off errors arising from the difference between real-number computations and their floating-point counterparts may affect the correctness of the algorithm. In fact, floating-point implementations of point-in-polygon methods are very sensitive to round-off errors. For instance, the presence of floating-point computations in Boolean expressions of conditional statements may cause the control flow of the floating-point program to diverge from the ideal real-number program, resulting in the wrong computation of the winding number. This may happen even when the point is far from the edges of the polygon.

This paper presents a formally verified floating-point C implementation of the winding number algorithm. This implementation is obtained by applying a program transformation to the original algorithm. This transformation replaces numerically unstable conditions with more restrictive ones that preserve the control flow of the ideal real number specification. The transformed program is guaranteed to return a warning when real and floating-point flows may diverge. The program transformation used is an extension of the one defined in [32] and it has been implemented within PRECiSA[4] (Program Round-off Error Certifier via Static Analysis), a static analyzer of floating-point programs [24,30].

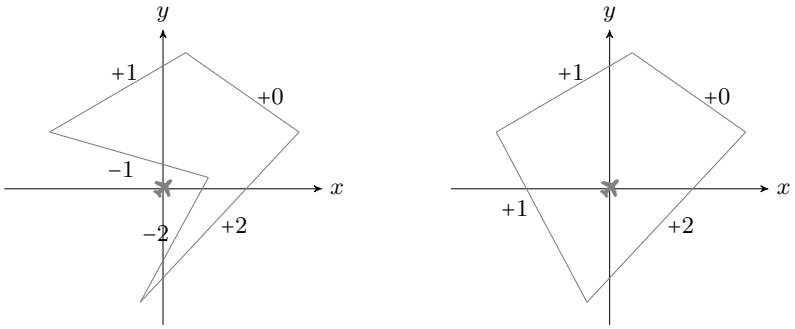
Frama-C [20] is used to formally verify the correctness of the generated C program. Frama-C is a collaborative platform that hosts several plugins for the verification and analysis of C code. In particular, in this work, an extension of the Frama-C/WP (Weakest Precondition calculus) plugin is implemented to automatically generate verification conditions that can be discharged in PVS.

The rest of this paper is organized as follows. Section 2 presents the definition of the winding number. An extension of the program transformation defined in [30] is presented in Section 3. In Section 4, the transformed floating-point version of the winding number is introduced. The verification approach used to prove the correctness of the C floating-point implementation of the transformed program is explained in Section 5. Related work is discussed in Section 6. Finally, Section 7 concludes the paper.

2 The Winding Number Algorithm

The winding number of a point s with respect to a polygon P is defined as the number of times the perimeter of P travels counterclockwise around s. For simple polygons, i.e., the ones that do not contain intersecting edges, this function can be used to determine whether s is inside or outside P. In [25], the winding number of s with respect to P is computed by applying a geometric translation that sets

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



(a) The sum of the contributions is 0 and the point is outside.

(b) The sum of the contributions is 4 and the point is inside.

Fig. 1: Winding number edge contributions

s as the origin of coordinates. For each edge e of P, the algorithm counts how many axes e intersects. This contribution can be positive or negative, depending on the direction of the edge e. If the sum of all contributions from all edges is 0 then s is outside the perimeter of P, otherwise, it is inside. Fig. 1 shows the edge contributions in the computation of the winding number for two different polygons.

Mathematical functions that define the winding number algorithm are presented in Fig. 2. Given a point $v = (v_x, v_y)$, the function *Quadrant* returns the quadrant in which v is located. Given the endpoints of an edge e, $v = (v_x, v_y)$ and $v' = (v'_x, v'_y)$, and the point under test $s = (s_x, s_y)$, the function $EdgeContrib(v_x, v_y, v'_x, v'_y, s_x, s_y)$ computes the number of axes e intersects in the coordinate system centered in s. This function checks in which quadrants v and v' are located and counts how many axes are crossed by the edge e. If v and v' belong to the same quadrant, the contribution of the edge to the winding number is 0 since no axis is crossed. If v and v' lie in adjacent quadrants, the contribution is 1 (respectively -1) if moving from v to v' along the edge is in counterclockwise (respectively clockwise) direction. In the case v and v' are in opposite quadrants, the determinant is computed to check the direction of the edge. If it is counterclockwise, the contribution is 2; otherwise, it is -2. The function *WindingNumber* takes as input a point $s = (s_x, s_y)$ and a polygon P of size n, which is represented as a couple of arrays $\langle P_x, P_y \rangle$ modeling the coordinates of its vertices $(P_x(0), P_y(0)) \ldots (P_x(n-1), P_y(n-1))$. The size of a polygon is defined as the number of its vertices. The winding number of s with respect to the polygon P is obtained as the sum of the contributions of all the edges in P. The result of the winding number is 0 if and only if the polygon P does not wind around the point s, hence s lies outside P.

$$
\begin{aligned}
&Quadrant(v_x, v_y) = \ \mathit{if}\ v_x \geq 0 \wedge v_y \geq 0\ \mathit{then}\ 1 \\
&\qquad\qquad\quad \mathit{elsif}\ v_x < 0 \wedge v_y \geq 0\ \mathit{then}\ 2 \\
&\qquad\qquad\quad \mathit{elsif}\ v_x < 0 \wedge v_y < 0\ \mathit{then}\ 3 \\
&\qquad\qquad\quad \mathit{else}\ 4
\end{aligned}
$$

$$
\begin{aligned}
&EdgeContrib(v_x, v_y, v'_x, v'_y, s_x, s_y) = \\
&\quad \mathit{let}\ this_x = v_x - s_x, this_y = v_y - s_y, next_x = v'_x - s_x, next_y = v'_y - s_y, \\
&\qquad dist_x = next_x - this_x, dist_y = next_y - this_y, \\
&\qquad det = dist_x \cdot this_y - dist_y \cdot this_x \\
&\qquad q_{this} = Quadrant(this_x, this_y), q_{next} = Quadrant(next_x, next_y)\ \mathit{in} \\
&\quad \mathit{if}\ q_{this} = q_{next}\ \mathit{then}\ 0 \\
&\quad \mathit{elsif}\ q_{next} - 1 = \mathsf{mod}\,(q_{this}, 4))\ \mathit{then}\ 1 \\
&\quad \mathit{elsif}\ q_{this} - 1 = \mathsf{mod}\,(q_{next}, 4))\ \mathit{then}\ -1 \\
&\quad \mathit{elsif}\ det \leq 0\ \mathit{then}\ 2 \\
&\quad \mathit{else}\ -2
\end{aligned}
$$

$$
\begin{aligned}
&WindingNumber(P_x, P_y, s_x, s_y, i) = \\
&\quad \mathit{if}\ i < size(P_x) - 1 \\
&\quad \mathit{then}\ EdgeContrib(P_x(i), P_y(i), P_x(i+1), P_y(i+1), s_x, s_y) \\
&\qquad + WindingNumber(P_x, P_y, s_x, s_y, i+1) \\
&\quad \mathit{else}\ EdgeContrib(P_x(i), P_y(i), P_x(0), P_y(0), s_x, s_y)
\end{aligned}
$$

Fig. 2: Winding number algorithm

It has been formally verified in PVS, that the algorithm presented in Fig. 2 is equivalent to an alternative point-in-polygon algorithm.[5] The following property is therefore assumed.

Property 1. Given a simple polygon $P = \langle P_x, P_y \rangle$ and a point $s = (s_x, s_y)$, s lies outside P if and only if $WindingNumber(P_x, P_y, s_x, s_y, 0) = 0$.

A formal proof of Property 1 that does not rely on an alternative algorithmic method to check point containment is a hard problem beyond the scope of this paper. In particular, a proof of this statement involving a non-algorithmic definition of containment may require the formal development of fundamental topological concepts such as the Jordan Curve theorem.

3 Program Transformation to Avoid Unstable Tests

Floating-point numbers are widely used to represent real numbers in computer programs since they offer a good trade-off between efficiency and precision. A

[5] `https://github.com/nasa/PolyCARP`.

floating-point number can be formalized as a pair of integers $(m, e) \in \mathbb{Z}^2$, where m is called the *significand* and e the *exponent* of the float [7,13]. Henceforth, $\mathbb{F}$ will denote the set of floating-point numbers. A conversion function $R : \mathbb{F} \to \mathbb{R}$ is defined to refer to the real number represented by a given float, i.e., $R((m, e)) = m \cdot b^e$ where b is the base of the representation. According to the IEEE-754 standard [19], each floating-point operation must be computed as if its result is first calculated correct to infinite precision and with unbounded range and then rounded to fit a particular floating-point format.

The main drawback of using floating-point numbers is the presence of *round-off errors* that originate from the difference between the ideal computation in real arithmetic and the actual floating-point computation. Let $\tilde{v}$ be a floating-point number that represents a real number r, the difference $|R(\tilde{v}) - r|$ is called the *round-off error* (or *rounding error*) of $\tilde{v}$ with respect to r. Rounding errors accumulate during the program execution and may affect the evaluation of both arithmetic and Boolean expressions. As a consequence, when guards of if-then-else statements contain floating-point expressions, as in the case of the winding number, the output of a program is not only directly influenced by rounding errors, but also by the error of taking the opposite branch with respect to the real number intended behavior. This problem is known as *test instability*. A conditional statement (or test) *if* $\tilde{\phi}$ *then* S_1 *else* S_2 is said to be *unstable* when $\tilde{\phi}$ evaluates to a different Boolean value than its real-valued counterpart.

In [32], a formally proven[6] program transformation is proposed to detect and correct the effects of unstable tests for a simple language with conditionals and let-in expressions. The output of the transformation is a floating-point program that is guaranteed to return either the result of the original floating-point one, when it can be assured that both the real and its floating-point flows agree, or a warning, when these flows may diverge. In this paper, the transformation defined in [32] has been extended to handle non-recursive function calls and simple for-loops. This extended transformation is then applied to the winding number algorithm.

Henceforth, the symbols $\mathbb{A}$ and $\widetilde{\mathbb{A}}$ denote the domain of arithmetic expressions over real and floating-point numbers, respectively. 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 function $R_{\widetilde{\mathbb{A}}} : \widetilde{\mathbb{A}} \to \mathbb{A}$ converts an arithmetic expression on floating-point numbers to an arithmetic expression on real numbers. This function is defined by simply 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. By abuse of notation, floating-point expressions are interpreted as their real number evaluation when occurring inside a real-valued expression. The symbols $\mathbb{B}$ and $\widetilde{\mathbb{B}}$ denote the domain of Boolean expressions over real and floating-point numbers, respectively. The function $R_{\widetilde{\mathbb{B}}} : \widetilde{\mathbb{B}} \to \mathbb{B}$ converts a Boolean expression on floating-point numbers to a Boolean expression on real numbers. Given a variable assignment

[6] The PVS formalization is available at `https://shemesh.larc.nasa.gov/fm/PRECiSA`.

$\sigma : \mathbb{V} \to \mathbb{R}$, $eval_{\mathbb{B}}(\sigma, B) \in \{true, false\}$ denotes the evaluation of the real Boolean expression B. Similarly, given $\widetilde{B} \in \widetilde{\mathbb{B}}$ and $\widetilde{\sigma} : \widetilde{\mathbb{V}} \to \mathbb{F}$, $\widetilde{eval}_{\widetilde{\mathbb{B}}}(\widetilde{\sigma}, \widetilde{B}) \in \{true, false\}$ denotes the evaluation of the floating-point Boolean expression $\widetilde{B}$. A program is defined as a set of function declarations of the form $f(\tilde{x}_1, \ldots, \tilde{x}_n) = S$, where S is a program expression that can contain binary and n-ary conditionals, let expressions, arithmetic expressions, non-recursive function calls, for-loops, and a warning exceptional statement ω. Given a set Σ of function symbols, the syntax of program expressions S is given by the following grammar.

$$\begin{aligned} S ::= & \widetilde{A} \mid \textit{if } \widetilde{B} \textit{ then } S \textit{ else } S \mid \textit{if } \widetilde{B} \textit{ then } S \; [\textit{elsif } \widetilde{B} \textit{ then } S]_{i=1}^{m} \textit{ else } S \\ & \mid \textit{let } \tilde{x} = \widetilde{A} \textit{ in } S \mid \textit{for}(i_0, i_n, acc_0, \lambda(i, acc).S) \mid \; g(\widetilde{A}, \ldots, \widetilde{A}) \mid \omega, \end{aligned} \tag{3.1}$$

where $\widetilde{A} \in \widetilde{\mathbb{A}}$, $\widetilde{B} \in \widetilde{\mathbb{B}}$, $\tilde{x}, i, acc \in \widetilde{\mathbb{V}}$, $g \in \Sigma$, $m \in \mathbb{N}^{>0}$, and $i_0, i_n, acc_0 \in \mathbb{N}$. The notation $[\textit{elsif } \widetilde{B} \textit{ then } S]_{i=1}^{m}$ denotes a list of m *elsif* branches. The *for* expression emulates a for loop where i 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 S is the body of the loop. For instance, $\textit{for}(1, 10, 0, \lambda(i, acc).i + acc)$ represents the value $f(1, 0)$, where f is the recursive function $f(i, acc) \equiv \textit{if } i > 10 \textit{ then } acc \textit{ else } f(i + 1, acc + i)$. The set of program expressions is denoted as $\mathbb{S}$, while the set of programs is denoted as $\mathbb{P}$.

The proposed transformation takes into account round-off errors by replacing the Boolean expressions in the guards of the original program with more restrictive ones. This is done by means of two abstractions $\beta^+, \beta^- : \widetilde{\mathbb{B}} \to \widetilde{\mathbb{B}}$ defined as follows for conjunctions and disjunctions of sign tests, where $\widetilde{expr} \in \widetilde{\mathbb{A}}$ and $\epsilon \in \widetilde{\mathbb{V}}$ is a variable that represents the rounding error of $\widetilde{expr}$ such that $|\widetilde{expr} - R_{\mathbb{A}}(\widetilde{expr})| \leq \epsilon$ and $\epsilon \geq 0$.

$$\begin{aligned}
&\beta^+(\widetilde{expr} \leq 0) = \widetilde{expr} \leq -\epsilon && \beta^-(\widetilde{expr} \leq 0) = \widetilde{expr} > \epsilon \\
&\beta^+(\widetilde{expr} \geq 0) = \widetilde{expr} \geq \epsilon && \beta^-(\widetilde{expr} \geq 0) = \widetilde{expr} < -\epsilon \\
&\beta^+(\widetilde{expr} < 0) = \widetilde{expr} < -\epsilon && \beta^-(\widetilde{expr} < 0) = \widetilde{expr} \geq \epsilon \\
&\beta^+(\widetilde{expr} > 0) = \widetilde{expr} > \epsilon && \beta^-(\widetilde{expr} > 0) = \widetilde{expr} \leq -\epsilon \\
&\beta^+(\tilde{\phi}_1 \wedge \tilde{\phi}_2) = \beta^+(\tilde{\phi}_1) \wedge \beta^+(\tilde{\phi}_2) && \beta^-(\tilde{\phi}_1 \wedge \tilde{\phi}_2) = \beta^-(\tilde{\phi}_1) \vee \beta^-(\tilde{\phi}_2) \\
&\beta^+(\tilde{\phi}_1 \vee \tilde{\phi}_2) = \beta^+(\tilde{\phi}_1) \vee \beta^+(\tilde{\phi}_2) && \beta^-(\tilde{\phi}_1 \vee \tilde{\phi}_2) = \beta^-(\tilde{\phi}_1) \wedge \beta^-(\tilde{\phi}_2) \\
&\beta^+(\neg\tilde{\phi}) = \beta^-(\tilde{\phi}) && \beta^-(\neg\tilde{\phi}) = \beta^+(\tilde{\phi})
\end{aligned}$$

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 $\beta^+(\tilde{\phi})$ implies both $\tilde{\phi}$ and its real counterpart, while $\beta^-(\tilde{\phi})$ implies both the negation of $\tilde{\phi}$ and the negation of its real counterpart. The proof is available as part of the PVS formalization defined in [32].

Lemma 1. *Given $\tilde{\phi} \in \widetilde{B}$, let $fv(\tilde{\phi})$ be the set of free variables in $\tilde{\phi}$. For all $\sigma : \{\chi_r(\tilde{x}) \mid \tilde{x} \in fv(\tilde{\phi})\} \to \mathbb{R}$, $\tilde{\sigma} : fv(\tilde{\phi}) \to \mathbb{F}$, and $\tilde{x} \in fv(\tilde{\phi})$ such that $R(\tilde{\sigma}(\tilde{x})) = \sigma(\chi_r(\tilde{x}))$, 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. $\widetilde{eval}_{\widetilde{\mathbb{B}}}(\tilde{\sigma}, \beta^-(\tilde{\phi})) \Rightarrow \widetilde{eval}_{\widetilde{\mathbb{B}}}(\tilde{\sigma}, \neg\tilde{\phi}) \wedge eval_{\mathbb{B}}(\sigma, \neg\, R_{\mathbb{B}}(\tilde{\phi}))$.

The transformation function $\tau : \mathbb{S} \to \mathbb{S}$ applies β^+ and β^- to the guards in the conditionals. For binary conditional statements, τ is defined as follows.

– If $\tilde{\phi} \neq \beta^+(\tilde{\phi})$ or $\tilde{\phi} \neq \beta^-(\tilde{\phi})$:

$$\tau(\textit{if } \tilde{\phi} \textit{ then } S_1 \textit{ else } S_2) = \textit{if } \beta^+(\tilde{\phi}) \textit{ then } \tau(S_1) \textit{ elseif } \beta^-(\tilde{\phi}) \textit{ then } \tau(S_2) \textit{ else } \omega;$$

– If $\tilde{\phi} = \beta^+(\tilde{\phi})$ and $\tilde{\phi} = \beta^-(\tilde{\phi})$:

$$\tau(\textit{if } \tilde{\phi} \textit{ then } S_1 \textit{ else } S_2) = \textit{if } \tilde{\phi} \textit{ then } \tau(S_1) \textit{ else } \tau(S_2).$$

When the round-off error does not affect the evaluation of the Boolean expression, i.e., $\tilde{\phi} = \beta^+(\tilde{\phi})$ and $\tilde{\phi} = \beta^-(\tilde{\phi})$, the transformation is just applied to the subprograms S_1 and S_2. Otherwise, the *then* branch of the transformed program is taken when $\beta^+(\tilde{\phi})$ is satisfied. From Lemma 1, it follows that both $\tilde{\phi}$ and $R(\tilde{\phi})$ hold and, thus, the *then* branch is taken in both real and floating-point control flows. Similarly, the *else* branch of the transformed program is taken when $\beta^-(\tilde{\phi})$ holds. This means 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. In the case of the for-loop, the transformation is applied to the body of the loop.

$$\tau(for(i_0, i_n, acc_0, \lambda(i, acc).S)) = for(i_0, i_n, acc_0, \lambda(i, acc).\tau(S)). \tag{3.2}$$

Given a program $P \in \mathbb{P}$, the transformation $\bar{\tau} : \mathbb{P} \to \mathbb{P}$ is defined as follows.

$$\bar{\tau}(P) = \bigcup\{f^\tau(\tilde{x}_1, \ldots, \tilde{x}_n, e_1, \ldots, e_m) = \tau(S) \mid f(\tilde{x}_1, \ldots, \tilde{x}_n) = S \in P\}, \tag{3.3}$$

where τ is applied to the body of the function and new arguments $e_1, \ldots, e_m$ are added to represent the round-off error of the arithmetic expressions occurring in the body of each test in S. When either β^+ or β^- is applied to a test in the body of S, e.g. $\widetilde{expr} < 0$, a new fresh variable e is introduced representing the round-off error of the arithmetic expression occurring in the test. This fresh variable becomes a new argument of the function and a pre-condition is imposed stating that $|\widetilde{expr} - R_{\mathbb{A}}(\widetilde{expr})| \leq e$. In addition, for every function call $g(A_1, \ldots, A_n, e'_1, \ldots, e'_k)$ occurring in S, the error variables of g, $e'_1, \ldots, e'_k$, are added as additional arguments to f.

When a function g is called, it is necessary to check if the returning value is a warning ω. Let $g(A_1, \ldots, A_n)$ be a call to the function $g(x_1, \ldots, x_n) = S$ in the original program with actual parameters $A_1, \ldots, A_n \in \widetilde{A}$. Additionally, let $g^\tau(x_1, \ldots, x_n, e_1, \ldots, e_m) = \tau(S)$ be the corresponding function declaration in the transformed program such that for all $i = 1 \ldots m$, $\widetilde{expr}_i$ is an arithmetic expression occurring in a transformed test and $|\widetilde{expr}_i - R_{\mathbb{A}}(\widetilde{expr}_i)| \leq e_i$. The transformation of the function call is defined as follows:

$$\tau(g(A_1, \ldots, A_n)) = \textit{if } A_1 = \omega \textit{ then } \omega$$

$$
\begin{array}{l}
\vdots \\
\mathit{elsif}\ A_n = \omega\ \mathit{then}\ \omega \\
\mathit{else}\ g^{\tau}(A_1, \ldots, A_n, e'_1, \ldots, e'_m),
\end{array}
$$

where for all $i = 1 \ldots m$, e'_i is such that $|\widetilde{expr}_i[x_i/A_i]_{i=1}^n - R_{\mathbb{A}}(\widetilde{expr}_i[x_i/A_i]_{i=1}^n)| \leq e'_i$. In this case, the information regarding the error variables is instantiated with the actual parameters of the function.

The following theorem states the correctness of the program transformation. The transformed program is guaranteed to return either the result of the original floating-point program, when it can be assured that both its real and floating-point flows agree or a warning ω when these flows may diverge.

Theorem 1 (Program Transformation Correctness). *Given $P \in \mathbb{P}$, for all $f(\tilde{x}_1,\ldots,\tilde{x}_n) = S \in P$, $\sigma : \{x_1 \ldots x_n\} \to \mathbb{R}$, and $\tilde{\sigma} : \{\tilde{x}_1 \ldots \tilde{x}_n\} \to \mathbb{F}$, such that for all $i \in \{1, \ldots, n\}$, $R(\tilde{\sigma}(\tilde{x}_i)) = \sigma(x_i)$:*

$$f^{\tau}(\tilde{x}_1, \ldots, \tilde{x}_n, e_1, \ldots, e_m) \neq \omega \iff f(x_1, \ldots, x_n) = f^{\tau}(\tilde{x}_1, \ldots, \tilde{x}_n, e_1, \ldots, e_m)$$

where $f^{\tau}(\tilde{x}_1, \ldots, \tilde{x}_n, e_1, \ldots, e_m) \in \bar{\tau}(P)$.

Theorem 1 follows from Lemma 1 and the definition of the program transformation $\bar{\tau}$. It has been formally proved in PVS for the particular case of the winding number transformation. A general PVS proof of this statement for an arbitrary program is under development.

4 Test-Stable Version of the Winding Number

The use of floating-point numbers to represent real values introduces test instability in the program defined in Section 2. A technique used in PolyCARP to mitigate the uncertainty of floating-point computations in the winding number algorithm is to consider a buffer area around the perimeter of the polygon that is assumed to contain the points that may produce instability. As part of this work, the PRECiSA static analyzer is used to validate if a buffer that protects against instability exists. PRECiSA accepts as input a floating-point program and computes a sound over-approximation of the floating-point accumulated round-off error that may occur in each computational path of the program. In addition, the corresponding path conditions are also collected for both stable and unstable cases. When real and floating-point flows diverge, PRECiSA outputs the Boolean conditions under which the instability occurs.

Given the unstable conditions produced by PRECiSA for the winding number algorithm, an over-approximation of the region of instability is generated by using the paving functionality of the Kodiak global optimizer [26]. Concrete examples for these instability conditions are searched in the instability region by using the FPRoCK [29] solver, a tool able to check the satisfiability of mixed real and floating-point Boolean expressions. As an example, consider the edge (v, v'), where $v = (1, 1)$ and $v' = (3, 2)$, in the polygon depicted in Fig. 3. The

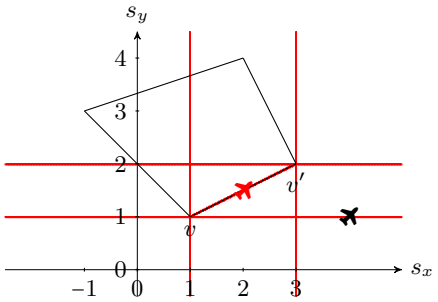


Fig. 3: Points that cause instability in *EdgeContrib* and *WindingNumber*.

red lines represent a guaranteed over-approximation of the values for s_x and s_y that may cause instability in the function *EdgeContrib* with respect to the considered edge. The black aircraft denotes a case in which the contribution of the edge (v, v') has a different value in real and floating-point arithmetic. In fact, when $s_x = 4$ and $s_y \approx 1.0000000000000001$, the real function *EdgeContrib* returns -1, indicating that v and v' are located in adjacent quadrants. However, its floating-point counterpart returns 0 meaning that the vertices are located in the same quadrant. The red aircraft represents the point $s_x \approx 2.0000000000000002$, $s_y = 1.5$, for which the main function *WindingNumber* returns 0, i.e., the point is outside, when evaluated with real arithmetics, and it returns 4, i.e., the point is inside, when evaluated in floating-point arithmetic. This figure suggests that simply considering a buffer around the edge is not enough to guarantee the correct behavior of the *EdgeContrib* function since errors in the contribution can happen also when the point is far from the boundaries. It has been conjectured that, for this algorithm, when the checked point is far from the edges of the polygon, the error occurring in one edge is compensated with the error of another edge of the polygon in the computation of the winding number. To the authors' knowledge, no formal proof of this statement exists.

The floating-point program depicted in Fig. 4 is obtained by applying the transformation $\bar{\tau}$ from Section 3 to the real-number winding number algorithm presented in Fig. 2. The function $Quadrant^{\tau}$ has two additional arguments, e_x and e_y, modeling the round-off errors of v_x and v_y, respectively. Thus,

$$|v_x - \chi_r(v_x)| \leq e_x,\ |v_y - \chi_r(v_y)| \leq e_y, \text{ and } e_x, e_v \geq 0. \tag{4.1}$$

The tests are approximated by means of the functions β^+ and β^- by replacing the value 0 with the error variables e_x and e_y.

The function $EdgeContrib^{\tau}$ contains two calls to $Quadrant^{\tau}$. Therefore, it is necessary to check if any of these calls return a warning ω. If this is the case, $EdgeContrib^{\tau}$ also returns ω since a potential instability has been detected in the calculation of $Quadrant^{\tau}$. The function $EdgeContrib^{\tau}$ has five additional arguments with respect to its real number counterpart *EdgeContrib*. Besides e_{det} that represents the error of the expression calculating the determinant, the

error variables appearing in the calls to $Quadrant^\tau$ are considered: e_{this_x}, e_{this_y}, e_{next_x}, and e_{next_y}. The new parameters are such that:

$$\begin{aligned}
&|this_x - R_{\mathbb{A}}(this_x)| \le e_{this_x},\ |this_y - R_{\mathbb{A}}(this_y)| \le e_{this_y},\\
&|next_x - R_{\mathbb{A}}(next_x)| \le e_{next_x}\ |next_y - R_{\mathbb{A}}(next_y)| \le e_{next_y}\\
&|det - R_{\mathbb{A}}(det)| \le e_{det},\ \text{and } e_{this_x}, e_{this_y}, e_{next_x}, e_{next_y}, e_{det} \ge 0.
\end{aligned} \tag{4.2}$$

The conditional in the main function $WindingNumber^\tau$ does not introduce any new error variable, therefore just the error parameters in the calls to *EdgeContrib* are considered. Let $n = size(P_x)$ be the size of the polygon P, and let *fdet* be the function calculating the determinant, which is defined as follows

$$\begin{aligned}
fdet(v_x, v_y, v'_x, v'_y, s_x, s_y) = \quad & ((v_x - s_x) - (v'_x - s_x)) \cdot (v'_y - s_y)\\
& -((v_y - s_y) - (v'_y - s_y)) \cdot (v'_x - s_x).
\end{aligned} \tag{4.3}$$

The error variables e_x, e_y, and e_{det} are such that:

$$\begin{aligned}
&e_x, e_y, e_{det} \ge 0,\\
&\forall i = 0 \ldots n-1 : |(P_x(i) - s_x) - R_{\mathbb{A}}(P_x(i) - s_x)| \le e_x,\\
&\qquad\qquad\quad |(P_y(i) - s_y) - R_{\mathbb{A}}(P_y(i) - s_y)| \le e_y,\\
&\forall i = 0 \ldots n-2 : |fdet(P_x(i+1), P_y(i+1), P_x(i), P_y(i), s_x, s_y)\\
&\qquad\qquad\quad - R_{\mathbb{A}}(fdet(P_x(i+1), P_y(i+1), P_x(i), P_y(i), s_x, s_y))| \le e_{det},\\
&\qquad\qquad\quad |fdet(P_x(0), P_y(0), P_x(n-1), P_y(n-1), s_x, s_y)\\
&\qquad\qquad\quad - R_{\mathbb{A}}(fdet(P_x(0), P_y(0), P_x(n-1), P_y(n-1), s_x, s_y))| \le e_{det}.
\end{aligned} \tag{4.4}$$

5 Verification Approach

This section presents the approach used to obtain a formally verified test-stable C implementation of the winding number algorithm that uses floating-point numbers. The toolchain is comprised of the PVS interactive prover, the static analyzer PRECiSA, and the Frama-C analyzer. The input is a real-valued program P expressed in the PVS specification language. The output is a C implementation of P that correctly detects and corrects unstable tests. An overview of the approach is depicted in Fig. 5.

As already mentioned, PRECiSA is a static analyzer that computes an overestimation of the round-off error that may occur in a program. In addition, it automatically generates a PVS proof certificate ensuring the correctness of the computed bound. In this work, PRECiSA is extended to implement the transformation defined in Section 3 and to generate the corresponding C code. Given a desired floating-point format (single or double precision), PRECiSA is used to convert the PVS real-number version of the winding-number algorithm defined in Section 2 into a floating-point program. This is done by replacing all the real operators with their floating-point counterpart and by approximating

$$
\begin{aligned}
&Quadrant^{\tau}(v_x, v_y, e_x, e_y) = \mathit{if}\ v_x \geq e_x \wedge v_y \geq e_y\ \mathit{then}\ 1\\
&\qquad\qquad\qquad\qquad\quad \mathit{elsif}\ v_x < -e_x \wedge v_y \geq e_y\ \mathit{then}\ 2\\
&\qquad\qquad\qquad\qquad\quad \mathit{elsif}\ v_x < -e_x \wedge v_y < -e_y\ \mathit{then}\ 3\\
&\qquad\qquad\qquad\qquad\quad \mathit{elsif}\ v_x \geq e_x \wedge v_y < -e_y\ \mathit{then}\ 4\\
&\qquad\qquad\qquad\qquad\quad \mathit{else}\ \omega
\end{aligned}
$$

$$
\begin{aligned}
&EdgeContrib^{\tau}(v_x, v_y, v'_x, v'_y, s_x, s_y, e_{this_x}, e_{this_y}, e_{next_x}, e_{next_y}, e_{det}) =\\
&\quad \mathit{let}\ this_x = v_x - s_x, this_y = v_y - s_y, next_x = v'_x - s_x, next_y = v'_y - s_y,\\
&\qquad dist_x = next_x - this_x, dist_y = next_y - this_y, det = dist_x \cdot this_y - dist_y \cdot this_x,\\
&\qquad q_{this} = Quadrant^{\tau}(this_x, this_y, e_{this_x}, e_{this_y}),\\
&\qquad q_{next} = Quadrant^{\tau}(next_x, next_y, e_{next_x}, e_{next_y})\ \mathit{in}\\
&\quad \mathit{if}\ q_{this} = \omega\ \mathit{or}\ q_{next} = \omega\ \mathit{then}\ \omega\\
&\quad \mathit{elsif}\ q_{this} = q_{next}\ \mathit{then}\ 0\\
&\quad \mathit{elsif}\ q_{next} - 1 = \mathsf{mod}\,(q_{this}, 4)\ \mathit{then}\ 1\\
&\quad \mathit{elsif}\ q_{this} - 1 = \mathsf{mod}\,(q_{next}, 4)\ \mathit{then}\ -1\\
&\quad \mathit{elsif}\ (det \leq -e_{det})\ \mathit{then}\ 2\\
&\quad \mathit{elsif}\ (det > e_{det})\ \mathit{then}\ -2\\
&\quad \mathit{else}\ \omega
\end{aligned}
$$

$$
\begin{aligned}
&WindingNumber^{\tau}(P_x, P_y, s_x, s_y, i, e_x, e_y, e_{det}) =\\
&\ \mathit{if}\ i < n-1\ \mathit{then}\\
&\quad (\mathit{if}\ EdgeContrib^{\tau}(P_x(i), P_y(i), P_x(i+1), P_y(i+1), s_x, s_y, e_x, e_y, e_x, e_y, e_{det}) = \omega\\
&\quad\ \mathit{then}\ \omega\\
&\quad\ \mathit{else}\ EdgeContrib^{\tau}(P_x(i), P_y(i), P_x(i+1), P_y(i+1), s_x, s_y, e_x, e_y, e_x, e_y, e_{det})\\
&\qquad + WindingNumber^{\tau}(P_x, P_y, s_x, s_y, i+1, e_x, e_y, e_x, e_y, e_{det}))\\
&\ \mathit{else}\\
&\quad (\mathit{if}\ EdgeContrib^{\tau}(P_x(i), P_y(i), P_x(0), P_y(0), s_x, s_y, e_x, e_y, e_x, e_y, e_{det}) = \omega\\
&\quad\ \mathit{then}\ \omega\\
&\quad\ \mathit{else}\ EdgeContrib^{\tau}(P_x(i), P_y(i), P_x(0), P_y(0), s_x, s_y, e_x, e_y, e_x, e_y, e_{det})
\end{aligned}
$$

Fig. 4: Pseudo-code on floating-point arithmetic of the transformed winding number algorithm

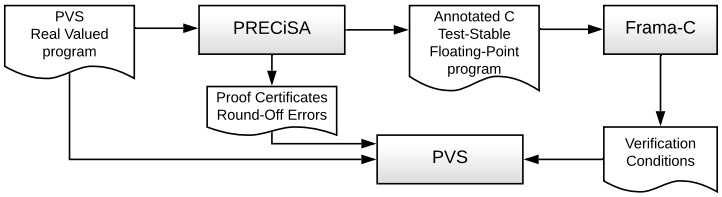


Fig. 5: Verification approach.

the real variables and constants with their floating-point representation. The integer operations, variables, and constants are left unchanged since they do not carry round-off errors. Subsequently, the transformation presented in Section 3 is applied. To facilitate the translation from PVS to C syntax, the function *WindingNumber* has been reformulated using the for-iterate scheme introduced in Equation (3.1) that emulates an imperative for-loop in a functional setting.

$$\begin{aligned} &\mathit{WindingNumber}(P_x, P_y, s_x, s_y, i) = \\ &\quad \mathit{for}(0, \mathit{size}(P_x) - 1, 0, \lambda i, \mathit{acc}.\ \ \mathit{if}\ i < \mathit{size}(P_x) - 2 \\ &\qquad \mathit{then}\ \mathit{acc} + \mathit{EdgeContrib}(P_x(i), P_y(i), P_x(i+1), P_y(i+1), s_x, s_y) \\ &\qquad \mathit{else}\ \ \mathit{acc} + \mathit{EdgeContrib}(P_x(i), P_y(i), P_x(0), P_y(0), s_x, s_y)). \end{aligned} \tag{5.1}$$

The result of the transformation is the program shown in Fig. 4 where a for-loop replaces the recursive call in the main function $\mathit{WindingNumber}^{\tau}$.

The transformed program is then converted in C syntax with ACSL annotations. The ANSI/ISO C Specification Langage (ACSL [1]) is a behavioral specification language for C programs centered on the notion of function contract. For each function in the transformed program, a C procedure is automatically generated. In addition, the functions in the original version of the winding number algorithm, defined in Section 2, are rephrased as ACSL axiomatic logic functions. For each function, ACSL preconditions are added to relate C floating-point expressions with their corresponding logic real-valued counterpart through the error variable representing their round-off error. As mentioned in Section 4, a fresh error variable e is introduced for each floating-point arithmetic expression $\widetilde{expr}$ occurring in the conditional tests. For each new error variable, a precondition stating that $|\widetilde{expr} - R_{\mathbb{A}}(\widetilde{expr})| \leq e$ is added.

The loop invariant of the function $\mathit{WindingNumber}^{\tau}$ is specified as an ACSL annotation before the for-loop as follows

$$\begin{aligned} \forall i = 0 \ldots \mathit{size}(P_x).\ \ &\mathit{if}\ \ \mathit{acc} = 0\ \mathit{then}\ 0 \\ &\mathit{else}\ \mathit{acc} = \mathit{WindingNumber}^{\tau}(P_x, P_y, s_x, s_y, i-1, e_x, e_y, e_{det}). \end{aligned}$$

This information is required in order to prove the correctness of each iteration of the for-loop and has to be provided as an input to PRECiSA together with the input program. In addition, PRECiSA identifies the for-loop variant $\mathit{size}(P_x) - i$ that is also needed for the verification of the loop. For each function, a post-condition is added stating that if the result is different from ω, then the result

of the C function is the same as the real-valued logic function that corresponds to the initial PVS specification.

To verify the correctness of the C code generated by PRECiSA with respect to the accompanying ACSL contracts, an extension of the Weakest Precondition (WP) plug-in of Frama-C has been developed. This plug-in implements the weakest precondition calculus for ACSL annotations of C programs. For each ACSL annotation, the plug-in generates a set of verification conditions (VCs) that can be discharged by a suite of external provers. In this work, support for generating PVS VCs is added to the Frama-C/WP plug-in. This extension links the generated VCs with the formal certificates generated by PRECiSA regarding the round-off errors and the original PVS formalization of the winding number. Frama-C/WP generates a set of PVS declarations from the ACSL logic definitions. These declarations are proved to be mathematically equivalent to the original winding number PVS formalization (Fig. 2) in the PVS theorem prover. In addition, Frama-C/WP computes a set of verification conditions from the pre and post conditions stating the correctness of the C program with respect to the ACSL logic definitions. The verification conditions generated for the functions $Quadrant^{\tau}$ and $EdgeContrib^{\tau}$ are formalized in the following lemmas.

Lemma 2. *Let* $v_x, v_y, e_x, e_y \in \widetilde{\mathbb{V}}$ *such that* $|v_x - \chi_r(v_x)| \leq e_x$ *and* $|v_y - \chi_r(v_y)| \leq e_y$, *if* $Quadrant^{\tau}(v_x, v_y, e_x, e_y) \neq \omega$, *then* $Quadrant(v_x, v_y) = Quadrant^{\tau}(v_x, v_y, e_x, e_y)$.

Lemma 3. *Let* $v_x, v_y, v'_x, v'_y, s_x, s_y, e_{this_x}, e_{this_y}, e_{next_x}, e_{next_y}, e_{det} \in \widetilde{\mathbb{V}}$ *such that the inequalities in Equation* (4.2) *hold.*

If $EdgeContrib^{\tau}(v_x, v_y, v'_x, v'_y, s_x, s_y, e_{this_x}, e_{this_y}, e_{next_x}, e_{next_y}, e_{det}) \neq \omega$, *then* $EdgeContrib(v_x, v_y, v'_x, v'_y, s_x, s_y) = EdgeContrib^{\tau}(v_x, v_y, v'_x, v'_y, s_x, s_y, e_{this_x}, e_{this_y}, e_{next_x}, e_{next_y}, e_{det})$.

The following theorem summarizes the verification conditions generated for the main function $WindingNumber^{\tau}$. All these verification conditions are proven with the help of the PVS theorem prover[7].

Theorem 2. *Let* $v_x, v_y, v'_x, v'_y, s_x, s_y, e_x, e_y, e_{det} \in \widetilde{\mathbb{V}}$ *and* $P = \langle P_x, P_y \rangle$ *a polygon of size* n *such that for all* $i = 0 \ldots n-1$ $P_x(i), P_y(i) \in \widetilde{\mathbb{V}}$ *and the inequalities in Equation* (4.4) *hold.*

If $WindingNumber^{\tau}(P_x, P_y, s_x, s_y, i, e_x, e_y, e_x, e_y, e_{det}) \neq \omega$, *then* $WindingNumber(P_x, P_y, s_x, s_y, i) = WindingNumber^{\tau}(P_x, P_y, s_x, s_y, i, e_x, e_y, e_{det})$.

The parameters representing the round-off errors of the arithmetic expressions occurring in the body of each function can be instantiated with concrete numerical values. Given numerical bounds for the input variables, the numerical error values are automatically computed by PRECiSA by means of the Kodiak global optimizer [26]. For example, assuming $P_x(i), s_x \in [-1000, 1000]$ for all $i = 0..size(P_x)$, PRECiSA computes the upper bound $3.637978807091714 \times 10^{-12}$ for the error variable e_x meaning that $|(P_x(i) - s_x) - R_{\mathbb{A}}(P_x(i) - s_x)| \leq$

[7] The PVS verification conditions generated by Frama-C and their proofs can be found at `https://shemesh.larc.nasa.gov/fm/PolyCARP`.

$3.637978807091714 \times 10^{-12}$. PRECiSA also emits the proof certificates ensuring that the numerical result computed by Kodiak is a correct over-approximation of the round-off error occurring in the considered expression.

The PRECiSA certificates prove the correctness of the round-off error bounds used in the program transformation. They are essential to ensure that the transformed program is correct, i.e., the Boolean abstractions β^+ and β^- are correctly over-estimating the conditional tests and, thus, Lemma 1 holds. Additionally, they are used to prove the verification conditions generated by Frama-C/WP, for instance, the preconditions on the error defined in Equations (4.2) and (4.3).

6 Related Work

Several techniques and tools have been developed to formally verify properties of C programs related to floating-point numbers. Fluctuat and Astrée are commercial tools based on abstract interpretation [11], which have been successfully used to verify and analyze numerical properties for industrial and safety-critical C code, including aerospace software. Fluctuat [18] is a static analyzer that computes round-off error bounds for C programs with annotations. Astrée [12] is a fully-automatic static analyzer that uses sound floating-point abstract domains [9,23] to uncover the presence of run-time exceptions such as division by zero and under and over-flows. Astrée has been applied to automatically check the absence of runtime errors associated with floating-point computations in aerospace control software [2]. For instance, the fly-by-wire primary software of commercial airplanes is verified with the help of Astrée [14]. Moreover, Astrée and Fluctuat have been used in combination to analyze on-board software acting in the Monitoring and Safing Unit of the ATV space vehicle [8]. In contrast to the technique presented in this paper, the above-mentioned approaches do not provide formal proof certificates that can be discharged in an external prover. This is particularly useful for safety-critical systems since the proof certificates improve the trustworthiness of the approach. In addition, in contrast with the tools used in this paper, Fluctuat and Astrée are not open-source.

Caduceus [5,16] is a tool that produces verification conditions from annotated C code with the help of the platform Why [3]. Similarly, in [6], a chain of tools composed of Frama-C, the Jessie plug-in [22], and Why is used to automatically generate verification conditions, which are checked by several external provers. These approaches were used to formally verify wave propagation differential equations [4], a pairwise state-based conflict detection algorithm [17], and numerical properties of industrial software related to inertial navigation [21]. In [31], a combination of Frama-C and PVS was used to verify a numerically improved version of the Compact Position Reporting (CPR) algorithm, a key component of the ADS-B protocol allowing aircraft to share their position. In this case, Frama-C was used to generate verification conditions discharged using the SMT solver Alt-Ergo [10] and the prover Gappa [15]. PVS was employed to prove the equivalence between the original implementation of the CPR algorithm and the improved one. In contrast to [31], the verified C code presented in

this paper is automatically generated from the PVS specification. None of the approaches mentioned before tackles the problem of detecting unstable tests.

7 Conclusion

In this paper, a formal approach is proposed to generate and to verify a test-stable version of the winding number algorithm. This version is obtained by applying an extension of the program transformation defined in [30] that over-approximates the Boolean expressions occurring in conditional statements. The over-approximation soundly handles round-off errors that may occur in the numerical computation of the expression. The transformed program is guaranteed to return the same output with respect to the original algorithm when real and floating-point flows match. When the correct output cannot be guaranteed, a warning is issued. The static analyzer PRECiSA [24,30] is enhanced with a module implementing this transformation and with a C/ACSL code generator. Thus, given the PVS program specification of the winding number assuming real numbers arithmetics, PRECiSA automatically generates its test-stable floating-point version in C syntax enriched with ACSL annotations. This approach can be applied to generic algorithms involving non-recursive function calls, conditionals, and let-in expressions.

The generated C implementation of the winding number is analyzed within the Frama-C tool suite. In this work, the Frama-C/WP [20] plug-in is extended to generate verification conditions in PVS syntax. These verification conditions state that the transformed floating-point version of the winding number is correct with respect to its real-valued specification, meaning that if the C implementation answers that a point is inside (or outside) a polygon the same answer would be obtained in the ideal real number implementation of the original algorithm. The verification conditions generated by Frama-C are proven correct within the PVS theorem prover.

The verification of the correctness of the transformed C program relies on three different tools: the PVS interactive prover, the Frama-C analyzer, and PRECiSA. All of these tools are based on rigorous mathematical foundations and have been used in the verification of industrial and safety-critical systems. The C floating-point transformed program, the PVS verification conditions, and the round-off errors bounds are automatically generated. However, the verification approach proposed in this work requires some level of expertise for proving the PVS verification conditions generated by Frama-C. In the future, the authors plan to define proof strategies that automatically discharge these PVS verification conditions.

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. Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Static Analysis and Verification of Aerospace Software by Abstract Interpretation. Foundations and Trends in Programming Languages **2**(2-3), 71–190 (2015)
3. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Let's verify this with Why3. International Journal on Software Tools for Technology Transfer **17**(6), 709–727 (2015)
4. Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P.: Wave equation numerical resolution: A comprehensive mechanized proof of a C program. Journal of Automatic Reasoning **50**(4), 423–456 (2013)
5. Boldo, S., Filliâtre, J.C.: Formal verification of floating-point programs. In: Proceedings of ARITH18 2007. pp. 187–194. IEEE Computer Society (2007)
6. 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)
7. Boldo, S., Muñoz, C.: A high-level formalization of floating-point numbers in PVS. Tech. Rep. CR-2006-214298, NASA (2006)
8. Bouissou, O., Conquet, E., Cousot, P., Cousot, R., Feret, J., Goubault, E., Ghorbal, K., Lesens, D., Mauborgne, L., Miné, A., Putot, S., Rival, X., Turin, M.: Space Software Validation using Abstract Interpretation. In: Proceedings of the International Space System Engineering Conference, Data Systems in Aerospace, DASIA 2009. pp. 1–7. ESA publications (2009)
9. Chen, L., Miné, A., Cousot, P.: A Sound Floating-Point Polyhedra Abstract Domain. In: Proceedings of the 6th Asian Symposium on Programming Languages and Systems, APLAS 2008. Lecture Notes in Computer Science, vol. 5356, pp. 3–18. Springer (2008)
10. Conchon, S., Contejean, E., Kanig, J., Lescuyer, S.: CC(X): Semantic Combination of Congruence Closure with Solvable Theories. Electronic Notes in Theoretical Computer Science **198**(2), 51 – 69 (2008)
11. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Conference Record of tha 4th ACM Symposium on Principles of Programming Languages, POPL 1977. pp. 238–252. ACM (1977)
12. 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)
13. Daumas, M., Rideau, L., Théry, L.: A Generic Library for Floating-Point Numbers and Its Application to Exact Computing. In: Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics. pp. 169–184. Springer Berlin Heidelberg (2001)
14. Delmas, D., Souyris, J.: Astrée: From research to industry. In: Proceedings of the 14th International Symposium on Static Analysis, SAS 2007. pp. 437–451 (2007)
15. 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)
16. Filliâtre, J.C., Marché, C.: Multi-prover verification of C programs. In: Proceedings of the 6th International Conference on Formal Engineering Methods, ICFEM 2004. Lecture Notes in Computer Science, vol. 3308, pp. 15–29. Springer (2004)
17. 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)

18. 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)
19. IEEE: IEEE standard for binary floating-point arithmetic. Tech. rep., Institute of Electrical and Electronics Engineers (2008)
20. 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)
21. Marché, C.: Verification of the functional behavior of a floating-point program: An industrial case study. Science of Computer Programming **96**, 279–296 (2014)
22. Marché, C., Moy, Y.: The Jessie Plugin for Deductive Verification in Frama-C (2017)
23. Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In: Proceedings of the 13th European Symposium on Programming Languages and Systems, ESOP 2004. Lecture Notes in Computer Science, vol. 2986, pp. 3–17. Springer (2004)
24. Moscato, M.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)
25. Narkawicz, A., Hagen, G.: Algorithms for collision detection between a point and a moving polygon, with applications to aircraft weather avoidance. In: Proceedings of the AIAA Aviation Conference (2016)
26. Narkawicz, A., Muñoz, C.: A formally verified generic branching algorithm for global optimization. In: Revised Selected Papers of VSTTE 2013. Lecture Notes in Computer Science, vol. 8164, pp. 326–343. Springer (2013)
27. Narkawicz, A., Muñoz, C., Dutle, A.: The MINERVA software development process. In: 6th Workshop on Automated Formal Methods, AFM 2017 (2017)
28. Owre, S., Rushby, J., Shankar, N.: PVS: A prototype verification system. In: Proceedings of CADE 1992. vol. 607, pp. 748–752. Springer (1992)
29. Salvia, R., Titolo, L., Feliú, M., Moscato, M., Muñoz, C., Rakamaric, Z.: A Mixed Real and Floating-Point Solver. In: 11th Annual NASA Formal Methods Symposium (NFM 2019) (2019)
30. 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 2018. vol. 10747, pp. 516–537. Springer (2018)
31. Titolo, L., Moscato, M.M., Muñoz, C.A., 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)
32. 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. vol. 10747, pp. 169–183. Springer (2018)