A provably correct floating-point implementation of Well Clear Avionics Concepts

Nikson Bernardes Fernandes Ferreira*, Mariano M. Moscato†, Laura Titolo†, and Mauricio Ayala-Rincón*‡
*Department of Computer Science
‡Department of Mathematics
University of Brasilia, Brasilia, Brazil
Email: niksonber@gmail.com, ayala@unb.br
†Analytical Mechanics Associates, Hampton, USA
Email: {mariano.moscato,laura.titolo}@ama-inc.com

***Abstract*—The NASA DAIDALUS library provides formal definitions for Detect-and-Avoid avionics concepts such as when an aircraft is well-clear with respect to the surrounding air traffic, i.e., it does not operate in such proximity to create a collision hazard. While several properties are proven correct for DAIDALUS assuming ideal real number arithmetic, an actual implementation that uses floating-point numbers may behave unexpectedly because of round-off errors and run-time exceptions. This paper presents an experience report on the application of a formal methods toolchain to extract and verify floating-point C code from a real-valued specification of the well-clear module of DAIDALUS. This toolchain comprises the PVS theorem prover, the PRECiSA floating-point analyzer and code generator, and the Frama-C analysis suite. The generated code is automatically instrumented to detect when the control flow of the floating-point program may diverge from the ideal real number specification, and it is annotated with contracts that state the maximum accumulated round-off error. The absence of overflows is also formally verified for the generated code. In order to apply the toolchain to an industrial case study such as DAIDALUS, a formally verified pre-processing of the input specification is performed, which includes a program slicing and several semantic-preserving simplifications.**

***Index Terms*—Program Verification, Floating-Point, PVS, Detect-and-Avoid**

I. Introduction

Midair conflicts are one of the most dangerous situations that may occur in the airspace domain. The USA Federal Aviation Administration (FAA) reported that over forty midair collisions occurred from January 2009 through December 2013 [1]. The primary mitigation to such situations is the longstanding principle of *See and Avoid*. In short, it states that a person operating an aircraft has the responsibility to remain vigilant to see and avoid nearby traffic [2]. The advent of Unmanned Aerial Systems (UAS) and their incorporation into the airspace introduced the need to restate this concept in terms suitable for aircraft with no crew onboard. The *Detect and Avoid* (DAA) concept emerged then as an effort to support the integration of UAVs into civil airspace. A DAA system is required to provide alerting and guidance to avoid potential conflicts.

Diverse industrial and governmental actors proposed algorithmic DAA solutions. Among them, NASA developed the Detect and Avoid Alerting Logic for Unmanned Systems library (DAIDALUS[1]) [3]. DAIDALUS provides prototypical open-source implementations in Java and C++, which were included as reference implementations of the DAA functional requirements described in RTCA's Minimum Operational Performance Standards (MOPS) DO-365 [4]. One distinguishing characteristic of DAIDALUS is that it also provides formal specifications of the algorithms along with proofs for correctness and safety properties on them, mechanically checked within the Prototype Verification System (PVS) [5]. These proofs assume ideal real number arithmetic. However, when implemented using floating-point representations, the properties may no longer hold because of round-off errors and runtime exceptions. The adherence of the implementations to the behavior modeled by the formal specifications was checked using a testing-based approach [6]. While such an approach is usually enough for non-critical applications, the correctness of DAA implementations calls for a higher level of assurance. Given the numerical nature of several functions in DAIDALUS, it is important to provide formal guarantees on the finite-precision implementation concerning the expected behavior specified using real-numbers arithmetic.

In the past, an integrated toolchain has been proposed to automatically extract and verify floating-point C code from real-valued specifications [7]. This toolchain consists of the PVS theorem prover, the PRECiSA floating-point analyzer and code generator ([8], [9]), and the Frama-C tool suite [10]. In a nutshell, PRECiSA automatically generates a floating-point C implementation from a PVS real number specification. The extracted C code contains program contracts that relate the floating-point computations with their ideal counterpart by the maximum round-off error that may occur. These contracts enable the use of the Frama-C analysis suite which automatically generates a set of verification conditions that can be proven correct with the help of diverse backends. The toolchain proposed in [7] included a customization on Frama-C that allowed it to generate the verification conditions in the language of PVS and connect them with the NASA PVS library (NASALib).

In [7], this technique was applied to one of the core

[1]DAIDALUS is available at https://github.com/nasa/daidalus.

functions of DAIDALUS. This paper describes the application and adaptation of this technique to one of the main modules in DAIDALUS which is devoted to the definition of *well-clear* concepts. Two aircraft are considered to be well clear of each other if appropriate distance and time variables determined by the relative aircraft states remain outside a set of predefined threshold values. Remaining outside of these threshold values guarantees they have adequate separation in relation to the surrounding traffic; therefore, midair collisions are not expected.

The toolchain presented in [7] could not be applied directly to the DAIDALUS specification because the code generation capability of PRECiSA, at its current stage, does not support some of the features of the PVS language used to formally define Well-Clear, such as abstract data types and higher-order functions. In addition, the complexity of the target module, given by the number and nature of the interactions between the functions composing it and the wide ramification of the control flow graph of the whole library, impacts on the efficiency of the analysis performed by PRECiSA and the legibility of the results of this analysis. In order to make the DAIDALUS specification manageable by the toolchain, this paper proposes to apply a semantic-preserving program slicing on a simplification from higher-order to first-order declarations. This program rewriting improved the performance of the generation and verification of the C code significantly. The obtained program is formally proven equivalent to the original specification within the PVS theorem prover. In addition, a new PVS floating-point formalization is used. This formalization extends the one used in [7] with explicit handling for special values such as NaNs and infinities. This change positively impacted the analysis by enabling the verification of the absence of these values. It also significantly improved the performance of the type checking in PVS. However, it resulted in many of the proof strategies developed in the past being unusable. Part of the work presented in this paper focuses on fixing and adapting the proofs generated by PRECiSA to this new formalization. More information about the PRECiSA project and the files related to the work presented in this paper can be found at https://shemesh.larc.nasa.gov/fm/PRECiSA/.

The paper is organized as follows. Section II describes DAIDALUS and explains the well-clear concept. An overview of the analysis approach is presented in Section III. The application of the slicing technique to the original specification is detailed in Section IV. Then, Section V explains the code extraction and the program instrumentation used to detect control-flow divergences between real and floating-point computations and how these conditions are verified using Frama-C and PVS. Finally, Section VI provides a brief discussion of the most relevant outcomes of this work, Section VII discusses the related work, and Section VIII concludes the paper.

II. THE DAIDALUS LIBRARY

DAIDALUS is a software library developed at NASA that implements a Detect-and-Avoid alerting logic for unmanned systems. In DAIDALUS, the condition of Well-Clear is defined in the context of an encounter between two aircraft, usually called the *ownship* and the *intruder*. These conditions are stated in an *intruder-centric* manner, meaning that the information describing the encounter is expressed relative to the state of the intruder. In particular, DAIDALUS includes definitions determining when the aircraft are in a situation of violation of well-clear. This violation occurs when (a) the two aircraft are already close enough, or (b) they will be close enough if they keep the same orientation and velocity. This notion is expressed in terms of horizontal (1) and vertical (2) well-clear violation.

This section presents a high-level description of the well-clear concepts defined in [11]. For details and further explanation, the reader is referred to that work. In the following, $\mathbf{s}$ and $\mathbf{v}$ denote vectors of dimension 3 and the subindices x, y, and z are used to indicate their first, second, and third component respectively. Two-dimensional vectors are used to describe the horizontal position ($\mathbf{s}_h \stackrel{\text{def}}{=} (s_x, s_y)$) and velocity ($\mathbf{v}_h \stackrel{\text{def}}{=} (v_x, v_y)$) of the ownship with respect to the intruder. Additionally, $\|\cdot\|$ denotes the Euclidean norm.

$$\text{WCV}_\text{H}(\mathbf{s}_h, \mathbf{v}_h) \stackrel{\text{def}}{=} \|\mathbf{s}_h\| \leq \delta_d \vee (0 \leq \tau_{mod}(\mathbf{s}_h, \mathbf{v}_h) \leq \delta_t \wedge d_{cpa}(\mathbf{s}_h, \mathbf{v}_h) \leq \delta_{hmd}) \quad (1)$$

The values δ_d, δ_t, and δ_{hmd} are parameters of the model, used as thresholds for distance and time. The function τ_{mod}, defined below, is an approximation for the time of closest point of approach, i.e., the instant in which both aircraft would be closer to each other than in any other moment. Below, and in the rest of this paper, the dot product between two vectors (for example, $\mathbf{a}$ and $\mathbf{b}$) is denoted by their juxtaposition ($\mathbf{ab}$).

$$\tau_{mod}(\mathbf{s}_h, \mathbf{v}_h) \stackrel{\text{def}}{=} \begin{cases} \frac{\delta_d^2 - \mathbf{s}_h^2}{\mathbf{s}_h \mathbf{v}_h} & \text{if } \mathbf{s}_h\mathbf{v}_h < 0 \\ -1 & \text{otherwise} \end{cases} \quad (2)$$

The function d_{cpa} calculates the projected horizontal distance between the aircraft at their closest point of approach, assuming the velocity and orientation remain constant. The definition of d_{cpa} relies on the actual calculation of the *time of closest point of approach* (t_{cpa}). Both notions are formally stated below.

$$d_{cpa}(\mathbf{s}_h, \mathbf{v}_h) \stackrel{\text{def}}{=} \|\mathbf{s}_h + t_{cpa}(\mathbf{s}_h, \mathbf{v}_h)\mathbf{v}_h\| \quad (3)$$

$$t_{cpa}(\mathbf{s}_h, \mathbf{v}_h) \stackrel{\text{def}}{=} \begin{cases} \frac{\mathbf{s}_h \cdot \mathbf{v}_h}{v_h^2} & \text{if } \|\mathbf{v}_h\| > 0 \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

The violation of vertical well clear is defined analogously to its horizontal counterpart; using the scalars vertical position s_z and velocity v_z, and the time to co-altitude (t_{coa}) instead of the time to the closest point of approach.

$$\text{WCV}_\text{V}(s_z, v_z) \stackrel{\text{def}}{=} |s_z| \leq \delta_z \vee 0 \leq t_{coa}(s_z, v_z) \leq \delta_{tcoa} \quad (5)$$

$$t_{coa}(s_z, v_z) \stackrel{\text{def}}{=} \begin{cases} \frac{-s_z}{v_z} & \text{if } s_z v_z < 0 \\ -1 & \text{otherwise} \end{cases} \quad (6)$$

Given their relative position and velocity, two aircraft are considered to be in well-clear violation when both horizontal and vertical violations occur.

$$\text{WCV}(\mathbf{s}, \mathbf{v}) \iff \text{WCV}_\text{V}(s_z, v_z) \wedge \text{WCV}_\text{H}(\mathbf{s}_h, \mathbf{v}_h) \quad (7)$$

The DAIDALUS library also provides conflict detection algorithms whose purpose is to check whether the well-clear condition is predicted to be violated within a given timeframe. The function WCVint_{V} computes a time interval, included in a given lookahead interval $\mathbf{t} = [b, t] \subset \mathbb{R}$, in which vertical well-clear is violated at every moment. If no such interval exists, the empty set is returned.

$$\text{WCV}int_{\mathsf{V}}(\mathbf{t}, s_z, v_z) \stackrel{\text{def}}{=} \begin{cases} \mathbf{t} & \text{if } v_z = 0 \wedge |s_z| \leq \delta_z \\ \varnothing & \text{if } v_z = 0 \wedge |s_z| > \delta_z \\ [\max(b, c_0), \min(t, c_F)] & \text{if } v_z \neq 0 \wedge b \leq c_0, c_F \leq t \\ \varnothing & \text{otherwise} \end{cases} \tag{8}$$

where $c_0 \stackrel{\text{def}}{=} \frac{-\text{sign}(v_z)\max(\delta_z, \delta_{tcoa}|v_z|) - s_z}{v_z}$ and $c_F \stackrel{\text{def}}{=} \frac{-\text{sign}(v_z)\delta_z - s_z}{v_z}$. Definitions such as c_0 and c_F, for which no equation numbers are provided, should be understood as syntactic abbreviations used to improve the presentation.

Similarly, the function WCVint_{H} returns a time interval included in $[0, t]$ in which the condition of horizontal well-clear is violated at every moment, if such interval exists.

$$\text{WCV}int_{\mathsf{H}}(t, \mathbf{s}_h, \mathbf{v}_h) \stackrel{\text{def}}{=} \begin{cases} [0, t] & \text{if } a = 0 \wedge \mathbf{s}_h^2 \leq \delta_D^2 \\ \left[0, \min(t, \Theta^+_{\mathbf{s}_h, \mathbf{v}_h})\right] & \text{if } a \neq 0 \wedge \mathbf{s}_h^2 \leq \delta_D^2 \\ \varnothing & \text{if } \mathbf{s}_h^2 > \delta_D^2 \wedge (\mathbf{s}_h\mathbf{v}_h \geq 0 \vee \Delta_{a,b,c} < 0) \\ [\max(0, r^-_{a,b,c}), \min(t, \Theta^+_{\mathbf{s}_h, \mathbf{v}_h})] & \\ & \text{if } \mathbf{s}_h^2 > \delta_D^2 \wedge \mathbf{s}_h\mathbf{v}_h < 0 \wedge \Delta_{a,b,c} \geq 0 \wedge \\ & \quad \Delta^{\mathbb{R}\times\mathbb{R}}_{\mathbf{s}_h, \mathbf{v}_h} \geq 0 \wedge r^-_{a,b,c} \leq t \\ \varnothing & \text{otherwise} \end{cases} \tag{9}$$

where

- $a \stackrel{\text{def}}{=} \mathbf{v}_h^2$,
- $b \stackrel{\text{def}}{=} 2\ \mathbf{s}_h\mathbf{v}_h + \delta_t\mathbf{v}_h^2$,
- $c \stackrel{\text{def}}{=} \mathbf{s}_h^2 + \delta_t\mathbf{s}_h\mathbf{v}_h - \delta_D^2$,
- $\Delta_{a,b,c} \stackrel{\text{def}}{=} b^2 - 4ac$,
- $r^-_{a,b,c} \stackrel{\text{def}}{=} \frac{1}{2a}\left(-b - \sqrt{b^2 - 4ac}\right)$,
- $\Theta^+_{\mathbf{s}_h, \mathbf{v}_h} \stackrel{\text{def}}{=} \frac{1}{\mathbf{v}_h^2}\left(-\mathbf{s}_h\mathbf{v}_h + \sqrt{(\mathbf{s}_h\mathbf{v}_h)^2 - 4\mathbf{v}_h^2(\mathbf{s}_h^2 - \delta_D^2)}\right)$, and
- $\Delta^{\mathbb{R}\times\mathbb{R}}_{\mathbf{s}_h, \mathbf{v}_h} \stackrel{\text{def}}{=} \delta_D^2\mathbf{v}_h^2 - (\mathbf{s}_h\mathbf{v}_h{}^{\perp})^2$.

The two functions defined above can be used to calculate a time interval of well-clear violation. In the following, $\mathsf{V} \stackrel{\text{def}}{=} \text{WCV}int_{\mathsf{V}}(\mathbf{t}, s_z, v_z)$, and $\mathsf{H} \stackrel{\text{def}}{=} \text{WCV}int_{\mathsf{H}}(t_{end} - t_{begin}, \mathbf{s}_h + lb(\mathsf{V})\mathbf{v}_h, \mathbf{v}_h)$ where $[t_{begin}, t_{end}] = \mathbf{t}$.

$$\text{WCV}int(\mathbf{t}, \mathbf{s}, \mathbf{v}) \stackrel{\text{def}}{=} \begin{cases} \varnothing & \text{if } \mathsf{V} = \varnothing \\ \mathsf{V} & \text{if } lb(\mathsf{V}) = ub(\mathsf{V}) \wedge \text{WCV}_{\mathsf{H}}(\mathbf{s}_h + lb(\mathsf{V})\mathbf{v}_h, \mathbf{v}_h) \\ \varnothing & \text{if } lb(\mathsf{V}) = ub(\mathsf{V}) \wedge \neg\text{WCV}_{\mathsf{H}}(\mathbf{s}_h + lb(\mathsf{V})\mathbf{v}_h, \mathbf{v}_h) \\ \varnothing & \text{if } ub(\mathsf{V}) - lb(\mathsf{V}) > 0 \wedge \mathsf{H} = \varnothing \\ [lb(\mathsf{H}) + lb(\mathsf{V}), ub(\mathsf{H}) + lb(\mathsf{V})] & \text{otherwise} \end{cases} \tag{10}$$

where lb and ub return the lower and upper end-point of a given non-empty closed interval, respectively.

The predicate WCV? determines if there is a subinterval of $\mathbf{t}$ where a violation of well-clear occurs.

$$\text{WCV?}(\mathbf{t}, \mathbf{s}, \mathbf{v}) \iff \text{WCV}int(\mathbf{t}, \mathbf{s}, \mathbf{v}) \neq \varnothing \tag{11}$$

The equations in this Section are a simplified version of the definitions originally presented in [12] where properties and additional definitions can be found.

III. Verification approach

The verification approach used in this paper relies on the integrated toolchain presented in [7] which is composed of several formal methods tools:

- PRECiSA [8], [9], a static analyzer for floating-point programs,[2]
- the global optimizer Kodiak [13],[3]
- Frama-C [10], a collaborative tool suite for the analysis of C code, and
- the Prototype Verification System (PVS) [14], a verification environment consisting of a specification language, a large number of predefined theories, and an interactive theorem prover.

PRECiSA is a static analyzer for floating-point programs that computes sound and accurate round-off error estimations and provides support for a large variety of mathematical operators and programming language constructs. Given a floating-point program, PRECiSA generates a symbolic error expression modeling an over-approximation of the round-off error that may occur in the program. This error expression is a function of the input variables of the program and their associated rounding error. Given input ranges for these variables, PRECiSA uses the Kodiak global optimizer to maximize the round-off error expressions. Additionally, PRECiSA generates formal certificates ensuring that these bounds are correct with respect to the IEEE Standard for Floating-Point Arithmetic (IEEE 754). These certificates are output in the language of PVS, which can be used to mechanically check their validity. Even though proofs in PVS are expected to be carried out following user guidance in general, this process is automatic thanks to an available collection of proof strategies targeted to this particular application.

One of the more recent extensions of PRECiSA [7], includes the addition of a code-extraction capability that automatically generates a floating-point C implementation from a real-number function expressed in the language of PVS. The generated C code is instrumented to detect whether the floating-point computational flow diverges from its ideal real number counterpart, and it is automatically annotated with *program contracts* stating the formal relationship between real and floating-point computations. These contracts are written in the ANSI/ISO C Specification Language (ACSL) which can be processed by Frama-C. Frama-C is a collaborative modular platform for the analysis of C programs. In this work, the Frama-C weakest precondition (WP) plug-in is used to

[2]PRECiSA is available at https://github.com/nasa/PRECiSA.

[3]Kodiak is available at https://shemesh.larc.nasa.gov/fm/Kodiak/.

generate verification conditions in the language of PVS and it is customized to integrate the PVS certificates generated by PRECiSA into the proof of such verification conditions.

An overview of the verification approach applied to the well-clear calculations in DAIDALUS is depicted in Fig. 1. First, the PVS higher-order specification of DAIDALUS is manually rewritten using only first-order constructs. This is achieved by replacing each higher-order argument with a specific function instantiation. For instance, the original definition for the violation of vertical well-clear is parametric on the technique used to approximate the time of closest point of approach. Such parameter was replaced by a particular concrete calculation of this time, resulting in the definition shown in (5). While this kind of simplification cannot be performed in general by preserving the semantics of the program, the nature of the higher-order parameters used in DAIDALUS allowed to simplify the definitions even though this change resulted in a less general specification. The first-order specification is mechanically proved equivalent to the higher-order one within PVS. The higher-order specification is instantiated with the specific functions used in the fist-order one as parameters. This simplification was necessary since PRECiSA does not yet provide support for the use of higher-order arguments in the input program.

Then, a program slicing technique is applied to the first-order specification to obtain a set of simpler descriptions. This program slicing is proved to be semantically equivalent to the original specification. The next section provides more details on the slicing process and the resulting fragmentation of the specification.

Each specification slice is input to PRECiSA which automatically extracts the corresponding annotated floating-point C code and generates the corresponding PVS proof certificates ensuring the correctness of the round-off error estimations used in the code extraction and instrumentation. Since the extracted C code implements each of the slices of the original specification, it is necessary to develop a top-level module in C providing the same functionality as the involved functions in DAIDALUS. Basically, this top-level function selects the proper slice given an unrestricted input and calls the corresponding C function. The top-level function was manually developed and annotated with specific program contracts to ensure its compliance with the original specification. The details about this function are explained in Sect. V.

Frama-C was used to analyze both the automatically generated C functions from each slice and the top-level function. Finally, the verification conditions output by Frama-C were proved in the PVS theorem prover. While these proofs were made interactively for this particular application of the technique, they can be automated since they rely heavily on the structure of the program. The automation of the proofs is left as future work.

IV. SPECIFICATION SLICING

Program Slicing [15], [16] is a technique generally applied on source code to analyze particular behaviors of software.

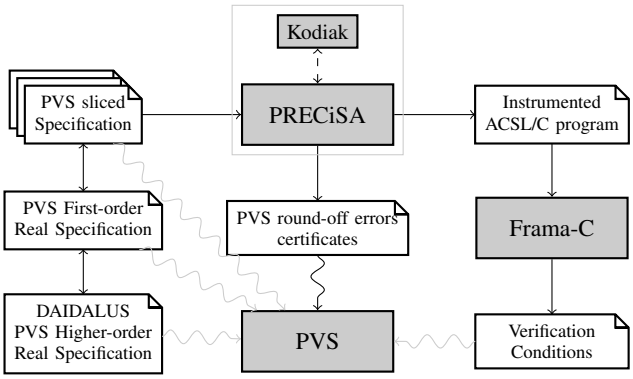

Fig. 1. Workflow of the verification approach.

TABLE I
NAME OF THE MAIN PREDICATE ON EACH SLICE.

horizontal separation	*vertical separation*	*decrease* $\mathbf{v}_z < 0$	*maintain* $\mathbf{v}_z = 0$	*increase* $\mathbf{v}_z > 0$
alter	$\mathbf{v}_x \neq 0 \vee \mathbf{v}_y \neq 0$	$\text{WCV}?^{\leftrightarrow}_{\downarrow}$	$\text{WCV}?^{\leftrightarrow}_{\cdot}$	$\text{WCV}?^{\leftrightarrow}_{\uparrow}$
maintain	$\mathbf{v}_x = 0 \wedge \mathbf{v}_y = 0$	$\text{WCV}?_{\downarrow}$	$\text{WCV}?_{\cdot}$	$\text{WCV}?_{\uparrow}$

However, in this work, it was applied to the specification of the definitions presented in Section II as a way to address scalability issues in the PRECiSA code extraction. The slicing approach used in this work was first introduced by Canfora et al. [17] and Ning et al. [18] and it is known as *Conditioned Slicing* [19]. Essentially, it proposes the decomposition of a program into independent simpler parts, called *slices*, according to its control flow graph as defined by the guards in the branching instructions appearing in the program. Each slice runs under the assumption of specific restrictions on the inputs, determining the execution of a particular path in the control flow graph of the original program.

For this case study, the criterion used to select the restriction on the inputs producing the slices was focused on the different cases determined by the possible relative velocities of the aircraft, as given in the branches of the source code. Three possible situations regarding relative vertical velocity were considered: maintaining separation (null relative vertical velocity), increasing separation (positive relative vertical velocity), and decreasing separation (negative relative vertical velocity). In terms of horizontal relative velocity, only the cases *altering separation* or *maintaining separation* were considered. Hence, a total of six slices were defined by applying this criterion on the predicate presented in (11) which is the top-most declaration in the Well-Clear module. Table I shows the name of the top-most predicate in each slice.

To exemplify how the slices are actually defined, Equation (12) shows the entry point for the slice describing a situation of maintaining vertical separation and altering horizontal separation, given by the conditions $\mathbf{v}_z = 0$ and $\mathbf{v}_x \neq 0 \vee \mathbf{v}_y \neq 0$.

$$\text{WCV}?^{\leftrightarrow}_{\cdot}(\mathbf{t},\mathbf{s},\mathbf{v}) \iff (|s_z| \leq \delta_z \wedge \text{WCV}_H?^{\leftrightarrow}_{\cdot}(\mathbf{t}_F - \mathbf{t}_0, \mathbf{s}_x + \mathbf{t}_0\mathbf{v}_x, \mathbf{s}_y + \mathbf{t}_0\mathbf{v}_y, \mathbf{v}_x, \mathbf{v}_y)) \quad (12)$$

where $\text{WCV}_H?^{\leftrightarrow}_{\cdot}$ is a predicate checking whether the $\text{WCV}int_{\mathsf{H}}$

function from (9) returns a non-empty interval when the restrictions imposed by the conditions defining the slice are assumed to hold on the inputs.

$$\text{WCV}_H?^{\leftrightarrow}_{\cdot}(t,\mathbf{s}_h,\mathbf{v}_h) \iff \begin{cases} r^+_{a,2\mathbf{s}_h\mathbf{v}_h,\mathbf{s}_h^2-\delta_D^2} \geq 0 & \text{if } \mathbf{s}_h^2 \leq \delta_D^2 \\ \left(0 \leq r^-_{a,b,c} \leq t \wedge r^-_{a,b,c} \leq r^+_{a,2\mathbf{s}_h\mathbf{v}_h,\mathbf{s}_h^2-\delta_D^2}\right) \vee \\ \left(r^-_{a,b,c} < 0 \wedge r^+_{a,2\mathbf{s}_h\mathbf{v}_h,\mathbf{s}_h^2-\delta_D^2} \geq 0\right) \\ \quad \text{if } \mathbf{s}_h^2 > \delta_D^2 \wedge \mathbf{s}_h\mathbf{v}_h < 0 \wedge \Delta_{a,b,c} \geq 0 \wedge \\ \quad \Delta^{\mathbb{R}\times\mathbb{R}}_{\mathbf{s}_h,\mathbf{v}_h} \geq 0 \wedge r^-_{a,b,c} \leq t \\ \textbf{false} & \text{otherwise} \end{cases} \tag{13}$$

Above, a, b, c, $\Delta_{a,b,c}$, $r^-_{a,b,c}$, and $\Delta^{\mathbb{R}\times\mathbb{R}}_{\mathbf{s}_h,\mathbf{v}_h}$ are as in (9), and $r^+_{a,b,c} \stackrel{\text{def}}{=} \frac{-b+\sqrt{\Delta_{a,b,c}}}{2a}$. The following theorem validates that the decomposition proposed by the slicing process correctly captures the semantics of the original specification.

Theorem 1: [Slicing Correctness] For all time interval $\mathbf{t} \subset \mathbb{R}$ and all pair of three-dimensional vectors $\mathbf{s},\mathbf{v} \in \mathbb{R}^3$, $\text{WCV}?(\mathbf{t},\mathbf{s},\mathbf{v})$ holds if and only if

$$\begin{cases} \text{WCV}?^{\leftrightarrow}_{\downarrow}(\mathbf{t},\mathbf{s},\mathbf{v}) & \text{when } \mathbf{v}_z < 0 \wedge (\mathbf{v}_x \neq 0 \vee \mathbf{v}_y \neq 0) \\ \text{WCV}?^{\downarrow}_{\downarrow}(\mathbf{t},\mathbf{s},\mathbf{v}) & \text{when } \mathbf{v}_z < 0 \wedge (\mathbf{v}_x = 0 \wedge \mathbf{v}_y = 0) \\ \text{WCV}?^{\leftrightarrow}_{\uparrow}(\mathbf{t},\mathbf{s},\mathbf{v}) & \text{when } \mathbf{v}_z > 0 \wedge (\mathbf{v}_x \neq 0 \vee \mathbf{v}_y \neq 0) \\ \text{WCV}?^{\downarrow}_{\uparrow}(\mathbf{t},\mathbf{s},\mathbf{v}) & \text{when } \mathbf{v}_z > 0 \wedge (\mathbf{v}_x = 0 \wedge \mathbf{v}_y = 0) \\ \text{WCV}?^{\leftrightarrow}_{\cdot}(\mathbf{t},\mathbf{s},\mathbf{v}) & \text{when } \mathbf{v}_z = 0 \wedge (\mathbf{v}_x \neq 0 \vee \mathbf{v}_y \neq 0) \\ \text{WCV}?^{\cdot}_{\cdot}(\mathbf{t},\mathbf{s},\mathbf{v}) & \text{when } \mathbf{v}_z = 0 \wedge (\mathbf{v}_x = 0 \wedge \mathbf{v}_y = 0) \end{cases}$$

As one of the contributions of the present work, the definition of the predicates in Table I and the theorem above, along with all the ad-hoc lemmas needed in its proof, were mechanically checked using the PVS theorem prover.

V. Code Extraction and Verification

The round-off error occurring in the guards of conditionals can provoke the floating-point control flow to diverge with respect to its ideal real-numbers counterpart. The guards in a program where such phenomenon can occur are called *unstable conditions*. As another of its features, the code extracted by PRECiSA is instrumented to emit a warning when such conditions may occur. This instrumentation is based on the program transformation presented in [20]. In the rest of this section, the code extraction procedure is outlined. As part of the verification presented in this paper, this procedure was applied to each of the slices of the specification described in the previous section.

A. Processing the slices

Given the specification of a real-valued program, understood as a collection of functions collaborating to compute a determined result, and the desired floating-point format (single or double precision), PRECiSA replaces each real arithmetic operator with its floating-point counterpart. Then, it modifies all the conditional statements. Each guard is replaced by a more restrictive one that takes into account the round-off error that may occur. This round-off error is computed with PRECiSA. Additionally, a warning is emitted when the original guard may be evaluated differently in real and floating-point arithmetic. This warning is denoted by a distinguished value disjoint from the floating-point domain. Getting such a warning as the result of a computation implies that, for the inputs provided, it cannot be guaranteed that the floating-point execution follows the same control flow than its real-valued counterpart. This divergence could provoke a much bigger error in the numerical final result than the accumulated round-off error occurred in the evaluation of an arithmetic expression. It is worth noting that, since the round-off error estimation computed by PRECiSA is a sound over-approximation of the error that may occur, false warnings may arise. However, it is guaranteed that all the instabilities are detected.

For instance, the floating-point function depicted below is the result of applying this instrumentation on the function τ_{mod}, defined by (2), whose goal is to approximate the time of closest point of approach of two aircraft. Here and in the rest of this paper, the tilde over a variable, an operator, or function denotes the fact that it belongs to the floating-point domain.

$$\begin{aligned} &\widetilde{\tau'_{mod}}(\widetilde{s_x},\widetilde{v_x},\widetilde{s_y},\widetilde{v_y},\widetilde{\epsilon}) \\ &= \mathit{if}\ \widetilde{s_x} \mathbin{\tilde{*}} \widetilde{v_x} \mathbin{\tilde{+}} \widetilde{s_y} \mathbin{\tilde{*}} \widetilde{v_y} < -\widetilde{\epsilon} \\ &\quad \mathit{then}\ (\delta_d \mathbin{\tilde{*}} \delta_d \mathbin{\tilde{-}} \widetilde{s_x} \mathbin{\tilde{*}} \widetilde{s_x} \mathbin{\tilde{+}} \widetilde{s_y} \mathbin{\tilde{*}} \widetilde{s_y}) \mathbin{\tilde{/}} (\widetilde{s_x} \mathbin{\tilde{*}} \widetilde{v_x} \mathbin{\tilde{+}} \widetilde{s_y} \mathbin{\tilde{*}} \widetilde{v_y}) \\ &\quad \mathit{elsif}\ \widetilde{s_x} \mathbin{\tilde{*}} \widetilde{v_x} \mathbin{\tilde{+}} \widetilde{s_y} \mathbin{\tilde{*}} \widetilde{v_y} \geq \widetilde{\epsilon}\ \mathit{then}\ -1 \\ &\quad \mathit{else}\ \omega \end{aligned} \tag{14}$$

When the evaluation of $\widetilde{s_x} \mathbin{\tilde{*}} \widetilde{v_x} \mathbin{\tilde{+}} \widetilde{s_y} \mathbin{\tilde{*}} \widetilde{v_y}$ lies in the interval $[-\epsilon,\epsilon)$ the function above signals a warning by returning the value ω. The new argument of the function, ϵ, is expected to be an over-approximation of the round-off error that may occur when computing $\widetilde{s_x} \mathbin{\tilde{*}} \widetilde{v_x} \mathbin{\tilde{+}} \widetilde{s_y} \mathbin{\tilde{*}} \widetilde{v_y}$.

Listing 1 shows the C code and the ACSL annotations generated by PRECiSA for the function τ_{mod}. The C function taumod_fp mimics the definition of $\widetilde{\tau'_{mod}}$, while the annotations express the contracts enforcing the properties explained above. The type **double′** is the implementation of a union type consisting of the **double** datatype and the ω warning value[4]. In ACSL, the keywords **requires** and **ensures** are used to describe preconditions and postconditions of a function, respectively. The main precondition of taumod_fp (line 9) restricts ϵ to be a non-negative and not infinite numeric value, i.e., it cannot be a *NaN* neither. The postcondition on line 10 ensures that when the result is not ω, it is the same as the one computed by the floating point version of τ_{mod} (before the instrumentation). The following postcondition (line 11) states that, if additionally to the result not being ω, the argument ϵ is a sound approximation of the round-off error of the guard, then no unstable conditions occur, meaning that the guard has the same value under both floating-point and real-valued arithmetic. This latter condition is expressed by the predicate $stable\_paths_{\tau_m}$ defined on lines 5-7.

[4]For ease of reading no explicit projection of the values in the union type are used.

```
1 /*@
2 double taumod_fp(double ŝ_x, v̂_x, ŝ_y, v̂_y) = \let ĝ = ŝ_x ⊛ v̂_x ⊞ ŝ_y ⊛ v̂_y;
3   ĝ < 0 ? (δ_d ⊛ δ_d ⊟ ŝ_x ⊛ ŝ_x ⊞ ŝ_y ⊛ ŝ_y) ⊘ ĝ : −1.0;
4
5 predicate stable_paths_τm(real v_x, v_y, s_x, s_y, double v̂_x, v̂_y, ŝ_x, ŝ_y) =
6   \let ĝ = ŝ_x ⊛ v̂_x ⊞ ŝ_y ⊛ v̂_y; \let g = s_x * v_x + s_y * v_y;
7   (g < 0 ∧ ĝ < 0) ∨ (g ≥ 0 ∧ ĝ ≥ 0);
8
9 requires: \is_finite?(ϵ) ∧ ϵ ≥ 0;
10 ensures: \result ≠ ω ⇒ \result = taumod_fp(ŝ_x, v̂_x, ŝ_y, v̂_y)
11 ensures: ∀ real v_x, v_y, s_x, s_y;
12           \result ≠ ω ∧ |(ŝ_x ⊛ v̂_x ⊞ ŝ_y ⊛ v̂_y) − (s_x * v_x + s_y * v_y)| ≤ ϵ
13            ⇒ stable_paths_τm(v_x, v_y, s_x, s_y, v̂_x, v̂_y, ŝ_x, ŝ_y);
14 */
15 double′ taumod_fp(double ŝ_x, v̂_x, ŝ_y, v̂_y, ϵ̂){
16   if (ŝ_x ⊛ v̂_x ⊞ ŝ_y ⊛ v̂_y < −ϵ̂)
17     return (δ_d ⊛ δ_d ⊟ ŝ_x ⊛ ŝ_x ⊞ ŝ_y ⊛ ŝ_y) ⊘ (ŝ_x ⊛ v̂_x ⊞ ŝ_y ⊛ v̂_y);
18   else if (ŝ_x ⊛ v̂_x ⊞ ŝ_y ⊛ v̂_y ≥ ϵ̂)
19     return −1.0;
20   else
21     return ω;
22 }
```

Listing 1. C function and annotations generated by PRECiSA for τ_{mod}. Some syntactic simplifications were applied to the code in this listing for ease of reading, e.g., the use of the infix version of some operators and avoiding the repetition of the type of the function parameters, among others.

As already mentioned, PRECiSA is able to compute concrete error bounds for the guards when the user provides specific ranges for the arguments. For instance, if the values of the input variables are assumed to lie in the range $[1, 2]$ and double precision floating-point precision is selected, PRECiSA computes the round-off error bound $\epsilon = 3.55 \times 10^{-15}$ for the expression $s_x * v_x + s_y * v_y$. Notably, PRECiSA also generates a formal certificate of the validity of this bound, materialized as a theorem that can be mechanically checked in the PVS theorem prover. For the τ_{mod} example, such a theorem can be expressed as it is shown below.

Theorem 2 (Error bound for the guard in τ_{mod}): For all real values v_x, v_y, s_x, s_y and floating-point numbers $\widehat{v_x}, \widehat{v_y}, \widehat{s_x}, \widehat{s_y}$, if $1 \leq v_x, v_y, s_x, s_y \leq 2$ and each float is the rounding of the respective real, then

$$|(\widehat{s_x} \tilde{*} \widehat{v_x} \tilde{+} \widehat{s_y} \tilde{*} \widehat{v_y}) - (s_x * v_x + s_y * v_y)| \leq 3.55 \times 10^{-15} \ .$$

This theorem can be used to prove that one of the hypothesis of the ensures clause in lines 11-13 of Listing 1 holds when velocities and positions are in the specified ranges and ϵ is instantiated with the value from the theorem. Then, under these assumptions, such *ensures* guarantees that float and real flows do not diverge. Furthermore, the accumulated round-off error in the final result of taumod_fp is the maximum between the accumulated round-off errors in the expressions of each branch of the if-then-else that does not return a warning (ω). Again, PRECiSA is used to calculate a bound for such an error for every one of these expressions under the same assumption on the input values. In the case of τ_{mod}, these bounds are 1.08×10^{-14} for the first branch and 0 for the second, since -1 is a value that can be exactly representable in floating points. This kind of deduction can be repeated for each collection of input ranges provided by the user. PRECiSA summarizes it in a new annotated C function. This kind of function is called *concrete* or *numerical* in the context of this work and it only consists of a call to the function in Listing 1 instantiated with the error estimation computed by PRECiSA; the latter function, for contraposition, is called *generic*.

```
1 /*@
2 real τ_mod(real s_x, v_x, s_y, v_y) = \let g = s_x * v_x + s_y * v_y;
3   g < 0 ? (δ_d * δ_d ⊟ s_x * s_x + s_y * s_y)/g : −1;
4
5 ensures: ∀ real v_x, v_y, s_x, s_y;
6     1 ≤ v_x ≤ 2 ∧ 1 ≤ v_y ≤ 2 ∧ 1 ≤ s_x ≤ 2 ∧ 1 ≤ s_y ≤ 2 ∧
7     |v̂_x − v_x| ≤ ulp(v_x)/2 ∧ |v̂_y − v_y| ≤ ulp(v_y)/2 ∧
8     |ŝ_x − s_x| ≤ ulp(s_x)/2 ∧ |ŝ_y − s_y| ≤ ulp(s_y)/2 ∧
9     \result ≠ ω
10    ⇒ |\result − τ_mod(s_x, v_x, s_y, v_y)| ≤ 1.08 × 10^−14;
11 */
12 double′ taumod_num(double ŝ_x, v̂_x, ŝ_y, v̂_y){
13   return taumod_fp(ŝ_x, v̂_x, ŝ_y, v̂_y, 0x1.0000000000001p − 48);
14 }
```

Listing 2. Concrete C function generated by PRECiSA for τ_{mod} assuming that all the input values lie in the interval $[1, 2]$.

Listing 2 shows the concrete function and its associated annotations for τ_{mod} under the assumptions on the inputs described above. The formula on line 6 enforces the restriction on the inputs. Lines 7-8 states the relation between the real and the corresponding floating-point values, as in the hypothesis in Theorem 2. The program contract finishes ensuring that under the mentioned conditions, the difference between the result of the C function and its real-valued specification is at most the estimation computed by PRECiSA ($0x1.0000000000001p-48$ is the hexadecimal representation of the value 3.55×10^{-15}).

While Listings 1 and 2 serve as a useful hint to picture the implementation and contracts of more complex functions returning numeric values, the application of the code extraction process to the predicates present in the sliced DAIDALUS specification, e.g., WCV?$^{\leftrightarrow}_{\downarrow}$, WCV?$^{\downarrow}_{\downarrow}$, etc., deserves a closer look. For each predicate in the input specification, PRECiSA generates two pairs of C functions. Each of these pairs, as in the case of the functions with numeric return values, consists of a generic and a concrete C function. One of the pairs describes the cases in which the original predicate returns an affirmative answer (true) while the other characterizes the inputs for which a negative answer (false) is obtained. For instance, Listing 3 shows a fragment of the program contracts for the C function WCVint_inc_mtn_plus (line 10, Listing 3), that is extracted from the predicate WCV?$^{\downarrow}_{\uparrow}$. The predicate *wcv_inc_mtn_plus*, whose actual definition is omitted because of space limitations, is such that every set of real values for which it holds, make WCV?$^{\downarrow}_{\uparrow}$ hold as well. Respectively, the predicate *wcv_inc_mtn_plus_fp* is such that if it holds for a set of floating-point inputs, the floating-point version of WCV?$^{\downarrow}_{\uparrow}$ holds as well for those inputs. On the other hand, WCV?$^{\downarrow}_{\uparrow}$ (respectively, its floating-point version) does not hold for the values for which *wcv_inc_mtn_mns* holds (resp., *wcv_inc_mtn_mns_fp*.) Finally, the return type of the C function (**bool′**) represents the implementation of the union type between the **bool** datatype and the ω value. Hence, in lines 6 and 17, if the **result** keyword does not denote a warning, it is used to represent the Boolean value resulting from the computation of the function.

```
/*@
predicate wcv_inc_mtn_plus(real b,t,v_x,v_y,v_z,s_x,s_y,s_z) = ···;
predicate wcv_inc_mtn_plus_fp(double b̃,t̃,s̃_x,s̃_y,s̃_z,ṽ_x,ṽ_y,ṽ_z) = ···;
···
ensures: ∀ real b,t,v_x,v_y,v_z,s_x,s_y,s_z;
    \result ≠ ω ∧ \result
    ⇒ (wcv_inc_mtn_plus(b,t,v_x,v_y,v_z,s_x,s_y,s_z) ∧
        wcv_inc_mtn_plus_fp(b̃,t̃,s̃_x,s̃_y,s̃_z,ṽ_x,ṽ_y,ṽ_z));
*/
bool′ WCVint_inc_mtn_plus (double b̃,t̃,s̃_x,s̃_y,s̃_z,ṽ_x,ṽ_y,ṽ_z){···}

/*@
predicate wcv_inc_mtn_mns(real b,t,v_x,v_y,v_z,s_x,s_y,s_z) = ···;
predicate wcv_inc_mtn_mns_fp(double b̃,t̃,s̃_x,s̃_y,s̃_z,ṽ_x,ṽ_y,ṽ_z) = ···;
···
ensures: ∀ real b,t,v_x,v_y,v_z,s_x,s_y,s_z;
    \result ≠ ω ∧ \result
    ⇒ (wcv_inc_mtn_mns(b,t,v_x,v_y,v_z,s_x,s_y,s_z) ∧
        wcv_inc_mtn_mns_fp(b̃,t̃,s̃_x,s̃_y,s̃_z,ṽ_x,ṽ_y,ṽ_z));
*/
bool′ WCVint_inc_mtn_minus (double b̃,t̃,s̃_x,s̃_y,s̃_z,ṽ_x,ṽ_y,ṽ_z){···}
```

Listing 3. Excerpt from the program contracts in the generic function generated by PRECiSA for the WCV?$_{\uparrow}$ predicate.

Once each slice of the specification was input to PRECiSA to obtain the corresponding annotated C code, Frama-C was used to verify that this implementation actually fulfills the contracts stated by the annotations. As explained in the paragraphs above, the validity of these contracts is mainly supported by the error-bound certificates generated by PRECiSA, which are output in PVS language and, in its turn, they depend on the definitions and properties declared in the axiomatic floating-point formalization from NASALib. For this reason, a particular customization was applied to Frama-C in order to generate the verification conditions in the language of PVS and using the aforementioned floating-point formalization.

B. The top-level function

The process described above allowed to generate code for each slice of the specification and verify its compliance to the corresponding predicate from Table I. Nevertheless, in order to generate code with the same applicability than the original target, i.e., the predicate WCV? from (11), an additional layer of C code is needed. This layer is responsible for selecting the slice activated by the inputs and invoking the corresponding function.

Listing 4 shows an excerpt from the generic top-level function. The postcondition states that if the computation does not raise a warning and the ϵ parameters actually denote bounds for the errors in the conditionals defining the control flow graph of the whole program, then the result is equivalent to the original Well-Clear predicate WCV? defined in (11). The proof of the verification condition generated from this contract relies on the contracts of the invoked functions, e.g., WCVint_inc_mtn_plus and WCVint_inc_mtn_minus in the excerpt, and the Slicing Correctness Theorem 1. As in the lower layers, accompanying concrete C functions were defined, where the error-bound parameters ϵ are instantiated with concrete values computed by PRECiSA, given user-provided ranges for the rest of the inputs.

```
/*@
predicate wcv_in_range(real b,t,v_x,v_y,v_z,s_x,s_y,s_z) =
  // WCV? ((b,t),(v_x,v_y,v_z),(s_x,s_y,s_z)) from Eq. (11)
···
requires: \is_finite(ε̃_0) ∧ ε̃_0 ≥ 0 ∧ ··· ∧ \is_finite(ε̃_3) ∧ ε̃_3 ≥ 0;
ensures: ∀ real b,t,v_x,v_y,v_z,s_x,s_y,s_z;
    |(δ̃_z ⊖̃ ṽ_z ⊛̃ δ̃_tcoa) − (δ_z − v_z ∗ δ_tcoa)| ≤ ε̃_0 ∧
    |(t̃ ⊖̃ coalt_t_inc_vz_fp(s̃_z,ṽ_z)) − (t − coalt_t_inc_vz(s_z,v_z))| ≤ ε̃_1 ∧
    |(coalt_b_inc_vz_fp(s̃_z,ṽ_z) ⊖̃ b̃) − (coalt_b_inc_vz(s_z,v_z) − b)| ≤ ε̃_2 ∧
    ···
    \result ≠ ω
    ⇒ (\result ⇔ wcv_in_range(b,t,v_x,v_y,v_z,s_x,s_y,s_z));
*/
bool′ WCV_interval(double b̃,t̃,s̃_x,s̃_y,s̃_z,ṽ_x,ṽ_y,ṽ_z,ε̃_0,ε̃_1,ε̃_2,ε̃_3,···){
 bool′ res;
 if (ṽ_z > 0.0) // increasing vertical separation
  if (ṽ_x == 0.0 && ṽ_y == 0.0){ // maintaining horizontal separation
   res = WCVint_inc_mtn_plus(b̃,t̃,s̃_x,s̃_y,s̃_z,ṽ_x,ṽ_y,ṽ_z,ε̃_0,ε̃_1,ε̃_2,ε̃_3);
   if (res == ω || res) return res;
   res = WCVint_inc_mtn_minus(b̃,t̃,s̃_x,s̃_y,s̃_z,ṽ_x,ṽ_y,ṽ_z,ε̃_0,ε̃_1,ε̃_2,ε̃_3);
   if (res == ω) return ω;
   if (res) return false;
   return ω;
  } else {
   ···
  }
 else {
  ···
 }
}
```

Listing 4. Excerpt from the generic top-level function.

One may wonder if the top-level function implementation is subject to conditional instability. However, it can be noticed that the guards used to select the program slice are sign checks on input values that comes from an external sensor or data. In these cases, the rounding error corresponds to the representation error on this value which does not affect its sign.

The top-level functions and the accompanying annotations were developed by hand for this case study. Nevertheless, once the criteria to be used to define the slicing is selected, the development of these functions and their annotations is almost mechanic, at least for applications like this one, where quite simple slicing conditions are used. The automation of this stage of the technique is one of the possible extensions to this work.

VI. Discussion

The goal of the work presented in this paper is the extraction and verification of a floating-point C implementation from a proven correct real-valued specification of an algorithmic solution for a safety- and mission-critical problem. When trying to apply the tool chain presented in [7] several practical issues were addressed and new improvements were proposed. This section provides a brief summary of the most significant of them.

The first impediment that prevented the existent toolchain to be applied as in the past was the occurrence of higher-order elements in the input specification. This issue was addressed by restating several of the declarations into a more concrete form, avoiding the use of higher-order parameters. The level of effort involved in the application of this simplification could be seen as non-trivial since there were changes in

many of the lines of the original specification. Nevertheless, in the majority of the cases, each change was simple and it could be applied in a mechanical way. For this concrete case study, the transformation to first order is a process that could have been performed automatically. In fact, part of the future lines of research is devoted to the development of an automatic procedure to simplify higher-order features from a PVS specification.

After simplifying the original declarations, the resulting first-order specification was fed to PRECiSA. Nevertheless, the process had to be aborted after reaching a time-out of three hours without response. This impact on the performance could possibly be explained by the number of different flows starting at the top-level function WCVint, presented in 10, which provokes the generation of huge error expressions. The manipulation of such expressions deemed the code-generation process to be impractical. The step that allowed pushing PRECiSA beyond its scalability limit was the application of a slicing-based simplification on the first-order specification. The automatic generation of code performed by PRECiSA took less than 15 minutes to finish for the whole collection of slices on the same machine. This improvement is related to the fact that in the DAIDALUS specification some checks on the velocities and positions are repeated along the same branch in the control flow tree. In its turn, this phenomenon is repeated in several different branches. The slicing of the specification lifted to the top-level several of these checks, reducing the complexity of each individual slice. While the selection of the slicing criteria would depend on human insight in the general case, once it is decided, the automation of most of the tasks related with the process and integration of the slices into the final analysis is expected to be feasible, at least in examples with a complexity similar to the one presented in this paper.

Another distinguishing feature of this work is the use of a new formalization for floating-point numbers[5]. This formalization is different from the one used in previous works in several aspects. Mainly, it is defined in an axiomatic way, which has a significant impact in the type checking time of PVS, improving it by a factor of six. Since the verification conditions output by Frama-C are expressed in terms of floating-point and real-valued operations, the PVS libraries where these arithmetic domains are defined need to be type checked. The reduction in the time spent in type checking improved not only the flow of the work while the proofs for the verification conditions were developed, but it also decreased the time needed to rerun such proofs once they were done.

Additionally, this new formalization follows the IEEE-754 standard more closely, including representations for special values such as Not-a-Numbers (NaN) and infinities. While the use of a more detailed model usually complicates its interaction with the rest of the specification, in this work it was possible to reduce such impact to a minimum. For instance, the only place where a restriction about finiteness of the floating-point representations is explicitly used is for predefined constants and error-bound parameters, as can be seen in the *requires* of all the listings above.

On the bright side, the gain of using this more detailed formalization is at the semantic stance. It is not uncommon to simplify some aspects of the models when a formalization work is performed. For the case of floating-point numbers, usually some aspects of the IEEE-754 standard, such as the special values, are left aside because they complicate the formalization by introducing the need to handle a nontrivial number of special cases. Nevertheless, since the special values are supported by the C language, working with a conceptual model that does not support them could introduce space for flaws undetectable by the analysis.

It is important to note that the almost seamless integration with this new formalization was possible because the check for finiteness was encapsulated in the error-bound certificates generated by PRECiSA. As part of the automatic proof for certificates as the one expressed by Theorem 2, the numeric expressions (including subexpressions) appearing in them are checked to remain in the floating-point representable domain, therefore no infinite values or overflows occur. This check is done by using a branch-and-bound optimization algorithm implemented in the logic of PVS itself [21]. Notably, this process provides hints on *overflow detection* since if the solver cannot decide whether the numeric expressions remain in the representable range for the inputs provided by the user, the proof of the certificate cannot be completed. In other words, if PVS cannot automatically prove the error certificate using the PRECiSA proof strategies, the user is directed to look for a possible overflow condition in their program.

VII. Related work

Different tools have been proposed to reason about the numerical aspects of C programs. In this work, a combination of PRECiSA, PVS, and Frama-C [10] is used. Support for floating-point round-off error analysis in Frama-C is also provided by the integration with the tool Gappa [22]. However, the applicability of Gappa is limited to straight-line programs without conditionals, and it often requires providing additional ACSL intermediate assertions and hints through annotation that may be unfeasible to generate automatically. The interactive theorem prover Coq can also be applied to prove verification conditions on floating-point numbers thanks to the formalization defined in [23]. Nevertheless, Coq [24] tactics are not available 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 [25]–[30]. In contrast to the present work, the techniques above are not fully automatic and require user intervention in both the specification and verification processes.

In [31], 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 [7], the verification approach is presented and applied to a small fragment of

[5]Available at https://github.com/nasa/pvslib/tree/master/float/axm_bnd.

DAIDALUS. Note, in both [31] and [7] overflow detection is not performed.

Besides Frama-C, other formal methods tools are available to analyze the numerical properties of C code. Fluctuat [32] is a 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 [33], but does not instrument the program to emit a warning in these cases. The static analyzer Astrée [34] detects the presence of run-time exceptions such as division by zero and under and over-flows employing sound floating-point abstract domains. In contrast to the approach presented here, neither Fluctuat nor Astrée emits proof certificates that an external prover can externally check.

VIII. Conclusion and future work

In this paper, a formal approach is applied to generate and verify a floating-point implementation from the DAIDALUS well-clear specification. This implementation is obtained by manually simplifying and slicing the original specification and then utilizing each slice as input to the PRECiSA code generator. PRECiSA automatically generates a floating-point version of each slice in C syntax enriched with ACSL contracts stating the relationship between the ideal real number specification and the floating-point implementation. In addition, PRECiSA instruments the code to detect control flow divergences due to rounding errors. The generated C implementation of each slice is analyzed within the Frama-C analyzer. In particular, the WP plugin is used to compute a set of verification conditions that are proved within the PVS theorem prover. These verification conditions ensure that the accumulated rounding error is bounded, all flow divergences are detected, and no overflow occurs.

The verification of the DAIDALUS well-clear C implementation 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 error bounds are automatically generated. However, a certain level of expertise is needed for proving the PVS verification conditions generated by Frama-C and for proving the equivalence between the original DAIDALUS specification and the simplified and sliced one.

In the future, the authors plan to improve the automation degree of the slicing and top-layer function generation. Static analysis techniques may be used since the slices are built according to the program branches. The authors also plan to simplify the structure of the ACSL contracts generated by PRECiSA to facilitate human inspection and to produce simpler verification conditions. Automatic strategies are already available in PRECiSA to discharge the PVS certificates ensuring the correctness of the rounding error bounds and to prove certain verification conditions generated by the WP analysis. However, additional work needs to be done to fully automatize this process because of the new extended floating-point formalization used in this paper.

Another line of future research is motivated by the evaluation of the impact of the process of generation of a safer program on its final performance with respect to existing implementations. For instance, the reference implementation for DAIDALUS is expected to outperform the code generated by PRECiSA since some overhead is introduced by checking the stability of every guard. For instance, in such a checking for a single condition, the evaluation of two conditions is needed in the worst case. Nevertheless, in aerospace software such as the DAIDALUS library in which no iterative statements are allowed, this kind of overhead could result to be negligible or at least acceptable in real-world deployments, weighting the higher level of safety provided by the code generated by PRECiSA.

References

[1] Advisory Circular, U.S. Dept. of Transportation, Federal Aviation Administration, *AC 90-48D - Pilots' Role in Collision Avoidance*. U.S. Government, 2016.

[2] U.S. Government, *Aeronautics and Space*. 14 CFR § 91.113, 2004.

[3] C. Muñoz, A. Narkawicz, G. Hagen, J. Upchurch, A. Dutle, and M. Consiglio, "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.

[4] RTCA DO-365A, *Minimum Operational Performance Standards (MOPS) for Detect and Avoid (DAA) Systems, Appendix H*. RTCA, February 2020.

[5] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A Prototype Verification System," in *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction*, ser. LNCS, vol. 607. Springer, 1992, pp. 748–752. [Online]. Available: https://doi.org/10.1007/3-540-55602-8_217

[6] A. Narkawicz, C. Muñoz, and A. Dutle, "The MINERVA software development process," in *Automated Formal Methods*, ser. Kalpa Publications in Computing, vol. 5. EasyChair, 2018, pp. 93–108. [Online]. Available: https://easychair.org/publications/paper/g1Rs

[7] L. Titolo, M. Moscato, M. Feliú, and C. Muñoz, "Automatic generation of guard-stable floating-point code," in *Proceedings of the 16th International Conference on Integrated Formal Methods (IFM 2020)*, ser. LNCS, vol. 12546. Springer, 2020, pp. 141–159.

[8] M. Moscato, L. Titolo, A. Dutle, and C. Muñoz, "Automatic estimation of verified floating-point round-off errors via static analysis," in *Proceedings of the 36th International Conference on Computer Safety, Reliability, and Security, SAFECOMP 2017*. Springer, 2017.

[9] L. Titolo, M. Feliú, M. Moscato, and C. Muñoz, "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)*. Springer, 2018, pp. 516–537.

[10] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C: A software analysis perspective," *Form. Asp. of Comput.*, vol. 27, no. 3, pp. 573–609, 2015.

[11] C. Muñoz and A. Narkawicz, "Formal analysis of extended well-clear boundaries for unmanned aircraft," in *Proceedings of the 8th NASA FM Symp. (NFM 2016)*, ser. LNCS, vol. 9690. Minneapolis, MN: Springer, June 2016, pp. 221–226.

[12] C. Muñoz, A. Narkawicz, J. Chamberlain, M. Consiglio, and J. Upchurch, "A family of well-clear boundary models for the integration of UAS in the NAS," in *Proceedings of the 14th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference*, Georgia, Atlanta, USA, June 2014.

[13] A. P. Smith, C. Muñoz, A. J. Narkawicz, and M. Markevicius, "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*, 2015, pp. 71–78.

[14] S. Owre, J. Rushby, and N. Shankar, "PVS: A prototype verification system," in *Proceedings of the 11th International Conference on Automated Deduction (CADE)*. Springer, 1992, pp. 748–752.

[15] M. D. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981*. IEEE Computer Society, 1981, pp. 439–449.

[16] ——, "Program slicing," *IEEE Trans. Software Eng.*, vol. 10, no. 4, pp. 352–357, 1984.

[17] G. Canfora, A. Cimitile, A. D. Lucia, and G. A. D. Lucca, "Software salvaging based on conditions," in *Proceedings of the International Conference on Software Maintenance, ICSM 1994, Victoria, BC, Canada, September 1994*, H. A. Müller and M. Georges, Eds. IEEE Computer Society, 1994, pp. 424–433.

[18] J. Q. Ning, A. Engberts, and W. Kozaczynski, "Automated support for legacy code understanding," *Commun. ACM*, vol. 37, no. 5, pp. 50–57, 1994.

[19] J. Silva, "A vocabulary of program slicing-based techniques," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 12:1–12:41, 2012.

[20] L. Titolo, C. Muñoz, M. Feliú, and M. Moscato, "Eliminating unstable tests in floating-point programs," in *Proceedings of the 28th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2018)*. Springer, 2018, pp. 169–183.

[21] A. Narkawicz and C. Muñoz, "A formally verified generic branching algorithm for global optimization," in *Proceedings of the 5th International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2013)*, ser. Lecture Notes in Computer Science, E. Cohen and A. Rybalchenko, Eds., vol. 8164. Menlo Park, CA, US: Springer, May 2014, pp. 326–343.

[22] F. de Dinechin, C. Lauter, and G. Melquiond, "Certifying the floating-point implementation of an elementary function using Gappa," *IEEE Trans. on Computers*, vol. 60, no. 2, pp. 242–253, 2011.

[23] S. Boldo and G. Melquiond, "Flocq: A unified library for proving floating-point algorithms in Coq," in *20th IEEE Symposium on Computer Arithmetic, ARITH 2011*. IEEE Computer Society, 2011, pp. 243–252.

[24] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[25] S. Boldo and J. C. Filliâtre, "Formal verification of floating-point programs," in *Proceedings of ARITH18 2007*. IEEE Computer Society, 2007, pp. 187–194.

[26] S. Boldo and C. Marché, "Formal verification of numerical programs: From C annotated programs to mechanical proofs," *Mathematics in Computer Science*, vol. 5, no. 4, pp. 377–393, 2011.

[27] S. Boldo, F. Clément, J. C. Filliâtre, M. Mayero, G. Melquiond, and P. Weis, "Wave equation numerical resolution: A comprehensive mechanized proof of a C program," *Journal of Automated Reasoning*, vol. 50, no. 4, pp. 423–456, 2013.

[28] A. Goodloe, C. Muñoz, F. Kirchner, and L. Correnson, "Verification of numerical programs: From real numbers to floating point numbers," in *Proceedings of the NASA FM Symp. NFM 2013*, ser. LNCS, vol. 7871. Springer, 2013, pp. 441–446.

[29] C. Marché, "Verification of the functional behavior of a floating-point program: An industrial case study," *Science of Computer Programming*, vol. 96, pp. 279–296, 2014.

[30] L. Titolo, M. Moscato, C. Muñoz, A. Dutle, and F. Bobot, "A formally verified floating-point implementation of the compact position reporting algorithm," in *Proceedings of the 22nd International Symposium on Formal Methods (FM 2018)*, ser. LNCS, vol. 10951. Springer, 2018, pp. 364–381.

[31] M. Moscato, L. Titolo, M. Feliú, and C. Muñoz, "Provably correct floating-point implementation of a point-in-polygon algorithm," in *Proceedings of the 23nd International Symposium on Formal Methods (FM 2019)*, ser. LNCS, vol. 11800. Springer, 2019, pp. 21–37.

[32] E. Goubault and S. Putot, "Static analysis of numerical algorithms," in *Proceedings of SAS 2006*, ser. LNCS, vol. 4134. Springer, 2006, pp. 18–34.

[33] ——, "Robustness analysis of finite precision implementations," in *Proceedings of APLAS 2013*, ser. LNCS, vol. 8301. Springer, 2013, pp. 50–57.

[34] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and Rival, "The ASTREÉ Analyzer," in *Proceedings of the 14th European Symposium on Programming (ESOP 2005)*, ser. LNCS, vol. 3444. Springer, 2005, pp. 21–30.