Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Improving the Safety of Numerical Programs

Nikson Bernardes Fernandes Ferreira

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Supervisor
Prof. Dr. Mauricio Ayala-Rincón

Co-Supervisor
Dr. Mariano Miguel Moscato

Brasília
2023

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Improving the Safety of Numerical Programs

Nikson Bernardes Fernandes Ferreira

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof. Dr. Mauricio Ayala-Rincón (Supervisor)
PPG Informática/UnB

Dr. Laura Titolo
AMA/NASA LaRC FM

Prof. Dr. Vander Ramos Alves
PPG Informática/UnB

Dr. Aaron Dutle
NASA LaRC Formal Methods

Prof. Dr. Ricardo Pezzuol Jacobi
Coordenador do Programa de Pós-graduação em Informática

Brasília, July 21, 2023

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

Este trabalho discute como a precisão dos erros de arredondamento envolvidos em implementações reais do sistema de gerenciamento da NASA para veículos não tripulados DAIDALUS afetam a segurança geral do sistema. A biblioteca DAIDALUS fornece definições formais para os conceitos de Detecção e Evasão em aviônica demonstrados mecanicamente no assistente de provas PVS. No entanto, tais verificações são apenas certificados do bom comportamento da especificação do ponto de vista lógico, o que não garante a precisão dos algoritmos implementados sob restrições aritméticas de ponto flutuante. Nossa análise assume o padrão IEEE 754 de ponto flutuante, implementados em diversas linguagens de programação, e a técnica de verificação se baseia na geração de uma especificação de primeira ordem dos cálculos numéricos. Uma característica proeminente da abordagem é dividir a especificação em fatias definidas de acordo com os diferentes ramos de computação. O fatiamento é crucial para simplificar a análise formal das computações com aritmética de ponto flutuante.

Palavras-chave: Análise de Programas, Aritmética de Ponto Flutuante, Detecção e Evasão, Métodos Formais, PVS

Abstract

This work discusses how the presence of round errors involved in real-world implementations of DAIDALUS, a NASA library for uncrewed vehicles, affects the system's overall safety. The DAIDALUS library provides formal definitions for avionics' Detect and Avoid concepts mechanically proven correct in the proof assistant PVS. However, such verification assures only the well-behavior of the specification assuming real-numbers semantics, which does not guarantee the correctness of the algorithms when implemented using floating-point arithmetic. The applied technique assumes the IEEE-754 floating-point standard, the most widely spread approach to developing numerical-computation software in real-life applications. This work describes in detail the applied technique and the modifications needed for this particular analysis. Notably, these modifications included the translation into a first-order specification of the numerical computations and the splitting of the specification into slices defined according to the different computation branches. Slicing was crucial to simplify the formal analysis of floating point arithmetic computations.

Keywords: Program Analysis, Floating point arithmetic, Detect and Avoid, Formal Methods, PVS

Contents

Chapter 1

Introduction

1.1 Motivation

The so-called *Midair Conflicts* are among the most dangerous situations in the aerospace domain. As reported by the USA Federal Aviation Administration, more than forty midair collisions have occurred during the five years from January 2009 through December 2013 [1].

With the objective of mitigating such situations, the mentioned agency stated in Title 14 of the Code of Federal Regulations (14 CFR) part 91, the concept of *See and Avoid*. In short, it poses the responsibility to remain vigilant to see and avoid nearby traffic on the persons operating the aircraft [15]. The advent of Unmanned Aerial Systems (UAS) and their incorporation into the airspace provoked 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. Noticeably, DAA poses more collision-avoidance responsibilities on the system, or even all of them in the case of autonomous ships.

Diverse industrial and governmental actors proposed algorithmic detect and avoid solutions. Among them, the Detect and AvoID Alerting Logic for Unmanned Systems library (DAIDALUS), a suite of detect and avoid-related algorithms developed by NASA, stands out [30]. DAIDALUS provides prototypical open-source implementations in Java and C++, which were included as the reference implementations of the functional requirements presented in DO-365, the Minimum Operational Performance Standards for Unmanned Aircraft Systems developed by RTCA Special Committee 228 [37]. One distinguishing characteristic of DAIDALUS is that it also provides formal specifications of the algorithms along with proofs of correctness and safety properties on them, mechanically checked within the Prototype Verification System (PVS) [34]. Nevertheless, the adherence of the implementations to the behavior modeled by the formal specifications

was checked using a testing-based approach [31]. While this kind of approach is usually enough for non-critical applications, the correctness of DAA implementations requires a higher level of assurance. Especially given the nature of the algorithms in DAIDALUS, since most of them perform numerical computations specified using real-numbers arithmetic but implemented using floating-point representations.

Since its last version, the PRECiSA tool [45], an analyzer for floating-point programs based on abstract interpretation, provides a feature that automatically extracts floating-point C code from a real-valued PVS specification. Remarkably, the extracted C code contains annotations enabling the use of Frama-C [23], a static program analyzer, to externally verify properties on the C code, such as its compliance with the original specification. The Frama-C analyzer generates verification conditions that are seamlessly processed by diverse backends. The workflow proposed in [45] includes customization of Frama-C that allowed it to generate the verification conditions in PVS and connect them with the NASA Library of PVS formalizations (NASALib)[1].

The case study presented in [45] focuses on applying this new feature of PRECiSA to a core function in DAIDALUS. This work aims to continue with the application of the code-extraction feature that automatically generates a formally verified instrumented stable-guard C-code from a real specification to cover one of the main modules in DAIDALUS, which is devoted to the definition of the condition of *Well Clear* of an aircraft with respect to surrounding traffic. The Well Clear condition guarantees a collision-free situation. While PRECiSA was well suited to analyze the mentioned function, the high number of function calls and predicates in the rest of the declarations in the targeted module prevented the tool from being able to process them directly, just timing out the execution. This report proposes diverse modifications to the technique and the tool, enabling the analysis of the whole Well Clear module. Mainly, the slicing of the original specification, which incremented the number of invocations to PRECiSA but simplified the input in each call. This made the process manageable by the tool and simplified the annotations on the code, producing more straightforward program analysis obligations. It also uses a different PVS floating-point formalization which significantly improved the performance of the analysis of the C code.

1.2 Organization

Chapter 2 presents the required background on floating-point IEEE standard representation. Chapter 3 introduces the general lines of the proposed methodology and describes the main elements in the toolchain, namely, PRECiSA, Frama-C, and PVS. Chapter 4

[1] `https://github.com/nasa/pvslib`

describes the DAIDALUS library, used as the case study in this report. It also explains how some of its properties were verified assuming real-numbers arithmetic. Chapter 5 presents the slicing technique applied to simplify the overall analysis, the equivalence theorem between the original specification and the sliced declarations, the code extraction process used on each slice, and the addition of a top-layer function to complete the verification of the actual floating-point implementation of DAIDALUS. Finally, Chapter 6 discusses related work, and Chapter 7 concludes and briefly presents some possible lines of future work.

Chapter 2

Background

2.1 Floating-point representation

2.1.1 IEEE-754

The objective of following the IEEE-754 standard for floating-point arithmetic is to make the result of any operation independent of the application of any hardware/software. Floating-point arithmetic intends to approximate real arithmetic using finite resources, i.e., (finite) hardware should be able to represent and operate over it. Only a finite subset of the extended reals domain is representable using the floating-point representation. This subset is defined by the parameters of its format, i.e., the base (2 or 10), precision p (significant length), and minimum/maximum exponent (e_{min}/e_{max}). Each floating-point data is either $\pm\infty$, NaN (Not a number) or a triple:

$$\langle sign, exponent, significant \rangle \tag{2.1}$$

where *sign* is an integer (0 or 1) that indicates the signal of the number, *exponent* an integer in the interval $[e_{min}, e_{max}]$ expressing a scale factor, and *significant* is the mantissa, a string of integer digits in the interval $[0, base)$ in the form $d_0.d_1d_2...d_{p-1}$. The number of digits, p, is the significant length. The triple represents the number:

$$(-1)^{sign} \times significant \times base^{exponent} \tag{2.2}$$

The floating-point data is stored in three fields $\langle S, E, T \rangle$ in the radix-2 format in Figure 2.1.

Figure 2.1: Floating-point binary format

In this format, S, the *sign*, is stored as a one-bit; E, stored in w-bits, is the positive biased integer *exponent* E, $E = exponent + bias$; T stored in t-bits characterize a positive integer representing the *significant*. It is interpreted as:

1. *NaN*, if $E = 2^w - 1$ and $T \neq 0$;
2. $(-1)^{sign}\ \infty$, if $E = 2^w - 1$ and $T = 0$;
3. $(-1)^{sign} \times significant \times base^{exponent}$, where $significant = (1 + T \times base^{1-p})$ and $exponent = E - bias$, if $1 \leq E \leq 2^w - 2$, called *normal* form;
4. $(-1)^{sign} \times significant \times base^{e_{min}}$, where $significant = (0 + T \times base^{1-p})$, if $E = 0$ and $T \neq 0$, called *subnormal* form;
5. $(-1)^{sign}\ 0$, $E = 0$ and $T = 0$

The IEEE-754 standard presents three basic bit-encoding lengths: 32 (single precision), 64 (double precision), and 128 (long double precision) and extensions [22]. Table 2.1 shows concrete parameters for each binary floating-point format specified. $base^{e_{min}}$ is the minimum positive representable *normal* number and $base^{e_{max}} \times (base - base^{1-p})$ is the largest. Every number above the maximum positive number is represented as infinity since the number extrapolates the representable range. It is called *overflow*. The same occurs for the negative bound. Numbers closer to zero than the minimum positive number are represented as zero. It is called *underflow* since the number goes out of the lower representable bound.

parameter	32-bit (float)	64-bit (double)	128-bit (long double)
p	24	53	113
e_{max}	127	1023	16383
minimum positive number	1.175494e-38	2.225074e-308	3.362103e-4932
maximum positive number	3.402823e+38	1.797693e+308	1.189731e+4932

Table 2.1: Concrete floating-point bounds for each format with $base = 2$.

Since the floats representation is given in finite exponential terms, the exactly representable numbers are not uniformly distributed in the representable range as shown in 2.2. The closer to zero, the more numbers are representable; Closer to the representable limit, the fewer numbers are representable.

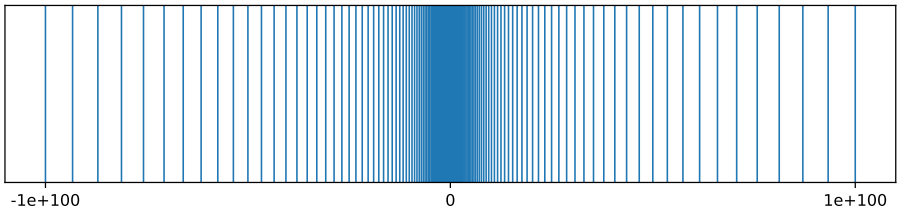


Figure 2.2: Representable floating-point numbers distribution.

Rounding is an operation that converts a real number to fit a finite representation if it is not exactly representable, signaling when an exception occurs during this process, such as *overflow*. The standard defined four possible modes:

- *round-to-nearest* - Convert to nearest representable value. If it has the same distance to two representations, then it can:
  - *ties-to-even* - Converto to the even one, i.e., the one in which the least significant bit is zero.
  - *ties-to-away* - Convert to largest one.
- *round-towards-positive* - Convert the real number to the smallest representation greater than it;
- *round-towards-negative* - Convert the real number to the largest positive representation less than it;
- *round-towards-zero* - Convert the real number with *round-towards-minus-infinity* if it is positive and with *round-towards-plus-infinity* if it is negative.

For instance, the real number `0.10`, which is not exactly representable in the floating-point domain, is rounded to `0.0999999940` in the single precision floats format using *round-towards-negative* or *round-towards-zero* mode; While using *round-towards-positive* or *round-to-nearest* it is represented as `0.1000000015`.

The difference between the ideal number with infinite precision and the finite precision representation is called round-off error. In the previous example, the round-off errors are 5.96e-09 and -1.19e-09, respectively. The default implementation defined by the standard is *round-to-nearest* and *ties-to-even*. It prevents statistical bias and provides numerical stability, i.e., the round-off error is bounded even with multiple sequential roundings operations in arithmetic expressions. Since an infinite precision number is rounded to the nearest representation, it introduces an error of, at most, half the gap between the two consecutive representations around the value. As depicted in Figure 2.2, the representable values are not uniformly distributed, then this gap depends on the real value intended to be represented and the representation precision.

The concept of *Unit in the Last Place* (*ulp*) expresses the minimum difference representable in a floating-point scale, i.e., the *significant*'s last digit maximum contribution to represent the value. A finite real number r is represented in the floating-point domain as:

$$\begin{aligned} round(r) &= (-1)^{sign} \times significant \times base^{exponent} \\ &= (-1)^{sign} \times (1 + T \times base^{1-p}) \times base^{exponent} \end{aligned} \tag{2.3}$$

since T is a positive integer number represented in binary format, the maximum contribution of the least significative binary digit of T in *significant* is $1 \times base^{1-p}$, so:

$$ulp(r) = base^{exponent(r)-p+1} \tag{2.4}$$

For instance, the *ulp* of 0.1 in single precision is 7.450581e-09 since its exponent is -4 and precision p is 24, as shown in Table 2.1, while the *ulp* of 1e7 is 1 in the same format. Note that the round-off error using the *round-to-nearest* mode is less than or equal to half *ulp*. Still, this error could appear irrelevant at first glance; In a hypothetical situation where banks use the single-precision format, it could allow someone to steal up to 50 cents from each bank account with more than 10 million dollars without altering its balance, i.e., without leaving a clear trace of the crime behind. Since this process does not affect the balance, it could be repeated endlessly if not discovered.

The IEEE-754 standard also defines how floating-point operations should be performed. Infinite is interpreted as an arbitrarily large number greater than all others. Regardless of invalid operations, such as $\infty * 0$ and $\infty - \infty$, or operations with special operands, such as addition with infinities or *NaN*, every basic operation should be performed as if calculated with infinite precision and then rounded to a floating-point value. For example, a floating-point sum is performed as:

$$\texttt{eval}_{\mathbb{F}}(f_1 + f_2) = round(\texttt{eval}_{\mathbb{R}}(\text{Float2Real}(f_1) + \text{Float2Real}(f_2))) \tag{2.5}$$

where `eval` means evaluating an expression using the subscript arithmetic $\mathbb{R}$ to reals and $\mathbb{F}$ to floating-point. Float2Real is the conversion from the floating-point domain to reals. For instance, the sum $0.10 + 0.10$ is evaluated over floating-point as:

$$round(\texttt{eval}_{\mathbb{R}}(0.1000000015 + 0.1000000015)) = 0.2000000030 \tag{2.6}$$

Unstable guards, also known as unstable tests [42], are guards in a conditional sentence that can evaluate different results when using real-valued or floating-point arithmetic lead-

ing the program to take a different control flow. [45]. A stable guard is a complementary definition, i.e., in both representations, the guard has convergent evaluations, then, the execution flow is the same. For instance, the branch expression presented in Specification 2.1 returns 100 if the expression is less than zero. Otherwise, returns 1.

```
100 if floor ((4/3 - 1) * 3 - 1) < 0 else 1
```

Specification 2.1: Unstable guard example

The expression's result is 1 when evaluated using real arithmetic, but using floating-point is 100, since $(4/3 - 1) * 3 - 1)$ evaluation is `-2.22e-16` instead of 0. As seen above, unstable guards can lead to absolutely different behavior under ideal real and floating-point computations. In the Specification 2.1, the round-off error is 99 due to the guard instability.

More formally, the stable-guard behavior could be expressed by the logical formula:

$$\texttt{eval}_{\mathbb{R}}(\texttt{guard}(x_1, ..., x_n)) \equiv \texttt{eval}_{\mathbb{F}}(\texttt{guard}(x_1, ..., x_n)) \tag{2.7}$$

where `eval` means evaluating a boolean expression using the subscript arithmetic $\mathbb{R}$ to reals and $\mathbb{F}$ to floating-point. $\texttt{guard}(x_1, ..., x_n)$ is a guard expression with n attributes. The Formula 2.7 can also be expressed as:

$$\begin{aligned}(\texttt{eval}_{\mathbb{R}}(\texttt{guard}(x_1, ..., x_n)) \wedge \texttt{eval}_{\mathbb{F}}(\texttt{guard}(x_1, ..., x_n))) \vee \\ (\neg\texttt{eval}_{\mathbb{R}}(\texttt{guard}(x_1, ..., x_n)) \wedge \neg\texttt{eval}_{\mathbb{F}}(\texttt{guard}(x_1, ..., x_n)))\end{aligned} \tag{2.8}$$

The expression $((4/3-1)*3-1)$ in previous example can be rewritten equivalently as $((4*3)/3-3)-1)$ using distributivity. However, when evaluated using floating-point arithmetic, these expressions are not equivalent, resulting in `-2.22e-16` and `0.0`, respectively. Indeed, real field properties such as associativity and commutativity are not preserved in floating-point arithmetic.

In general, designing correct and safe floating-point programs could be challenging. Moreover, it is crucial in safety-critical applications, where a minimum error could lead to catastrophic consequences. For instance, the Patriot missile failure (1991) failed to track and intercept a missile, which led to 28 deaths [17]. Another example of round-off error failure was reported in 2007 by Air-services Australia, where the ADS-B (Automatic Dependent Surveillance-Broadcast) system, currently used by thousands of aircraft, reported the position of an aircraft 220 nautical miles away from the actual position [40]. In [14] was demonstrated that CPR (Compact Position Reporting) algorithm responsible for encoding and decoding aircraft positions' original requirements does not guarantee the claimed precision.

2.1.2 Floating Point notation

For simplicity, in this document, boxed operators and expressions will be used to discriminate floating-point-evaluated expressions. For instance, the guard $(4/3 - 1) * 3 - 1$ in the if-then-else expression used in the previous subsection's example means that all operations are performed in f floating-point representation.

$$\boxed{\lfloor (4/3 - 1) * 3 - 1 \rfloor}$$

To discriminate operator variables and constants represented in floating point notation, they are boxed separately, as in the example below.

$$\boxed{\mathit{floor}}(\boxed{4} \boxtimes \boxed{3} \boxminus \boxed{1} \boxtimes \boxed{3} \boxminus \boxed{1})$$

So, since both expressions are considered to be performed in floating point arithmetic, they are equal, but

$$\boxed{\lfloor (4/3 - 1) * 3 - 1 \rfloor} \neq \lfloor (4/3 - 1) * 3 - 1 \rfloor$$

since the right-hand side of the inequality above is computed in real arithmetic.

Abusing in notation, a real number operation with floating-point operands includes a hidden transformation from floats to real. For instance, consider the real multiplication between floats $\boxed{v_1}$ and $\boxed{v_2}$:

$$\boxed{v_1} * \boxed{v_2} = \text{Float2Real}\left(\boxed{v_1}\right) * \text{Float2Real}\left(\boxed{v_2}\right)$$

In addition, an operation $\mathtt{op}$ with arity n, can be expressed as:

$$\mathtt{op}(v_i)_{i=0}^{n-1} = \mathtt{op}(v_0, v_1, ..., v_{n-2}, v_{n-1})$$

Chapter 3

Methodology

This work proposes a verification methodology that extends the approach described in previous works [42, 45]. Figure 3.1 shows the tailored and expanded version of the original approach applied to the analysis of DAIDALUS. The different steps of the approach are summarized below according to the case study analyzed in the present work.

1. Due to the restrictions of the PRECiSA tool, we translate the higher-order specification to equivalent first-order declarations. In the case study, the higher-order definitions in the DAIDALUS library are manually translated into semantic-preserving declarations in which the higher-order parameters are downgraded to first-order arguments.

2. We prove the equivalence between the original specification and the proposed first-order version using the PVS theorem prover.

3. We split the specification lifting the conditions on the relative velocities of the aircraft. This process resulted in six slices determining each of the following cases:

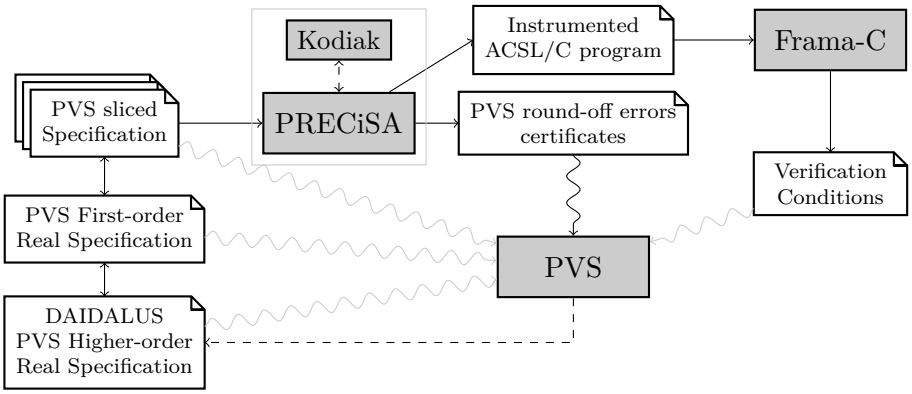


Figure 3.1: Workflow of the verification approach.

- aircraft maintain both their horizontal and vertical separation unmodified;
- the vertical separation increases, but the horizontal one is kept unmodified;
- the vertical separation decreases, but the horizontal one is kept unmodified;
- aircraft alter only their horizontal separation;
- aircraft increase vertical and alter their horizontal separation;
- aircraft decrease vertical and alter their horizontal separation.

4. We prove the equivalence between each slice specification and the original higher-order declarations assuming the slice input restriction.
5. We prove the soundness of the slicing, i.e., selecting the correct slice in each situation covers all input possibilities, and it is equivalent to the higher-order declaration.
6. For each slice, we follow the same methodology proposed in [45]:
   - provide a real-valuated specification, as well as input variable range to PRECiSA, generating a stable-guard ASCL instrumented C-Code, numerical errors bounds, and PVS certificates ensuring round-off errors properties;
   - apply Frama-C/WP to output verification conditions (VCs) from the generated C-code, and;
   - use PVS to prove the verification conditions.
7. We manually add a top-level C function with the purpose of selecting the slice corresponding to a specific input. This function is annotated with pre- and post-conditions allowing to verify its adherence to the first-order declarations from 1.

The proposed methodology adds to the approach [42, 45] the steps related to restating from higher-order to first-order and splitting the specification into slices. Conditional slicing has the advantage of simplifying the formal analysis substantially. Since the analysis looks for unstable guards, the more possible control flows, the more complexity in the analysis. However, it assumes the stability of the guards associated with determining the slice to be executed in the program flow of the first-order specification. So, it is important to highlight that this approach considers unstable guards only inside each slice, leaving aside the possibility of dealing with unstable tests at the top-level function. This issue will be addressed in future works.

Figure 3.1 depicts the actors' interaction in the approach, and their roles are described below.

- The translation from higher- to first-order and the slicing of the declarations are performed manually.

- The functional equivalence of these specifications is formalized in PVS.
- PRECiSA generates round-off certificates and stable-guard C-code from each slice.
- Frama-C analyses the generated code and output verification conditions to be proved in PVS.

In the next sections, we will present each tool participating in our approach's toolchain.

3.1 PRECiSA

PRECiSA (**P**rogram **R**ound-off **E**rror **C**ertifier v**i**a **S**tatic **A**nalysis) is a static analyzer for floating-point programs. Given a floating-point specification, computes symbolic error expressions and, if ranges for the input values are provided, it uses a global optimizer (Kodiak) to provide an over-approximation of roundoff-error bounds, along with PVS certificates ensuring its correctness [45, 42].

In fact, since IEEE-754 establishes that basic operations should be performed as if they had infinite precision and then rounded back to the desired precision, the evaluation of the application of an arbitrary floating-point operation has two main errors when compared with its ideal real counterpart: the rounding error already accumulated in the variables appearing in the expression, and the error originated in the rounding of the resulting value. Equation 3.1 express a bound for the rounding-error of any real value x included between the largest negative and the largest positive representable numbers, under the *to-the-nearest* rounding modality.

$$| \boxed{x} - x | \leq \frac{1}{2} \, ulp(x) \tag{3.1}$$

In the equation above, $\boxed{x}$ is the floating-point counterpart of x and $ulp(x)$ is *unit in the last place*, which can be seen as a measure of the floating-point representation precision. $ulp(x)$ is expressed by $base^{\exp(\boxed{x})-p+1}$, where $\exp(\boxed{x})$ is the exponent of the floating-point number $\boxed{x}$, p is the precision and *base* is the base of format under use.

Similarly for an arbitrary operation $\mathtt{op}$ with arity $n+1$, for any $n \in \mathbb{N}_0$:

$$| \boxed{\mathtt{op}(v_i)_{i=0}^{n}} - \mathtt{op}(\boxed{v_i})_{i=0}^{n} | \leq \frac{1}{2} \, ulp(\mathtt{op}(\boxed{v_i})_{i=0}^{n}) \tag{3.2}$$

where $\mathtt{op}(v_i)_{i=0}^{n}$ is an abbreviation for $\mathtt{op}(v_0, v_1, ..., v_n)$. Note that in this notation, when applying a real operation on floating-point arguments an implicit conversion from floats to reals its assumed. For example, Equation 3.2 shows the particular instance of Equation 3.3

obtained by instantiating op with the multiplication operation.

$$| \boxed{v_1 * v_2} - \boxed{v_1} * \boxed{v_2} | \leq \frac{1}{2}\, ulp(\boxed{v_1} * \boxed{v_2}) \tag{3.3}$$

The propagation of the round-off error depends on the operation and input values. For instance, for the multiplication, the propagation error for two floats, $\boxed{v_1}$ and $\boxed{v_2}$, with respective non-negative round-off errors bounds e_1, e_2, is given below.

$$\begin{aligned} \boxed{v_1} * \boxed{v_2} &= (v_1 \pm e_1) * (v_2 \pm e_2) \\ &= (v_1 * v_2) + (\pm v_1 e_2 \pm v_2 e_1 \pm e_1 e_2) \end{aligned}$$

Thus, one possible expression that bounds the error propagation produced when multiplying two values, denoted as $\epsilon_*(v_1, e_1, v_2, e_2)$, is:

$$\begin{aligned} |(\boxed{v_1} * \boxed{v_2}) - (v_1 * v_2)| = | \pm v_1 e_2 \pm v_2 e_1 \pm e_1 e_2 | &\leq \epsilon_*(v_1, e_1, v_2, e_2) \\ &\leq |v_1| e_2 + |v_2| e_1 + e_1 e_2 \end{aligned} \tag{3.4}$$

In the following, the greek letter ϵ will be used to denote error propagation for any operator op, i.e., $\epsilon_{\mathtt{op}} = |\mathtt{op}(\boxed{v_i})_{i=0}^{n} - \mathtt{op}(v_i)_{i=0}^{n}|$. Also, the total error of an arbitrary expression $\boxed{E}$ will be denoted as $e_E = | \boxed{E} - E |$. Finally, the error of a floating-point operation op, will be expressed by $e_{\mathtt{op}} = | \boxed{\mathtt{op}(v_i)_{i=0}^{n}} - \mathtt{op}(v_i)_{i=0}^{n} |$.

Expanding the module in the inequalities 3.3 and 3.4, one has:

$$\begin{aligned} -\frac{1}{2}\, ulp(\boxed{v_1} * \boxed{v_2}) \leq \boxed{v_1 * v_2} - (\boxed{v_1} * \boxed{v_2}) &\leq \frac{1}{2}\, ulp(\boxed{v_1} * \boxed{v_2}) \\ -\epsilon_*(v_1, e_1, v_2, e_2) \leq (\boxed{v_1} * \boxed{v_2}) - (v_1 * v_2) &\leq \epsilon_*(v_1, e_1, v_2, e_2) \end{aligned}$$

Summing both inequalities, you get:

$$| \boxed{v_1 * v_2} - (v_1 * v_2) | \leq \frac{1}{2}\, ulp(\boxed{v_1} * \boxed{v_2}) + \epsilon_*(v_1, e_1, v_2, e_2) \tag{3.5}$$

Since *ulp* is monotonically increasing:

$$ulp(\boxed{v_1} * \boxed{v_2}) \leq ulp((|v_1| + e_1) * (|v_2| + e_2)) \tag{3.6}$$

More generically, for an arbitrary operation op of arity n:

$$| \boxed{\mathtt{op}(v_i)_{i=0}^{n}} - \mathtt{op}(v_i)_{i=0}^{n} | \leq e_{\mathtt{op}}(v_i, e_i)_{i=0}^{n} \tag{3.7}$$

$$= \epsilon_{\mathtt{op}}(v_i, e_i)_{i=0}^{n} + \frac{1}{2}\, ulp(\mathtt{op}(\boxed{v_i})_{i=0}^{n}) \tag{3.8}$$

For instance, consider the Specification 3.1, which shows the defintion of a function computing the time of the closest point approach between two aircraft. In that function, the round-off error bound of expression named $\boxed{v}$ by the *let-in* statement can be expressed as:

$$e_v = e_+(v_x^2, e_{v_x^2}, v_y^2, e_{v_y^2}) = e_{v_x^2} + e_{v_y^2} + ulp(|v_x^2 + v_y^2| + e_{v_x^2} + e_{v_y^2}) \tag{3.9}$$

where $e_{v_x^2}$ and $e_{v_y^2}$ are the errors of $\boxed{v_x^2}$ and $\boxed{v_y^2}$, respectively, and the error expression $e_{v_x^2}$ (also $e_{v_y^2}$ replacing x by y) is defined as:

$$\begin{aligned} e_{v_x^2} = e_*(v_x, e_x, v_x, e_x) &= |v_x|e_{v_x} + |v_x|e_{v_x} + e_{v_x}e_{v_x} + ulp((|v_x| + e_{v_x}) * (|v_x| + e_{v_x})) \\ &= 2|v_x|e_{v_x} + e_{v_x}^2 + ulp((|v_x| + e_{v_x})^2) \end{aligned}$$

where e_{v_x} is the error bound of $\boxed{v_x}$.

While calculating the error expressions for a given program, PRECiSA keeps the expression and a set of conditional error bounds. When possible, it is also able to compute tighter bounds. For instance, the subtraction $\boxed{v_1 - v_2}$ can be exactly calculated, i.e., without an additional rounding error when $v_2/2 \leq v_1 \leq 2v_2$. This is particularly useful once a branch statement is followed, where an additional bound is added to this set.

Once every symbolic error expression is derived, the global optimizer Kodiak is called with assumptions on input ranges, for instance $(1 < s_x < 1000)$, to overapproximate its numeric error bounds. Applying the optimizer to symbolic e_v previously defined in the ranges $1 < v_x < 1000$ and $1 < v_y < 1000$, we get the numeric value $4.602043e-10$.

PRECiSA does not assume stable guards (described in Section 2.1.1). Thus, the computed error bound considers the possibility of taking a wrong branch in the conditional statements. For example, The error bound of function `tcpa` is calculated as:

$$|\boxed{\texttt{tcpa}(s_x, v_x, s_y, v_y)} - \texttt{tcpa}(s_x, v_x, s_y, v_y)| \leq max(e_{-(s_x * v_x + s_y * v_y)/v}, 0) \tag{3.10}$$

In a different mode of operation, PRECiSA can receive a real-valued program and concrete range for its arguments and generate a stable-guard floating-point C-code, that

$\boxed{\texttt{tcpa}(s_x, v_x, s_y, v_y)}$ =
  let $\boxed{v}$:= $\boxed{v_x * v_x + v_y * v_y}$ in
  if ($\boxed{v > 0}$) then $\boxed{-(s_x * v_x + s_y * v_y)/v}$
  else $\boxed{0}$

Specification 3.1: Function computing the closest point of approach of two aircraft given relative positions and velicities.

```
tcpa(s_x, v_x, s_y, v_y, ε) =
    let v := v_x * v_x + v_y * v_y in
        if v > ε then −(s_x * v_x + s_y * v_y)/v
        else if v < −ε then 0
             else ω
             endif
        endif
```

Specification 3.2: Transformed version of the function in Specification 3.1.

is, a program that emits a warning if an unstable guard could occur. The generated C-code is annotated using a domain-specific specification language (ACSL) enabling its automatic verification [45].

In fact, in this mode, PRECiSA replaces the original conditionals with more restrictive guards. For instance,

$$\texttt{IF guard}(x_1,\ldots,\ x_n)\ \texttt{THEN}\ x_1\ \texttt{ELSE}\ x_2\ \texttt{ENDIF} \rightsquigarrow \begin{array}{l}\texttt{IF guard}_{\beta+}(x_1,\ldots,\ x_n,\ e_s)\\ \texttt{THEN}\ x_1\\ \texttt{ELSE}\\ \quad\texttt{IF guard}_{\beta-}(x_1,\ldots,\ x_n,\ e_s)\\ \quad\texttt{THEN}\ x_2\\ \quad\texttt{ELSE}\ \omega\ \texttt{ENDIF}\\ \texttt{ENDIF}\end{array}$$

where ω is a distinguished value represented a warning, e_s is a bound for the round-off error bound of the expression s, and `guard` is an expression of form:

$$\begin{aligned}\texttt{guard}(x_1,\ldots,x_n) &\equiv s > 0\\ \texttt{guard}_{\beta+}(x_1,\ldots,x_n,,e_s) &\equiv s > e_s\\ \texttt{guard}_{\beta-}(x_1,\ldots,x_n,,e_s) &\equiv s < -e_s\end{aligned}$$

In the `tcpa` example, the transformation generates the function shown in Specification 3.2. There, PRECiSA adds one symbolic error variable to each branch expression as a parameter. Compared with just adding one error variable to each input parameter, according to the authors, this approach can solve one main problem: avoid huge symbolic expressions, which can be more time-consuming and have worse readability. In the `tcpa` example, only the variable ϵ was added.

In this work, PRECiSA is applied to each real specification slice, generating its stable-guard C-code. Its output is then analyzed by Frama-C which generates corresponding

```
1 /*@
2 logic real tcpa (real sx, real vx, real sy, real vy) =
3     \let v = (vx * vx) + (vy * vy);
4         (v > 0) ? (-((sx * vx) + (sy * vy)) / v) : 0;
5 */
```

Specification 3.3: tcpa specification in ACSL language.

verification conditions as explained in the following section.

3.2 Frama C

Frama-C is a collaborative framework for the static analysis of C programs [23]. One of its main features is extensibility, which allows the addition of diverse plugins. Among them, the WP plugin [2] implements a Weakest Precondition calculus, translating annotations into mathematical properties, and allowing the use of external tools to prove the correctness of program contracts written in the code using the ANSI/ISO C Specification Language (ACSL). ACSL contracts are introduced as regular C comments, but starting with the at-sign (@). Their semantics is based on a restricted form of typed first-order logic, where only total functions are allowed. The syntax of ACSL is similar to a subset of C syntax [2]. For instance, the real version of the `tcpa` function could be expressed as shown in Specification 3.3. In line 3, we have a *let-in* statement that introduces the name `v` for the square of horizontal velocity expression $(vx * vx) + (vy * vy)$. In ACSL, special keywords begin with a backslash. For example, the let-in syntax has form:

```
\let name = expression1; expression2;
```

where `expression2` is the scope where `expression1` is named as `name`. In Specification 3.3, the name `v` is defined in line 3 and its scope is line 4.

Following the declaration of the `tcpa` function in ACSL presented in Specification 3.3, there is a conditional expression in line 4 similar to the in-line C conditional, this is the unique way that ACSL provides support to branching expressions. Their syntax is:

```
(guard_expression)? expression1: expression2;
```

where `guard_expression` is the branch guard, if the guard is evaluated as true then it evaluates to `expression1`, otherwise, `expression2`.

The syntax for the quantifiers is similar to the let-in statements:

```
\forall type variables; expression;
```

where `type` could be any ACSL type, multiple variables are separated by coma, and `expression` is the scope of the quantifier.

Program contracts such as preconditions and postconditions are introduced before each function declaration with keywords `requires` and `ensures`, respectively. For instance, a function that calculates the squared norm of a 2D vector is shown below.

```
/*@
assigns \nothing;
ensures \result >= 0;
*/
double squared_norm(double vx, double vy) {
  double v = vx * vx + vy * vy;
  return v;
}
```

The expression `assigns \nothing` means that no variable is assigned by reference. The keyword `\result` denotes the result of the annotated function, and the postcondition states that the result of this function is greater or equal to zero.

Once preconditions and postconditions are introduced, the WP plugin can be applied. It implements a weakest-precondition calculus to generate verification conditions to be processed by off-the-shelf solvers, such as theorem provers [2]. Dijkstra originally introduced the weakest-precondition calculus [13], which is based on the notion of Hoare triples:

$$\{P\}\ f\ \{Q\}$$

where f is a program, and P and Q are predicates on the states of f. The intuitive meaning of these structures can be stated as "If P is true before running a program f, then Q will hold after its execution" [21]. Being A and B two formulas, A is said to be *weaker* than B when:

$$B \Rightarrow A \tag{3.11}$$

The Weakest Precondition for a program f allowing it to reach a state described by Q, and noted as $wp(f, Q)$, is the precondition of f that all other such preconditions imply, i.e., the "necessary and sufficient precondition" [13]. The weakest precondition calculus applies the predicate transformer wp to each imperative program statement, transforming a postcondition into a weak precondition while transversing the program backwards [13]. For instance, consider the previous code snippet. Since there is a variable assignment, the postcondition is transformed using the rule:

$$wp(x := E, Q) = Q[x/E] \tag{3.12}$$

meaning that the weakest precondition for the assignment of an expression E to a variable

```
sqrt(nnx: nnreal): {nnz : nnreal | nnz*nnz = nnx}
```

Specification 3.4: PVS signature of the square root function.

x and a postcondition Q is the substitution of x by E in Q. In the Specification 3.2, the postcondition $v \geq 0$ is transformed in $vx * vx + vy * vy \geq 0$.

When validating the program contracts, we aim to verify that the precondition P, the program f, and the postcondition Q is a Hoare triple $\{P\}\ f\ \{Q\}$. Using the fact that weakest precondition wp, f and Q is a Hoare triple, one needs to prove that precondition P implies the wp, as shown in [2]:

$$\frac{P \Rightarrow wp(f, Q) \qquad wp(f, Q)\ \{f\}\ Q}{\{P\}\ f\ \{Q\}}$$

In this work, Frama-C/WP receives the instrumented stable-guard C code annotated with ACSL generated by PRECiSA and generates verification conditions in the PVS language.

3.3 PVS

Introduced by SRI International in 1996, the Prototype Verification System (PVS) is a complete formal specification and verification environment. Mainly composed of a strongly-typed specification language supporting higher-order logic and an interactive theorem prover with a robust automated deduction engine [34].

The PVS specification language is functional, supporting pre-defined basic types and user-defined type definitions, subtyping and dependent types. Type definitions cannot be recursive and use only previously declared symbols. In PVS, types are definitionally equivalent; that is, two types are equal if their declaration match, i.e., types are handled as different unless their definitions are equivalent. Subtyping is supported by allowing to add constraints to a given type by using a predicate. This is particularly useful to specify a function or predicate that is only defined for restricted input ranges. For instance, Specification 3.4 shows the PVS signature of the square root on real numbers, a function that requires a non-negative input. PVS provides built-in sub-types, as in the previous case, the non-negative reals (`nnreal`). The PVS semantics is even more powerful with the support of dependent types, where a previously defined variable appears in the predicate used to restrain the type. For instance, it is possible to define an ideal square root function as it is shown by Specification 3.4 (taken from the NASALib). This function requires a non-negative real as an argument, specified by `nnx` type `nnreal`, and returns a non-negative real called `nnz` such as the input is equal to `nnz` squared. Another example

```
root((a: real | a ≠ 0), b, (c: real | b^2 - 4*a*c >=0),
     (sign: nat | sign = 1 OR sign = -1)): real =
    (-b + sign * sqrt(b*b - 4*a*c)) / (2*a)
```

Specification 3.5: PVS definition of a function calculating the roots of a quadratic polynomial.

```
sqrt_TCC1: OBLIGATION
    EXISTS (x: [nnx: nnreal -> {nnz: nnreal | nnz * nnz = nnx}]): TRUE
```

Specification 3.6: One of the TCCs generated by type checking the definition in Specification 3.4.

is shown in Specification 3.5, where the dependent types are used in the arguments of a function that calculates de roots of a polynomial of order 2. The `root` function receives as an argument the polynomial coefficients called `a, b, c` and the sign (indicating plus or minus in the Baskara rule). This function requires a non-zero `a`, a `c` that ensures a non-negative discriminant, and a natural number that can be interpreted as a sign, i.e., 1 or −1.

The use of subtypes and dependent types adds some delicate issues, such as the matching between the types required by a function or predicate and its calling arguments or even the existence of some value that satisfy the restriction. The type checker in PVS uses a static analysis engine to generate proof obligations, called *Type Check Conditions* (TCCs). Some of them are usually extremely simple and can be automatically proven using the PVS proof engine, but there are exceptions because compatibility/type-equivalence is undecidable [35]. For instance, consider the definition of square root in Specification 3.4, where the return type is a dependent type. The type-checking system generates a proof obligation presented in Specification 3.6. This TCC requires proving the existence of a function mapping from non-negative real numbers to non-negative real numbers, where the square of the image of an input number `nnx` under this function is the actual input number. Another example is a TCC that ensures there is no division by zero, as presented in Specification 3.7.

The specification language of PVS supports if-then-else and case statements. While these kinds of conditionals are not usually defined in classical logic systems, they are

```
tcoa_TCC1: OBLIGATION
    FORALL (sz, vz: real): (sz * vz < 0) IMPLIES vz ≠ 0
```

Specification 3.7: Example of TCC.

```
first_conflict_step(CD,B,T,traj,k,ts,MaxN,AL): RECURSIVE
(first_conf_step?(CD,B,T,traj,k,ts,MaxN,AL)) =
    IF k > MaxN THEN -1
    ELSIF first_conflict_aircraft(CD,B,T,traj,k,ts,0,AL) >= 0
        THEN k
    ELSE first_conflict_step(CD,B,T,traj,k+1,ts,MaxN,AL)
    ENDIF
MEASURE max(MaxN-k+1,0)
```

Specification 3.8: Example of recursive definition in PVS.

interpreted as follows in PVS.

$$\text{if } A \text{ then } B \text{ else } C \equiv A \Rightarrow B \wedge \neg A \Rightarrow C \tag{3.13}$$

Recursive definitions are also allowed in PVS and are introduced using the keyword `RECURSIVE`. The keyword `MEASURE` has to be used to provide a function on the same parameters of the declaration. This auxiliary function has to decrease after each recursive call. This property is expressed in the TCCs that ensure the termination of the function. For instance, Specification 3.8 shows a function taken from the DAIDALUS specification whose goal is to detect if a red band (high-priority alert region) exists. where the function returns an integer in which the `first_conf_step?` predicate is valid and the parameter `CD` is a conflict detection function, i.e., a function that maps from a lookahead time and relative states to a boolean indicating a conflict; `B` and `T` are non-negative reals indicating the limits of lookahead time; `traj` is the aircraft trajectory,i.e., a function mapping from a non-negative real (time) to the aircraft state (space and velocity vectors); `k` is a natural number indicating the timestep interaction; `ts` is a non-negative real indicating the timestep value; `MaxN` is a natural number indicating the máximum number of timesteps; and, `AL` the aircraft list (state and name). In one of the generated TCCs, the user must prove the measure is decreasing `max(maxN-k+1, 0)` in the recursive call. This is easily demonstrated for this function since it is called increasing `k` and maintaining `MaxN`. This is a way to emulate interactive for-loops using recursive functions.

The PVS deduction engine is based on Gentzen's sequent calculus (SC), where each moment in a proof is represented by a pair of finite collections of formulas called *antecedent* and *consequent*, respectively. Each of these pairs of collections is called a *sequent* and, if the symbols Σ and Λ are used to represent the antecedent and the consequent, the sequent is noted as

$$\Sigma \vdash \Lambda \tag{3.14}$$

The intuitive idea behind a sequent is that it represents the implication between

```
horizontal_WCV_taumod_interval_equivalence: LEMMA
 FORALL(sx, sy, vx, vy, T):
    LET s = (# x := sx, y := sy #),
        v = (# x := vx, y := vy#) IN
    LET hni = hor_WCV_ti(T, s, v)
    IN hor_WCV_ti(T, sx, sy, vx, vy, TAUMOD, DTHR)
    IFF hni`entry <= hni`exit
```

Specification 3.9: Example of a lemma in PVS.

the conjunction of formulas in the antecedent and the disjunction of formulas in the consequent. The proof evolves by the application of so-called *proof rules*. This application could close the proof or generate one or more new sequents, generating new proof branches. The proof is considered finished when all branches in the proof tree are closed. The PVS proofs start with the objective and end when obviously true sequents are reached. Note that the proof trees are generated in the inverse order with respect to the classic sequent calculus. For instance, consider the lemma of equivalence between the original predicate `horizontal_WCV_taumod_interval`, for short `hor_WCV_ti`, and its simplified first-order version, as shown in Specification 3.9. In this case, the first sequent of the corresponding proof is shown below.

```
|-------------------------------------
{1}   FORALL (sx, sy, vx, vy, T):
        LET s: [# x: real, y: real #] = (# x := sx, y := sy #),
            v: [# x: real, y: real #] = (# x := vx, y := vy #)
          IN
          LET hni: EntryExit[0, T] = hor_WCV_ti(T, s, v)
            IN
            hor_WCV_ti(T, sx, sy, vx, vy, TAUMOD, DTHR)
            IFF hni`entry <= hni`exit
```

where the dotted line represents the turnstile $\vdash$, and the formulas below it, i.e., with a positive number, are the ones in the consequent. Some proof branch ends when one of the axiomatic rules (Ax, FALSE$\vdash$, $\vdash$TRUE) is applied.

$$\frac{}{\Sigma, a \vdash a, \Lambda}\,(\text{Ax}) \qquad \frac{}{\Sigma, \text{FALSE} \vdash \Lambda}\,(\text{FALSE}\vdash) \qquad \frac{}{\Sigma \vdash \text{TRUE}, \Lambda}\,(\vdash\text{TRUE})$$

The first step of the proof in the example is the elimination of the quantifier in the consequent performing a Skolemization using the PVS rule called `skeep`:

$$\frac{\Gamma \vdash A[x/y], \Lambda}{\Gamma \vdash \forall_x A, \Lambda}\ (\text{skeep } +) \qquad \frac{\Gamma, A[x/y] \vdash \Lambda}{\Gamma, \exists_x A, \vdash \Lambda}\ (\text{skeep -})$$

where y is a fresh name. Note that PVS rules are the inverse of SC rules, i.e., the bottom part of rules is before the rule application and the top after. The `skeep` command replaces the variable with the same name if there is no name collision. In the example, the following sequent is generated by the application of `skeep`.

```
|-------------------------------
{1}   LET s: [# x: real, y: real #] = (# x := sx, y := sy #),
          v: [# x: real, y: real #] = (# x := vx, y := vy #)
        IN
        LET hni: EntryExit[0, T] = hor_WCV_ti(T, s, v)
          IN
          hor_WCV_ti(T, sx, sy, vx, vy, TAUMOD, DTHR)
           IFF hni`entry <= hni`exit
```

The next step is to simplify the sequent eliminating the let-in expressions, using the proof rule implementing β-reduction:

$$\frac{}{\vdash (\lambda(x:T):a)(b) = a[b/x]}\ \beta$$

The application of such a rule generates the sequent shown below.

```
|------------------------
{1}   hor_WCV_ti(T, sx, sy, vx, vy, TAUMOD, DTHR)
      IFF
       hor_WCV_ti(T,(# x := sx, y := sy #),
                    (# x := vx, y := vy #))`entry
        <=
        hor_WCV_ti(T,(# x := sx, y := sy #),
                     (# x := vx, y := vy #))`exit
```

The next step is to introduce a hypothesis about the equivalence of an internal function. This process is equivalent to using the cut rule of SC:

$$\frac{\Gamma, a \vdash \Lambda \qquad \Gamma \vdash a, \Lambda}{\Gamma \vdash \Lambda}\ \text{Cut}$$

One introduces a new hypothesis (α) in the left branch and needs to prove it in the right branch. This can be done in two ways: in the same proof using the PVS command `case` or in a separate lemma, which is introduced using the `lemma` command. Usually, separate lemmas are specified in terms of universal quantifiers just like in this case, so

is necessary to instantiate the lemma with suitable parameters using the PVS command `inst`:

$$\frac{\forall_x A, \Gamma, a \vdash \Lambda}{A[x/y], \Gamma \vdash \Lambda}\,(\text{inst-})$$

Obtaining after introducing the lemma `theta_d_equivalence`, which states the equivalence of `Theta_D` function, beta reduction and renaming some terms:

```
{-1}   ((vx /= 0 OR vy /= 0) AND Delta(sx, sy, vx, vy, DTHR) >= 0) =>
       (Theta_D[DTHR](s, v, 1) = Theta_D_pos(DTHR, sx, sy, vx, vy) AND
         Theta_D[DTHR](s, v, -1) = Theta_D_neg(DTHR, sx, sy, vx, vy))
|-------------------------------------
{1}   hor_WCV_ti(T, sx, sy, vx, vy, TAUMOD, DTHR)
      IFF
       hor_WCV_ti(T, s, v)`entry <= hor_WCV_ti(T, s, v)`exit
```

There is an implication in the antecedent, so one can apply the sequent calculus rule L→ rule executed by the PVS command `split`, which also executes L∨ and R∧ SC rules and splits the if-then-else, which is interpreted as a conjunction of implications.

$$\frac{B, \Gamma \vdash, \Lambda \qquad \Gamma \vdash A, \Lambda}{A \to B, \Gamma \vdash \Lambda}\,\text{L}\to \qquad \frac{A, \Gamma \vdash, \Lambda \qquad B, \Gamma \vdash \Lambda}{A \vee B, \Gamma \vdash \Lambda}\,\text{L}\vee$$

$$\frac{\Gamma \vdash A, \Lambda \qquad \Gamma \vdash B, \Lambda}{\Gamma \vdash A \wedge B, \Lambda}\,\text{R}\wedge \qquad \frac{A, B, \Gamma \vdash, \Lambda \qquad C, \Gamma \vdash A, \Lambda}{\Gamma \vdash \text{if } A \text{ then } B \text{ else } C, \Lambda}\,\vdash\text{IF}$$

The first branch after the application of the previous command is:

```
{-1} Theta_D[DTHR](s, v, 1) = Theta_D_pos(DTHR, sx, sy, vx, vy)
     AND
     Theta_D[DTHR](s, v, -1) = Theta_D_neg(DTHR, sx, sy, vx, vy)
|-------------------------------------
[1] hor_WCV_ti(T, sx, sy, vx, vy, TAUMOD, DTHR)
    IFF
    hor_WCV_ti(T, s, v)`entry <= hor_WCV_ti(T, s, v)`exit
```

Since the antecedent has a conjunction, the L∧ rule of the sequent calculus can be applied. This is done using the command `flatten`, which also executes R∨ and R→, and IF⊢.

$$\frac{A, B, \Gamma \vdash, \Lambda}{A \wedge B, \Gamma \vdash \Lambda}\,\text{L}\wedge \qquad \frac{A \to B, \neg A \to C, \Gamma \vdash \Lambda}{\text{if } A \text{ then } B \text{ else } C, \Gamma \vdash \Lambda}\,\text{IF}\vdash$$

$$\frac{\Gamma \vdash A, B, \Lambda}{\Gamma \vdash A \vee B, \Lambda}\,\text{R}\vee \qquad \frac{A, \Gamma \vdash B, \Lambda}{\Gamma \vdash A \to B, \Lambda}\,\text{R}\to$$

Another possible command is `prop`, which applies all the possible propositional simplifications, i.e., `split`, `flatten`, etc. Now, there are two equality formulas in the antecedent. At some point in the proof, one needs to use the equality rule provided by the command `replace`:

$$\frac{a = b, \Gamma[b] \vdash \Lambda[b]}{a = b, \Gamma[a] \vdash \Lambda[a]}\ \text{Repl}$$

Sometimes some formulas are irrelevant and can be omitted through structural rules. For instance, in some moments of the previous proof, one arrives at the following sequent.

```
[-1]   sqv(s) - sq(DTHR) <= 0
[-2]   min_(T, Theta_D_pos(DTHR, sx, sy, vx, vy)) =
       min(T, Theta_D_pos(DTHR, sx, sy, vx, vy))
[-3]   FALSE => s * v = 0
[-4]   DTHR * DTHR = sq(DTHR)
[-5]   Delta(sx, sy, vx, vy, DTHR) = Delta[DTHR](s, v)
[-6]   inner_product(sx, sy, vx, vy) = s * v
[-7]   sq(sx, sy) = sqv(s)
[-8]   sq(vx, vy) = sqv(v)
|-------------------------------------
{1}    (vx ≠ 0 OR vy ≠ 0)
[2]    sqv(v) = 0
[3]    0 <= min_(T, Theta_D_pos(DTHR, sx, sy, vx, vy)) IFF
        0 <= min(T, Theta_D[DTHR](s, v, 1))
```

Here, only formulas 1 and 2 are necessary to close this branch since either one of the components is different from zero or the squared norm of velocity is zero. So one could use structural rules (WL, WR) from sequent calculus to hide the unused formulas; this is done using the PVS command `hide`:

$$\frac{\Gamma \vdash \Lambda}{A, \Gamma \vdash \Lambda}\ \text{WL} \qquad\qquad \frac{\Gamma \vdash \Lambda}{\Gamma \vdash A, \Lambda}\ \text{WR}$$

PVS formally supports the earlier software and hardware development life-cycle stages. Usually, it happens when the system is still in a modeling stage [34]. However, in the current work, following the toolchain proposed in [45], PVS provides formal support in the latter stages to prove verification conditions generated from the final concrete code.

Chapter 4

Case Study: DAIDALUS

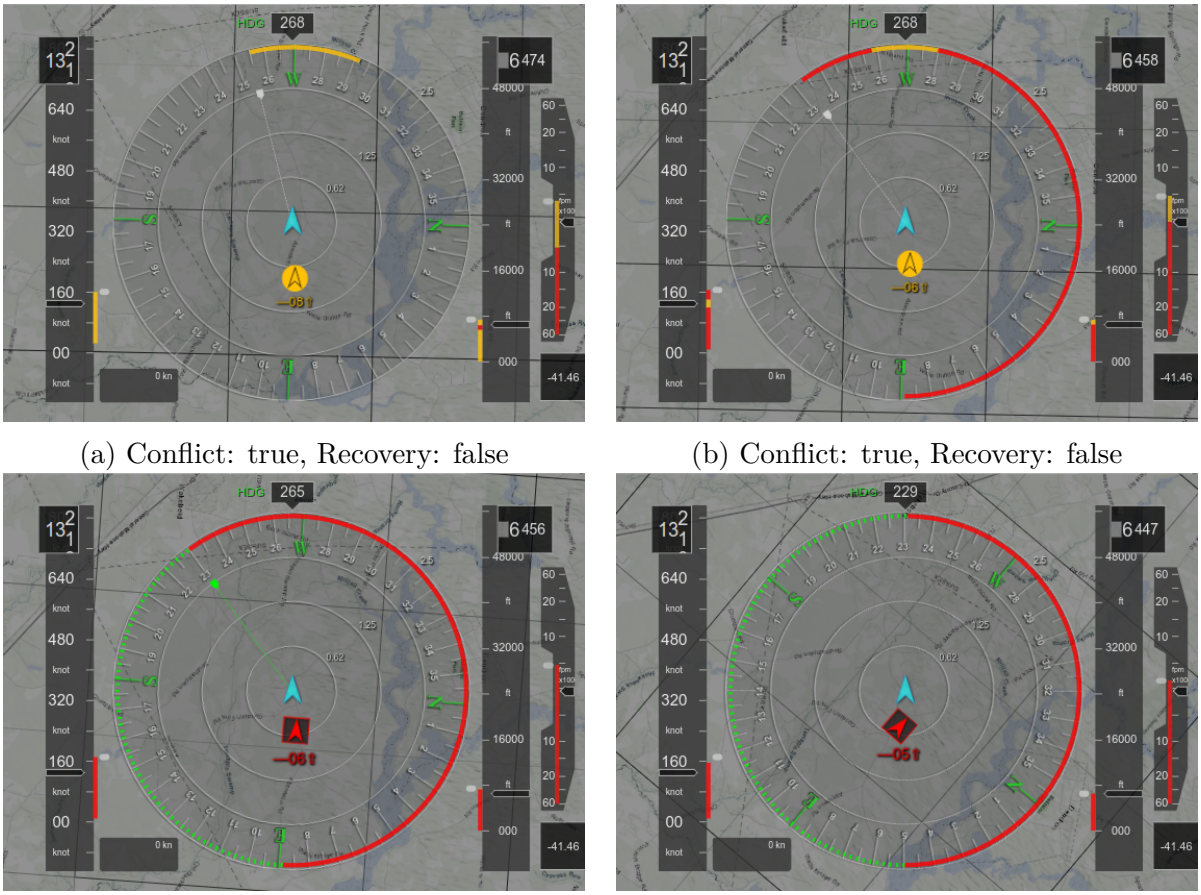
4.1 DAIDALUS description

The Detect and AvoID Alerting Logic for Unmanned Systems library (DAIDALUS) is a collection of algorithms providing detect and avoid (DAA) capabilities [29]. DAIDALUS was included in RTCA/FAA Minimum Operational Performance Standards (MOPS) DO365 as the reference implementation. Its goal is to advance the integration of Unmanned Aircraft Systems into the US National Airspace System (NAS) in non-segregated civil airspace. Several functions in the DAIDALUS libraries are actually implemented in C and Java. These implementations are the result of a manual translation from formal specifications in a more expressive language [29]. In this chapter, we present a description of the PVS DAIDALUS higher-order specification (see Figure 3.1). It provides three main features:

- Conflict Detection
- Maneuver Guidance
- Alerting

Conflict detection determines the current status of aircraft regarding the surrounding air traffic and its prediction interval of occurrence given some assumptions as straight-line trajectory. Maneuver Guidance calculates the possible maneuvers to keep or recover the well-clear status. Alerting emits different levels of hazard signals.

Figures 4.1 and 4.2 are part of an example of an application to present information computed by the DAIDALUS algorithms to pilots. This application, called Danti, is also being developed by researchers at NASA Langley [26]. In these figures, the ownship is displayed in the center of the display as a cyan chevron, and the color used in the near traffic represents the different kinds of alerting signals. As is common in cockpit instru-



(a) Conflict: true, Recovery: false

(b) Conflict: true, Recovery: false

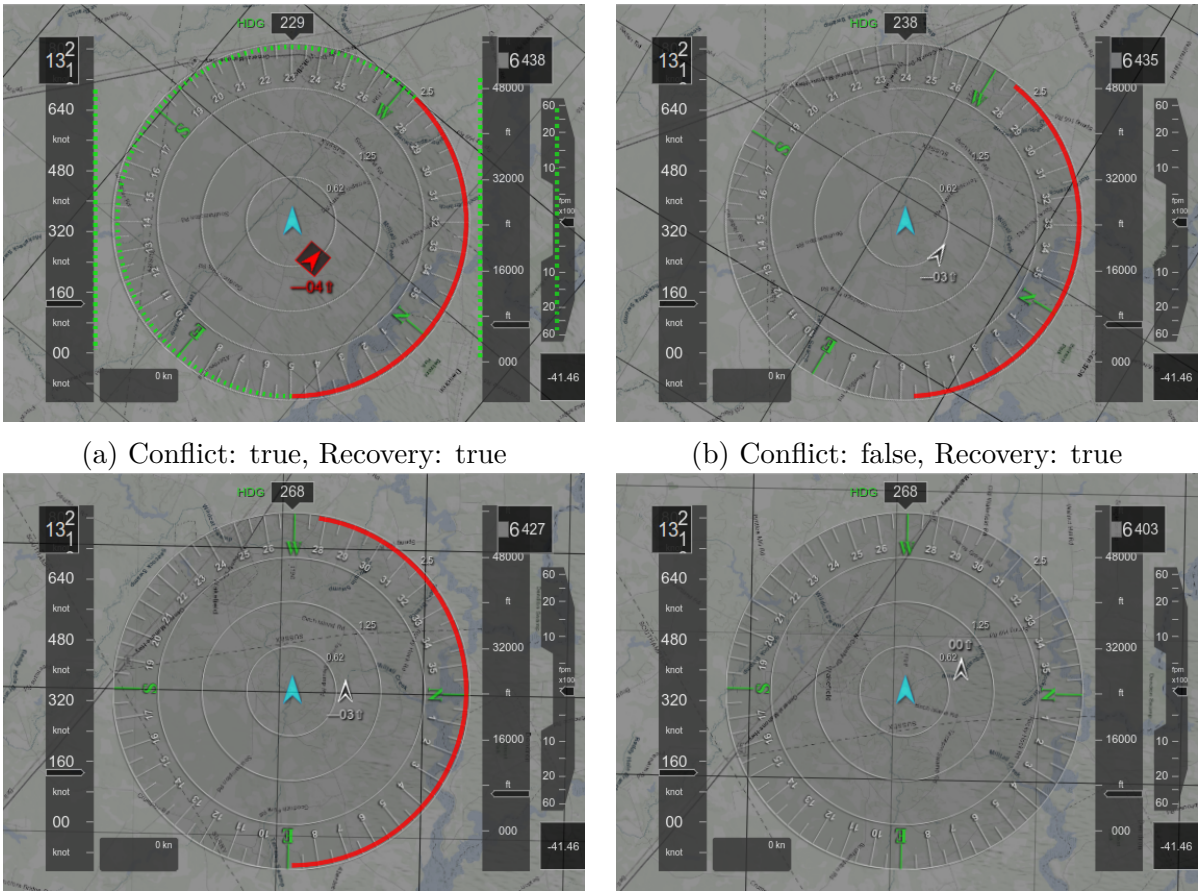
(c) Conflict: true, Recovery: true

(d) Conflict: true, Recovery: true

Figure 4.1: DAIDALUS conflict detection and maneuvering guidance using DAA-display. Example extracted from [26].

ments, the display shows the airspeed, altitude, and vertical velocity of the ownship using vertical strips on the sides of the screen. The heading is displayed using a compass centered at the ownship representation. Figures 4.1 and 4.2 show a sequence of frames from the same encounter, exemplifying how a conflict situation and some recovery guidance are presented to the pilot. Initially, the alert is presented in Figure 4.1a, but its level is 2 (MID), and thus no action is taken. The yellow bands on the compass's perimeter and the strips depict heading, velocities, and altitude that must be avoided to disarm the conflict. In Figure 4.1b, the bands are larger and mostly red. As expected, the red color means a more critical situation than yellow. In Figure 4.1c, the recovery maneuver begins. The conflict ends in Figure 4.2a, but the aircraft is still in the recovering maneuver to return to the original route. The original route is recovered in the last frame, shown in Figure 4.2d.

The detection of midair conflicts needs to be very carefully implemented since numerical errors can make the program not detect an actual conflict, leading to catastrophic



(a) Conflict: true, Recovery: true

(b) Conflict: false, Recovery: true

(c) Conflict: false, Recovery: true

(d) Conflict: false, Recovery: false

Figure 4.2: DAIDALUS conflict detection and maneuvering guidance using DAA-display. Example extracted from [26] second part

consequences. For instance, in Figure 4.2, the best conflict resolution is to turn left. However, a numerical error could make the algorithm take the wrong branch turning to the right, provoking a mid-air collision. Specification 4.1 shows the PVS specification of a function from DAIDALUS that could be used to determine which side the turn should be done to avoid a collision. This function computes the dot product between the relative velocity and position; if it is positive, then returns 1, indicating a right turn, otherwise -1, indicating a left turn. The actual implementation of the `eps_line` function using floating-point arithmetic can present one unstable guard, leading the airship to turn in

```
eps_line(vx, vy, sx, sy: real) : real =
    IF(sx*vx+sy*vy)*(sx*vx-sy*vy) > 0 THEN 1 % right turn
    ELSE -1 % left turn
    ENDIF
```

Specification 4.1: Function from the DAIDALUS module.

the wrong direction and provoking a collision.

The core of detection logic is the mathematical definition of Well Clear violation (WCV) and its predicted occurrence interval. WCV definition is strongly related to resolution advisories (RA) from Traffic Alerting and Collision Avoidance Systems (TCAS) being its extension. WCV is based on the self-separation status between two UAVs, i.e. relative distance, time variables, and respective thresholds (minimum space-time distance to one eventual maneuver). Self-separation volume is an extension of TCAS Collision avoidance Threshold (CAT) [29]. TCAS was designed to reduce mid-air collision occurrence through a set of embedded devices in airships. Its second generation, called TCAS II, provides resolution advisories (RA) to instruct pilots to maintain or increase the relative distance between aircraft to avoid collisions. RAs are based on the τ concept, defined division of range r (relative distance norm) over closure rate (minus range change rate $\dot{r}$), as expressed by:

$$\tau = -\frac{r}{\dot{r}} \tag{4.1}$$

τ is an overapproximation of the closest point approach time (see Lemma 4.4). Both definitions only give the same result if the aircraft is in a perfect rectilinear route. Otherwise, τ is more conservative. However, this definition can produce undesirable behaviors, such as alerting a hazard when the closure rate is too low [27]. Thus, TCAS II uses another definition for τ called τ_{mod} expressed by:

$$\tau_{mod} = -\frac{r^2 - \texttt{DMOD}^2}{r\dot{r}} \tag{4.2}$$

where `DMOD` is a distance threshold.

Using a vectorized representation where $\boldsymbol{s}$ is a 2D vector with form (s_x, s_y), where $\boldsymbol{s}$ and $\boldsymbol{v}$ the relative distance and velocity vectors, and $s(t)$ denotes the relative distance vector in time t assuming constant velocity, i.e:

$$\boldsymbol{s}(t) = \boldsymbol{s} + \boldsymbol{v}t \tag{4.3}$$

the range (relative distance norm) is expressed by:

$$\begin{aligned} r(t) &= \|\boldsymbol{s}(t)\| \\ &= \sqrt{(\boldsymbol{s} + \boldsymbol{v}t) \cdot (\boldsymbol{s} + \boldsymbol{v}t)} \\ &= \sqrt{(\boldsymbol{s} \cdot \boldsymbol{s}) + 2t(\boldsymbol{s} \cdot \boldsymbol{v}) + t^2(\boldsymbol{v} \cdot \boldsymbol{v})} \end{aligned}$$

```
horizontal_tca(s:real, nzv:nzreal) : real = -(s*nzv)/sqv(nzv)
```

Specification 4.2: Formalization of t_{cpa}

deriving $r(t)$ regarding to t, one obtain the range rate:

$$\begin{aligned}\dot{r}(t) &= -\frac{1}{2||\boldsymbol{s}(t)||} \times (2t\boldsymbol{s}\dot{\boldsymbol{v}} + 2t\boldsymbol{v}\cdot\boldsymbol{v}) \\ &= \frac{\boldsymbol{s}\cdot\boldsymbol{v} + t||\boldsymbol{v}||^2}{||\boldsymbol{s}(t)||}\end{aligned}$$

Since the closest point approach occurs when the closure rate is zero, the time of closest point approach t_{cpa} is expressed as:

$$t_{cpa}(\boldsymbol{s},\boldsymbol{v}) = -\frac{\boldsymbol{s}\cdot\boldsymbol{v}}{||\boldsymbol{v}||^2} \tag{4.4}$$

The values τ and τ_{mod} can be rewritten using the previous equations as:

$$\tau = -\frac{r}{\dot{r}} = -\frac{r(0)}{\dot{r}(0)} = -\frac{||\boldsymbol{s}||^2}{\boldsymbol{s}\cdot\boldsymbol{v}} \tag{4.5}$$

$$\tau_{mod} = -\frac{r^2 - \texttt{DMOD}^2}{r\dot{r}} = -\frac{||\boldsymbol{s}||^2 - \texttt{DMOD}^2}{\boldsymbol{s}\cdot\boldsymbol{v}} \tag{4.6}$$

Another important time function is the co-altitude time t_{coa}, also called *vertical* τ, and is expressed as:

$$t_{coa} = -\frac{s_z}{v_z} \tag{4.7}$$

Formulas 4.4, 4.5, 4.6 and 4.7 are undefined under certain conditions. Thus, for completeness reasons, in the final specification they return 0 (for t_{cpa}) and -1 (for τ, τ_{mod} and t_{coa}) in undefined cases or when aircraft are not converging.

4.2 Verification under real-valued declarations

The verification of DAIDALUS properties was fully formalized in [28], assuming exact computations over real numbers. This section illustrates how real arithmetic calculations of DAIDALUS fragments were logically verified. DAIDALUS detection is based on time variables. It is defined as a function that maps the relative states (distance and velocity vectors) to a real value that is negative if aircraft are diverging ($\boldsymbol{s}\cdot\boldsymbol{v} > 0$), and negative otherwise. As already stated, some of these time variables are t_{cpa} and τ. Their specifications are shown in Specification 4.2 and 4.3, respectively.

```
tau(s:Vect2,(v:Vect2|(s*v /= 0))) : real = -sqv(s)/(s*v)
```

Specification 4.3: Formalization of τ

```
tau_tca: LEMMA FORALL (s,v:Vect2): s*v<0 IMPLIES
    tau(s,v) >= horizontal_tca(s,v)
```

Specification 4.4: Lemma `tau_tca`

Specification 4.4 shows the PVS lemma `tau_tca`, which states that τ is one overapproximation of t_{cpa} as discussed above.

Proof of lemma `tau_tca`. (Sketch) This lemma considers logical aspects and its proof uses known reals properties. Initially expanding the definitions of `tau` and `horizontal_tcpa`, one get:

$$-\frac{(\boldsymbol{s}\cdot\boldsymbol{s})}{\boldsymbol{s}\cdot\boldsymbol{v}} \geq -\frac{\boldsymbol{s}\cdot\boldsymbol{v}}{\boldsymbol{v}\cdot\boldsymbol{v}} \tag{4.8}$$

At this point, one needs to include a known reals property called `cross_mult`. It states that an equality/inequality of fractions can be replaced by its cross-multiplication once none of the denominators is zero. PVS provides a large set of proven properties/lemmas in its prelude, including this one and, as described in Section 3.3, it is possible to include an external lemma using the `lemma` rule, corresponding to the `cut` rule of the sequent calculus. Once the lemma is added, instantiated, and simplified, the consequent shows the rule to be proven:

$$(\boldsymbol{s}\cdot\boldsymbol{v})(\boldsymbol{s}\cdot\boldsymbol{v}) \leq (\boldsymbol{s}\cdot\boldsymbol{s})(\boldsymbol{v}\cdot\boldsymbol{v}) \tag{4.9}$$

This inequality is equal to the Cauchy-Schwarz inequality. Thus, to finish the proof, one needs to invoke the Cauchy-Schwarz inequality lemma present in Specification 4.5. This lemma is available on PVS NASALib, a collaborative corpus of mathematical theorems in a variety of areas such as algebra, analysis, geometry, topology, etc. □

Another important property is the symmetry of these time functions since both the owner and the intruder must have the same conflict. Let t_{var} denote any of these 3 functions: τ, τ_{mod} or t_{cpa}, then:

$$t_{var}(\boldsymbol{s},\boldsymbol{v}) = t_{var}(-\boldsymbol{s},-\boldsymbol{v}) \tag{4.10}$$

```
cauchy_schwarz : LEMMA sq(u*v) <= sqv(u)*sqv(v)
```

Specification 4.5: Cauchy-Schwarz theorem as formalized in PVS.

```
WCV(tvar:TimeVar)(s,v: Vect3): bool =
  horizontal_WCV(tvar)(s,v) AND vertical_WCV(s`z,v`z)

vertical_WCV(sz,vz:real): bool =
  abs(sz) <= ZTHR OR
  (0 <= tcoa(sz,vz) AND tcoa(sz,vz) <= TCOA)

tcoa(sz,vz:real): real =
    IF (sz*vz < 0) THEN -(sz/vz)
    ELSE -1
    ENDIF

horizontal_WCV(tvar:TimeVar)(s,v:Vect2): bool =
    sqv(s) <= sq(DTHR) OR
    (sqv(s+tcpa(s,v)*v) <= sq(DTHR) AND
     0 <= tvar(s,v) AND tvar(s,v) <= TTHR)
```

Specification 4.6: Formalizaiton of the WCV function.

Theorem 1 (Symmetry of time functions)**.** $\tau, \tau_{mod}, t_{cpa}$ are symmetric

Proof. (Sketch) After expanding, the symmetric and t_{var} definitions, introduce a hypothesis of the branch statement guard in both t_{var} instances; for example, in the t_{cpa}, introduce using the `case` PVS rule (`cut` SC rule or LEM), $\boldsymbol{v} \neq \mathbf{0}$ and $-\boldsymbol{v} \neq \mathbf{0}$, then expand the definitions and apply some arithmetic simplifications. □

However, these properties are not necessarily inherited by floating-point arithmetic. Thus, the lemma does not remain valid by simply replacing the variables and operations with their floating-point counterparts.

WCV logic extends TCAS RA for more time-variable functions. Thus it defines a violation as simultaneous horizontal and vertical violations [32]. Its formalization is shown in Specification 4.6. There, `ZTHR` is the vertical distance threshold, `TCOA` is the time threshold for co-altitude, and `tcoa` function calculates the co-altitude time assuming constant velocity. The time of co-altitude is only relevant if aircraft are converging in this way. In the `horizontal_WCV` function, `s` and `v` are relative horizontal distance and velocity, `DTHR` and `TTHR` are distance and time threshold, respectively, `tcpa` is a function that calculates the closest point approach time expressed in Specification 4.7 and `tvar` is any of previously defined time variables, such as `taumod` is expressed in Specification 4.8.

Several properties of the DAIDALUS implementation using real representation were proved. For instance, the symmetry of the `WCV` expressed as:

Theorem 2. A time variable t_{var} to be symmetric implies `WCV(`t_{var}`)` is symmetric.

```
tcpa(sx,vx,sy,vy:real) : real =
  IF (vx*vx + vy*vy /= 0) THEN
  -(sx*vx + sy*vy)/(vx*vx + vy*vy)
  ELSE 0
  ENDIF
```

Specification 4.7: Formalizaiton of the `tcpa` function.

```
taumod(sx,vx,sy,vy,DTHR:real) : real =
  IF (sx*vx)+(sy*vy) < 0
  THEN (DTHR*DTHR - (sx*sx + sy*sy))/(sx*vx + sy*vy)
  ELSE -1
  ENDIF
```

Specification 4.8: Formalizaiton of the `taumod` function.

Proof. (Sketch) `WCV_vertical` is independent from t_{var}. To prove the symmetry of `WCV_horizontal`, one need to remove the `forall` quantifier using Skolen simplification through the command `skeep`, them invoke the lemmas that state the symmetry of the square of a vector (`sqv`) and `tcpa`. Then induce the hypothesis:

```
-s + tcpa(s, v) * -v = -(s + tcpa(s, v) * v)
```

Then split the rules using the `split command` and each branch applies some propositional simplifications followed by the use of the assumption that `tvar` is symmetric. □

The importance of the previous lemma is related to the desired behavior of both owner and intruder aircraft, calculating the same violation from each's perspective.

Another behavior of the well-clear functions is the most rigid, providing the most conservative prediction.

Theorem 3. (WCV inclusion) For all relative states (distance and velocity vectors) and the same thresholds, the fact that a time variable function is an upper bound of another implies that detection of WCV using the first one is included in the second one, that is, a WCV model with the first one implies a WCV with second one:

```
WCV_inclusion : THEOREM
FORALL (tvar1,tvar2:(pre_timevar?)):
  tvar2 <= tvar1 IMPLIES
  (WCV(tvar1)(s,v) IMPLIES WCV(tvar2)(s,v))
```

`(pre_timevar?)` type indicates that t_{var1} and t_{var2} are time-variable functions in which $\boldsymbol{s} \cdot \boldsymbol{v} < 0 \wedge \boldsymbol{s} \cdot \boldsymbol{s} > \mathtt{DTHR}^2 \wedge \mathtt{sqv}(\boldsymbol{s} + t_{cpa}(\boldsymbol{s}, \boldsymbol{v}) * \boldsymbol{v}) \leq \mathtt{DTHR}^2$ implies $t_{var}(\boldsymbol{s}, \boldsymbol{v}) > 0$.

Proof. (Sketch) As shown in Specification 4.6, `WCV` is separated into vertical and horizontal. The vertical branch is trivial since it is independent of the time variable. In the second one, one expands the definitions and does propositional simplifications until one gets the sequent below.

$$t_{var1} \leq t_{var2}, 0 \leq t_{var2}(\boldsymbol{s}, \boldsymbol{v}) \leq \texttt{TTHR}, \Gamma \vdash 0 \leq t_{var1}(\boldsymbol{s}, \boldsymbol{v}) \wedge t_{var1}(\boldsymbol{s}, \boldsymbol{v}) \leq \texttt{TTHR}, \Lambda$$

where Γ includes the t_{var1} and t_{var2} type predicate `pre_timevar?` previously cited. One introduces a case where $\boldsymbol{s} \cdot \boldsymbol{v} = 0$, which is closed trivially since implies in a null t_{cpa}, but it is used in another branch where it is assumed false. At this point, one needs to expand the definition of $\leq$ for two-time variables, which claims:

```
FORALL (s, v: Vect2):
  s*v < 0 AND sqv(s) > sq(DTHR) AND sqv(s+tcpa(s,v)*v) <= sq(DTHR)
  IMPLIES tvar1(s, v) <= tvar2(s, v)
```

Then, splitting this implication, the first branch is closed using the transitivity property and the second one using the assumed types. □

As a Cololary of the previous lemma, the following properties hold for `WCV` predicate expressed in Specification 4.6. `WCV` (well clear violation) depends on the aircraft's relative states (distance and relative velocity vectors) and indicates if the aircraft remain outside of distance and time thresholds.

- `WCV(`τ`)(s,v)` $\Rightarrow$ `WCV(`t_{cpa}`)(s,v)`
- `WCV(`t_{cpa}`)(s,v)` $\Rightarrow$ `WCV(`τ`)(s,v)`

Theorem 4. (Local convexity) The aircraft is in a WCV at most one time in a time interval in a straight-line trajectory, i.e.,

```
WCV_locally_convex?(tvar) : bool =
  FORALL (B,(T|B<T),s,v: Vect2):
    NOT EXISTS(t1,t2,t3:Lookahead[B,T]): t1 <= t2 AND t2 <= t3 AND
    WCV(tvar)(s+t1*v,v) AND
    NOT WCV(tvar)(s+t2*v,v) AND
    WCV(tvar)(s+t3*v,v)
```

Proof. (Sketch) Invoke the lemma that states the correctness of the function that predicts the interval of occurrence is correctness, instantiate it and do some propositional simplifications. □

```
horizontal_WCV_taumod_interval(T,s,v): EntryExit[0,T] =
  LET a=sqv(v),
      b=2*(s*v)+TAUMOD*sqv(v),
    c=sqv(s)+TAUMOD*(s*v)-sq(DTHR) IN
  IF a = 0 AND sqv(s)<=sq(DTHR) THEN WholeInterval[0,T]
  ELSIF sqv(s)<=sq(DTHR) THEN (# entry :=0 ,exit:=min(T,Theta_D[DTHR](s,v,1))#)
  ELSIF s*v>=0 OR discr(a,b,c)<0 THEN EmptyInterval[0,T]
  ELSIF Delta[DTHR](s,v)>=0 AND root(a,b,c,-1)<=T THEN
      (#entry := max(0,root(a,b,c,-1)),exit:=min(T,Theta_D[DTHR](s,v,1))#)
  ELSE EmptyInterval[0,T]
  ENDIF
```

Specification 4.9: Function computing the interval of occurrence of a horizontal well clear violation for time variable τ_{mod}.

The function `horizontal_WCV_interval` predicts the time interval of a future violation assuming a uniform rectilinear motion. Specification 4.9 is for τ_{mod} and Specification 4.10 for t_{cpa}.

The correctness and soundness of the `horizontal_WCV_interval` of τ_{mod} are expressed in the lemma presented in Specification 4.11 and soundness in Specification 4.13.

The above predicate `horizontal_interval_correct?` is curried and uses two parameters. The argument `taumod` is a variable time function. It is specified in Specification 4.12.

```
horizontal_interval_correct?(tvar:TimeVar)(hi:HorizontalInterval) : bool =
  FORALL (T:posreal,s,v:Vect2,t:Lookahead[0,T]) :
    LET interval = hi(T,s,v),
      tentry = interval`entry,
         texit  = interval`exit IN
    horizontal_WCV(tvar)(s+t*v,v) IFF
    (tentry<=t AND t<=texit)
```

Specification 4.12: Predicate indicating the correctness of the function `horizontal_WCV_taumod_interval`, i.e, there is a well-clear violation only within the computed interval.

In the specification of this predicate, the parameter `tvar` is a time variable function, and the parameter `hi` is a function that calculates a time for violation and a function that calculates the violation interval. This predicate states that for all lookahead time `T`, relative 2D distance `s`, relative 2D velocity `v`, and time `t` in lookahead interval, there is a violation in the future position at time `t` (`s+t*v`) if and only if the is between the entry and exit violation time calculated by `hi`.

```
horizontal_WCV_tcpa_interval(T,s,v): EntryExit[0,T] =
  IF    sqv(v) = 0 AND sqv(s)<=sq(DTHR) THEN WholeInterval[0,T]
  ELSIF sqv(v)=0 THEN EmptyInterval[0,T] % detection returns false
  ELSIF sqv(s)<=sq(DTHR) THEN (# entry:=0,exit:=min(T,Theta_D[DTHR](s,v,1)) #)
  ELSIF s*v>0 THEN EmptyInterval[0,T] % detection returns false
  ELSIF sqv(s + horizontal_tca(s,v)*v)>sq(DTHR) THEN (# entry:=T,exit:=0 #)
  ELSIF Delta[DTHR](s,v)<0 AND horizontal_tca(s,v)-TCPA>T
    THEN EmptyInterval[0,T] % detection returns false
  ELSIF Delta[DTHR](s,v)<0 THEN
        (# entry:=max(0,horizontal_tca(s,v)-TCPA),
          exit:=min(T,horizontal_tca(s,v)) #)
  ELSE LET tmin = min(Theta_D[DTHR](s,v,-1),horizontal_tca(s,v)-TCPA) IN
    IF tmin > T THEN EmptyInterval[0,T] % detection returns false
    ELSE  (# entry:= max(0,tmin), exit:=min(T,Theta_D[DTHR](s,v,1)) #)
    ENDIF
  ENDIF
```

Specification 4.10: Function computing the interval of occurrence of a horizontal well clear violation for time variable t_{cpa}.

```
horizontal_WCV_taumod_interval_correct : LEMMA
  horizontal_interval_correct?(taumod)(horizontal_WCV_taumod_interval)
```

Specification 4.11: Correctness of the predicate `horizontal_WCV_taumod_interval`

Proof. (Sketch) As the WCV with time variable τ_{mod} is equivalent to the TCAS RA definition, the lemma involves the equivalence between both definitions. □

```
horizontal_WCV_taumod_interval_sound : LEMMA
  horizontal_interval_sound?(taumod)(horizontal_WCV_taumod_interval)
```

Specification 4.13: Soundness of the `horizontal_WCV_taumod_interval`

Above the predicate `horizontal_interval_sound?` is specified below, and `taumod` is a time variable function.

```
horizontal_interval_sound?(tvar:TimeVar)(hi:HorizontalInterval) : bool =
  FORALL (T:posreal,s,v:Vect2) :
    LET interval = hi(T,s,v),
      tentry = interval`entry,
      texit  = interval`exit IN
    (nonempty?(interval) IMPLIES
     FORALL (t:Lookahead[0,T]):
       ((t < tentry OR t > texit) IMPLIES NOT horizontal_WCV(tvar)(s+t*v,v)) AND
```

```
      ((t = tentry OR t = texit) IMPLIES horizontal_WCV(tvar)(s+t*v,v)))
    AND
    ((EXISTS (t:Lookahead[0,T]): horizontal_WCV(tvar)(s+t*v,v)) IMPLIES
      nonempty?(interval))
```

Specification 4.14: Predicate indicating the soundness of the function `horizontal_WCV_taumod_interval`

This predicate has two parameters: a function that calculates a time for violation and a function that calculates the violation interval. It is composed of two main clauses in conjunction. The first one claims that if the function returns a violation interval (expressed by predicate `nonempty?`), then the entry time for violation calculated is the first time that there is a violation, and the exit time is the last one, i.e., for all time in the lookahead in the interval (expressed by `Lookharead[0, T]`) the time before entry time or after exit time implies that there is no violation considering that the aircraft keep the same velocity, that is, the position is expressed by `s+t*v`, and exactly in entry and exit time, is the violation. The second clause ensures that the function returns a nonempty violation interval if there is a violation in the lookahead interval.

Proof. (Sketch) The Lemma 4.11 is proven by relating `horizontal_WCV_taumod_interval` definition with the TCASS II RA2D definition shown in Specification 4.15 which is similar to ir regardless of some minor differences in the return type (tuple instead of LoakHead interval).

```
RA2D_interval(so,vo,si,vi): [real,real] =
  LET s=so-si,v=vo-vi,aa=sqv(v),bb=2*(s*v)+TAU*sqv(v),
    cc=sqv(s)+TAU*(s*v)-sq(DMOD)
  IN (IF aa = 0 AND sqv(s)<=sq(DMOD) THEN (B,T)
      ELSIF sqv(s)<=sq(DMOD) THEN (B,root(sqv(v),2*(s*v),sqv(s)-sq(DMOD),1))
      ELSIF s*v>=0 OR discr(aa,bb,cc)<0 THEN (T+1,0)
      ELSIF Delta[DMOD](s,v)>=0 THEN (root(aa,bb,cc,-1),Theta_D[DMOD](s,v,1))
      ELSE (root(aa,bb,cc,-1),root(aa,bb,cc,1)) ENDIF)
```

Specification 4.15: TCASS II RA2D declaration in PVS formalization

To finish the proof, only propositional simplifications need to be applied. □

As mentioned, in [29], real arithmetic computations were formally verified. Properties such as local convexity are proven using real arithmetic. Nevertheless, they cannot be automatically assumed when using floating-point representations instead.

Chapter 5

Verification of actual computations on DAIDALUS

5.1 Slicing

This work aims to generate an instrumented stable-guard floating point program, that is, a program that detects potentially unstable inputs and emits a warning. The complexity of this analysis is directly correlated with the size of the program's possible control flow graph. Thus, reducing the branching of the control-flow graph has a positive impact on the performance of the analysis. Among the program simplification analysis techniques, Slicing is one of the more classical approaches and has been applied to simplify code analysis for more than four decades since its invention by Mark D. Weiser [46, 47]. In particular, the so-called *conditional slicing* technique decomposes programs into simpler parts, named slices, according to its control flow graph defined by guards and cases in its branching instructions (case-of and if-then-else instructions). Each slice assumes specific input restrictions, and its execution corresponds to a particular subset of the control-flow paths of the original program. This slicing technique was introduced by Canfora et al. and Ning et al. [7, 33, 39]. Slicing first-order PVS specifications was a crucial step in this work since it allowed us to simplify the inputs to PRECiSA and make feasible the overall analysis. The *conditional slicing* was applied manually on the real-valued declaration of the `WCV_interval` predicate (Specification 5.1) and produced six cases, determined by the relative velocity between the aircraft participating in the encounter. Each slice represented a particular kind of encounter, as described below.

- aircraft maintain both their horizontal and vertical separation unmodified;
- the vertical separation increases, but the horizontal one is kept unmodified;
- the vertical separation decreases, but the horizontal one is kept unmodified;

- aircraft alter only their horizontal separation;
- aircraft increase vertical and alter their horizontal separation;
- aircraft decrease vertical and alter their horizontal separation.

Example 5.1.1. Specification 5.1 shows the definition of `WCV_interval` predicate used to detect whether there is a violation in the lookahead time interval. The definition was extracted from the DAIDALUS specification and manually translated into a first-order declaration.

```
WCV_interval(B, T, sx, sy, sz, vx, vy, vz, TAUMOD, TCOA, DTHR, TTHR, ZTHR): bool =
LET vexit_minus_ventry = vertical_WCV_exit_minus_entry(B, T, sz, vz, TCOA, ZTHR),
    ventry = vertical_WCV_entry(B, T, sz, vz, TCOA, ZTHR),
    proj_x = sx + vx * ventry,
    proj_y = sy + vy * ventry
IN      IF vexit_minus_ventry < 0 THEN FALSE
        ELSIF vexit_minus_ventry /= 0 THEN
            horizontal_WCV_taumod_interval(vexit_minus_ventry, proj_x, proj_y,
                                           vx, vy, TAUMOD, DTHR)
        ELSIF horizontal_wcv(proj_x, proj_y, vx, vy, DTHR, TTHR) THEN
            TRUE
        ELSE FALSE
        ENDIF
```

Specification 5.1: First-order WCV interval predicate

The slices corresponding to the cases in which the aircraft maintain both their horizontal and vertical separation unmodified and the aircraft alter only their horizontal separation are presented in Specification 5.2, and Specification 5.3, respectively.

```
WCV_interval_{v_z=0∧v=0}(B, T, sx, sy, sz, nzvx, nzvy, pvz,
                                    DTHR, ZTHR: real): bool =
  abs(sz) - ZTHR <= 0 AND sq(sx, sy) - DTHR*DTHR <= 0
```

Specification 5.2: slice of `WCV_interval` predicate in which the aircraft maintain both their horizontal and vertical separation unmodified, i.e., null vertical and horizontal relative velocities.

```
WCV_interval_{v_z=0∧v≠0}(B, (T | T > B), sx, sy, sz, nzvx,
                                        (nzvy | nzvx /= 0 OR nzvy /= 0), pvz,
                                        TAUMOD, DTHR, ZTHR): bool =
    LET proj_x112 = sx + nzvx * B,
            proj_y112 = sy + nzvy * B
```

```
    IN  abs(sz) - ZTHR <= 0 AND
        horizontal_WCV_taumod_interval_non_zero_velocity(T-B, proj_x112,
                                                         proj_y112, nzvx, nzvy,
                                                         TAUMOD, DTHR)
```

Specification 5.3: slice of `WCV_interval` predicate in which the aircraft alter only their horizontal separation, i.e., null vertical relative velocity and non-null horizontal.

It is important to note that program slicing is usually applied to reduce the complexity of concrete implementations; in this work, it is applied to the specification level, aiming to overcome scalability issues on PRECiSA.

5.1.1 Equivalence theorem

Since the objective of the module is to detect *well-clear violations* in a look ahead time interval under all conditions, instead of under a specific situation, it is necessary to aggregate all slices together, providing the same functionality as the original specification. It is expressed through the equivalence theorem shown in Specification 5.4.

```
1  slicing_equivalence: THEOREM
2  FORALL(s_x, s_y, s_z, v_x, v_y, v_z, B, (T| B < T)):
3    LET s = (# x := s_x, y := s_y, z := s_z #),
4        v = (# x := v_x, y := v_y, z := v_z #) IN
5    LET wcv_interval = WCV_taumod_interval(B, T, s, v)
6    IN  (wcv_interval`entry <= wcv_interval`exit) IFF
7        IF v_z < 0 THEN
8            IF v_x /= 0 OR v_y /= 0 THEN
9                 WCV_interval_{v_z<0∧v≠0}(B, T, s, v)
10           ELSE WCV_interval_{v_z<0∧v=0}(B, T, s, v)
11           ENDIF
12       ELSIF v_z > 0 THEN
13         IF v_x /= 0 OR v_y /= 0 THEN
14              WCV_interval_{v_z>0∧v≠0}(B, T, s, v)
15         ELSE WCV_interval_{v_z<0∧v=0}(B, T, s, v)
16         ENDIF
17       ELSE
18         IF v_x /= 0 OR v_y /= 0 THEN
19              WCV_interval_{v_z=0∧v≠0}(B, T, s, v)
20         ELSE WCV_interval_zero_{v_z=0∧v=0}(B, T, s, v)
```

```
21          ENDIF
22        ENDIF
```

Specification 5.4: Slicing Equivalence Theorem

In Specification 5.4, lines 3-4 express the three-dimensional space and velocity, and lines 5-6 express there is an original WCV detection, i.e., a valid conflict interval if and only if the corresponding slice detects a violation too (lines 7-22). The latter is called the top-layer function and combines the slices by selecting the correct slice for each situation. The slice assumptions are expressed as subscripts. For instance, in lines 7-9, there is a decrement in vertical and an alteration in horizontal separation.

The Equivalence theorem is proven using each slice equivalence lemma. For instance, the slice's equivalence lemma where the vertical separation decreases, but the horizontal one is kept unmodified is shown in Specification 5.5.

```
1 WCV_interval_taumod_interval_equivalence: LEMMA
2  FORALL(sx, sy, sz, nvz, B, (T| B < T)):
3     LET s = (# x := sx, y := sy , z := sz #),
4         v = (# x := 0, y := 0, z := nvz #) IN
5     LET wcv_interval = WCV_taumod_interval(B, T, s, v)
6     IN  WCV_interval_{v_z<0∧v=0}(B, T, s, v) IFF
7         (wcv_interval`entry <= wcv_interval`exit)
```

Specification 5.5: Slice decreasing vertical separation, keeping the horizontal one unmodified equivalence lemma

Line 4 states no changes in horizontal separation, i.e., null horizontal relative velocity, and a decrease in vertical separation, i.e., negative vertical relative velocity. Thus, slice detection only occurs if and only if the original specification returns a valid conflict interval. The WCV_interval$_{v_z<0 \wedge v=0}$ is defined in Specification 5.6.

```
WCV_interval_{v_z<0∧v=0}(B, T, sx, sy, sz, zvx, zvy, nvz): bool =
    LET centry1 = coalt_entry_{v_z<0}(sz, nvz, TCOA, ZTHR),
        cexit1 = coalt_exit_{v_z<0}(sz, nvz, ZTHR)
    IN T - centry1 >= 0 AND cexit1 - B >= 0 AND
       sq(sx, sy) - DTHR*DTHR <= 0
```

Specification 5.6: *WCV* associated interval for slice decreasing vertical separation and keeping horizontal one unmodified

Above, `coalt_entry` and `coalt_exit` calculate the time to co-altitude between the owner and intruder; `B` and `T` are the endpoints of the look-ahead time interval; `TCOA`, `DTHR`, `ZTHR`

are the time and distances threshold constants; and `sq` is a function that calculates the squared norm of a vector.

5.2 Code extraction

As mentioned, the DAIDALUS specification was verified assuming real-valuated operations. Still, its floating-point implementation could present different behaviors since round-off errors in guards of conditionals could lead to choosing a different branch, leading the program to take a different control flow than its ideal verified version. In this manner, the program could either not detect a safety-critical situation or detect when there is no one. Usually, estimating round-off errors is a tough task that requires floating-point expertise. Thus, automatically generating an instrumented implementation that emits a warning when an unstable guard could occur following the methodology given in Chapter 3 has significant value.

5.2.1 Processing slices

As described in Section 3.1, the program transformation implemented in PRECiSA replaces all real arithmetic operations from a real-valuated program by their floating-point counterpart, then replaces each guard statement with a parametrized one that is more restrictive. This new parameter takes into account the possible round-off error. PRECiSA also calculates such a round-off error for a desired floating-point precision. The original guard's replacement introduces some inputs in which the transformed program is not defined, while the original program returns a numerical value. In this case, PRECiSA adds a warning indicating a possible divergence between the evaluation of the original guard using reals and floating-point arithmetic. PRECiSA provides a sound over-approximation of the round-off bound, which means that false warnings may be emitted. However, it also guarantees that all actual instabilities raise a warning.

The transformation of the t_{cpa} function is depicted in 3.2. At the same time, the actual output of PRECiSA is shown below with some syntactic simplifications.

```
1 /*@
2 real tcpa(real s_x, real v_x, real s_y, real v_y) =
3 v_x^2 + v_y^2 > 0 ? -(s_x * v_x + s_y * v_y) / (v_x^2 + v_y^2) : 0;
4
5 double tcpa(double s_x, double v_x, double s_y, double v_y) =
6 v_x^2 + v_y^2 > 0 ? -(s_x * v_x + s_y * v_y)/(v_x^2 + v_y^2) : 0;
7
```

```
8  predicate tcpa_stable_paths(real s_x, real v_x, real s_y, real v_y,
9                              double [s_x], double [v_x], double [s_y], double [v_y]) =
10  v_x^2 + v_y^2 > 0 ∧ [v_x^2 + v_y^2 > 0]
11 ∨ ¬ (v_x^2 + v_y^2 > 0) ∧ ¬ [v_x^2 + v_y^2 > 0];
12
13 requires: [0 <= ε] ∧ finite?([ε]) ;
14 ensures: (result ≠ ω ⇒ result = tcpa_fp
15 ensures: result ≠ ω ⇒ (|([v_x^2 + v_y^2])-(v_x^2 + v_y^2)| ≤ ε => tcpa_stable_paths)
16 */
17 double* tcpa_fp(double [s_x], double [v_x], double [s_y], double [v_y], double ε){
18     if([v_x^2 + v_y^2 > ε]){
19         return [-(s_x * v_x + s_y * v_y)/(v_x^2 + v_y^2)];
20     }
21     else{
22         if([v_x^2 + v_y^2 ≤ -ε])
23             return [0];
24         else
25             return ω;
26     }
27 }
```

Specification 5.7: C function and annotations generated by PRECiSA for t_{cpa}. Some syntactic simplifications were applied for ease of reading.

The Specification 5.7 shows the C code and its ACSL annotations generated by PRECiSA for the t_{cpa} function. Lines 1-16 contain the ACSL annotations expressing the contracts explained below, and lines 17-27 contain the [tcpa] function that is equivalent to those presented in the Specification 3.2. The type `double*` expresses a type that is the union of type `double` and the warning ω. The specification of the function over reals and its floating-point counterpart are given in lines 2-3 and 5-6. The precondition in line 13 restricts the argument ϵ to be positive and finite, i.e., it cannot be infinity nor *NaN*. The postcondition (in line 14) states that if the `tcpa_fp` function does not output a warning, then it equals the non-instrumented floating-point version of t_{cpa}. This post-condition is called *structural behavior*. The floating-point counterparts replace the real number operations with their floating-point counterparts. The postcondition in line 15 is called *stable-path behavior* and expresses that if the result is not a warning and the ϵ is the guard expression round-off sound error approximation, then the predicate `tcpa_stable_paths`

holds. The latter predicate is defined in lines 8-11 and expresses that the evaluation using both the real number and the floating-point arithmetic is the same. This means that there are no unstable conditions. Here, it is important to remark that the variable ϵ is expected to be a sound-over approximation of guard expression round-off error since when the floating-point evaluation of the guard expression is in the interval $[-\epsilon, +\epsilon)$ the function returns a warning expected when an unstable guard may occur.

Given ranges for the arguments, PRECiSA also computes numeric round-off error over-approximations. For instance, if the velocities remain in the interval $(1, 1000)$, the bound for the round-off error of the expression $v_x^2 + v_y$ computed is $\epsilon = 4.602043e-10$. PRECiSA also generates formal certificates claiming these bounds that can be mechanically verified using PVS. These certificates are indeed theorems; for instance, the Theorem 5 refers to the above case of t_{cpa}.

Theorem 5 (Error bound for the guard in t_{cpa})**.** For all real values v_x, v_y in the interval $(0, 1000)$ and its rounding to floating-point represetation $\boxed{v_x}$, $\boxed{v_y}$:

$$\left| \boxed{v_x^2 + v_y^2} - (v_x^2 + v_y^2) \right| \leq 4.602043e-10 \tag{5.1}$$

PRECiSA also computes bounds for the round-off errors in the output of function `tcpa_fp` as the maximum between the error bounds of all branches which do not return a warning as expressed in Equation 3.10.

```
1  /*@
2  ensures (∀ real sx, sy, vx, vy;
3       1 < vx < 1000 ∧ 1 < vy < 1000 ∧ 1 < sx < 1000 ∧ 1 < sy < 1000∧
4       |[sx] − sx| ≤ ulp(sx)/2 ∧ |[sy] − sy| ≤ ulp(sy)/2 ∧
5       |[vx] − vx| ≤ ulp(sx)/2 ∧ |[vy] − vy| ≤ ulp(vy)/2
6       ⇒ |(result - tcpa(sx, vx, sy, vy)| ≤ 9.106570e − 07)| ;
7  */
8  double* tcpa_num(double [sx], double [vx], double [sy], double [vy]) {
9    return tcpa_fp (sx, vx, sy, vy, 4.602043e-10);
10 }
```

Specification 5.8: Concrete C function and annotations generated by PRECiSA for t_{cpa} . Some syntactic simplifications were applied for ease of reading.

Specification 5.8 shows the concrete version of `tcpa_fp` assuming the inputs remain in the provided input range. In fact, `tcpa_num` implementation (lines 8-10) is just `tcpa_fp` with the concrete value $\epsilon = 4.602043e-10$ calculated by PRECiSA. The postcondition

(lines 2-7) states that if the inputs remain in the input range (line 3) and the relation between the real input and floats is the same as the Theorem 5 premise (lines 4-5), then the difference between the concrete C-function and its real-valued specification is at most $9.106570e-07$ (line 9).

The previous case is an excellent example of how the code transformation and the contracts are for more complex non-boolean functions. However, predicates are a particular case that deserves a separate analysis. The code extraction process applied in predicates, such as `WCV_interval`, produces two abstract functions and their corresponding concrete implementation. It is similar to the code extraction of two distinct functions that return numeric values previously shown. One of those pairs (abstract and concrete implementation), called beta plus, represents a more restrictive implementation of the original predicate for the positive (true) considering the round-off error. The other, called beta minus, is the negative case.

```
1  /*@
2  predicate WCV_interval_{v_z>0∧v=0}(real b,t,s_x,s_y,s_z,v_x,v_y,v_z) = ....
3  predicate WCV_interval_{v_z>0∧v=0}_fp(double [b,t,s_x,s_y,s_z,v_x,v_y,v_z]) = ....;
4
5  ensures: ∀ real b,t,s_x,s_y,s_z,v_x,v_y,v_z;
6      \result ≠ ω∧ \result
7      => WCV_interval_{v_z>0∧v=0}(b,t,s_x,s_y,s_z,v_x,v_y,v_z) ∧
8         WCV_interval_{v_z>0∧v=0}_fp([b,t,s_x,s_y,s_z,v_x,v_y,v_z])
9  */
10 bool* WCV_interval_{v_z>0∧v=0}_plus(double [b,t,s_x,s_y,s_z,v_x,v_y,v_z,e_1,e_2,e_3]){...}
11
12 /*@
13
14 ensures: ∀ real b,t,s_x,s_y,s_z,v_x,v_y,v_z;
15     \result ≠ ω∧ \result
16     => ¬ WCV_interval_{v_z>0∧v=0}(b,t,s_x,s_y,s_z,v_x,v_y,v_z) ∧
17        ¬ WCV_interval_{v_z>0∧v=0}_fp([b,t,s_x,s_y,s_z,v_x,v_y,v_z])
18 */
19 bool* WCV_interval_{v_z>0∧v=0}_mns(double [b,t,s_x,s_y,s_z,v_x,v_y,v_z,e_1,e_2,e_3]){...}
```

Specification 5.9: Piece of the C function and annotations generated by PRECiSA for slice increasing vertical and keeping horizontal separation for `WCV_interval` predicate. Some syntactic simplifications were applied for ease of reading.

The Specification 5.9 depicts PRECiSA output for the slice in which aircraft increase vertical and maintain horizontal separation for predicate `WCV_interval`, called

`WCV_interval`$_{v_z>0\wedge v=0}$. For simplicity, the actual C-code of the functions and predicate definitions are omitted. The predicate `WCV_interval`$_{v_z>0\wedge v=0}$ (line 2) is the original sliced predicate evaluated over reals, and `WCV_interval`$_{v_z>0\wedge v=0}$`_fp` (line 3) is its floating-point non-instrumented counterpart. Notice that there are no error variables as arguments. The function in line 10 (`WCV_interval`$_{v_z>0\wedge v=0}$`_plus`) is the instrumented implementation to the beta plus that is omitted. The type `bool`*, similarly to `double`*, is the datatype that represents the disjoint union between the booleans and the warning ω. The contract in lines 5-8 claims that if the result of the instrumented plus version is not a warning and holds, then the real predicate and its floating-point counterpart also hold. Similarly, the floating point to the negation of the `WCV_interval`$_{v_z>0\wedge v=0}$ predicate is shown in line 19. The contract to the beta minus is slightly different and states that if the beta minus implementation holds, then the real predicate does not hold as well as its floating-point counterpart. PRECiSA also produces concrete versions for these functions. The difference between the concrete function for predicates and functions that return numeric values is that no round-off error must obviously be estimated for the result. Thus, the contract states that if the function holds, its real version also does.

5.2.2 Top-layer function

As mentioned above, the transformation is applied to each slice, generating instrumented C code that warns whenever the program presents an unstable guard. However, it is necessary to recover the semantics of the original code, i.e., a code with the same functionality instead of several snippets.

Therefore, an addition of a top-layer function is required. This layer is responsible for selecting the suitable slice for each situation, and it is annotated with contracts that relate full-definition and joined slices as depicted in Specification 5.4.

```
1  /*@
2  predicate wcv_in_range(real b, t, s_x, s_y, s_z, v_x, v_y, v_z) =
3    // WCV?((b, t), (vx, vy, vz), (sx, sy, sz)) previously defined
4
5  requires: \is_finite(e_0) ∧ e_0 > 0 ∧ ... ∧\is_finite(e_n) ∧ e_n > 0;
6  ensures: ∀ real b, t, s_x, s_y, s_z, v_x, v_y, v_z;
7    |δ − v_z ∗ δ_tcoa − δ − v_z ∗ δ_tcoa| < e_0 ∧
8    |(t − coalt_t_asc_fp(s_z, v_z)) − (t − coalt_t_asc_fp(s_z, v_z)| < e_1 ∧
9    ...
10   \result ≠ ω
```

```
11    ⇒ (\result ⟺ wcv_in_range(b, t, s_x, s_y, s_z, v_x, v_y, v_z))
12  */
13  bool* WCV_interval(double b, t, s_x, s_y, s_z, v_x, v_y, v_z, e_1, e_2, e_3...){
14    bool* res;
15    if (v_z > 0){ // increasing vertical separation
16      if (v_x == 0.0 ∧ v_y == 0.0){ // maintaining horizontal separation
17        res = WCV_int_{vz>0∧v=0}_plus(b, t, s_x, s_y, s_z, v_x, v_y, v_z, e_1, e_2, e_3);
18        if (res == ω ∨ res) return res;
19        res = WCV_int_{vz>0∧v=0}_minus(b, t, s_x, s_y, s_z, v_x, v_y, v_z, e_1, e_2, e_3);
20        if (res == ω) return ω;
21        if (res) return false;
22        return ω;
23      } else{  // altering horizontal separation
24        res = WCV_int_{vz>0∧v≠0}_plus(b, t, s_x, s_y, s_z, v_x, v_y, v_z, e_1, e_2, e_3);
25        if (res == ω ∨ res) return res;
26        res = WCV_int_{vz>0∧v≠0}_minus(b, t, s_x, s_y, s_z, v_x, v_y, v_z, e_1, e_2, e_3);
27        if (res == ω) return ω;
28        if (res) return false;
29        return ω;
30      }
31    } else if (v_z < 0){ // decreasing vertical separation
32      ...
33    } else { // maintaining vertical separation
34     ...
35    }
36  }
```

Specification 5.10: Top-layer function draft

Specification 5.4 shows a fragment of the top-layer function. Line 5 precondition states that every error variable e must be strictly positive and finite. The postcondition in Lines 6-11 states that for all real counterparts, each error variable denotes an error bound of the conditionals present in the whole program. This implies the equivalence between the top-layer function and the original not-sliced specification expressed by the predicate `wcv_in_range`. The top-layer function is depicted in lines 13-36. The slice selection is made through nested conditionals expressions, lines 15-30 contain the scope of increasing vertical separation cases, whereas, in lines 16-22, the keeping horizontal distance and lines 23-29 changing, both for increasing vertical distance cases. Inside each slice, the top layer

checks the stronger positive version of the slice predicate (lines 17-18), and if it returns either true or a warning, the top layer returns it. Otherwise, the top layer checks the negative stronger version of the slice (lines 19-21). If it is either true or a warning, the top layer will output it (inverting the logic value). Otherwise, it returns a warning. The rest of the cases are handled in a similar way.

One important point to highlight is the absence of floating-point operations inside these added guards, i.e., the values used by the top-layer function guards are assumed to be coming directly from sensors. So, the round-off errors are just representational errors that do not affect the sign.

One can see the similarities between this annotated code and the Specification 5.5. The proof that ensures the contracts prevalence uses the Specification 5.5, jointly with contracts of called functions.

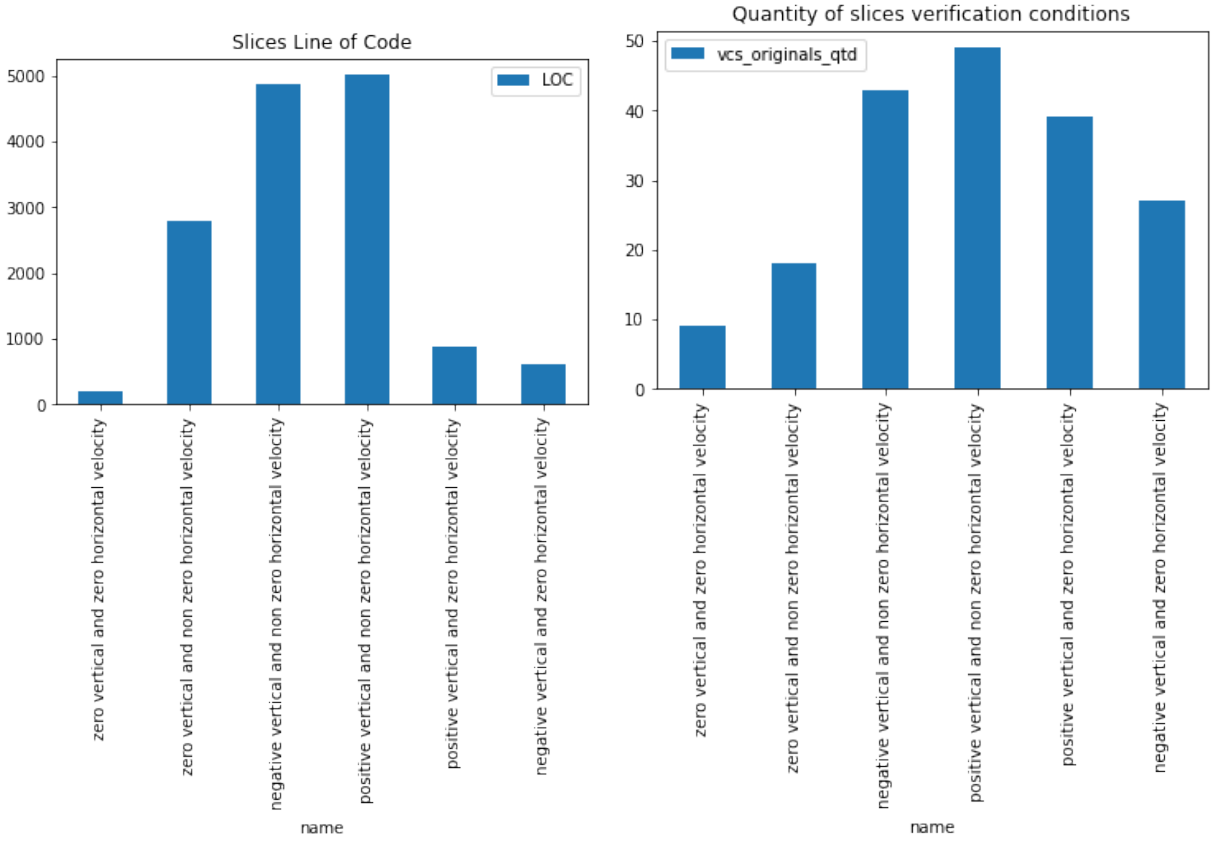
Currently, both the slicing processes and the addition of the top-layer function are implemented manually. However, these tasks can be done automatically with user hints about the slicing conditions.

Figure 5.1a[1] shows the number of lines of C-code (LOC) generated for each slice. Such metrics are correlated with algorithm complexity. It is possible to see that the more complex slices are those in which aircraft are altering horizontal separation (increasing, decreasing, and maintaining vertical separation in order). However, the number of verification conditions per slice, presented in Figure 5.1b, demonstrates that it does not follow the same distribution. Still, the two slices with more verification conditions are those in which aircraft alter both horizontal and vertical separation as well in the 5.1a; It is followed by those in which the aircraft are altering only the vertical separation, then comes the slice that alters only horizontal distance, followed by the slice that maintains the separation. It indicates that there is no direct relation between LOC and the number of verification conditions and altering the vertical separation is more determinant in the number of VCs than altering the horizontal.

5.3 Verification of the floating-point implementation of DAIDALUS

Once the annotated instrumented stable-guard C code is generated for each slice, Frama-C is used to process the code and the ACSL annotations with the plugin `WP` (weakest pre-condition). For each function, it generates verification conditions (`VCs`) ensuring the validity of the program contracts in the PVS language (Figure 3.1). These `VCs` could be

[1]These statistics are from a preliminary version of the work and will be updated in the document's final version.



(a) Number of lines of code in each slice

(b) Number of verification conditions generated for each slice

Figure 5.1: Slicing statistics of a preliminary version

discharged using *ad-hoc* strategies generated, since the proofs are heavily driven by the syntactic structure of the program.

Frama-C generates three `VC`s for the function t_{cpa}. The first two are related to the contracts annotated in lines 14-15 on Specification 5.7 and the last one in lines 2-6 of Specification 5.8. The Specification 5.11 shows the **stable-path behavior** `VC` (theorem) in one intermediary language between the PVS and mathematical languages. It expresses that if the pre-condition holds (lines 3), the computation gives a result without producing a warning (line 3), the highlighted post-condition premise to the stable-path behavior holds (line 4), and, in addition, if types are preserved (lines 5), then the stable-paths condition holds (line 9). Lines 6-8 express whether the result of the computation in double floating-point precision equals the desired result. Input parameter **result** is the result of the function implemented in C. The type of **result** contains its value and a boolean flag of its validity, and its type could be defined in a disjoint union between double domain and warning. Recall that boxed expressions are floating-point operations, and cross-operations between boxed and not boxed expressions have an underlying conversion from floats to

reals.

```
1  tcpa_fp_stable_paths_ensures: THEOREM
2    FORALL (v_x, v_y : real, [ε, s_x, s_y, v_x, v_y] : double, result : double*):
3    ([0 ≤ ε]) => result ≠ ω =>
4    |[v_x * v_x + v_y * v_y] − (v_x^2 + v_y^2)| ≤ ε =>
5    (finite?([ε]) =>
6    (IF [ε < v_x^2 + v_y^2]
7      THEN [result = −(s_x * v_x + s_y * v_y)/(v_x^2 + v_y^2)]
8      ELSE ([result = 0] AND [v_x^2 + v_y^2 ≤ −ε] ENDIF) =>
9    p_tcpa_stable_paths(v_x, v_y, [v_x, v_y]) )
```

Specification 5.11: Stable-path behavior VC for t_{cpa} function.

Simplifying, and omitting typing conditions, the VC theorem in Specification 5.11 might be abbreviated as the formula below.

$$0 \le \epsilon \to \mathbf{result} \neq \omega \to |\boxed{v_x^2 + v_y^2} - (v_x^2 + v_y^2)| \le \epsilon \to \left(\begin{array}{l} \mathit{if}\ \boxed{\epsilon < v_x^2 + v_y^2}\ \mathit{then}\ \boxed{\mathbf{result} = \dfrac{-(s_x * v_x + s_y * v_y)}{v_x^2 + v_y^2}} \\ \mathit{else}\ \boxed{v_x^2 + v_y^2 \le -\epsilon} \wedge \boxed{\mathbf{result} = 0} \end{array} \right) \to \mathtt{p\_tcpa\_stable\_paths}(v_x, v_y, \boxed{v_x, v_y}) \tag{5.2}$$

To prove the **stable-paths behavior** VC in Specification 5.11, one uses the definitions available in the NASALib axiomatic floating-point formalizations and takes advantage of the premisses that state that the error variable ϵ as the absolute bound of guard expression round-off error; i.e., $|(\boxed{v_x^2 + v_y^2}) - (v_x^2 + v_y^2)| < \epsilon$. Thus, if the more restrict guard β_+ $(\boxed{v_x^2 + v_y^2 > \epsilon})$ is satisfied, then necessarily, the original real-expressed $({v_x}^2 + {v_y}^2 > 0)$ also is. The same happens for β_- $(\boxed{v_x^2 + v_y^2 < -\epsilon})$. That is:

$$\begin{array}{r} (\boxed{v_x^2 + v_y^2 > \epsilon} \Rightarrow ({v_x}^2 + {v_y}^2 > 0) \wedge \boxed{v_x^2 + v_y^2 > 0} \wedge \\ (\boxed{v_x^2 + v_y^2 \le -\epsilon} \Rightarrow \neg({v_x}^2 + {v_y}^2 > 0) \wedge \boxed{\neg v_x^2 + v_y^2 > 0}) \end{array} \tag{5.3}$$

The **structural behavior** VC in Specification 5.12 comes from line 14 of Specification 5.7. It claims that if the precondition holds (line 3), the result is not a warning (highlighted in line 3), and types are preserved (lines 4), then the result of the instrumented floating-point C implementation is the same as the result of the non-instrumented floating-point

implementation (line 8). Lines 5-8 are similar to the previous VC, expressing the desired result of the computed floating-point implementation.

```
1 tcpa_fp_structural_ensures: THEOREM
2 FORALL (ϵ, s_x, s_y, v_x, v_y : double, result : double*):
3 (0 ≤ ϵ) => result ≠ ω =>
4 (finite?(ϵ)) =>
5 (IF ϵ < v_x^2 + v_y^2
6   THEN result = −(s_x * v_x + s_y * v_y)/(v_x^2 + v_y^2)
7   ELSE (result = 0 AND v_x^2 + v_y^2 ≤ −ϵ ENDIF) =>
8 result = l_tcpa_fp(s_x, s_y, v_x, v_y))
```

Specification 5.12: Structural behavior VC for t_{cpa} function.

The proof of the **structural behavior** VC, in Specification 5.12, is based on precondition clams that once error variable ϵ is strictly positive. Thus, the new guard β_+ is stronger than the original, also satisfying the floating-point non-instrumented guard.

$$\begin{aligned}(v_x^2 + v_y^2 > \epsilon \Rightarrow ({v_x}^2 + {v_y}^2 > 0) \wedge v_x^2 + v_y^2 > 0 \wedge \\ (v_x^2 + v_y^2 \leq -\epsilon \Rightarrow \neg({v_x}^2 + {v_y}^2 > 0) \wedge \neg v_x^2 + v_y^2 > 0)\end{aligned} \tag{5.4}$$

The **numerical ensures** VC shown in Specification 5.13 is related to lines 2-6 of Specification 5.8. It claims that if the result is not a warning (line 3), the input variables remain in the input range (line 4). The difference between the real variable in its floating-point representation is at most half of ulp (lines 5-6), the structure behavior holds (line 7), stable path behavior holds for all v_x and v_y where the difference of the expression compared in the guard using real arithmetic and floating point is at most the explicit value (lines 8-11). Then the difference between the instrumented floating-point implementation and the real specification is, at most, the value calculated by PRECiSA and presented in line 13.

```
1 tcpa_num_ensures: THEOREM
2 FORALL (s_x, v_x, s_y, v_y:real, ϵ, s_x, s_y, v_x, v_y :double, result:double*):
3     result ≠ ω =>
4     1 < v_x < 1000 => 1 < v_y < 1000 => 1 < s_x < 1000 => 1 < s_y < 1000 =>
5     |s_x − s_x| ≤ ulp(s_x)/2 => |s_y − s_y| ≤ ulp(s_y)/2 =>
6     |v_x − v_x| ≤ ulp(s_x)/2 => |v_y − v_y| ≤ ulp(v_y)/2
```

```
7       result = l_tcpa_fp(s_x, s_y, v_x, v_y) =>
8        (FORALL (v_x, v_y:real):
9           | v_x * v_x + v_y * v_y - (v_x^2 + v_y^2) | <=
10          (8901646138474497 / 19342813113834066795298816)) =>
11              p_tcpa_stable_paths(v_x, v_y, v_x, v_y)) =>
12      |result - tcpa(s_x, v_x, s_y, v_y)| <=
13      (4300455909721841 / 4722366482869645213696
```

Specification 5.13: Numerical ensures VC for t_{cpa} function.

The proof of **numerical ensures** VC relies on the PRECiSA's certificates that are automatically discharged and relate the original real specification with its not-instrument floating-point counterpart assuming stable-guards (only replacing the real operations by their finite precision versions, and constants and values by their float representations). The Specification 5.14 shows one of these certificates expressing this implementation's error bound (lines 8-9). Since the representational error between the real variables and their float counterparts is at most half of `ulp` (lines 3-4), there is no division by zero (line 5). The input variables are in the input range (line 6), and assume stable-paths (line 8). Worth remarking that this error bond is the maximum between the round-off errors of the branches as expressed in Equation 3.10.

```
1  tcpa_fp_c_0 : LEMMA
2  FORALL(s_x, v_x, s_y, v_y: real, ε, s_x, s_y, v_x, v_y : double):
3       |s_x - s_x| <= ulp(s_x)/2 ∧ |s_y - s_y| <= ulp(s_y)/2 ∧
4       |v_x - v_x| <= ulp(s_x)/2 ∧ |v_y - v_y| <= ulp(v_y)/2 ∧
5        v_x^2 + v_y^2 ≠ 0 ∧ v_x^2 + v_y^2 ≠ 0 ∧
6        s_x ∈ [1,1000] ∧ v_x ∈ [1,1000] ∧ s_y ∈ [1,1000] ∧ v_y ∈ [1,1000]) =>
7          p_tcpa_stable_paths(v_x, v_y, v_x, v_y) =>
8             |tcpa_fp(s_x, v_x, s_y, v_y) - tcpa(s_x, v_x, s_y, v_y)| <=
9              4300455909721841 / 4722366482869645213696
```

Specification 5.14: PRECiSA tcpa numerical certificate

It is important to highlight there is nothing about the symbolic error variable in Specification 5.14, just the error between the real implementation and its floating-point counterpart, assuming no unstable guards.

Every function that returns numeric values generates at least structural and numeric VCs to be proven. In cases of predicates, it generates twice as many VCs due to the β_+ and β_- versions. Thus, creating *ah-hoc* strategies based on the proof structure to prove them is an interesting approach.

Chapter 6

Related work

Several approaches are available to analyze the numerical properties of floating-point software and improve its safeness and quality [4, 5, 3, 18, 25, 44, 20, 38]. Nevertheless, most of them can be differentiated from the present work in at least one of the following aspects:

- Handling of unstable guards;
- Instrumentation of the final code to provide a warning when an unstable test is detected;
- Generation of a proof certificate able to be verified using an external proof assistant;
- Requiring hints from the user;
- Requiring floating-point specialist knowledge;
- Level of automation.

The current work uses the toolchain proposed in [45] integrating the formal methods tools PRECiSA, Frama-C and PVS. Frama-C was already used jointly with Gappa to analyze numerical properties of C source code [12]. However, it was only applied to straight-line code and required additional annotations and expert-provided hints.

The Coq prover also has been integrated with Gappa to check verification conditions [4, 5, 3, 25] using a floating-point formalization developed in [6]. Nevertheless, this approach relies on some interaction from the user to complete the proofs.

Besides PRECiSA and Frama-C, other tools have been used to analyze floating-point round-off errors. These tools can be segregated into two main groups: precision allocation and optimization [42]. Precision allocation tools select the lowest floating-point encoding length to reach target accuracy improving the program performance as Rosa [11], Precimonius [38] and FPTuner [8]. Rosa can receive a real-valuated implementation

and provide a floating-point implementation using a compilation algorithm. Rosa even deals with unstable guards but does not instrument the code to provide warnings when they occur. Program optimization tools rewrite floating-point arithmetic expressions by equivalent ones with a lower accumulated round-off error. This category include tools as CoHD [41], Herbie [36], AutoRNP [48] and Salsa [10].

Fluctuat [19] and Astrée [9] are two commercial analyzers for C based on abstract interpretation. Fluctuat correctly estimates the rounding error of a program and detects possible unstable guards, as reported by Goubault and Putot [20], but it does not provide any warning in this situation. Astrée is a tool that detects the presence of runtime exceptions such as floating-point overflows, computations returning not-a-number values, and divisions by zero.

A previous version of the approach applied in this work was presented in [42] and was used to generate code from a point-in-polygon algorithm. Later, the integrated toolchain used in this work was presented and applied to obtain a small fragment of DAIDALUSverified implementation [43].

In [24], the authors face the limitation of currently available approaches bound to the round-off error for more complex code fragments, such as N-body collision simulation, which contains a loop. The authors propose the identification of numerical kernels (most complex computations) and a two-step phase approach. The first phase uses abstract interpretation to infer the sound input ranges of the kernel pre-conditions, and the second phase uses an SMT-solver. Such a technique, as the approach in the current work, reduces the analysis of a more complex code in small pieces using pre-conditions; however, the analysis focuses on some computation inside a for-loop. In contrast, our approach handles depth function calls with branches.

The Floating-point Error Analyzer FErA [16] applies a methodology that provides an over-approximation of round-off error bound using a branch-and-bond global optimizer and an under-approximation. The main limitation of this tool is given by the operations it supports, including only the basic arithmetic functions.

Chapter 7

Conclusions and future work

The present work generated a verified instrumented floating-point implementation of a core module of DAIDALUS using a formal approach. This implementation emits a warning whenever an unstable guard may occur. A crucial step in the approach applied in this work was to slice the original declarations of the Well Clear module into an equivalent collection of simpler conceptual units. The discrimination in slices enables the application of the code generation feature of PRECiSA on each of these slices, generating a C-program annotated with ACSL contracts ensuring its correct behavior. Frama-C analyses the generated code for each slice and outputs verification conditions in PVS language that is used to prove them. The VCs ensure that all unstable guards are detected, the bound of round-off error, and there is no overflow.

In the process, we discovered several issues and opportunities for improvement in the tool; all these were opportunely communicated to the PRECiSA development team. The analysis of the slices also required modifying PRECiSA to add support for operators and modalities that were not supported yet.

The availability of a new floating-point formalization from NASALib was essential in improving the efficiency of the analysis. In particular, it significantly reduced the time spent by PVS in the type checking of the verification conditions output by Frama-C. Nevertheless, this change impacted the existing proof strategies, which are no longer usable. One of the main tasks that remain to be done is to fix those strategies to allow automatic verification of the verification conditions.

Regarding threats to validity, our approach assumes and builds upon the correctness of all the tools participating in the analysis process. However, the integration of these tools introduced instances of cross-validation. Each tool validates the results of the previous steps at least partially. For instance, if PRECiSA generates an ill-formed contract, Frama-C would detect it. Furthermore, if PRECiSA generates annotated code syntactically accepted by Frama-C but not compliant with its contract, then the VCs expressed as

PVS theorems cannot be proved. Consequently, such integration provides a higher level of confidence than the one provided by each individual tool.

In future work, the goal is to improve the automation level of the approach applied in this work. In principle, it is not viable to fully automate the slicing process. However, once the slicing criteria are defined, the definition of each slice is basically mechanical. Thus, developing a tool that provides slices of declarations and states corresponding equivalence theorems could be doable. The author also intends to improve the automation degree of the proofs, maintaining and developing more proof strategies. Another line of future work is applying the technique to other case studies, for instance, other modules of the DAIDALUS library.

Bibliography

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

[2] Patrick Baudin, François Bobot, Loïc Correnson, Zaynah Dargaye, and Allan Blanchard. *WP Plug-in Manual*, 2023. 16, 17, 18

[3] 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*, 50(4):423–456, 2013. 52

[4] S. Boldo and J. C. Filliâtre. Formal verification of floating-point programs. In *18th IEEE Symposium on Computer Arithmetic, ARITH*, pages 187–194. IEEE Computer Society, 2007. 52

[5] S. Boldo and C. Marché. Formal verification of numerical programs: From C annotated programs to mechanical proofs. *Mathematics in Computer Science*, 5(4):377–393, 2011. 52

[6] S. Boldo and G. Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *20th IEEE Symposium on Computer Arithmetic, ARITH*, pages 243–252. IEEE Computer Society, 2011. 52

[7] Gerardo Canfora, Aniello Cimitile, Andrea De Lucia, and Giuseppe A. Di Lucca. Software salvaging based on conditions. In *Proceedings of the International Conference on Software Maintenance, ICSM*, pages 424–433. IEEE Computer Society, 1994. 37

[8] W. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić. Rigorous floating-point mixed-precision tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*, pages 300–315. ACM, 2017. 52

[9] 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)*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005. 53

[10] N. Damouche and M. Martel. Salsa: An Automatic Tool to Improve the Numerical Accuracy of Programs. *6th Workshop on Automated Formal Methods, AFM*, 2017. 53

[11] E. Darulova and V. Kuncak. Sound compilation of reals. In *Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 235–248. ACM, 2014. 52

[12] F. de Dinechin, C. Lauter, and G. Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Trans. on Computers*, 60(2):242–253, 2011. 52

[13] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, aug 1975. 17

[14] A. Dutle, M. Moscato, L. Titolo, and C. Muñoz. A formal analysis of the compact position reporting algorithm. *9th Working Conference on Verified Software: Theories, Tools, and Experiments, VSTTE 2017, Revised Selected Papers*, 10712:19–34, 2017. 8

[15] FAA. *Code of Federal Regulations Title 14, Part 91 Aeronautics and Space (Last amended 6/15/2023)*. U.S. Federal Aviation Administration, 2004. 1

[16] Rémy Garcia, Claude Michel, and Michel Rueher. Rigorous enclosure of round-off errors in floating-point computations. In *Software Verification: 12th International Conference, VSTTE, and 13th International Workshop NSV, Revised Selected Papers*, page 196–212, Berlin, Heidelberg, 2020. Springer-Verlag. 53

[17] United States of America General Accounting Office. Patriot missile defense: Software problem led to system failure at dhahran, saudi arabi. Technical Report GAO/IMTEC-92-26, U.S. General Accounting Office, Washington, D.C. 20548, February 1992. 8

[18] 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 Formal Methods Symposium NFM*, volume 7871 of *LNCS*, pages 441–446. Springer, 2013. 52

[19] E. Goubault and S. Putot. Static analysis of numerical algorithms. In *Proc. of the 13th International Symposium on Static Analysis, SAS*, volume 4134 of *LNCS*, pages 18–34. Springer, 2006. 53

[20] E. Goubault and S. Putot. Robustness analysis of finite precision implementations. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems, APLAS*, volume 8301 of *LNCS*, pages 50–57. Springer, 2013. 52, 53

[21] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969. 17

[22] IEEE. IEEE standard for binary floating-point arithmetic. Technical report, Institute of Electrical and Electronics Engineers, 2008. 5

[23] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A Software Analysis Perspective. *Form. Asp. of Comput.*, 27(3):573–609, 2015. 2, 16

[24] Debasmita Lohar, Clothilde Jeangoudoux, Joshua Sobel, Eva Darulova, and Maria Christakis. A two-phase approach for conditional floating-point verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 43–63, Cham, 2021. Springer International Publishing. 53

[25] C. Marché. Verification of the functional behavior of a floating-point program: An industrial case study. *Science of Computer Programming*, 96:279–296, 2014. 52

[26] Paolo Masci and César Muñoz. A graphical toolkit for the validation of requirements for detect and avoid systems. In *Proceedings of the 14th International Conference on Tests and Proofs TAP*, volume 12165 of *Lecture Notes in Computer Science*, pages 155–166. Springer, June 2020. 25, 26, 27

[27] César Muñoz, Anthony Narkawicz, and James Chamberlain. A TCAS-II resolution advisory detection algorithm. In *Proceedings of the AIAA Guidance Navigation, and Control Conference and Exhibit 2013*, August 2013. 28

[28] César Muñoz, Anthony Narkawicz, James Chamberlain, María Consiglio, and Jason 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*, June 2014. 29

[29] César Muñoz, Anthony Narkawicz, George Hagen, Jason Upchurch, Aaron Dutle, and María Consiglio. DAIDALUS: Detect and Avoid Alerting Logic for Unmanned Systems. In *Proceedings of the 34th Digital Avionics Systems Conference DASC*, September 2015. 25, 28, 36

[30] César Muñoz, Anthony Narkawicz, George Hagen, Jason Upchurch, Aaron Dutle, and María Consiglio. DAIDALUS: Detect and Avoid Alerting Logic for Unmanned Systems. In *Proceedings of the 34th Digital Avionics Systems Conference DASC*, 2015. 1

[31] Anthony Narkawicz, César Muñoz, and Aaron Dutle. The MINERVA software development process. In *Automated Formal Methods*, volume 5 of *Kalpa Publications in Computing*, pages 93–108. EasyChair, 2018. 2

[32] Anthony J. Narkawicz, César A. Muñoz, Jason M. Upchurch, James P. Chamberlain, and María C. Consiglio. A well-clear volume based on time to entry point. Technical Memorandum NASA/TM-2014-218155, NASA, Langley Research Center, Hampton VA 23681-2199, USA, January 2014. 31

[33] Jim Q. Ning, Andre Engberts, and Wojtek Kozaczynski. Automated support for legacy code understanding. *Commun. ACM*, 37(5):50–57, 1994. 37

[34] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction, CADE*, pages 748–752. Springer, 1992. 1, 18, 24

[35] Sam Owre and Natarajan Shankar. The Formal Semantics of PVS. Technical Report CSL-97-2R, SRI International, 1999. 19

[36] P. Panchekha, A. Sanchez-Stern, J.R. Wilcox, and Tatlock Z. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 1–11. ACM, 2015. 53

[37] RTCA DO-365A. *Minimum Operational Performance Standards (MOPS) for Detect and Avoid (DAA) Systems, Appendix H.* RTCA, February 2020. 1

[38] C. Rubio-González, C. Nguyen, H.D. Nguyen, J. Demmel, W. Kahan, K. Sen, D.H. Bailey, C. Iancu, and D. Hough. Precimonious: tuning assistant for floating-point precision. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13*, page 27. ACM, 2013. 52

[39] J. Silva. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.*, 44(3):12:1–12:41, 2012. 37

[40] Busyairah Syd Ali, Wolfgang Schuster, Washington Ochieng, and Arnab Majumdar. Analysis of anomalies in ads-b and its gps data. *GPS Solutions*, 20, 03 2015. 8

[41] L. Thévenoux, P. Langlois, and M. Martel. Automatic source-to-source error compensation of floating-point programs. In *18th IEEE International Conference on Computational Science and Engineering, CSE*, pages 9–16. IEEE Computer Society, 2015. 53

[42] 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)*, pages 516–537. Springer, 2018. 7, 10, 11, 12, 52, 53

[43] 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)*, volume 12546 of *LNCS*, pages 141–159. Springer, 2020. 53

[44] 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)*, volume 10951 of *LNCS*, pages 364–381. Springer, 2018. 52

[45] Laura Titolo, Mariano Moscato, Marco A. Feliu, and César A. Muñoz. Automatic generation of guard-stable floating-point code. In *Integrated Formal Methods*, volume 12546 of *LNCS*, pages 141–159, Cham, 2020. Springer International Publishing. 2, 8, 10, 11, 12, 15, 24, 52

[46] Mark D. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981*, pages 439–449. IEEE Computer Society, 1981. 37

[47] Mark D. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984. 37

[48] X. Yi, L. Chen, X. Mao, and T. Ji. Efficient automated repair of high floating-point errors in numerical libraries. *PACMPL*, 3(POPL):56:1–56:29, 2019. 53