# Verification of Numerical Programs: From Real Numbers to Floating Point Numbers

Alwyn E. Goodloe[1], César Muñoz[1], Florent Kirchner[2], and Loïc Correnson[2]

[1] NASA Langley Research Center, USA
{a.goodloe,cesar.a.munoz}@nasa.gov
[2] CEA, LIST, France
{florent.kirchner,loic.correnson}@cea.fr

**Abstract.** Numerical algorithms lie at the heart of many safety-critical aerospace systems. The complexity and hybrid nature of these systems often requires the use of interactive theorem provers to verify that these algorithms are logically correct. Usually, proofs involving numerical computations are conducted in the infinitely precise realm of the field of real numbers. However, numerical computations in these algorithms are often implemented using floating point numbers. The use of a finite representation of real numbers introduces uncertainties as to whether the properties verified in the theoretical setting hold in practice. This short paper describes work in progress aimed at addressing these concerns. Given a formally proven algorithm, written in the Program Verification System (PVS), the Frama-C suite of tools is used to identify sufficient conditions and verify that under such conditions the rounding errors arising in a C implementation of the algorithm do not affect its correctness. The technique is illustrated using an algorithm for detecting loss of separation among aircraft.

## 1 Introduction

Virtually every aerospace application is composed of numerical algorithms. The mathematics in these algorithms is both continuous and discrete. The hybrid nature of aerospace applications often means that interactive theorem provers are required to reason about their logical correctness. As the models and algorithms are refined into an implementation, care must be taken so that assumptions made in the abstract models are not violated by the implementation. Of particular concern are the issues that arise when moving from the infinitely precise field of real numbers to an implementation using a floating point representation [4,8] such as the IEEE 754 standard [5]. It is well-known that overflows, underflows, and accumulated rounding errors in floating point arithmetic can produce results that significantly differ from the ideal. Hence, properties that were demonstrated to hold in the abstract models may be violated in a concrete implementation. Therefore one cannot assert that theorems proven in the setting of the real numbers carry over to the implementation without additional arguments.

The domain of application of the case study in this paper is *air traffic management* (ATM). Advances in surveillance and communication systems allow for

G. Brat, N. Rungta, and A. Venet (Eds.): NFM 2013, LNCS 7871, pp. 441–446, 2013.

ATM concepts where computer programs provide safety-critical functionality. For instance, the self-separation operational concept proposed by NASA [10] relies on airborne conflict detection and resolution (CD&R) systems that assist pilots and air traffic controllers to maintain safety in the airspace by keeping aircraft separated. Computer-based separation assurance systems are critical elements of air/ground distributed operational concepts for the next generation of air traffic management systems.

The Formal Methods group at NASA Langley has developed the Aiborne Coordinated Conflict Resolution and Detection (ACCoRD) formal framework for reasoning about aircraft separation assurance systems.[1] The framework, which is written in the Program Verification System (PVS) [9], consists of more than 1500 lemmas and includes formally verified algorithms for conflict detection, conflict resolution, conflict recovery, loss of separation recovery, and conflict prevention bands. This paper reports work in progress on a verification approach that is being applied to formally prove the correctness of the C implementations of some of these algorithms.

## 2 Conflict Detection

This paper concerns a conflict detection algorithm, namely CD2D, developed by NASA as part of the ACCoRD framework. CD2D is pairwise state-based 2-D conflict detection algorithm. Pairwise refers to the fact that CD2D only considers two aircraft called the *ownship* and the *intruder*. State-based refers to the use of an Euclidean airspace where the aircraft fly at constant velocity. In particular, in CD2D, the position and velocity of the ownship are represented by 2-D position $\mathbf{s}_o = (s_{ox}, s_{oy})$ and vector $\mathbf{v}_o = (v_{ox}, v_{oy})$, respectively, and the position and velocity of the intruder are represented by $\mathbf{s}_i = (s_{ix}, s_{iy})$ and $\mathbf{v}_i = (v_{ix}, v_{iy})$, respectively. As it simplifies the mathematical development, most definitions in ACCoRD use a relative coordinate system where the intruder is static at the center of the system. In this relative system, the ownship is located at $\mathbf{s} = \mathbf{s}_o - \mathbf{s}_i$ and moves at relative velocity $\mathbf{v} = \mathbf{v}_o - \mathbf{v}_i$.

In air traffic management, a *loss of separation* is a violation of the separation requirement between two aircraft. If the vertical dimension is ignored, the separation requirement is given by a minimum horizontal distance $D$. A *conflict* is a predicted loss of separation within a lookahead time $T$. In this paper, $D$ and $T$ are global constants. Loss of separation and conflict are formalized in ACCoRD as follows.

$$\mathit{los?}(\mathbf{s}) \equiv \sqrt{s_x^2 + s_y^2} < D, \quad \mathit{conflict?}(\mathbf{s}, \mathbf{v}) \equiv \exists 0 \leq t \leq T : \mathit{los?}(\mathbf{s} + t\mathbf{v}).$$

The PVS function *cd2d*, that models the CD2D algorithm, takes as parameters the state of the aircraft, i.e., $\mathbf{s}_o, \mathbf{v}_o, \mathbf{s}_i, \mathbf{v}_i$ and returns a Boolean value that indicates whether or not a loss of separation with respect to the minimum distance $D$ is predicted to occur within the lookahead time $T$.

[1] `http://shemesh.larc.nasa.gov/people/cam/ACCoRD`

$$cd2d(\mathbf{s}_o, \mathbf{v}_o, \mathbf{s}_i, \mathbf{v}_i) \equiv \texttt{let } \mathbf{s} = \mathbf{s}_o - \mathbf{s}_i, \mathbf{v} = \mathbf{v}_o - \mathbf{v}_i \texttt{ in } los?(\mathbf{s}) \text{ or } \omega(\mathbf{s}, \mathbf{v}) < 0,$$

where $\omega$ is a continuous function that characterizes conflicts. It is defined as follows.

$$\omega(\mathbf{s}, \mathbf{v}) \equiv \begin{cases} \mathbf{s} \cdot \mathbf{v} & \text{if } \mathbf{s}^2 = D^2, \\ \mathbf{v}^2\mathbf{s}^2 + 2\tau(\mathbf{s} \cdot \mathbf{v}) + \tau^2(\mathbf{s}, \mathbf{v}) - D^2\mathbf{v}^2 & \text{otherwise,} \end{cases}$$

where $\tau(\mathbf{s}, \mathbf{v}) \equiv \min(\max(0, -(\mathbf{s} \cdot \mathbf{v})), T\mathbf{v}^2)$. When $\mathbf{v}^2 \neq 0$, $\frac{\tau(s,\mathbf{v})}{\mathbf{v}^2}$ denotes the time of closest approach for the aircraft and $\frac{\omega(\mathbf{s},\mathbf{v})}{\mathbf{v}^2} + D$ denotes the minimum distance.

The ACCoRD development has a formal proof that the function *cd2d* is sound and complete with respect to the predicate *conflict*?, i.e., that the following statement holds.

**Proposition 1.** *Given a distance $D > 0$ and a lookahead time $T > 0$, for all vectors $\mathbf{s} = \mathbf{s}_o - \mathbf{s}_i$ and $\mathbf{v} = \mathbf{v}_o - \mathbf{v}_i$,*

**(soundness)** *If conflict?$(\mathbf{s}, \mathbf{v})$ holds then cd2d$(\mathbf{s}_o, \mathbf{v}_o, \mathbf{s}_i, \mathbf{v}_i)$ returns true.*
**(completeness)** *If cd2d$(\mathbf{s}_o, \mathbf{v}_o, \mathbf{s}_i, \mathbf{v}_i)$ returns true then conflict?$(\mathbf{s}, \mathbf{v})$ holds.*

Soundness and completeness are closely related to the concepts of *missed-alerts* and *false-alerts*, respectively.

It should be noted that the theoretical development presented in this section assumes infinite precision real numbers and does not consider physical limitations of the aircraft. In a concrete implementation of the CD2D algorithm, those considerations become significant. In particular, arbitrary large/small numbers in the presence of floating point numbers and the use of floating point arithmetic introduce uncertainties as to whether properties verified in the ideal theoretical setting, such as Proposition 1, hold in practice.

## 3 Verification in Practice

In order to formally prove a stament such as Proposition 1 for a C program, it is necessary to have a verification environment that provides a specification language supporting both real numbers and floating point arithmetic and that easily integrates with automated and interactive theorem provers. Frama-C is an open-source framework developed at CEA comprising a suite of tools for static analysis of C programs in the form of plugins implementing abstract interpretation, slicing, and deductive verification engines. In particular, Frama-C uses the deductive verification plugin Jessie [6], which generates verification conditions for C programs. These verification conditions are submitted to different theorem provers via the Why3 back-end [2]. In particular, Why3 connects to the Gappa [7] tool, which specializes in verifying properties of numerical programs. Frama-C

supports annotations written in the ANSI C Specification Language (ACSL) [1], an assertion language for specifying behavioral properties of C programs in a first-order logic. As PVS, ACSL supports mathematical expressions over the real numbers. Furthermore, ACSL has a built-in model of IEEE-754 arithmetic including the rounding modes, casts, and infinity. The analysis presented here assumes IEEE-754 in strict form, i.e., the generated verification conditions ensure no overflows or special values, and rounding to nearest with ties to even.

A straightforward C implementation of *cd2d* does not satisfy Proposition 1 due to the use of floating point arithmetic in C. Indeed, in the presence of computation errors, it is impossible to write a program that satisfies both correctness and completeness. In practice, there is a trade-off between soundness and completeness in any implementation of a conflict detection algorithm. From a safety point of view, soundness is usually considered the more desirable of the two properties since it eliminates the possibility for missed-alerts. Therefore, the target property for the verification presented here is soundness. However, it should be noted that completeness also has safety implications. For example, a program that always returns *true* would be trivially sound. Of course, such a program will have an unacceptable rate of false alerts and quickly erode the trust that a pilot may have on these kinds of systems.

This paper proposes a systematic construction of a C program, namely `cd2d`, from its PVS counterpart, namely *cd2d*, that is provably sound. The proof is conducted in the Frama-C environment and reuses Proposition 1 and other core geometric properties proved in PVS. The construction of `cd2d` starts by translating every real-valued function $f$ involved in the definition of *cd2d* into an identical logical ACSL function $f$ and into a C function `f`. Function $f$ uses real number arithmetic, while function `f` uses floating point arithmetic. The specification of the function `f` states that the absolute error of the floating point computation is bounded by a given positive constant $\epsilon_f$, i.e., $|f(x) - \mathtt{f}(x)| \leq \epsilon_f$. Here only the C basic types `double` and `int` are used for the translation. Therefore, vectors are represented by their components. For instance, the function $\tau$, used in Formula 2, is translated into ACSL-annotated C code as follows.[2]

```
/*@ logic real tauR(real s_x,real s_y,real v_x,real v_y,real t) =
   @   dmin(dmax(0.,-dotR(s_x,s_y,v_x,v_y)),t*sqvR(v_x,v_y)) ;
   @*/

/*@ requires -100. <= s_x <= 100. && ...;
   @ ensures \abs(\result - tauR(s_x,s_y,v_x,v_y,T)) <= E_tau;
   @*/

double tau(double s_x,double s_y,double v_x,double v_y) {
  return min(max(0,-dot(s_x,s_y,v_x,v_y)),T*sqv(v_x,v_y)); }
```

In ACSL, the precondition is denoted by the keyword `requires`, while the postcondition is denoted by the keyword `ensures`. By convention, real number

[2] Logical definitions in ACSL cannot refer to C constants. Hence, $t$ has been added as a parameter to `tauR`.

functions are written with the postfix `R`. If function `f` is proven to satisfy its specification for a certain value of $\epsilon_f$, this value is propagated into the specification of functions and Boolean conditions that depend on `f`. At the end of the process, the `cd2d` function is written as follows.

```
int cd2d(double so_x,double so_y,double vo_x,double vo_y,
         double si_x,double si_y,double vi_x,double vi_y) {
    double s_x = so_x − si_x; double s_y = si_x − si_y;
    double v_x = vo_x − vi_x; double v_y = vi_x − vi_y;
    return los(s_x,s_y) || omega(s_x,s_y,v_x,v_y) < E_cd2d; }
```

In order to appropriately bound the values of the input variables, a system of units needs to be chosen. As usual in air traffic management, distances are given in nautical miles, speeds are given in knots (nautical miles per hour), and, for unit consistency, times are given in hours. Typical bounds for state-based separation assurance algorithms such as CD2D are $|\texttt{so\_x}|, |\texttt{so\_y}|, |\texttt{si\_x}|, |\texttt{si\_y}| \leq 100$ nautical miles and $|\texttt{vo\_x}|, |\texttt{vo\_y}|, |\texttt{vi\_x}|, |\texttt{vi\_y}| \leq 600$ knots. Furthermore, the constants `D` and `T` are set to 5 nautical miles and 0.083 hours (about 5 minutes), respectively.

An approach to verify that `cd2d` verifies soundness consists in replaying the soundness proof of *cd2d* and adapting, on this process, every proof step to deal with floating point inaccuracies. This paper takes a different approach. Since the PVS function *cd2d* is known to be sound *and* complete, soundness of `cd2d` is equivalent to the following proposition.

**Proposition 2 (Soundness of `cd2d`).** *Given the specified values of `D` and `T`, for all `so_x`, `so_y`, `si_x`, `si_y`, `vo_x`, `vo_y`, `vi_x`, `vi_y` that satisfy the specified bounds, if cd2d(`so_x`, `so_y`, `vo_x`, `vo_y`, `si_x`, `si_y`, `vi_x`, `vi_y`) returns true, then `cd2d(so_x,so_y,vo_x,vo_y,si_x,si_y,vi_x,vi_y)` returns 1.*

This leaves the question of how to find the error bounds for each $f$, i.e., $\epsilon_f$. Sophisticated analytical techniques exist for estimating rounding errors [3] and while these are needed to analyze more complex computations, in many cases it is possible to exploit the capability of Frama-C to quickly and automatically prove assertions to discover an appropriate value for $\epsilon_f$. The process implements a search by dichotomy, hinging on the provability of the proof assertions.

Beginning with an initial estimate for $\epsilon_f$, the Frama-C/Jessie plugin is invoked generating a number of verification conditions. If the automated prover cannot show that $\epsilon_f$ is a good bound, the value of $\epsilon_f$ is increased. On the other hand, if the provers show that the bound holds, the value of $\epsilon_f$ is decreased. The process continues until convergence on a tight bound. In the case of `tau`, the initial value of `E_tau` was set to $2^{-30}$, but the Gappa solver on the back-end could not prove the postcondition. Next, `E_tau` was set to $2^{-10}$, which the solver easily discharged. The dichotomy process eventually reached a bound on an absolute error of $2^{-21}$. Proposition 2 is formally verified in Frama-C for the value $\texttt{E\_cd2d} = 2 \times 2^{-1}$.

## 4 Conclusion

This work in progress contributes a methodology for proving the correctness of implementations of numerical programs whose soundness and completeness have already been demonstrated in the ideal setting of real numbers. In particular, the approach proposed here focuses on discovering and proving the bounds on floating-point rounding errors that can invalidate in practice the theorems proven on reals. As a first case study, the technique was applied to candidate algorithms in the ACCoRD framework. These algorithms feature strong correctness conditions, use only bounded loops and conditionals, and employ well behaved mathematical operations. In addition, the algorithms have well defined bounded input, and units were chosen that kept the magnitude of the computed values from growing big enough to produce large rounding errors. Future work will apply the approach to more sophisticated programs and consider relative error in addition to the absolute error. Also, the task remains to validate the safety implications of the error bounds shown in the paper. As the methodology evolves, the Frama-C tool support is expected to evolve by incorporating new algorithms and plugins to aid in the verification of numerical programs.

## References

1. Baudin, P., Cuoq, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, version 1.6 (2012)
2. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland, pp. 53–64 (August 2011)
3. Boldo, S., Nguyen, T.M.T.: Hardware-independent proofs of numerical programs. In: NASA Formal Methods, pp. 14–23 (2010)
4. Goldberg, D.: What every computer scientist should know about floating point arithmetic. ACM Computing Surveys 23(1), 5–48 (1991)
5. IEEE Task P754. ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic. IEEE (1985)
6. Marhé, C., Moy, Y.: The Jessie Plugin for Deductive Verification in Frama-C. INRIA Saclay Île-de-France and LRI, CNRS UMR (2012)
7. Melquiond, G.: User's Guide for Gappa. INRIA (2012)
8. Muller, J.-M., Brisebarre, N., de Dinechin, F., Jeannerod, C.-P., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D., Torres, S.: Handbook of Floating-Point Arithmetic. Birkhäuser, Boston (2010)
9. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992)
10. Wing, D.J., Cotton, W.B.: Autonomous flight rules a concept for self-separation in U.S. domestic airspace. Technical Publication NASA/TP-2011-217174, NASA, Langley Research Center, Hampton VA 23681-2199, USA (November 2011)