A Formally Verified Floating-Point Implementation of the Compact Position Reporting Algorithm

Laura Titolo[1], Mariano M. Moscato[1], César A. Muñoz[2], Aaron Dutle[2], and François Bobot[3]*

[1] National Institute of Aerospace, Hampton, VA, USA
{laura.titolo,mariano.moscato}@nianet.org**
[2] NASA, Hampton, VA, USA
{cesar.a.munoz, aaron.dutle}@nasa.gov
[3] CEA LIST, Software Security Lab, Gif-sur-Yvette, France
francois.bobot@cea.fr***

Abstract. The Automatic Dependent Surveillance-Broadcast (ADS-B) system allows aircraft to communicate their current state, including position and velocity information, to other aircraft in their vicinity and to ground stations. The Compact Position Reporting (CPR) algorithm is the ADS-B module responsible for the encoding and decoding of aircraft positions. CPR is highly sensitive to computer arithmetic since it heavily relies on functions that are intrinsically unstable such as floor and modulo. In this paper, a formally-verified double-precision floating-point implementation of the CPR algorithm is presented. The verification proceeds in three steps. First, an alternative version of CPR, which reduces the floating-point rounding error is proposed. Then, the Prototype Verification System (PVS) is used to formally prove that the ideal real-number counterpart of the improved algorithm is mathematically equivalent to the standard CPR definition. Finally, the static analyzer Frama-C is used to verify that the double-precision implementation of the improved algorithm is correct with respect to its operational requirement. The alternative algorithm is currently being considered for inclusion in the revised version of the ADS-B standards document as the reference implementation of the CPR algorithm.

1 Introduction

The Automatic Dependent Surveillance-Broadcast (ADS-B) protocol [27] is a fundamental component of the next generation of air transportation systems.

* The authors are thankful to Guillaume Melquiond for his help and useful insights on the tool Gappa.

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

*** The work by the fifth author was partially funded by the National Aeronautics and Space Administration under NASA/NIA Cooperative Agreement NNL09AA00A and the grant ANR-14-CE28-0020.

It is intended to augment or replace ground-based surveillance systems such as radar by providing real-time accurate surveillance information based on global positioning systems. Aircraft equipped with ADS-B services broadcast a variety of information related to the current state of the aircraft, such as position and velocity, to other traffic aircraft and to ground stations. The use of ADS-B transponders is required to fly in some regions and, by 2020, it will become mandatory for most commercial aircraft in the US [11] and Europe [17] . Thousands of aircraft are currently equipped with ADS-B.[4]

The ADS-B broadcast message is defined to be 112 bits long. Its data frame takes 56 bits, while the rest is used to transmit aircraft identification, message type, and parity check information. When the data frame contains a position, 21 bits are devoted to the status information and altitude, leaving 35 bits in total for latitude and longitude. If raw latitude and longitude data were expressed as numbers of 17 bits each, the resulting position accuracy would be worse than 300 meters, which is inadequate for safe navigation. For this reason, the ADS-B protocol uses an algorithm called Compact Position Reporting (CPR) to encode/decode the aircraft position in 35 bits in a way that, for airborne applications, is intended to guarantee a position accuracy of approximately 5 meters. Unfortunately, pilots and manufacturers have reported errors in the positions obtained by encoding and decoding with the CPR algorithm.

In [16], it was formally proven that the original operational requirements of the CPR algorithm are not enough to guarantee the intended precision, even when computations are assumed to be performed using exact arithmetic. Additionally, the ideal real number implementation of CPR has been formally proven correct for a slightly tightened set of requirements [16]. Nevertheless, even assuming these more restrictive requirements, a straight-forward floating-point implementation of the CPR algorithm may still be unsound and produce incorrect results due to round-off error. For instance, using a standard single-precision floating-point implementation of CPR on a position whose latitude is $-77.368°$ and longitude is $180°$, the recovered position differs from the original one by approximately 1500 nautical miles.

In this paper, an alternative implementation of the CPR algorithm is presented. This version includes simplifications that decrease the numerical complexity of the expressions with respect to the original version presented in the ADS-B standard. In this way, the accumulated round-off error is reduced. Frama-C [21] is used to prove that the double-precision floating-point implementation of the proposed CPR algorithm is correct in the sense that the encoding has no rounding error and the decoded position satisfies the required operational accuracy of the algorithm. The Frama-C WP (Weakest Precondition) plug-in is used to generate verification conditions ultimately discharged with the aid of the automatic solvers Gappa [15] and Alt-Ergo [12]. In addition, the interactive theorem prover PVS [26] is used to formally prove that the real counterpart of the proposed alternative CPR algorithm is mathematically equivalent to the one

[4] `https://generalaviationnews.com/2017/09/18/more-than-40000-aircraft-now-equipped-with-ads-b/`.

defined in the standard [27]. It follows that the correctness results presented in [16] also hold for the proposed version of CPR. The PVS formalization of this equivalence is available at `https://shemesh.larc.nasa.gov/fm/CPR/`.

The remainder of the paper is organized as follows. In Section 2, the original definition of the CPR algorithm and the correctness of its real-valued version [16] are summarized. The alternative version of CPR is presented in Section 3 along with the results ensuring its mathematical equivalence with respect to the original algorithm. In Section 4, the verification approach used to prove the correctness of the double-precision implementation of the alternative algorithm is explained. Related work is discussed in Section 5. Finally, Section 6 concludes the paper.

2 The Compact Position Reporting Algorithm

In this section, the CPR algorithm is introduced, summarizing its definition in the ADS-B standard [27]. The CPR goal is to encode latitude and longitude in 17 bits while keeping a position resolution of approximately 5 meters. CPR is based on the fact that transmitting the entire latitude and longitude at each broadcasted message is inefficient since the higher order bits are very unlikely to change over a short period of time. In order to overcome this inefficiency, only an encoding of the least significant bits of the position is transmitted and two different techniques are used to recover the higher order bits.

CPR uses a special coordinate system where each direction (latitude and longitude) is divided into zones of approximately 360 nautical miles. There are two different subdivisions of the space in zones, based on the *format* of the message, either *even* or *odd*. The number of zones depends on the format and, in the case of the longitude, also on the current latitude of the target. Each zone is itself divided into 2^{17} parts, called *bins*. Fig. 1 shows how the latitude is divided into 60 zones (for the even subdivision) or into 59 zones (for the odd subdivision) and how each zone is then divided into 2^{17} bins. The CPR encoding procedure transforms degree coordinates into CPR coordinates and is parametric with respect to the chosen subdivision (even or odd). The decoding procedure recovers the position of the aircraft from the CPR coordinates. A CPR message coordinate is exactly the number corresponding to the bin where the target is located. The correct zone can be recovered from either a previously known position (for *local decoding*) or from a matched pair of even and odd messages (for *global decoding*). The decoding procedures return a coordinate which corresponds to the centerline of the bin where the target is located (see Fig. 1). In a latitude zone (respectively longitude zone), all the latitudes (respectively longitudes) inside a *bin* have the same encoding. This means that the recovered latitude (respectively longitude) corresponds to the *bin centerline*. Therefore, the difference between a given position and the result of encoding and decoding should be less than or equal to the size of half of a bin.

The modulo function is assumed to be computed as $\mathsf{mod}\,(x, y) = x - y\,\lfloor x/y \rfloor$. In this section, all computations are assumed to be performed in real arithmetic.

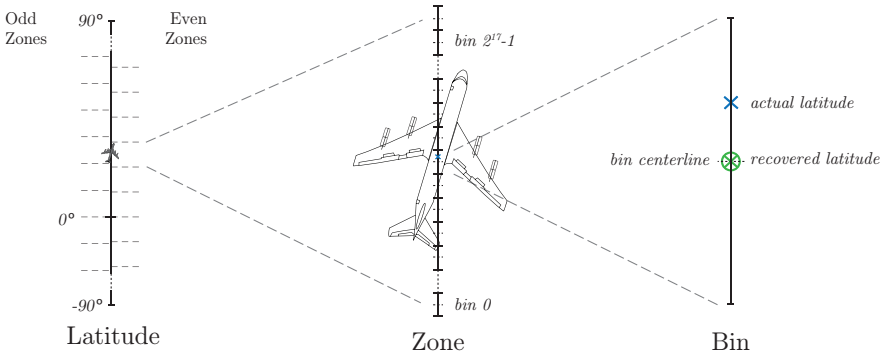

Fig. 1. CPR latitude coordinate system.

Therefore, no rounding error occurs. All the results presented in this section have been formally proven in a previous work [16].

2.1 Encoding

The CPR encoding translates latitude and longitude coordinates, expressed in degrees, into a pair of CPR coordinates, i.e., bin indices. Each CPR message is transmitted inside the data frame of an ADS-B message. The 35 bits composing the CPR message are grouped into three parts. One bit determines the format (0 for even and 1 for odd), 17 bits are devoted to the bin number for the latitude, and the other 17 bits to the bin number for the longitude.

Let $i \in \{0,1\}$ be the format of the message to be sent, the size of a latitude zone is defined as $dlat_i = 360/(60-i)$. Given a latitude in degrees $lat \in [-90, 90]$, the *latitude encoding* is defined as follows:

$$latEnc(i, lat) = \mathsf{mod}\left(\left\lfloor 2^{17}\frac{\mathsf{mod}\,(lat, dlat_i)}{dlat_i} + \frac{1}{2}\right\rfloor, 2^{17}\right). \tag{2.1}$$

In (2.1), $\mathsf{mod}\,(lat, dlat_i)$ is the distance between lat and the bottom of a zone edge. Thus, $\frac{\mathsf{mod}(lat, dlat_i)}{dlat_i}$ is the zone fraction of lat. Multiplying by 2^{17} gives a value between 0 and 2^{17}, while $\left\lfloor x + \frac{1}{2}\right\rfloor$ rounds a number x to the nearest integer. The external modulo ensures that the encoded latitude fits in 17 bits. It may appear that this final truncation can discard some useful information. However, it only affects half of a bin at the top of a zone, which is accounted for by the adjacent zone. For longitude, the CPR coordinate system keeps the size of zones approximately constant by reducing the number of longitude zones as the latitude increases. As a consequence, the number of longitude zones circling the globe is a function of the latitude. The function that determines the number of longitude zones is called *NL*. While its value can be calculated directly from a given latitude, in practice, it is determined from a pre-calculated lookup table. Since the construction of this table occurs off-line it can be computed with

enough precision to ensure its correctness during the encoding stage. Note that the latitude used to compute NL for encoding is actually the *recovered latitude*, which is the centerline of the bin containing the location. This ensures that the broadcaster and receiver can calculate the same value of NL for use in longitude decoding.

Given a recovered latitude value $rlat \in [-90, 90]$, the NL value is used to compute the longitude zone size as follows.

$$dlon_i(rlat) = 360/\max\{1, NL(rlat) - i\}. \tag{2.2}$$

Note that the denominator in the above expression uses the max operator when NL is 1, which occurs for latitudes beyond ±87 degrees. In this case, there is only one longitude zone and even and odd longitude encodings coincide.

Given a longitude value $lon \in [0, 360]$ and a recovered latitude value $rlat \in [-90, 90]$, the *longitude encoding* is defined similarly to latitude encoding:

$$lonEnc(i, rlat, lon) = \mathsf{mod}\left(\left\lfloor 2^{17}\frac{\mathsf{mod}\,(lon, dlon_i(rlat))}{dlon_i(rlat)} + \frac{1}{2}\right\rfloor, 2^{17}\right). \tag{2.3}$$

Let $\mathcal{BN}$ denote the domain of bin numbers which is composed by the integers in the interval $[0, 2^{17} - 1]$. The following lemma ensures the message is of the proper length.

Lemma 1. *Given $i \in \{0, 1\}$, $lat \in [-90, 90]$, and $lon \in [0, 360]$, then $latEnc(i, lat) \in \mathcal{BN}$ and $lonEnc(i, lat, lon) \in \mathcal{BN}$.*

2.2 Local Decoding

Each encoded coordinate broadcast in a CPR message identifies exactly one bin inside each zone. In order to unambiguously compute the decoded position, it suffices to determine the zone. To this end, the CPR *local decoding* uses a reference position that is known to be near the broadcast one. This reference position can be a previously decoded position or can be obtained by other means. The idea behind local decoding is simple. Observe that a one zone wide interval centered around a given reference position does not contain more than one occurrence of the same bin number. Therefore, as long as the target is close enough to the reference position (slightly less than half a zone), decoding can be performed correctly.

Given a format $i \in \{0, 1\}$, the encoded latitude $YZ_i \in \mathcal{BN}$, and a reference latitude $lat_{ref} \in [-90, 90]$, the local decoding uses the following formula to calculate the *zone index number* (zin).

$$latZin_{\mathsf{L}}(i, YZ_i, lat_{ref}) = \left\lfloor \frac{lat_{ref}}{dlat_i} \right\rfloor + \left\lfloor \frac{1}{2} + \frac{\mathsf{mod}\,(lat_{ref}, dlat_i)}{dlat_i} - \frac{YZ_i}{2^{17}} \right\rfloor. \tag{2.4}$$

The first term in this sum calculates which zone the reference latitude lies in, while the second term adjusts it by −1, 0, or 1 based on the difference between

the reference latitude and the received encoded latitude. The zone index number is then used to compute the recovered latitude using the following function.

$$rlat_{\mathsf{L}}(i, YZ_i, lat_{ref}) = dlat_i \cdot \left(latZin_{\mathsf{L}}(i, YZ_i, lat_{ref}) + \frac{YZ_i}{2^{17}} \right). \tag{2.5}$$

This recovered latitude is used to determine the *NL* value for computing the value of $dlon_i$ by Formula (2.2). Given a reference longitude $lon_{ref} \in [0, 360]$, the recovered latitude $rlat \in [-90, 90]$, and the encoded longitude $XZ_i \in \mathcal{BN}$, the longitude zone index and recovered longitude are computed similarly to the case of the latitude. In the following formulas, $dlon_i$ is used as an abbreviation for $dlon_i(rlat_{\mathsf{L}}(i, YZ_i, lat_{ref}))$.

$$lonZin_{\mathsf{L}}(i, XZ_i, lon_{ref}, rlat) = \left\lfloor \frac{lon_{ref}}{dlon_i} \right\rfloor + \left\lfloor \frac{1}{2} + \frac{\mathsf{mod}\,(lon_{ref}, dlon_i)}{dlon_i} - \frac{XZ_i}{2^{17}} \right\rfloor. \tag{2.6}$$

$$rlon_{\mathsf{L}}(i, XZ_i, lon_{ref}, rlat) = dlon_i \cdot \left(lonZin_{\mathsf{L}}(i, XZ_i, lon_{ref}, rlat) + \frac{XZ_i}{2^{17}} \right). \tag{2.7}$$

When the difference between original and reference latitude (respectively longitude) is less than half zone size minus half bin size, local decoding is correct. This means that the difference between the original and recovered latitude (respectively longitude) is at most half of a bin size.

Theorem 1 (Local Decoding Correctness). *Given a format $i \in \{0, 1\}$, a latitude $lat \in [-90, 90]$, and a reference latitude $lat_{ref} \in [-90, 90]$ such that $|lat - lat_{ref}| < \frac{dlat_i}{2} - \frac{dlat_i}{2^{18}}$,*

$$|lat - rlat_{\mathsf{L}}(i, latEnc(i, lat), lat_{ref})| \leq \frac{dlat_i}{2^{18}}.$$

Furthermore, given a recovered latitude $rlat \in [-90, 90]$, a longitude $lon \in [0, 360]$, and a reference longitude $lon_{ref} \in [0, 360]$ such that $|lon - lon_{ref}| < \frac{dlon_i(rlat)}{2} - \frac{dlon_i(rlat)}{2^{18}}$,

$$|lon - rlon_{\mathsf{L}}(i, lonEnc(i, rlat, lon), lon_{ref}, rlat)| \leq \frac{dlon_i(rlat)}{2^{18}}.$$

2.3 Global Decoding

Global decoding is used when a valid reference position is unknown. This can occur when a target is first encountered, or when messages have not been received for a significant amount of time. Similarly to the local decoding case, the correct zone in which the encoded position lies has to be determined. To accomplish this, the global decoding uses a pair of messages of different formats, one even and one odd. The algorithm computes the number of *zone offsets* (the difference between an odd zone length and an even zone length) from the origin (either equator or prime meridian) to the encoded position. This can be used to establish the zone for either message type, and hence used to decode the position.

The first step in global decoding is to determine the number of zone offsets between the southern boundaries of the two encoded latitudes. Given two integers $YZ_0, YZ_1 \in \mathcal{BN}$, the zone index number for the latitude is computed as follows.

$$latZin_{\mathsf{G}}(YZ_0, YZ_1) = \left\lfloor \frac{59\,YZ_0 - 60\,YZ_1}{2^{17}} + \frac{1}{2} \right\rfloor. \tag{2.8}$$

Note that $YZ_0/2^{17}$ is the fraction into the even zone that the encoded latitude lies in. Since exactly 59 zone offsets fit into each even zone, $59\,YZ_0/2^{17}$ is the number of zone offsets from the southern boundary of an even zone. Similarly, $60\,YZ_1/2^{17}$ is the number of zone offsets from the southern boundary of an odd zone. The difference between these gives the number of zone offsets between southern boundaries of the respective zones, which corresponds to the correct zone. For example, if both are in zone 0, the southern boundaries coincide. If both are in zone 1, the southern boundaries differ by 1 zone offset. The case when encoding zones differ is accounted for by the modulo operation.

Given $i \in \{0, 1\}$, the recovered latitude is calculated as shown below.

$$rlat_{\mathsf{G}}(i, YZ_0, YZ_1) = dlat_i \cdot \left(\mathsf{mod}\,(latZin_{\mathsf{G}}(YZ_0, YZ_1), 60 - i) + \frac{YZ_i}{2^{17}} \right). \tag{2.9}$$

For the global decoding of a longitude, it is essential to check that the even and odd messages being used were calculated with the same NL value. To this end, both even and odd latitude messages are decoded, and their NL values are calculated. If they differ, the messages are discarded, otherwise, the longitude decoding can proceed using the common NL value. Given $i \in \{0, 1\}$ and $XZ_0, XZ_1 \in \mathcal{BN}$, if $NL(rlat_{\mathsf{G}}(0, YZ_0, YZ_1)) = NL(rlat_{\mathsf{G}}(1, YZ_0, YZ_1))$ the zone index number is computed as follows, where nl denotes $NL(rlat_{\mathsf{G}}(i, YZ_0, YZ_1))$ for $i = 0, 1$.

$$lonZin_{\mathsf{G}}(XZ_0, XZ_1) = \left\lfloor \frac{(nl-1)XZ_0 - (nl)XZ_1}{2^{17}} + \frac{1}{2} \right\rfloor. \tag{2.10}$$

Using $rlat_{\mathsf{G}}(i, YZ_0, YZ_1)$ to compute $dlon_i$ and nl, and letting nl_i stand for $\max(nl - i, 1)$, the recovered longitude is computed as follows.

$$rlon_{\mathsf{G}}(i, XZ_0, XZ_1) = dlon_i \cdot \left(\mathsf{mod}\,(latZin_{\mathsf{G}}(XZ_0, XZ_1), nl_i) + \frac{XZ_i}{2^{17}} \right). \tag{2.11}$$

The zone offset represents the difference between an even and an odd zone. For the latitude it is defined as $ZO_{lat} = dlat_1 - dlat_0$, while for the longitude, given a latitude $rlat$, is defined as $ZO_{lon} = dlon_1(rlat) - dlon_0(rlat)$. When the difference between the original coordinates is less than half zone offset minus the size of one odd bin, global decoding is correct. This means that the difference between the original and recovered latitude and longitude is at most the size of half bin.

Theorem 2 (Global Decoding Correctness). *Given $i \in \{0, 1\}$, for all lat_0, $lat_1 \in [-90, 90]$ such that $|lat_0 - lat_1| < \frac{ZO_{lat}}{2} - \frac{dlat_1}{2^{17}}$,*

$$|lat_i - rlat_{\mathsf{G}}(i, latEnc(0, lat_0), latEnc(1, lat_1))| \leq \frac{dlat_i}{2^{18}}.$$

Furthermore, let $rlat_0 = rlat_{\mathsf{G}}(0, latEnc(0, lat_0), latEnc(1, lat_1))$ *and* $rlat_1 = rlat_{\mathsf{G}}(1, latEnc(0, lat_0), latEnc(1, lat_1))$ *be even and odd recovered latitudes, respectively. If* $NL(rlat_0) = NL(rlat_1)$*, then for all* $lon_0, lon_1 \in [0, 360]$ *such that* $|lon_0 - lon_1| < \frac{ZO_{lon}}{2} - \frac{dlon_1(rlat_i)}{2^{17}}$*,*

$$|lon_i - rlon_{\mathsf{G}}(i, lonEnc(0, rlat_0, lon_0), lonEnc(1, rlat_1, lon_1))| \leq \frac{dlon_i(rlat_i)}{2^{18}}.$$

3 An Alternative Implementation of CPR

In this section, an alternative implementation of CPR is presented. This version uses mathematical simplifications that decrease the numerical complexity of the expressions with respect to the original implementation presented in the ADS-B standard. The alternative version is designed to be more numerically stable and to minimize the accumulated floating-point round-off error. Whenever possible, the formulas are transformed in order to perform multiplications and divisions by a power of 2, which are known to produce no round-off error as long as no over or under-flow occurs. Other simplifications are applied to reduce the number of operations, especially the modulo and floor. These operations are particularly problematic because a small difference in the arguments can lead to a significant difference in the result. For instance, consider a variable x that has an ideal real value of 1, while its floating-point version $\tilde{x}$ has value 0.999999. The round-off error associated to x is $|x - \tilde{x}| = 0.000001$, but the error associated to the application of the floor operation is $|\lfloor x \rfloor - \lfloor \tilde{x} \rfloor| = 1$.

Assuming real arithmetic, the proposed implementation is shown to be equivalent to the original one. All the results presented in this section have been formally verified using the PVS theorem prover. The input coordinates for this CPR algorithm are assumed to be given in a format called *32 bit angular weighted binary* (AWB), a standard format for expressing geographical positions used by GPS manufacturers and many others. An AWB coordinate is a 32 bit integer in the interval $[0, 2^{32} - 1]$, where the value x corresponds to $\frac{360x}{2^{32}}$ degrees (negative latitudes are identified with their value modulo 360). In the following, $\mathcal{AWB}$ denotes the domain of AWB numbers and a hat is used to emphasize that a given variable denotes an AWB value.

3.1 Alternative Encoding

Given a latitude $\widehat{lat} \in \mathcal{AWB}$, Algorithm 1 encodes it in a bin index number. The encoding is slightly different for AWB latitudes greater than 2^{30} because the input latitude range for the original encoding is $[-90, 90]$ and the AWB interval from 2^{30} to 2^{32} corresponds to the range $[90, 360]$. Therefore, a shift must be performed to put the range $[270, 360]$ in the expected input format $[-90, 0]$.

Algorithm 2 implements the longitude encoding similarly to Algorithm 1. In this case, no shift is needed since the input longitude range is $[0, 360]$. The variable nz denotes the number of longitude zones, which is 1 when $nl = 1$ and $nl - i$

Algorithm 1 $latEnc'(i, \widehat{lat})$

$nz \leftarrow 60 - i$
if $\widehat{lat} \leq 2^{30}$ **then**
  $tmp_1 = (\widehat{lat} * nz + 2^{14}) * 2^{-15}$
  $tmp_2 = (\widehat{lat} * nz + 2^{14}) * 2^{-32}$
else
  $tmp_1 = ((\widehat{lat} - 2^{32}) * nz + 2^{14}) * 2^{-15}$
  $tmp_2 = ((\widehat{lat} - 2^{32}) * nz + 2^{14}) * 2^{-32}$
end if
return $\lfloor tmp_1 \rfloor - 2^{17} * \lfloor tmp_2 \rfloor$

Algorithm 2 $lonEnc'(i, nl, \widehat{lon})$

if $nl = 1$ **then**
  $nz \leftarrow 1$
else
  $nz \leftarrow nl - i$
end if
$tmp_1 = (\widehat{lon} * nz + 2^{14}) * 2^{-15}$
$tmp_2 = (\widehat{lon} * nz + 2^{14}) * 2^{-32}$
return $\lfloor tmp_1 \rfloor - 2^{17} * \lfloor tmp_2 \rfloor$

otherwise. This is equivalent to taking the maximum between 1 and $nl - i$ as done in the original version of the algorithm (see Formula (2.2)). The following theorem states the mathematical equivalence of the proposed alternative encoding with respect to the one described in Subsection 2.1 assuming ideal real-valued arithmetic.

Theorem 3. *Let* $lat, rlat \in [-90, 90]$, $lon \in [0, 360]$, $\widehat{lat}, \widehat{lon} \in \mathcal{AWB}$, *and* $i \in \{0, 1\}$, *if* $lat = \frac{360\widehat{lat}}{2^{32}}$, $lon = \frac{360\widehat{lon}}{2^{32}}$, *and* $nl = NL(rlat)$, *then*

$$latEnc'(i, \widehat{lat}) = latEnc(i, lat),$$
$$lonEnc'(i, nl, \widehat{lon}) = lonEnc(i, rlat, lon),$$

where $rlat = dlat_i \cdot \left(\left\lfloor \frac{lat}{dlat_i} \right\rfloor + \frac{latEnc(i,lat)}{2^{17}} \right)$.

To prove this lemma, it is necessary to use the following intermediate results. First, the following alternative formula for encoding is used, which avoids the external modulo of 2^{17} used in Equations (2.1) and (2.3).

Lemma 2. *Let* $lat, rlat \in [-90, 90]$, $lon \in [0, 360]$, *and* $i \in \{0, 1\}$,

$$latEnc(i, lat) = \left\lfloor 2^{17} \frac{\mathsf{mod}\left(lat + 2^{-18} dlat_i, dlat_i\right)}{dlat_i} \right\rfloor$$
$$lonEnc(i, rlat, lon) = \left\lfloor 2^{17} \frac{\mathsf{mod}\left(lon + 2^{-18} dlon_i(rlat), dlon_i(rlat)\right)}{dlon_i(rlat)} \right\rfloor.$$

The following two results, which have been formally proven correct in [16], are also used. When the modulo operator is divided by its second argument, the following simplification can be applied.

$$\frac{\mathsf{mod}\,(a, b)}{b} = \frac{a}{b} - \left\lfloor \frac{a}{b} \right\rfloor. \tag{3.1}$$

Additionally, given any number x and any integer n, the floor function and the addition of integers is commutative.

$$\lfloor x + n \rfloor = \lfloor x \rfloor + n. \tag{3.2}$$

Algorithm 3 $rlat'_{\mathsf{L}}(i, \widehat{lat}, YZ)$

$nz \leftarrow 60 - i$
$dlat \leftarrow 360/nz$
if $\widehat{lat} \leq 2^{30}$ **then**
  $zin \leftarrow \lfloor (\widehat{lat} * nz - (YZ - 2^{16}) * 2^{15}) * 2^{-32} \rfloor$
else
  $zin \leftarrow \lfloor ((\widehat{lat} - 2^{32}) * nz - (YZ - 2^{16}) * 2^{15}) * 2^{-32} \rfloor$
end if
return $dlat * (YZ * 2^{-17} + zin)$

Given l denoting either a latitude or a longitude and dl representing $dlat$ or $dlon$ respectively, the following equality holds.

$$\left\lfloor 2^{17} \frac{\mathsf{mod}\left(l + 2^{-18} dl, dl\right)}{dl} \right\rfloor = \left\lfloor 2^{17} \frac{l}{dl} + \frac{1}{2} \right\rfloor - 2^{17} \left\lfloor \frac{l}{dl} + \frac{1}{2^{18}} \right\rfloor. \tag{3.3}$$

Since the input coordinate l is assumed to correspond to an AWB, there exists $\hat{l} \in \mathcal{AWB}$ such that $l = \frac{360\hat{l}}{2^{32}}$. By replacing l, after some basic arithmetic simplifications, the formula used in Algorithms 1 and 2 is obtained as follows.

$$\left\lfloor 2^{17} \frac{l}{dl} + \frac{1}{2} \right\rfloor - 2^{17} \left\lfloor \frac{l}{dl} + \frac{1}{2^{18}} \right\rfloor = \left\lfloor (\hat{l} \cdot nz + 2^{14}) 2^{-15} \right\rfloor - 2^{17} \left\lfloor (\hat{l} \cdot nz + 2^{14}) 2^{-32} \right\rfloor. \tag{3.4}$$

3.2 Alternative Local Decoding

Given an encoded latitude YZ and a reference latitude in AWB format, Algorithm 3 recovers the latitude corresponding to the centerline of the bin where the original latitude was located. Similarly to the encoding algorithm, it is necessary to shift the AWB to correctly represent the latitudes between −90 and 0 degrees. Correspondingly, Algorithm 4 recovers the longitude centerline bin. Note that the two algorithms differ only in the computation of the zone index number (zin). Let ref be the reference latitude (respectively longitude) in degrees, dl be the zone size, and enc the 17-bit encoding. By applying Equations (3.1) and

Algorithm 4 $rlon'_{\mathsf{L}}(i, nl, \widehat{lon}, XZ)$

if $nl = 1$ **then**
  $nz \leftarrow 1$
else
  $nz \leftarrow nl - i$
end if
$dlon \leftarrow 360/nz$
$zin \leftarrow \lfloor (\widehat{lon} * nz - (XZ - 2^{16}) * 2^{15}) * 2^{-32} \rfloor$
return $dlon * (XZ * 2^{-17} + zin)$

Algorithm 5 $rlat'_{\mathsf{G}}(i, YZ_0, YZ_1)$

$dlat \leftarrow 360/(60 - i)$
$zin = \lfloor (59 * YZ_0 - 60 * YZ_1 + 2^{16}) * 2^{-17} \rfloor$
if $i = 0$ **then**
  return $dlat * ((zin - 60 * \lfloor zin/60 \rfloor) + YZ_0 * 2^{-17})$
else
  return $dlat * ((zin - 59 * \lfloor zin/59 \rfloor) + YZ_1 * 2^{-17})$
end if

(3.2), the latitude (respectively longitude) zone index number formulas (2.4) and (2.6) can be rewritten in the form

$$\left\lfloor \frac{1}{2} + \frac{ref}{dl} - \frac{enc}{2^{17}} \right\rfloor.$$

Since the reference coordinate ref is assumed to represent an AWB, there exists $\widehat{ref} \in \mathcal{AWB}$ such that $ref = \frac{360\widehat{ref}}{2^{32}}$. After some simple algebraic simplification, Theorem 4 directly follows.

Theorem 4. *Let* $i \in \{0, 1\}$, $YZ_i, XZ_i \in \mathcal{BN}$, *if* $lat_{ref} = \frac{360\widehat{lat}_{ref}}{2^{32}}$ *and* $lon_{ref} = \frac{360\widehat{lon}_{ref}}{2^{32}}$, *then*

$$rlat_{\mathsf{L}}(i, YZ_i, lat_{ref}) = rlat'_{\mathsf{L}}(i, YZ_i, \widehat{lat}_{ref})$$
$$rlon_{\mathsf{L}}(i, XZ_i, lon_{ref}, lat_{ref}) = rlon'_{\mathsf{L}}(i, NL(rlat'_{\mathsf{L}}(i, YZ_i, \widehat{lat}_{ref})), \widehat{lon}_{ref}, XZ_i).$$

3.3 Alternative Global Decoding

Algorithm 5 and Algorithm 6 perform the global decoding for latitude and longitude, respectively. Variable i represents the format of the most recent message received, which is used to determine the aircraft position. In Algorithm 6, nl is the common value computed using both latitudes recovered by Algorithm 5. When $nl = 1$, the computation is significantly simplified due to having only one zone. Otherwise, the recovered longitude is computed similarly to the latitude. Theorem 5 directly follows from simple algebraic manipulations. The sum of the two fractions inside the floor in Formula (2.8) is explicitly calculated and the modulo in Formulas (2.9) and (2.11) is expanded.

Theorem 5. *Let* $i \in \{0, 1\}$, $YZ_i, XZ_i \in \mathcal{BN}$, *and* $nl = NL(rlat'_{\mathsf{G}}(i, YZ_0, YZ_1))$,

$$rlat_{\mathsf{G}}(i, YZ_0, YZ_1) = rlat'_{\mathsf{G}}(i, YZ_0, YZ_1)$$
$$rlon_{\mathsf{G}}(i, XZ_0, XZ_1) = rlon'_{\mathsf{G}}(i, nl, XZ_0, XZ_1).$$

4 Verification Approach

This section presents the verification approach used to prove that double precision floating-point arithmetic is enough to obtain a correct implementation of

Algorithm 6 $rlon'_{\mathsf{G}}(i, nl, XZ_0, XZ_1)$

if $nl = 1$ **then**
  if $i = 0$ **then**
    return $360 * XZ_0 * 2^{-17}$
  else
    return $360 * XZ_1 * 2^{-17}$
  end if
else
  $dlon \leftarrow 360/(nl - i)$
  $zin \leftarrow \lfloor ((nl - 1) * XZ_0 - nl * XZ_1 + 2^{16}) * 2^{-17} \rfloor$
  $zin' \leftarrow zin/(nl - i)$
  if $i = 0$ **then**
    return $dlon * ((zin - (nl - i) * \lfloor zin' \rfloor) + XZ_0 * 2^{-17})$
  else
    return $dlon * ((zin - (nl - i) * \lfloor zin' \rfloor) + XZ_1 * 2^{-17})$
  end if
end if

the CPR algorithm. In the following, the double-precision floating-point counterpart of a real-valued function f will be represented with a tilde, as $\tilde{f}$. In the floating-point version, every mathematical operator on real numbers is replaced by the corresponding double-precision floating-point operator.

The floating-point encoding of CPR is considered correct if it returns exactly the same value of the real number implementation. This means that no round-off error affects the final outcome. The double precision implementation of encoding achieves this, as indicated by the following theorem.

Theorem 6 (Correctness Double-precision Encoding). *Let* $\widehat{lat} \in \mathcal{AWB}$, $\widehat{lon} \in \mathcal{AWB}$, *nl be an integer in the range* $[1, 59]$, *and* $i \in \{0, 1\}$,

$$latEnc'(i, \widehat{lat}) = \widetilde{latEnc}'(i, \widehat{lat})$$
$$lonEnc'(i, nl, \widehat{lon}) = \widetilde{lonEnc}'(i, nl, \widehat{lon}).$$

For decoding, note that Theorem 1 and Theorem 2 state that the original coordinate and the bin centerline differs by at most half the size of a bin. If the recovered coordinate computed with floating-point decoding differs from the bin-centerline computed with real numbers by at most half the size of a bin, then the original coordinate, the bin-centerline, and the recovered coordinate are all located in the same bin. Hence, a floating-point decoding function can be considered correct when the recovered coordinate differs from the bin-centerline by at most half the size of a bin. From the previous observation, it follows that a new table $\widetilde{NL}$, which takes as input the floating-point latitude resulting from $\widetilde{rlat_{\mathsf{L}}}'$, can be computed off-line with sufficient precision. For each transition latitude l in the original NL table the floating-point representation of the closest bin centerlines enclosing l are used to decide the corresponding nl value. Recall from Section 2 that the bin size for the even configuration is approximatively

4.578×10^{-5} degrees, and for the odd one is 4.655×10^{-5} degrees. In the following theorems, the lower bound for half the bin size of 2.2888×10^{-5} degrees is used.

Theorem 7 (Correctness Double-precision Local Decoding). *Let* $i \in \{0, 1\}$, $YZ_i, XZ_i \in \mathcal{BN}$, *and* $\widehat{lat}_{ref}, \widehat{lon}_{ref} \in \mathcal{AWB}$,

$$|rlat'_{\mathsf{L}}(i, \widehat{lat}_{ref}, YZ_i) - \widetilde{rlat_{\mathsf{L}}}'(i, \widehat{lat}_{ref}, YZ_i)| \leq 2.2888 \times 10^{-5}$$
$$|rlon'_{\mathsf{L}}(i, nl, \widehat{lon}_{ref}, XZ_i) - \widetilde{rlon_{\mathsf{L}}}'(i, \widetilde{nl}, \widehat{lon}_{ref}, XZ_i)| \leq 2.2888 \times 10^{-5}$$

where $nl = NL(rlat'_{\mathsf{L}}(i, YZ_i, \widehat{lat}_{ref}))$ *and* $\widetilde{nl} = \widetilde{NL}(\widetilde{rlat_{\mathsf{L}}}'(i, YZ_i, \widehat{lat}_{ref}))$.

Theorem 8 (Correctness Double-precision Global Decoding). *Let* $i \in \{0, 1\}$ *and* $YZ_i, XZ_i \in \mathcal{BN}$, *if* $NL(rlat'_{\mathsf{G}}(0, YZ_0, YZ_1)) = NL(rlat'_{\mathsf{G}}(1, YZ_0, YZ_1))$,

$$|rlat'_{\mathsf{G}}(i, YZ_0, YZ_1) - \widetilde{rlat_{\mathsf{G}}}'(i, YZ_0, YZ_1)| \leq 2.2888 \times 10^{-5}$$
$$|rlon'_{\mathsf{G}}(i, nl, YZ_0, YZ_1) - \widetilde{rlon_{\mathsf{G}}}'(i, \widetilde{nl}, YZ_0, YZ_1)| \leq 2.2888 \times 10^{-5}$$

where $nl = NL(rlat'_{\mathsf{G}}(j, YZ_0, YZ_1))$ *and* $\widetilde{nl} = \widetilde{NL}(\widetilde{rlat_{\mathsf{G}}}'(j, YZ_0, YZ_1))$ *for* $j = 0, 1$.

Fig. 2 depicts the verification approach followed in this work. Frama-C was used to formally verify that Theorems 6, 7 and 8 hold. Frama-C is a tool suite that collects several static analyzers for the C language. C programs can be annotated with ACSL [2] annotations that state function contracts, pre and post conditions, assertions, and invariants. The Frama-C WP plug-in implements a weakest precondition calculus for ACSL annotations through C programs. For each ACSL annotation, this plug-in generates a set of verification conditions (VCs) that can be discharged by external provers. In the analysis presented in this paper, the SMT solver Alt-Ergo and the prover Gappa are used.

Gappa [15] is a tool able to formally verify properties on finite precision computations and to bound the associated round-off error. Additionally, it generates a formal proof of the results that can be checked independently by an external proof assistant. This feature provides a higher degree of confidence in the analysis of the numerical code. Gappa models the propagation of the round-off error by using interval arithmetic and a battery of theorems on real and floating-point numbers. The main drawback of interval arithmetic is that it does not keep track of the correlation between expressions sharing subterms, which may lead to imprecise over-approximations. To improve precision, Gappa accepts hints from the user. These hints can be used to perform a bisection on the domain of an expression, or to propose rewriting rules that appear as hypotheses in the generated formal proof. Gappa is very efficient and precise for checking enclosures for floating-point rounding errors, but it is not always suited to tackle other types of verification conditions generated by Frama-C. For this reason, the SMT solver Alt-Ergo is used in combination with Gappa.

The real counterpart of each C function implementing the alternative version of CPR is expressed as an ACSL logic function. As mentioned in Section 3, PVS

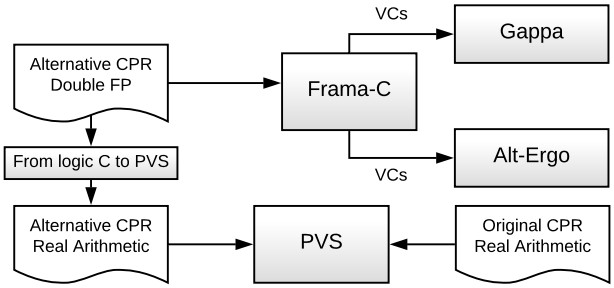

Fig. 2. Verification approach.

is used to formally verify the mathematical equivalence of these logic functions with respect to the PVS formalization of the original CPR definition. Pre and post-conditions are added to relate logic real-valued functions with the corresponding C double-precision floating-point implementation and to model Theorems 6, 7 and 8. Also, additional intermediate assertions are added after specific instructions to help the WP reasoning.

Algorithms 1 and 2 are annotated with assertions stating that tmp_1 and tmp_2 do not introduce rounding error. This generates VCs that are easily proved by Gappa because the computation just involves operations between integers and multiplications by powers of 2. Since the floor operation is applied to expressions that do not carry a round-off error, the computation of the floor is also exact and, therefore, Theorem 6 holds.

Algorithms 3 and 4 are annotated with assertions stating that the computation of the zone index number *zin* has no round-off error. This holds and can be easily discharged in Gappa since the computation of *zin* involves just integer sums and multiplications, and multiplications by powers of 2. The only calculation that carries a round-off error different from 0 is the one of the zone size (*dlat* and *dlon*) that involves a division. However, Gappa is able to prove that the propagation of this error in the result is bounded by half bin size (Theorem 7).

The verification of the global decoding procedures involves more complex reasoning. Similarly to the local decoding case, the code is annotated to explicitly state that the zone index number is not subject to rounding errors, and that its value is between −59 and 60. These two assertions are easily proved by Gappa. With nz denoting the number of zones (60 or 59 for latitude, and the maximum of $nl - 1$ and 1 for longitude), an annotation is added to assert that the real-valued and double-precision computation of $\lfloor zin/nz \rfloor$ coincide. In order to prove the verification conditions generated by this assertion, Gappa was provided with a hint on how to perform the bisection. It is important to remark that this hint does not add any hypothesis to the verification process. Given these intermediate assertions, Gappa is able to verify Theorem 8 as well.

5 Related Work

Besides Frama-C, other tools are available to formally verify and analyze numerical properties of C code. Fluctuat [20] is a commercial static analyzer that, given a C program with annotations about input bounds and uncertainties on its arguments, produces bounds for the round-off error of the program decomposed with respect to its provenance. Caduceus [18] produces verification conditions from annotated C code and discharges them in an independent theorem prover. In [7], the Caduceus tool is extended to reason about floating-point arithmetics. Here, Why [5] is used to generate verification conditions that are manually proven in the Coq proof assistant [3]. The static analyzer Astrée [13] detects the presence of run-time exceptions such as division by zero and under and over-flows by means of sound floating-point abstract domains [24,10].

The verification approach used in this work is similar to the analysis of numerical programs described in [8], where a chain of tools composed of Frama-C, the Jessie plug-in [23], and Why is used. The verification conditions obtained from the ACSL annotated C programs are checked by several external provers including Coq, Gappa, Z3 [25], CVC3 [1], and Alt-Ergo.

Recently, much work has been done on the verification of numerical properties for industrial and safety-critical C code, including aerospace software. The approach presented in [8] was applied to the formal verification of wave propagation differential equations [6] and to the verification of numerical properties of a pairwise state-based conflict detection algorithm [19]. A similar verification approach was employed to verify numerical properties of industrial software related to inertial navigation [22]. Astrée has been successfully applied to automatically check the absence of runtime errors associated with floating-point computations in aerospace control software [4]. More specifically, in [14] the fly-by-wire primary software of commercial airplanes is verified. Additionally, Astrée and Fluctuat were combined to analyze on-board software acting in the Monitoring and Safing Unit of the Automated Transfer Vehicle (ATV) [9].

6 Conclusion

In this paper, an alternative version of the CPR algorithm is proposed. This algorithm is an essential component of the ADS-B protocol which will soon be required in nearly all commercial aircraft in Europe and the USA. This alternative algorithm includes several simplifications aimed to reduce its numerical complexity. The equivalence between this version and the original algorithm in the ADS-B standard is formally proven in PVS. Additionally, it is shown that double-precision floating-point computation guarantees the correct operation of the alternative algorithm when implemented in C.

The verification approach applied in this work requires some level of expertise. A background in floating-point arithmetic is needed to express the properties to be verified and to properly annotate for the weakest precondition deductive reasoning. Deep understanding of the features of each tool is essential

for the analysis. Careful choice of types in the C implementation leads to fewer and simpler verification conditions. Also, Gappa requires user input to identify critical subexpressions when performing bisection.

The work presented here relies on several tools: the PVS interactive prover, the Frama-C analyzer, and the automatic provers AltErgo and Gappa. These tools are based on rigorous mathematical foundations and have been used in the verification of several industrial and safety-critical systems. In addition, proof certificates for significant parts of the analysis were generated (PVS and Gappa). However, the overall proof chain must be trusted. For instance, AltErgo does not generate any proof certificate that can be checked externally. Furthermore, though some effort has been made to formalize and verify the Frama-C WP plug-in, this endeavor is still incomplete. Nevertheless, the CPR algorithm is relatively simple, containing no complex features such as pointers or loops, and so the generation of verification conditions for CPR can be allegedly trusted.

References

1. Barrett, C., C., T.: CVC3. In: Proceedings of the 19th International Conference on Computer Aided Verification, CAV 2007. pp. 298–302 (2007)
2. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, version 1.12 (2016)
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004)
4. Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Static Analysis and Verification of Aerospace Software by Abstract Interpretation. Foundations and Trends in Programming Languages 2(2-3), 71–190 (2015)
5. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Let's verify this with Why3. International Journal on Software Tools for Technology Transfer 17(6), 709–727 (2015)
6. Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P.: Wave equation numerical resolution: A comprehensive mechanized proof of a C program. Journal of Automatic Reasoning 50(4), 423–456 (2013)
7. Boldo, S., Filliâtre, J.C.: Formal verification of floating-point programs. In: Proceedings of ARITH18 2007. pp. 187–194. IEEE Computer Society (2007)
8. Boldo, S., Marché, C.: Formal verification of numerical programs: From C annotated programs to mechanical proofs. Mathematics in Computer Science 5(4), 377–393 (2011)
9. Bouissou, O., Conquet, E., Cousot, P., Cousot, R., Feret, J., Goubault, E., Ghorbal, K., Lesens, D., Mauborgne, L., Miné, A., Putot, S., Rival, X., Turin, M.: Space Software Validation using Abstract Interpretation. In: Proceedings of the International Space System Engineering Conference, Data Systems in Aerospace (DASIA 2009). pp. 1–7. ESA publications (2009)
10. Chen, L., Miné, A., Cousot, P.: A Sound Floating-Point Polyhedra Abstract Domain. In: Proceedings of the 6th Asian Symposium on Programming Languages and Systems, APLAS 2008. Lecture Notes in Computer Science, vol. 5356, pp. 3–18. Springer (2008)

11. Code of Federal Regulations: Automatic Dependent Surveillance-Broadcast (ADS-B) Out, 91 c.f.r., section 225 (2015)
12. Conchon, S., Contejean, E., Kanig, J., Lescuyer, S.: CC(X): Semantic Combination of Congruence Closure with Solvable Theories. Electronic Notes in Theoretical Computer Science 198(2), 51 – 69 (2008)
13. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival: The ASTREÉ Analyzer. In: Proceedings of the 14th European Symposium on Programming (ESOP 2005). Lecture Notes in Computer Science, vol. 3444, pp. 21–30. Springer (2005)
14. Delmas, D., Souyris, J.: Astrée: From research to industry. In: Proceedings of the 14th International Symposium on Static Analysis, SAS 2007. pp. 437–451 (2007)
15. de Dinechin, F., Lauter, C., Melquiond, G.: Certifying the floating-point implementation of an elementary function using Gappa. IEEE Trans. on Computers 60(2), 242–253 (2011)
16. Dutle, A., Moscato, M., Titolo, L., Muñoz, C.: 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)
17. European Commission: Commission Implementing Regulation (EU) 2017/386 of 6 march 2017 amending Implementing Regulation (EU) No 1207/2011, C/2017/1426 (2017)
18. Filliâtre, J.C., Marché, C.: Multi-prover verification of C programs. In: Proceedings of the 6th International Conference on Formal Engineering Methods, ICFEM 2004. Lecture Notes in Computer Science, vol. 3308, pp. 15–29. Springer (2004)
19. Goodloe, A., Muñoz, C., Kirchner, F., Correnson, L.: Verification of numerical programs: From real numbers to floating point numbers. In: Proceedings of NFM 2013. Lecture Notes in Computer Science, vol. 7871, pp. 441–446. Springer (2013)
20. Goubault, E., Putot, S.: Static analysis of numerical algorithms. In: Proceedings of SAS 2006. Lecture Notes in Computer Science, vol. 4134, pp. 18–34. Springer (2006)
21. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Aspects of Computing 27(3), 573–609 (2015)
22. Marché, C.: Verification of the functional behavior of a floating-point program: An industrial case study. Science of Computer Programming 96, 279–296 (2014)
23. Marché, C., Moy, Y.: The Jessie Plugin for Deductive Verification in Frama-C (2017)
24. Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In: Proceedings of the 13th European Symposium on Programming Languages and Systems, ESOP 2004. Lecture Notes in Computer Science, vol. 2986, pp. 3–17. Springer (2004)
25. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
26. Owre, S., Rushby, J., Shankar, N.: PVS: A prototype verification system. In: Proceedings of CADE 1992. Lecture Notes in Artificial Intelligence, vol. 607, pp. 748–752. Springer (1992)
27. RTCA SC-186: Minimum Operational Performance Standards for 1090 MHz extended squitter Automatic Dependent Surveillance - Broadcast (ADS-B) and Traffic Information Services - Broadcast (TIS-B) (2009)