Under consideration for publication in Formal Aspects of Computing

Formal Analysis of the Compact Position Reporting Algorithm

Aaron Dutle[1], Mariano Moscato[2], Laura Titolo[2], César Muñoz[1], Gregory Anderson[3], François Bobot[4]

[1]NASA Langley Research Center, Hampton, VA, USA
[2]National Institute of Aerospace, Hampton, VA, USA
[3]University of Texas at Austin, Austin, TX, USA
[4]CEA List, Paris

Abstract. The Automatic Dependent Surveillance-Broadcast (ADS-B) system allows aircraft to communicate current state information, including position and velocity messages, to other aircraft in their vicinity and to ground stations. The Compact Position Reporting (CPR) algorithm is the ADS-B protocol responsible for the encoding and decoding of aircraft positions. CPR is sensitive to computer arithmetic since it relies on functions that are intrinsically unstable such as floor and modulus. In this paper, a formal verification of the CPR algorithm is presented. In contrast to previous work, the algorithm presented here encompasses the entire range of message types supported by ADS-B. The paper also presents two implementations of the CPR algorithm, one in double-precision floating-point and one in 32-bit unsigned integers, which are both formally verified against the real-number algorithm. The verification proceeds in three steps. For each implementation, a version of CPR, which is simplified and manipulated to reduce numerical instability and leverage features of the datatypes, is proposed. Then, the Prototype Verification System (PVS) is used to formally prove real conformance properties, which assert 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 software conformance properties, which say that the software implementation of the improved algorithm is correct with respect to its idealized real-number counterpart. In concert, the two properties guarantee that the implementation meets the original specification. The two implementations will be included in the revised version of the ADS-B standards document as the reference implementation of the CPR algorithm.

Keywords: Formal Analysis, Finite Precision Computation, Compact Position Reporting Algorithm, Frama-C, PVS

Correspondence and offprint requests to: Aaron Dutle, NASA Langley Research Center, Mail Stop 234, Hampton, VA 23681-2199, USA, e-mail: aaron.m.dutle@nasa.gov

1. Introduction

The Automatic Dependent Surveillance - Broadcast (ADS-B) protocol [RTC09] is a fundamental component of the next generation of air transportation systems. It is intended to augment or replace existing 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 [Cod15] and Europe [Eur17]. Thousands of aircraft are currently equipped with ADS-B.[1]

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.[2] If raw latitude and longitude data were expressed in 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 the aircraft position in 35 bits such that, for airborne applications, the decoded position 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.

Prior to the current work, a formal analysis of the *airborne* version of the CPR algorithm was conducted [DMTM17]. 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 was formally proven correct for a slightly tightened set of requirements. Nevertheless, even assuming these more restrictive requirements, subtleties in the implementation of the algorithm as well as accumulated error can cause incorrect computation. For instance, using a standard single-precision floating-point implementation of CPR for the position whose latitude is $-77.368°$ and longitude is $180°$, the recovered position differs from the original one by approximately 1500 nautical miles. Titolo et al. [TMM$^+$18] proposed an alternative version of the *airborne* version of the CPR algorithm. This version was proven correct when implemented using double-precision floating-point arithmetic.

This paper extends the previous work done on CPR ([DMTM17], [TMM$^+$18]) in several directions. First, the formal verification of the CPR algorithm under a set of tightened requirements is extended from covering only the *airborne* case, as in [DMTM17], to cover all four types of position messages that CPR allows (*coarse, intent, airborne,* and *surface*). The other main contributions of the paper are two implementations of the the CPR algorithm. The first is a double-precision floating-point implementation of the *airborne* algorithm, which was first detailed in [TMM$^+$18]. The second is a single-precision fixed-point implementation using unsigned integers, which implements all types of position messages. Both versions include algorithmic simplifications with respect to the original protocol presented in the ADS-B standard. This reduces the accumulated error in the floating-point implementation, and makes for faster and more straightforward computation in the integer version. Both implementations are verified using a similar approach. Each implementation is written in C and its real number counterpart is specified using the *logic* structure in ANSI/ISO C Specification Language (ACSL). Frama-C [KKP$^+$15] is used to prove that the encoding implementation always returns the same value as its idealized counterpart, and that the decoded position is close enough (in a sense that will be made precise later). The Frama-C WP plug-in is used to generate verification conditions ultimately discharged with the aid of the automatic solvers Gappa [dDLM11] and Alt-Ergo [CCKL08]. Finally, the Prototype Verification System (PVS) [ORS92] is used to formally prove that the real counterpart of the proposed CPR implementation is mathematically equivalent to the original algorithm defined in the standard [RTC09]. It follows that the correctness results presented in [DMTM17] and extended in this paper also hold for the proposed versions of CPR. The PVS formalization of this equivalence, as well as the ACSL annotated C code for both implementations, is available at `https://github.com/nasa/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 [DMTM17] are summarized. This section also discusses

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

[2] There are 35 bits available for *airborne* and *surface* message types. The *intent* and *coarse* messages use fewer bits for position, as described in section 2.

the extension to the other position formats. Section 3 details the verification approach used by both implementations of CPR. The double-precision floating-point implementation of CPR is presented in section 4, along with the results ensuring its mathematical equivalence with respect to the original algorithm. Section 5 presents the single-precision 32-bit unsigned integer implementation of CPR, and the related equivalence to the original algorithm. Related work is discussed in section 6. Finally, section 7 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 [RTC09]. In addition, the results of [DMTM17] proving the correctness of the decoding under a tightened set of requirements are extended to include all types of CPR messages. The CPR algorithm is used to produce four different types of position messages, depending on the aircraft state and the message type to be sent. The four types of messages are *coarse*, *intent*, *airborne*, and *surface*. The procedure for encoding and decoding that CPR uses is generally similar across the types, with different parameters and some small changes to the basic algorithm depending on the type. The goal of CPR is to encode latitude and longitude in a limited number of bits, while keeping a position resolution higher than would be possible by encoding raw latitude and longitude. The *coarse*, *intent*, *airborne*, and *surface* messages allow 12, 14, 17 and 17 bits (respectively) for each of latitude and longitude, and are meant to decode to a position within 164, 41, 5, and 1.25 meters (respectively) of the aircraft's actual GPS position. CPR is based on the fact that transmitting the entire latitude and longitude in each broadcast message is generally inefficient, since the higher order bits of such a transmission are very unlikely to change over a short period of time. In order to leverage this observation, only the least significant bits of the position are encoded and transmitted. Two different techniques for decoding are used to recover the higher order bits.

CPR uses a global 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^{nb} parts, called *bins*, where nb corresponds to the message type.[3] The value of nb is either 12, 14, 17, or 19 for message types *coarse, intent, airborne,* and *surface*, respectively. Throughout the remainder of the paper, the parameter nb will be used and the message type is inferred from its value. Figure 1 shows how latitude is divided into 60 zones (for the even subdivision) or into 59 zones (for the odd subdivision), and also how each zone is then divided into 2^{nb} bins.

The CPR encoding procedure transforms degree coordinates into CPR coordinates and is parametric with respect to the chosen subdivision (even or odd) and message type. A CPR message coordinate is exactly the number corresponding to the bin where the target is located. In isolation, such a message corresponds to many different positions on the globe, one in each zone. In order to choose the correct position, the reciever must somehow determine the correct zone. This 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 compute a coordinate that corresponds to the centerline of the bin where the target is located, as in figure 1. In a latitude zone (respectively longitude zone), all the latitudes (respectively longitudes) inside a *bin* have the same encoding. If the recovered latitude (respectively longitude) is taken to correspond to the *bin centerline*, then the difference between a given position and the result of a correct encoding and decoding should be less than or equal to half of the size of a bin.

The modulus function, which is in some cases assumed to be for integers only, is here taken to apply to real numbers as well, and is 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. Therefore, no rounding error occurs. The results presented in this section extend the work done in [DMTM17], where the airborne version of the algorithm was first analyzed. All of the results stated have been formally proven in PVS [ORS92].

[3] In fact, the *bin centerlines* separate each zone into 2^{nb} pieces, and the bins extend half of this distance from each centerline. This has the effect of placing a half-bin at the top and bottom of each zone, each of which correspond to a bin centerline of 0, as in figure 1. This makes the mathematics slightly more difficult, but has the aesthetically pleasing feature of allowing 0 degrees latitude as a return value.

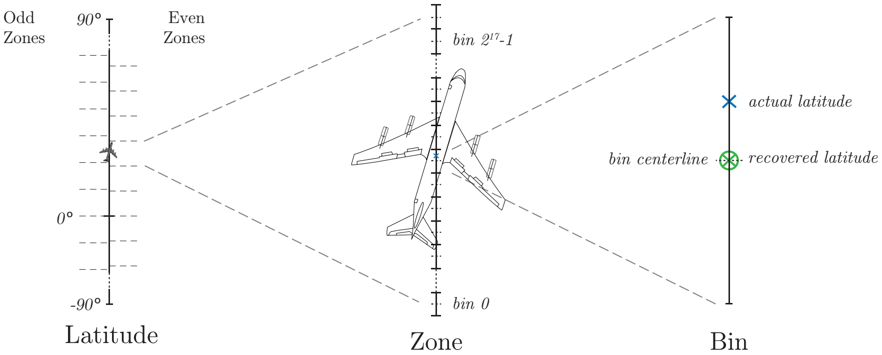

Fig. 1. CPR latitude coordinate system.

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. Coarse messages are 25 bits, intent messages are 28 bits, and airborne and surface messages are each 35 bits.

The bits composing the CPR message are grouped into three parts. One bit determines the format[4] (0 for even and 1 for odd). The remaining bits are divided evenly into a latitude message and a longitude message. This provides for 12, 14, 17, and 17 bits for each of longitude and latitude for *coarse, intent, airborne,* and *surface* message types, respectively.

Letting $i \in \{0,1\}$ 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. Let nb denote the number of bins for the encoding type, again noting nb is 12, 14, 17, or 19 for *coarse, intent, airborne,* or *surface* messages, respectively.

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

In formula (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^{nb} gives a value between 0 and 2^{nb}, while $\lfloor x + \frac{1}{2} \rfloor$ rounds a number x to the nearest integer. The external modulo ensures that the encoded latitude fits in the number of bits allocated to the message type. It may appear that this final truncation can discard some useful information. However, for all types except the surface message, it only affects half of a bin at the top of a zone, which is accounted for by the adjacent zone. For the surface message type, it actually removes the highest two bits of information. 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 latitude value $lat \in [-90, 90]$, the NL value is used to compute the longitude zone size as follows.

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

Note that the denominator in the above expression uses the max operator when NL is 1, which occurs for

[4] Except in the case of *intent* messages, which are always calculated using *even* format, and hence do not include a format bit, and only use local decoding.

latitudes beyond ±87 degrees. In this case, there is only one longitude zone, and so longitude encodings coincide for even and odd formats.

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

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

Let $\mathcal{BN}$ denote the domain of bin numbers, composed of the integers in the interval $[0, 2^{\max\{nb,17\}} - 1]$. The following lemma ensures the message is of the proper length. The proof of this lemma uses only simple properties of the modulus function.

Lemma 2.1 *Given $i \in \{0, 1\}$, $lat \in [-90, 90]$, and $lon \in [0, 360]$, $latEnc(i, lat) \in \mathcal{BN}$ 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 (except for surface messages, which have 4 equally spaced possible corresponding bins inside each zone). In order to unambiguously compute the decoded position, it suffices to determine the zone (quarter zone in the case of surface messages). To this aim, 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 (one-quarter-zone interval for surface) centered around a given reference position does not contain more than one occurrence of a centerline with a particular bin number. Therefore, as long as the target is known to be close enough to the reference position (slightly less than half a zone, or an eighth of a zone for surface messages), decoding can be performed uniquely.

Decoding uses the value of $dlat_i$ in its calculation, but the *surface* decoding uses an alternate value, to account for the four bits of the message removed at the end of encoding. In order to distinguish the two, define

$$dlat_i^* = \begin{cases} 90/(60 - i) \text{ for } \textit{surface} \text{ decoding} \\ 360/(60 - i) \text{ otherwise.} \end{cases} \tag{2.4}$$

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).

$$latZin_{\mathsf{L}}(i, YZ_i, lat_{ref}) = \left\lfloor \frac{lat_{ref}}{dlat_i^*} \right\rfloor + \left\lfloor \frac{1}{2} + \frac{\mathsf{mod}\,(lat_{ref}, dlat_i^*)}{dlat_i^*} - \frac{YZ_i}{2^{\max\{nb,17\}}} \right\rfloor. \tag{2.5}$$

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.

$$rlat_{\mathsf{L}}(i, YZ_i, lat_{ref}) = dlat_i^* \left(latZin_{\mathsf{L}}(i, YZ_i, lat_{ref}) + \frac{YZ_i}{2^{\max\{nb,17\}}} \right). \tag{2.6}$$

This recovered latitude is used to determine the NL value for computing the value of $dlon_i$. As with the latitude, *surface* longitude decoding uses a smaller value for computing with $dlon_i$. Define

$$dlon_i^*(lat) = \begin{cases} 90/\max\{1, NL(lat) - i\} \text{ for } \textit{surface} \text{ decoding} \\ 360/\max\{1, NL(lat) - i\} \text{ otherwise.} \end{cases} \tag{2.7}$$

In the following formulas, $dlon_i^*$ is used as an abbreviation for $dlon_i^*(rlat_{\mathsf{L}}(i, YZ_i, lat_{ref}))$. 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.

$$lonZin_{\mathsf{L}}(i, XZ_i, lon_{ref}, rlat) = \left\lfloor \frac{lon_{ref}}{dlon_i^*} \right\rfloor + \left\lfloor \frac{1}{2} + \frac{\mathsf{mod}\,(lon_{ref}, dlon_i^*)}{dlon_i^*} - \frac{XZ_i}{2^{\max\{nb,17\}}} \right\rfloor. \tag{2.8}$$

$$rlon_{\mathsf{L}}(i, XZ_i, lon_{ref}, rlat) = dlon_i^* \left(lonZin_{\mathsf{L}}(i, XZ_i, lon_{ref}, rlat) + \frac{XZ_i}{2^{\max\{nb,17\}}} \right). \tag{2.9}$$

When the difference between original and reference latitude (respectively longitude) is less than half zone size (one eight of a zone for *surface* messages) minus half of a 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 2.2 (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 $|lat - lat_{ref}| < \frac{dlat_i^*}{2} - \frac{dlat_i}{2^{nb+1}}$,*

$$|lat - rlat_{\mathsf{L}}(i, latEnc(i, lat), lat_{ref})| \leq \frac{dlat_i}{2^{nb+1}}.$$

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 $|lon - lon_{ref}| < \frac{dlon_i^*(rlat)}{2} - \frac{dlon_i(rlat)}{2^{nb+1}}$,

$$|lon - rlon_{\mathsf{L}}(i, lonEnc(i, rlat, lon), lon_{ref}, rlat)| \leq \frac{dlon_i(rlat)}{2^{nb+1}}.$$

This result was first formally proven in PVS for the *airborne* message type in [DMTM17]. The extension to the other message types is original to this paper. It should be noted here that the *surface* message type is able to increase the accuracy of the recovered position by a factor of 4 while keeping the same message size as *airborne* messages, but pays a penalty in the distance that a valid reference position can be from the actual position for valid decoding. This is acceptable because the rate distances are covered is much lower when an aircraft is on the ground.

2.3. Global Decoding

Global decoding is used when a valid reference position is unknown. This can occur when either a target is first encountered or when messages have not been received for a significant amount of time. For message types *coarse, intent, and airborne*, global decoding returns a unique position on the globe. For *surface* messages, the algorithm returns a unique position (in degrees) in the range $[0, 90]$ for each of latitude and longitude. This must be resolved into the correct actual position by adding or subtracting multiples of 90 degrees based on known information, for example the location of the receiving aircraft. Similar to the local decoding case, the correct zone in which the encoded position lies (modulo 90 degrees in the case of *surface*) has to be determined. To accomplish this, the global decoding uses a pair of messages with 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 the proper value of nb for the message type, and two integers $YZ_0, YZ_1 \in \mathcal{BN}$, the zone index number for the latitude is computed as follows.

$$latZin_{\mathsf{G}}(YZ_0, YZ_1) = \left\lfloor \frac{59\,YZ_0 - 60\,YZ_1}{2^{\max\{nb,17\}}} + \frac{1}{2} \right\rfloor. \tag{2.10}$$

Using either $i \in \{0,1\}$, the recovered latitude is calculated as shown below. In practice, the most recent message received is the one used to produce a position.

$$rlat_{\mathsf{G}}(i, YZ_0, YZ_1) = dlat_i^* \left(\mathsf{mod}\left(latZin_{\mathsf{G}}(YZ_0, YZ_1), 60 - i\right) + \frac{YZ_i}{2^{\max\{nb,17\}}} \right). \tag{2.11}$$

For the global decoding of a longitude, it is essential to check that the even and odd longitude messages being used were calculated with the same NL value. To this aim, *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 the common value of $NL(rlat_{\mathsf{G}}(i, YZ_0, YZ_1))$ for $i = 0, 1$.

$$lonZin_{\mathsf{G}}(XZ_0, XZ_1) = \left\lfloor \frac{(NL-1)XZ_0 - (NL)XZ_1}{2^{\max\{nb,17\}}} + \frac{1}{2} \right\rfloor. \tag{2.12}$$

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.

$$rlon_{\mathsf{G}}(i, XZ_0, XZ_1) = dlon_i^* \left(\mathsf{mod}\,(latZin_{\mathsf{G}}(XZ_0, XZ_1), nl_i) + \frac{XZ_i}{2^{\max\{nb,17\}}} \right). \tag{2.13}$$

The *zone offset* is the difference between the length of an even and the length of an odd zone. For latitude the zone offest is defined as $ZO_{lat} = dlat_1^* - dlat_0^*$, while for longitude, given a latitude $rlat$, the zone offset 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 can be proven correct, meaning that (modulo 90 degrees for *surface* messages) the difference between the original and recovered latitude and longitude are each at most half the size of a bin.

Theorem 2.3 (Global Decoding Correctness) *Given nb corresponding to the desired message type, and* $i \in \{0, 1\}$*, for all* lat_0*,* $lat_1 \in [-90, 90]$ *such that* $|lat_0 - lat_1| < \frac{ZO_{lat}}{2} - \frac{dlat_1}{2^{nb}}$*,*

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

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* $|lon_0 - lon_1| < \frac{ZO_{lon}}{2} - \frac{dlon_1(rlat_i)}{2^{nb}}$*,*

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

For the *surface* case, the positions are all assumed to be reduced modulo 90, except in the calculation of the NL value from the recovered latitudes. For these, the actual latitude for calculating is chosen to be the one nearest to the receiver. Since the possible choices are separated by 90 degrees, this is assumed to be sufficient. As in the case of local decoding, this result was proven for the *airborne* message type in [DMTM17], with the extension to the other three types being original to the present work.

Note also that the global decoding requires the original positions that were encoded to be *much* closer together than the local decoding. In the local case, the encoded position and the reference were required to be within a half a zone (one eighth for *surface*), which is approximately 180 nautical miles (45 for *surface*). For global decoding, the original positions are required to be within a half of a *zone offset*, which is approximately 3 nautical miles (0.75 nautical miles for *surface*). Here again, *surface* messages provide more accuracy than *airborne* messages using the same message size, but the requirements for correct decoding become more strict.

3. Verification Approach

This section presents the verification approach used to prove that each implementation of the CPR algorithm is precise enough to satisfy its operational requirements. In order to do so, several issues must be addressed, and assumptions made.

The first issue is that the algorithm as specified in the standards document takes real number latitude and longitude values as inputs. While this is fine for an idealized algorithm, an actual implementation must take some approximation of these values as input. For both implementations, the choice was made to use a format for the input values called *32-bit angular weighted binary* (AWB), a standard format for expressing geographical positions, used by GPS manufacturers and many others. Such a coordinate is a 32-bit integer in the interval $[0, 2^{32} - 1]$. An angular weighted binary value $\hat{x}$ corresponds to the latitude or longitude value of $\frac{360\hat{x}}{2^{32}}$ degrees (where negative latitudes are identified with their value modulo 360). An AWB value may be calculated from a real degree input in different ways by different parties, i.e., by truncation or rounding, but when taken as inputs, AWB values are assumed to correspond to exact positions. In the following, $\mathcal{AWB}$ denotes the domain of AWB numbers and a hat on a variable (as above) is used to emphasize that a given variable denotes an AWB value.

A similar issue is that the return value from decoding a CPR message is a real number, though in this case, is always one of a finite set of discrete rational values, corresponding to bin centerlines. Each possible output from a decoded CPR latitude message is of the form

$$dlat_i^*(k + z/2^{\max\{nb,17\}}),$$

where $k \in \mathbb{Z}$ is a zone number, $0 \le z < 2^{\max\{nb,17\}}$ is a latitude message, and the result is restricted to $[-90, 90]$. The set of these possible values will be denoted as $\mathcal{RLAT}$. Likewise for longitude, the possible outputs take the form

$$dlon_i^*(rlat)(k + z/2^{\max\{nb,17\}}),$$

where $rlat \in \mathcal{RLAT}$, $k \in \mathbb{Z}$ is a zone number, $0 \le z < 2^{\max\{nb,17\}}$ is a latitude message, and the result is restricted to $[0, 360]$. In the following, the set of all possible longitude outputs will be denoted $\mathcal{RLON}$.

While the outputs from decoding are finite, discrete, and rational, they are *not* all exactly expressible as double-precision floating-point values, nor as values in $\mathcal{AWB}$. The two implementations handle this issue in different ways. The double-precision floating-point implementation returns a floating-point value, and this value is proven to be close to the value computed by the real number counterpart in a way that will be made precise in the sequel. The unsigned integer implementation returns the value from $\mathcal{AWB}$ which is the closest such value to the result of the real number calculation. Along the way, this rounding is shown to be compatible with functions that must use the decoded output in further computation (particularly in the definition of the NL table).

The general approach to the verification of each implementation is depicted in figure 2. The approach relies on several verification tools. It uses Frama-C as the main engine, employing several of the provers that work with it, such as Gappa and Alt-Ergo. It also uses PVS to link the C code to the real-number version of CPR that was proven to encode and decode correctly in section 2. These tools and their uses are described below.

Frama-C is a tool suite that collects several static analyzers for the C language. C programs can be annotated with ACSL [BCF+16] annotations that state function contracts, pre- and postconditions, assertions, and invariants. For this work, one of the important annotations that ACSL supports is the *logic* construct. This allows the user to define custom functions on the real numbers, and use them in subsequent ACSL annotations. 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 [dDLM11] 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 some rewriting rules that will 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.

Frama-C also has the ability to translate verification conditions, and of particular interest ACSL *logic* functions, into PVS syntax. This allows the user to employ PVS in the proof of verification conditions if desired. In the present work, PVS is used to provide a verification link between an ACSL *logic* function,

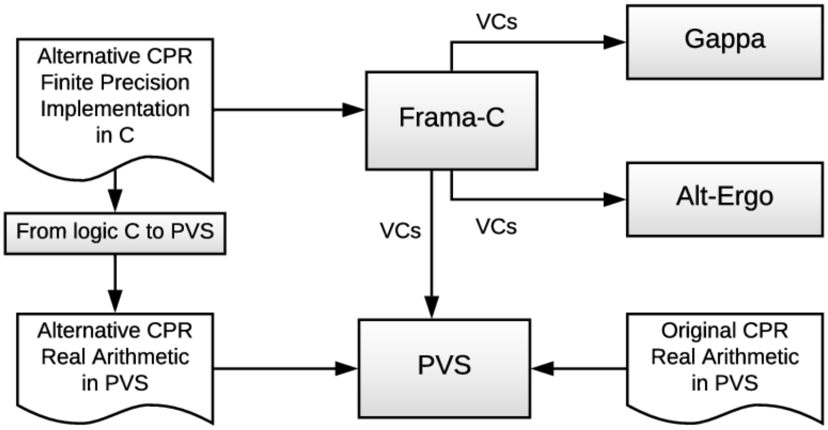

Fig. 2. Verification approach.

and the formally verified real number version of CPR discussed in section 2. With Frama-C providing the link between the ACSL *logic* function and the actual C implementation, the C implementation inherits the properties of the formally verified PVS version.

The general workflow is as follows. To begin, each of the alternate versions of CPR (described in section 4 and section 5) is implemented in C. Each C function is annotated with several ACSL statements. First, a corresponding *logic* ACSL function is specified that mirrors the C function, but operates over the real numbers. Pre- and postconditions are added to the C program, along with additional intermediate assertions which are added after specific program instructions to help the external provers in reasoning[5]. This program is analyzed by the WP plug-in of Frama-C and a set of verification conditions (VCs) is generated. These verification conditions are then discharged by the external provers Gappa and Alt-Ergo. Each ACSL *logic* counterpart that is specified in the annotation is translated by Frama-C into a PVS function. PVS is then used to formally verify the equivalence of these generated functions with respect to the PVS implementation of the CPR algorithm that has been proven to be correct in section 2.

For the purposes of this paper, a proof that connects an ACSL *logic* function considered over real numbers to the proven implementation of CPR from section section 2 is called a *real conformance* property, as it provides a link between two algorithms over the real numbers. An assertion proven in Frama-C that claims a link between the software version of a CPR function and the *logic* version of the same function is denoted a *software conformance* property. Taken together, a real conformance property and a software conformance property provide a link between a software implementation and the real number implementation proven to be correct from section 2.

Each implementation includes real and software conformance properties for encoding, local decoding, and global decoding. They are detailed for each implementation in section 4 and section 5. In each of the these sections, a C-implemented function (floating-point or integer) will be represented with a tilde, as $\tilde{f}$, while the corresponding real number function f specified in an ACSL *logic* construct will appear without one.

[5] As anyone familiar with Frama-C will recognize, these intermediate assertions are often identified and added incrementally, when the provers are not able to discharge a certain verification condition.

4. A Floating-Point Implementation of CPR

In this section, a floating-point implementation of the airborne CPR protocol and its verification is presented. The computations performed in this implementation leverage several 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.[6] Other simplifications are applied to reduce the number of operations, especially the modulus 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 $|\lfloor x \rfloor - \lfloor \tilde{x} \rfloor| = 1$.

4.1. Floating-Point 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 1 $fpLatEnc(i, \widehat{lat})$

```
$nz \leftarrow 60 - i$
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 $fpLonEnc(i, NL, \widehat{lon})$

```
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 $\lfloor tmp_1 \rfloor - 2^{17} * \lfloor tmp_2 \rfloor$
```

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 $NL - i$ 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 asserts the *real conformance* property for the proposed floating-point encoding with respect to the encoding described in subsection 2.1.

Theorem 4.1 (Real Conformance, Floating-Point Encoding) *Let* $lat \in [-90, 90]$, $lon \in [0, 360]$, $\widehat{lat} \in \mathcal{AWB}$, $\widehat{lon} \in \mathcal{AWB}$, *and* $i \in \{0, 1\}$. *If* $lat = \frac{360\widehat{lat}}{2^{32}}$, $lon = \frac{360\widehat{lon}}{2^{32}}$, *and* $NL = NL(rlat(lat))$, *then*

$$fpLatEnc(i, \widehat{lat}) = latEnc(i, lat) \text{ and}$$

$$fpLonEnc(i, NL, \widehat{lon}) = lonEnc(i, lat, lon).$$

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

[6] The verification tools must still determine and account for when overflow and underflow may occur, but when it is determined that no overflow or underflow happen, they use the fact that no additional error is introduced.

Lemma 4.2 *Let $lat \in [-90, 90]$, $lon \in [0, 360]$, and $i \in \{0, 1\}$,*

$$latEnc(i, lat) = \left\lfloor 2^{17} \frac{\mathsf{mod}\left(lat + 2^{-18} dlat_i, dlat_i\right)}{dlat_i} \right\rfloor \text{ and}$$

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

The following two results, which have been formally proven correct in [DMTM17], are also used. When the modulo operator is divided by its second argument, the following simplification can be applied.

$$\frac{\mathsf{mod}\,(a, b)}{b} = \frac{a}{b} - \left\lfloor \frac{a}{b} \right\rfloor. \tag{4.1}$$

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

$$\lfloor x + z \rfloor = \lfloor x \rfloor + z. \tag{4.2}$$

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

$$\left\lfloor 2^{17} \frac{\mathsf{mod}\left(l + 2^{-18} dl, dl\right)}{dl} \right\rfloor = \left\lfloor 2^{17} \frac{l}{dl} + \frac{1}{2} \right\rfloor - 2^{17} \left\lfloor \frac{l}{dl} + \frac{1}{2^{18}} \right\rfloor. \tag{4.3}$$

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.

$$\left\lfloor 2^{17} \frac{l}{dl} + \frac{1}{2} \right\rfloor - 2^{17} \left\lfloor \frac{l}{dl} + \frac{1}{2^{18}} \right\rfloor = \left\lfloor (\hat{l} \cdot nz + 2^{14}) 2^{-15} \right\rfloor - 2^{17} \left\lfloor (\hat{l} \cdot nz + 2^{14}) 2^{-32} \right\rfloor. \tag{4.4}$$

Because the encoding produces an integer which corresponds to the bin of the actual position, the floating-point encoding of CPR will satisfy the *software conformance* property if it returns exactly the same value as 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 4.3 (Software Conformance, Floating-Point Encoding) *Let $\widehat{lat} \in \mathcal{AWB}$, $\widehat{lon} \in \mathcal{AWB}$, NL be an integer in the range $[1, 59]$, and $i \in \{0, 1\}$. Then*

$$fpLatEnc(i, \widehat{lat}) = \widetilde{fpLatEnc}(i, \widehat{lat}) \text{ and}$$

$$fpLonEnc(i, NL, \widehat{lon}) = \widetilde{fpLonEnc}(i, NL, \widehat{lon}).$$

In order to prove this theorem, 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 automatically proved by Gappa since the computation only involves operations between integers and multiplications by powers of 2. Since the floor operation is applied to expressions that do not carry any round-off error, the computation of the floor is also exact and, therefore, theorem 4.3 holds.

Taken together, the *real* and *software conformance* theorems for encoding guarantee that the floating-point encoding algorithm calculates exactly the same value as the formally verified real-number version.

4.2. Floating-Point 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. Similar to the encoding algorithm, it is necessary to shift the AWB to correctly represent the latitudes between −90 and 0 degrees. Similarly, algorithm 4 recovers the longitude bin centerline. 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

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

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

Algorithm 4 $\mathit{fpDecLon}_{\mathsf{L}}(i, \mathit{NL}, \widehat{lon}, \mathit{XZ})$

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

degrees, *dl* be the zone size, and *enc* the 17-bit encoding. By applying Equations (4.1) and (4.2), the latitude (respectively, longitude) zone index number formulas (2.5) and (2.8) 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, the *real conformance* property of theorem 4.4 directly follows.

Theorem 4.4 (Real Conformance, Floating-Point Local Decoding) *Let $i \in \{0,1\}$, $\mathit{YZ}_i \in \mathcal{BN}$, $\mathit{XZ}_i \in \mathcal{BN}$, $\widehat{lat}_{ref} \in \mathcal{AWB}$, and $\widehat{lon}_{ref} \in \mathcal{AWB}$. If $lat_{ref} = \frac{360\widehat{lat}_{ref}}{2^{32}}$ and $lon_{ref} = \frac{360\widehat{lon}_{ref}}{2^{32}}$, then*

$$rlat_{\mathsf{L}}(i, \mathit{YZ}_i, lat_{ref}) = \mathit{fpDecLat}_{\mathsf{L}}(i, \mathit{YZ}_i, \widehat{lat}_{ref}) \text{ and}$$

$$rlon_{\mathsf{L}}(i, \mathit{XZ}_i, lon_{ref}, lat_{ref}) = \mathit{fpDecLon}_{\mathsf{L}}(i, \mathit{NL}(\mathit{fpDecLat}_{\mathsf{L}}(i, \mathit{YZ}_i, \widehat{lat}_{ref})), \widehat{lon}_{ref}, \mathit{XZ}_i).$$

The *software conformance* property for the floating-point implementation, for both global and local decoding, employs a concrete bound for how far the real number algorithm and the software implementation can differ. Note that theorem 2.2 and theorem 2.3 state that the original coordinate (latitude or longitude) and the recovered bin centerline differ 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 less than 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 is considered correct when the recovered coordinate differs from the bin centerline by less than half the size of a bin.

Recall from section 2 that the airborne bin size for the latitude even configuration is approximatively 4.578×10^{-5} degrees, and for the odd one is 4.655×10^{-5} degrees. Also note that longitude zones, regardless of the *NL* value, contain *more* degrees per zone than latitude zones. In the software conformance theorems for floating-point decoding (local and global), the lower bound for half bin size of 2.2888×10^{-5} degrees is used.

Theorem 4.5 (Software Conformance, Floating-Point Local Decoding) *Let $i \in \{0,1\}$, $\mathit{YZ}_i \in \mathcal{BN}$, $\mathit{XZ}_i \in \mathcal{BN}$, $\widehat{lat}_{ref} \in \mathcal{AWB}$, and $\widehat{lon}_{ref} \in \mathcal{AWB}$. Then*

$$|\mathit{fpDecLat}_{\mathsf{L}}(i, \widehat{lat}_{ref}, \mathit{YZ}_i) - \widetilde{\mathit{fpDecLat}}_{\mathsf{L}}(i, \widehat{lat}_{ref}, \mathit{YZ}_i)| \leq 2.2888 \times 10^{-5} \text{ and}$$

$$|\mathit{fpDecLon}_{\mathsf{L}}(i, \mathit{NL}, \widehat{lon}_{ref}, \mathit{XZ}_i) - \widetilde{\mathit{fpDecLon}}_{\mathsf{L}}(i, \widetilde{\mathit{NL}}, \widehat{lon}_{ref}, \mathit{XZ}_i)| \leq 2.2888 \times 10^{-5},$$

where $\mathit{NL} = \mathit{NL}(\mathit{fpDecLat}_{\mathsf{L}}(i, \mathit{YZ}_i, \widehat{lat}_{ref}))$ and $\widetilde{\mathit{NL}} = \widetilde{\mathit{NL}}(\widetilde{\mathit{fpDecLat}}_{\mathsf{L}}(i, \mathit{YZ}_i, \widehat{lat}_{ref}))$.

To verify this software conformance property for local decoding, 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 automatically 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*), which involves a division. However, Gappa is able to prove that the propagation of this error in the result is bounded by half bin size as stated in theorem 4.5.

Algorithm 5 $\mathit{fpDecLat}_{\mathsf{G}}(i, YZ_0, YZ_1)$

```
  $dlat \leftarrow 360/(60-i)$
  $zin = \lfloor (59 * YZ_0 - 60 * YZ_1 + 2^{16}) * 2^{-17} \rfloor$
  **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**
```

Algorithm 6 $\mathit{fpDecLon}_{\mathsf{G}}(i, NL, XZ_0, XZ_1)$

```
  **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**
```

4.3. Floating-Point 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.

The real conformance property, theorem 4.6, directly follows from simple algebraic manipulations. The sum of the two fractions inside the floor in formula (2.10) is explicitly calculated and the modulo in formulas (2.11) and (2.13) is expanded.

Theorem 4.6 (Real Conformance, Floating-Point Global Decoding) *Let* $i \in \{0,1\}$, $YZ_i \in \mathcal{BN}$, $XZ_i \in \mathcal{BN}$, *and* $nl = NL(\mathit{fpDecLat}_{\mathsf{G}}(i, YZ_0, YZ_1))$. *Then*

$$\mathit{rlat}_{\mathsf{G}}(i, YZ_0, YZ_1) = \mathit{fpDecLat}_{\mathsf{G}}(i, YZ_0, YZ_1) \text{ and}$$
$$\mathit{rlon}_{\mathsf{G}}(i, XZ_0, XZ_1) = \mathit{fpDecLon}_{\mathsf{G}}(i, NL, XZ_0, XZ_1).$$

The verification of the software conformance property for global decoding involves more complex reasoning. Similar 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 automatically 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. This hint is added to the Frama-C generated Gappa files. To the best of authors' knowledge, the direct specification of Gappa hints is not supported in Frama-C. 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 4.7 as well.

Theorem 4.7 (Software Conformance, Floating-Point Global Decoding) *Let* $i \in \{0,1\}$*,* $YZ_i \in \mathcal{BN}$*, and* $XZ_i \in \mathcal{BN}$*, if* $NL(\mathit{fpDecLat}_{\mathsf{G}}(0, YZ_0, YZ_1)) = NL(\mathit{fpDecLat}_{\mathsf{G}}(1, YZ_0, YZ_1))$*, then*

$$|\mathit{fpDecLat}_{\mathsf{G}}(i, YZ_0, YZ_1) - \mathit{fp}\widetilde{\mathit{DecLat}}_{\mathsf{G}}(i, YZ_0, YZ_1)| \leq 2.2888 \times 10^{-5} \text{ and}$$

$$|\mathit{fpDecLon}_{\mathsf{G}}(i, NL, XZ_0, XZ_1) - \mathit{fp}\widetilde{\mathit{DecLon}}_{\mathsf{G}}(i, NL, XZ_0, XZ_1)| \leq 2.2888 \times 10^{-5},$$

where $NL = NL(\mathit{fpDecLat}_{\mathsf{G}}(j, YZ_0, YZ_1))$ *and* $\widetilde{NL} = \widetilde{NL}(\mathit{fp}\widetilde{\mathit{DecLat}}_{\mathsf{G}}(j, YZ_0, YZ_1))$ *for* $j = 0, 1$*.*

4.4. A Note on *NL* Calculation

The bound used in the *software conformance* of decoding serves a secondary purpose as well. Because it is strictly less than half a bin, the recovered floating-point values for successive bins are guaranteed to be strictly separated from each other, in the sense that a value strictly between the possible return values for consecutive bins exists. It follows that a new table $\widetilde{NL}$, which takes as input the floating-point latitude resulting from $\mathit{fp}\widetilde{\mathit{DecLat}}_{\mathsf{L}}$, can be computed off-line with enough precision. This can be done as follows. For each transition latitude l in the original *NL* table, the two bin centerlines that surround it can be determined. The point centered between them is thus the transition between the two bins, and is guaranteed to be strictly separated from the possible return values for decoding either of the two flanking bins. This value can be calculated as precise as needed, and serve as the floating-point transition latitude. The only slight downside to this calculation is that separate *NL* tables must be computed for the odd and even formats, since odd and even bin breakpoints generally do not coincide.

5. An Unsigned Integer Implementation of CPR

This section presents another C implementation of CPR and its associated verification. This implementation uses 32-bit unsigned integers throughout the computation. As such, it is suitable for use on hardware with limited capabilities. It also implements the full version of CPR, allowing for *coarse, intent, airborne,* and *surface* message types.

The verification follows the approach described in section 3. In the floating-point implementation, the *logic* functions specified in Frama-C's ACSL operate natively on real numbers, and after some slight simplifications, very closely resembled the original CPR definition. This made the *real conformance* part of that verification relatively simple. The more difficult portion of the verification was in the *software conformance*, where the floating point and real functions had to be proven to compute closely. In the unsigned integer implementation, the ACSL *logic* functions are specified in a manner to exactly mimic the 32-bit unsigned integer computation, using the mod function to mimic overflow wrapping, and the floor function to truncate integer divisions. This makes the *software conformance* of the implementation and the logic functions simpler, since they are built to coincide exactly at every step. On the other hand, this makes the *real conformance* of these *logic* functions with respect to the version of CPR discussed in section 2 much more involved. In order to do this reasoning, a model of 32-bit unsigned integer computation was specified in PVS. The formalization includes functional specifications of all of the standard arithmetic operations (addition, subtraction, multiplication, and division, as well as exponentiation of powers of 2, integer modulus, and shifting operations). When shown in the following, the standard C version of each function will be used. For example, the integer modulus function $\mathrm{mod}(a, b)$ will be shown as $a\%b$.

The implementation uses many of the same mathematical simplifications that were employed in the floating point implementation of section 4 to decrease numerical complexity. It also uses the fact that fixed width unsigned integers never overflow as the result of any of the basic mathematical operations, instead returning the value of the operation modulo 2^{32}.

The limitations of 32-bit integer computation shows in several places in the computation, but is most apparent in expressions of the form $a \cdot b/c$, which occur often in the CPR specification. The issue with such expressions is that if the multiplication is performed first, the result might exceed 2^{32} and wrap around, making the final result incorrect after division. On the other hand, performing (one of the) divisions first loses the lower order bits of the numerator, and so the result is incorrect after multiplication. Because these expressions occur often, and a precise result is needed, the functions $\mathit{MultShiftDiv}(a, c, \mathit{exp})$ and $\mathit{MultDivShift}(a, b, \mathit{exp})$, defined in algorithms 7 and 8, are used in cases when at least one of the values involved is a power of 2.

Algorithm 7 *MultShiftDiv*(a, c, exp)

```
$tmp_1 = 2^{exp} * (a/c)$
$tmp_2 = 2^{exp+1} * (a\%c)$
$tmp_3 = ((tmp_2/c) + 1)/2$
**return** $tmp_1 + tmp_3$
```

Algorithm 8 *MultDivShift*(a, b, exp)

```
$tmp_1 = b * (a >> exp)$
$tmp_2 = (2 * b) * (a\%2^{exp})$
$tmp_3 = ((tmp_2 >> exp) + 1) >> 1$
**return** $tmp_1 + tmp_3$
```

The function *MultShiftDiv*(a, c, exp) is intended to compute an approximation of $a2^{exp}/c$, while the function *MultDivShift*(a, b, exp) computes an approximation to $ab/2^{exp}$. These functions are proven in PVS to return the closest unsigned integer value to what the corresponding real number expression would produce, given some restrictions on the input values. The precise formulations are given in the following two lemmas.

Lemma 5.1 *For unsigned integers* a, c, exp, *if* $0 < c$, $exp \leq 30$, *and* $2^{ex+1}(c-1) < 2^{32}$, *then*

$$MultShiftDiv(a, c, exp) = \mod(\lfloor a2^{exp}/c + 1/2 \rfloor, 2^{32}).$$

The restrictions on the above lemma are fairly lenient. Essentially they require that the denominator be non-zero, that the exponent be small enough to fit 2^{exp+1} in an unsigned integer, and that the computation of $2^{exp+1}(a\%c)$ in 32 bits does not overflow. Recalling that $\lfloor x+1/2 \rfloor$ returns the closest integer to x, lemma 5.1 guarantees that the result is (up to overflow wrapping) the closest integer to the desired value.

Lemma 5.2 *For unsigned integers* a, b, exp, *if* $b \leq 256$, $12 \leq exp \leq 20$, *then*

$$MultDivShift(a, b, exp) = \lfloor ab/2^{exp} + 1/2 \rfloor.$$

The restrictions on this lemma are more severe, requiring at least one of the numerator values to be small, and the exponent to be bounded above and below. The bound on the exponent is chosen very carefully to encompass the smallest value of nb up to the largest value of $nb + 1$, which are the values needed in the CPR implementation. On the other hand, the result of lemma 5.2 is stronger, in that it calculates the closest integer to the desired expression, and is known not to overflow.

5.1. Integer Encoding

For the integer implementation, each of the three main functions is specified in C generically. That is, a single function is given, and changing one of the parameters allows the function to compute for either latitude or longitude. The generic encoding function is specified in algorithm 9.

Algorithm 9 *intEnc*$(nz, \widehat{awb}, nb)$

```
$tmp_1 = \widehat{awb} * nz$
$tmp_2 = tmp_1 + (2^{32-(nb+1)})$
$tmp_3 = tmp_2 >> (32 - nb)$
**if** $nb = 19$ **then**
    $tmp_4 = tmp_3\%2^{17}$
**else**
    $tmp_4 = tmp_3$
**end if**
**return** $tmp_4$
```

Due to the way that the logic function is specified, exactly following the integer computation even down to overflow conditions, the *software conformance* property for the generic encoding is simple and exact. Indeed, because the output of encoding is a 32-bit integer, the logic function and the software implementation can produce identical output.

Theorem 5.3 (Software Conformance, Integer Encoding) *Let* $\widehat{awb} \in \mathcal{AWB}$, *nz be an unsigned integer, and* $nb \in \{12, 14, 17, 19\}$. *Then*

$$intEnc(nz, \widehat{awb}, nb) = \widetilde{intEnc}(nz, \widehat{awb}, nb)$$

The nz parameter in the encoding function is a placeholder for $60 - i$ in the latitude encoding, and $\max\{NL(rlat_i(lat)) - i, 1\}$ in the longitude encoding, where lat is the input latitude. In order to state the conformance property for longitude encoding, implementations of the two functions NL and $rlat_i$ need to be specified as well. The NL function will be discussed separately in subsection 5.4. The function defined there is NL_{awb}, which takes an input from $\mathcal{AWB}$. Recall from section 2 the the $rlat_i$ function translates a real-valued latitude to the centerline of the bin where it lies. This is done prior to computing the NL value used for encoding to ensure that the NL value used for encoding longitude (computed with $rlat_i(lat)$) is the same value used by a receiver of the message (computed with the decoded latitude).

Algorithm 10 $intRlat(i, \widehat{awb}, nb)$

```
  $tmp_1 = MultDivShift(\widehat{awb}, 60 - i, nb)$
  $tmp_2 = MultShiftDiv(tmp_1, 60 - i, 32 - nb)$
  return $tmp_2$
```

The unsigned integer specification of $rlat_i$ is given in algorithm 10. Both real and software conformance properties are proven for *intRlat*, but are omitted here. With the implementations of NL and $rlat_i$ defined, the *real conformance* property for integer encoding can be stated.

Theorem 5.4 (Real Conformance, Integer Encoding) *Let* $lat \in [-90, 90]$, $lon \in [0, 360]$, $\widehat{lat} \in \mathcal{AWB}$, $\widehat{lon} \in \mathcal{AWB}$, $nb \in \{12, 14, 17, 19\}$, $i \in \{0, 1\}$, *and* $nl = \max\{NL_{awb}(intRlat(\widehat{lat})) - i, 1\}$. *If* $lat = \frac{360\widehat{lat}}{2^{32}}$, *and* $lon = \frac{360\widehat{lon}}{2^{32}}$, *then*

$$\begin{aligned} intEnc(60 - i, \widehat{lat}, nb) &= latEnc(i, lat) \\ intEnc(nl, \widehat{lon}, nb) &= lonEnc(i, lat, lon). \end{aligned}$$

The *real* and *software conformance* theorems for encoding guarantee that the integer encoding algorithm calculates exactly the same value as the formally verified real number version.

5.2. Integer Local Decoding

Similar to the floating point implementation, the integer version of local decoding takes an encoded value YZ and a reference position in AWB format. The output of the integer implementation is another AWB value, which corresponds to the recovered position. As with the encoding, the integer algorithm is a generic decoding, with a parameter that can be set in order to decode either latitude or longitude messages. Algorithm 11 determines the *zone index*, which is then used by Algorithm 12 to produce the decoded position.

Algorithm 11 $intZone_L(nz, \widehat{awb}, YZ, nb)$

```
if $nb = 19$ then
    $nzz = 4 * nz$
    $mes = 4 * YZ$
else
    $nzz = nz$
    $mes = YZ$
end if
$tmp_1 = nzz * (\widehat{awb} >> (32 - nb))$
$tmp_2 = (nzz * (\widehat{awb}\%2^{32-nb})) >> (32 - nb)$
$tmp_3 = 2^{nb-1} + (tmp_1 + tmp_2)$
if $tmp_3 < mes$ then
    $tmp_4 = nzz - 1$
else
    $tmp_4 = (tmp_3 - mes) >> nb$
end if
return $tmp_4$
```

Algorithm 12 $intDec_L(nz, \widehat{awb}, YZ, nb)$

```
if $nb = 19$ then
    $nzz = 4 * nz$
    $mes = 4 * YZ$
else
    $nzz = nz$
    $mes = YZ$
end if
$tmp_1 = (2^{nb} * intZone_L(nz, \widehat{awb}, YZ)) + mes$
return $MultShiftDiv(tmp_1, nzz, 32 - nb)$
```

The *software conformance* property for the local decoding algorithm is simple and exact. The integer implementation computes the same value as the ACSL logic version.

Theorem 5.5 (Software Conformance, Integer Local Decoding) *Let $\widehat{awb} \in \mathcal{AWB}$, nz be an unsigned integer, $nb \in \{12, 14, 17, 19\}$, and $YZ \in \mathcal{BN}$. Then*

$$intZone_L(nz, \widehat{awb}, YZ, nb) = \widetilde{intZone}_L(nz, \widehat{awb}, YZ, nb)$$

and

$$intDec_L(nz, \widehat{awb}, YZ, nb) = \widetilde{intDec}_L(nz, \widehat{awb}, YZ, nb).$$

The *real conformance* property for local decoding (and global decoding) uses a function which converts a real valued coordinate in degrees into the closest corresponding value in $\mathcal{AWB}$.

$$awb(pos) = \lfloor (pos + \text{ IF } pos < 0 \text{ THEN } 360 \text{ ELSE } 0 \text{ ENDIF })2^{32}/360 + 1/2 \rfloor$$

Note that this function is only used in proofs of the real conformance properties, so the accuracy or ability to compute this value with 32-bit integers is irrelevant. theorem 5.6 states that the integer version of the algorithm returns the closest possible value to the idealized algorithm.

Theorem 5.6 (Real Conformance, Integer Local Decoding) *Let $i \in \{0, 1\}$, $YZ_i \in \mathcal{BN}$, $XZ_i \in \mathcal{BN}$, $\widehat{lat}_{ref} \in \mathcal{AWB}$, $\widehat{lon}_{ref} \in \mathcal{AWB}$, and $nb \in \{12, 14, 17, 19\}$. If $lat_{ref} = \frac{360\widehat{lat}_{ref}}{2^{32}}$ and $lon_{ref} = \frac{360\widehat{lon}_{ref}}{2^{32}}$, then*

$$awb(rlat_{\mathsf{L}}(i, YZ_i, lat_{ref})) = intDec_L(60 - i, \widehat{lat}_{ref}, YZ_i, nb).$$

Also, letting $nl = \max\{NL_{awb}(intDec_L(60 - i, \widehat{lat}_{ref}, YZ_i, nb)) - i, 1\}$,

$$awb(rlon_{\mathsf{L}}(i, XZ_i, lon_{ref}, lat_{ref})) = intDec_L(nl, \widehat{lon}_{ref}, XZ_i, nb).$$

The proof of the real conformance property proceeds in two steps. The first step proves that the local zone is calculated exactly. The second step shows that adding in the message information to the zone returns the closest AWB value. The proof of the correctness of the local zone is long and involved, but is not particularly enlightening. It utilizes the fact that many of the calculations performed in the original CPR specification can be significantly simplified mathematically when the input is in AWB format. As a simple example, note that

$$2^{nb} lat/dlat_i = 2^{nb}\frac{360\widehat{lat}/2^{32}}{360/(60 - i)} = \frac{\widehat{lat} \cdot (60 - i)}{2^{32-nb}}.$$

This last expression is calculated to the nearest integer by the function *MultDivShift*. The more tedious steps of the proof are in verifying that none of the integer functions encounter undesired overflow or underflow.

Algorithm 13 $intZone_G(nz, mes_0, mes_1, nb)$

```
if $nz = 1$ then
    $tmp_4 = 0$
else
    $tmp_1 = ((nz - 1) * mes_0) + (2^{\min\{nb,17\}-1})$

    $tmp_2 = nz * mes_1$
    $tmp_3 = (tmp_1 + ((nz - i) * 2^{18})) - tmp_2$
    $tmp_4 = (tmp_3 >> \min\{nb, 17\})\%(nz - i)$
end if
return $tmp_4$
```

Algorithm 14 $intDec_G(i, nz, mes_0, mes_1, nb)$

```
if $i = 0$ then
    $mes = mes_0$
else
    $mes = mes_1$
end if
if $nb = 19$ then
    $mmes = 4 * mes$
    $nzz = 4 * \max\{nz - i, 1\}$
else
    $mmes = mes$
    $nzz = \max\{nz - i, 1\}$
end if
$tmp_1 = (2^{nb} * intZone_G(i, nz, mes_0, mes_1, nb)) + mmes$
return $MultShiftDiv(tmp_1, nzz, 32 - nb)$
```

The casual reader is spared from this level of detail, while the full proof is available for inspection in the formal PVS development.

5.3. Integer Global Decoding

Algorithm 13 and algorithm 14 calculate the global zone index and global decoding value, respectively. Once again, the variable *nz* can be instantiated to allow for latitude or longitude decoding.

The software conformance theorem for global decoding, theorem 5.7, asserts that the implementation and the associated logic function perform precisely the same calculation. Similar to the local case, the assertion is shown for the zone index and then the entire decode.

Theorem 5.7 (Software Conformance, Integer Global Decoding) *Let* $i \in \{0, 1\}$, $mes_0 \in \mathcal{BN}$, $mes_1 \in \mathcal{BN}$, $nz < 64$, *and* $nb \in \{12, 14, 17, 19\}$. *Then*

$$intZone_G(i, nz, mes_0, mes_1, nb) = \widetilde{intZone}_G(i, nz, mes_0, mes_1, nb)$$

and

$$intDec_G(i, nz, mes_0, mes_1, nb) = \widetilde{intDec}_G(i, nz, mes_0, mes_1, nb)$$

The real conformance property, theorem 5.8, is more involved than the software conformance. This is again due to the presence of the *NL* function, but is made more complicated due to the way that the *surface* message decodes. For the other message types, the global decoding is truly global, in that each coordinate is intended to return the bin centerline of the actual broadcast location. Surface decoding returns a value in $[0, 90]$ which is the bin centerline of the actual broadcast *modulo 90.*

This difference is simple enough to account for in the case for latitude decoding. The real conformance property simply says that the value is correct modulo 90. The longitude decoding, on the other hand, employs the value of the latitude decoding to determine the correct *NL* value. To select between the two, a helper function is used to choose between the two possibilities. It uses an input latitude value that is assumed to be within 30 degrees of the actual broadcasting location. Generally this is taken as the receiver's current latitude, or due to the fact that this only occurs for surface messages, some known ground location for the broadcaster.

The function, called *NorthLat*?, compares a surface global decode and a reference latitude in AWB format, and if the values are within 30 degrees (computed in terms of AWB values), the function returns *true*, otherwise returning *false*. Since the surface global decode is always in the range $[0, 90]$ (translated to AWB), and because the values that correspond to the two possible decoded latitudes are separated by 90 degrees, this function returns *true* when the decoded latitude is already in the correct range. When it returns *false*, the associated AWB value can be obtained by adding $3 \cdot 2^{30}$ to the decoded value. The specification of the function itself, along with the associated software and real conformance properties, is simple enough to be omitted here. The interested reader can find the details and proofs in the formal PVS specification.

The following function adjusts the latitude if necessary.

$$\begin{aligned} \mathit{RecoveredGlobal}(i, YZ_0, YZ_i, \widehat{lat}_{ref}, nb) = & \mathit{intDec}_G(i, 60, YZ_0, YZ_1, nb) + \\ & \text{IF } nb \neq 19 \text{ OR } \mathit{NorthLat?}(i, YZ_0, YZ_1, \widehat{lat}_{ref}) \text{ THEN } 0 \\ & \text{ELSE } 3 \cdot 2^{30} \text{ ENDIF} . \end{aligned}$$

Theorem 5.8 (Real Conformance, Integer Global Decoding) *Let $i \in \{0,1\}$, $nb \in \{12,14,17,19\}$, $YZ_0 \in \mathcal{BN}$, $YZ_1 \in \mathcal{BN}$, $XZ_0 \in \mathcal{BN}$, $XZ_1 \in \mathcal{BN}$, and $\widehat{lat}_{ref} \in \mathcal{AWB}$. Then*

$$awb(\mathit{rlat}_{\mathsf{G}}(i, YZ_0, YZ_1)) = \mathit{intDec}_G(i, 60, YZ_0, YZ_1, nb).$$

Furthermore, let

$$nl_0 = NL_{awb}(\mathit{RecoveredGlobal}(0, YZ_0, YZ_i, \widehat{lat}_{ref}, nb)$$

and

$$nl_1 = NL_{awb}(\mathit{RecoveredGlobal}(1, YZ_0, YZ_i, \widehat{lat}_{ref}, nb).$$

If $nl_0 = nl_1$, then

$$awb(\mathit{rlon}_{\mathsf{G}}(i, XZ_0, XZ_1)) = \mathit{intDec}_G(i, nl_0, XZ_0, XZ_1, nb).$$

The theorem for longitude only applies when the computed NL values coincide, since otherwise the longitude messages are encoded using different values for the zone size.

The proof of the real conformance theorem, similar to the local decoding, is generally simple, though long and tedious. The process of proving the real conformance led to several refinements and adjustments of the algorithm. As an example, one particular statement in the global zone determination was added after attempting to prove the real conformance, but noticing the possibility of an underflow that was skewing the computation. The original instruction in the local zone algorithm was to assign $tmp_3 = tmp_1 - tmp_2$, which mimics the computation of $59\,YZ_0 - 60\,YZ_1$ in the real algorithm. Unfortunately, the values of the first two temporary variables can independently vary up to $60 \cdot 2^{17}$, making underflow of the subtraction possible. The solution is to pad tmp_1 by adding the value $((nz-i)2^{18})$, which guarantees that the subtraction of tmp_2 stays positive. The next two instructions are to right shift by 17 (or fewer) bits, and then perform a modulus by $nz - i$ operation. The right shift leaves the pad a multiple of $nz - i$, which is then removed by the modulus operation. Several such alterations and optimizations were required to be performed, and also proven to be equivalent to the real computation, in order to verify the real conformance theorem.

5.4. Integer NL Calculation

The computation of the correct NL value from a given latitude is essential for the reliable encoding and decoding with CPR. Indeed, some of the early issues with CPR errors can be traced back to miscalculation of the correct NL value.[7] Some of these issues were due to a misinterpretation of the requirements, in which the raw input latitude was being used to determine the NL value. This results in the possibility of the decoded latitude having a different NL value, in the case where the raw latitude and the bin centerline are separated by an NL boundary. Other errors result from the decoded latitude not having enough precision to place it on the correct side of an NL boundary.

The integer implementation of CPR includes a function for computing NL values where the inputs are assumed to be from $\mathcal{AWB}$. This allows for the encoding and decoding of longitudes using the input latitude, or the recovered latitude.

The function $NL_{awb}(\widehat{lat})$ first determines if the input is a northern hemisphere latitude (when $\widehat{lat} \leq 2^{30}$) and, if not, converts the southern latitude to a corresponding northern one by simply negating the input (this has the effect of subtracting the input from 2^{32} due to the unsigned integer specification). This northern hemisphere latitude value is the input to a look-up table, which compares it to pre-calculated breakpoint values to determine the return value.

[7] Personal communication with members of RTCA Special Committee 186.

The software implementation and the ACSL *logic* specification are identical, and are easily proven to coincide.

Theorem 5.9 (Software Conformance, Integer NL Value) *For all* $\widehat{lat} \in \mathcal{AWB}$,

$$NL_{awb}(\widehat{lat}) = \widetilde{NL_{awb}}(\widehat{lat}).$$

The *real* conformance property for the NL function is more subtle. The main observation that drives the property is that the only values that are intended to be inputs to the NL function are those that correspond to some bin centerline. This is because, whether encoding or decoding, the closest bin centerline is always used, in order to guarantee consistent computation. Hence, the only values input into the *idealized* version of the NL function come from $\mathcal{RLAT}$. Similarly, the three functions that compute recovered latitudes, i.e., $rlat, intDec_L$, and $intDec_G$, for the integer computation are guaranteed by their conformance theorems to compute the closest AWB value to some bin centerline. This suggests the appropriate real conformance property as given in theorem 5.10.

Theorem 5.10 *For all* $lat \in \mathcal{RLAT}, NL(lat) = NL_{awb}(awb(lat))$.

The proof of this conformance theorem is simple, if blunt. A function is specified that iterates through every possible value $lat \in \mathcal{RLAT}$, and compares the value of $NL(lat)$ and $NL_{awb}(awb(lat))$. In fact, first it is shown that the values in $\mathcal{RLAT}$ are all obtained as bin centerlines for the case of $nb = 19$. Including both odd and even divisions, the total number of cases to be evaluated is $119 * 2^{19}$, or approximately 62 million values, which is well within the range of efficient computation.

The real and software conformance properties proven for the NL_{awb} function complete the CPR implementation and facilitate the proof of the integer versions of longitude encoding and decoding real conformance.

6. Related Work

Recently, much work has been done on the verification of numerical properties for industrial and safety-critical C code, including aerospace software.

The verification approach used in this work is similar to the analysis of numerical programs described by Boldo et al. [BM11], where a chain of tools composed of Frama-C, the Jessie plug-in [MM17], and Why [BFMP15] is used. In that work, the verification conditions obtained from the ACSL annotated C programs are checked by several external provers including Coq, Gappa, Z3 [dMB08], CVC3 [BC07], and Alt-Ergo. This approach was applied to the formal verification of wave propagation differential equations [BCF+13] and to the verification of numerical properties of a pairwise state-based conflict detection algorithm [GMKC13]. A similar verification approach was employed to verify numerical properties of industrial software related to inertial navigation [Mar14].

Other similar work includes the verification of a floating-point implementation of a point-in-polygon algorithm [MTFM19] using a toolchain composed of Frama-C, PVS, and PRECiSA [MTDM17, TFMM18], a static analyzer for floating-point programs. In contrast to the technique presented in this paper, Moscato et al. automate the step linking a PVS specified function to an ACSL specified logic function. In fact, PRECiSA takes as input the real-valued PVS specification of the algorithm and generates an annotated floating-point C implementation. In addition, PRECiSA computes an upper bound for the round-off error occurring in the program and a PVS certificate ensuring its correctness. Frama-C is then used to generate a set of verification conditions that can be discharged in PVS by using the information contained in the PRECiSA round-off error certificates.

Fluctuat [GP06] and Astrée [CCF+05] are commercial tools based on abstract interpretation [CC77], which have been successfully used to verify and analyze numerical properties for industrial and safety-critical C code. Fluctuat [GP06] is a 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. Astrée is a fully-automatic static analyzer that uses sound floating-point abstract domains [CMC08, Min04] to uncover the presence of run-time exceptions such as division by zero, underflows, and overflows. Astrée has been successfully applied to automatically check the absence of runtime errors associated to floating-point computations in aerospace control software [BCC+15]. Specifically, Astrée was used to verify the fly-by-wire primary software of commercial airplanes [DS07]. Additionally, Astrée and Fluctuat were combined to analyze on-board software acting in the Monitoring and Safing Unit of the ATV space vehicle [BCC+09]. In contrast to the technique presented in this paper, Fluctuat and Astrée do not provide formal proof certificates that can be discharged in an external prover. This is particularly useful for safety-critical systems since the proof certificates improve the trustworthiness of the approach.

7. Conclusion

The CPR 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. In this paper, the formal specification and proof of the CPR algorithm is extended from [DMTM17] to encompass all message types supported by the ADS-B standard. Furthermore, two implementations of the CPR algorithm are proposed and analyzed, extending [TMM+18] to include a single precision unsigned integer implementation. The unsigned integer implementation encompasses all 4 message types, while the floating-point implementation covers only airborne messages. The floating-point implementation could be extended to cover the message formats beyond airborne following the same methodology, given sufficient time and resources. The formally verified C implementations of CPR and the verification artifacts presented in this paper are released under NASA's Open Source Agreement at `https://github.com/nasa/cpr`, and will be included in a revised version of the ADS-B international standard. These implementations will serve as reference implementations of the CPR algorithm to be used by avionics manufacturers of ADS-B devices.

The implementations presented in this paper each include several simplifications aimed to reduce the numerical complexity of the original algorithm and to compute precisely. The equivalence between each version and the original algorithm in the ADS-B standard is formally proven in PVS. Two types of properties are proven. *Real conformance* properties state equivalence between the original algorithm and an ideal version of each implementation. Additionally, it is shown that the software computation guarantees the correct operation of the algorithm when implemented in C, forming what are referred to as *software conformance* properties.

The verification approach applied in this work requires some level of expertise. A background in floating-point arithmetic and in unsigned integer computation is needed to express the properties to be verified and to properly annotate for the weakest precondition deductive reasoning employed by Frama-C. 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 Alt-Ergo 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, Alt-Ergo 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

[BC07] C. Barrett and Tinelli. C. CVC3. In *Proceedings of the 19th International Conference on Computer Aided Verification, CAV 2007*, pages 298–302, 2007.

[BCC+09] O. Bouissou, E. Conquet, P. Cousot, R. Cousot, J. Feret, E. Goubault, K. Ghorbal, D. Lesens, L. Mauborgne, A. Miné, S. Putot, X. Rival, and M. Turin. Space Software Validation using Abstract Interpretation. In *Proceedings of the International Space System Engineering Conference, Data Systems in Aerospace (DASIA 2009)*, pages 1–7. ESA publications, 2009.

[BCC+15] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static Analysis and Verification of Aerospace Software by Abstract Interpretation. *Foundations and Trends in Programming Languages*, 2(2-3):71–190, 2015.

[BCF+13] 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 Automatic Reasoning*, 50(4):423–456, 2013.

[BCF+16] P. Baudin, P. Cuoq, J. C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. ACSL: ANSI/ISO C Specification Language, version 1.12. 2016.

[BFMP15] F. Bobot, J. C. Filliâtre, C. Marché, and A. Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer*, 17(6):709–727, 2015.

[BM11] 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.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of POPL 1977*, pages 238–252. ACM, 1977.

[CCF+05] 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 *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[CCKL08] S. Conchon, E. Contejean, J. Kanig, and S. Lescuyer. CC(X): Semantic Combination of Congruence Closure with Solvable Theories. *Electronic Notes in Theoretical Computer Science*, 198(2):51 – 69, 2008.

[CMC08] L. Chen, A. Miné, and P. Cousot. A Sound Floating-Point Polyhedra Abstract Domain. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems, APLAS 2008*, volume 5356 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2008.

[Cod15] Code of Federal Regulations. Automatic Dependent Surveillance-Broadcast (ADS-B) Out, 91 c.f.r., section 225, 2015.

[dDLM11] 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.

[dMB08] L. de Moura and N. Bjørner. 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*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[DMTM17] 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.

[DS07] D. Delmas and J. Souyris. Astrée: From research to industry. In *Proceedings of the 14th International Symposium on Static Analysis, SAS 2007*, pages 437–451, 2007.

[Eur17] European Commission. Commission Implementing Regulation (EU) 2017/386 of 6 march 2017 amending Implementing Regulation (EU) No 1207/2011, C/2017/1426, 2017.

[GMKC13] A. Goodloe, C. Muñoz, F. Kirchner, and L. Correnson. Verification of numerical programs: From real numbers to floating point numbers. In *Proceedings of NFM 2013*, volume 7871 of *Lecture Notes in Computer Science*, pages 441–446. Springer, 2013.

[GP06] E. Goubault and S. Putot. Static analysis of numerical algorithms. In *Proceedings of SAS 2006*, volume 4134 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2006.

[KKP+15] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.

[Mar14] C. Marché. Verification of the functional behavior of a floating-point program: An industrial case study. *Science of Computer Programming*, 96:279–296, 2014.

[Min04] A. Miné. 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*, volume 2986 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2004.

[MM17] C. Marché and Y. Moy. The Jessie Plugin for Deductive Verification in Frama-C. 2017.

[MTDM17] M. M. Moscato, L. Titolo, A. Dutle, and C. Muñoz. Automatic estimation of verified floating-point round-off errors via static analysis. In *Proceedings of the 36th International Conference on Computer Safety, Reliablilty, and Security, SAFECOMP 2017*, 2017.

[MTFM19] M.M. Moscato, L. Titolo, M.A. Feliú, and C.A. Muñoz. Provably correct floating-point implementation of a point-in-polygon algorithm. In *Proceedings of the Third World Congress on Formal Methods - The Next 30 Years (FM 2019)*, volume 11800 of *Lecture Notes in Computer Science*, pages 21–37. Springer, 2019.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proceedings of CADE 1992*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer, 1992.

[RTC09] 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.

[TFMM18] 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 2018*, volume 10747, pages 516–537. Springer, 2018.

[TMM+18] L. Titolo, M. M. Moscato, C. A. Muñoz, A. Dutle, and F. Bobot. A formally verified floating-point implementation of the compact position reporting algorithm. In *Proceedings of the 22nd International Symposium on Formal Methods (FM 2018)*, volume 10951 of *Lecture Notes in Computer Science*, pages 364–381. Springer, 2018.