Time Representation and Interpretation in Simulation Interoperability – an Overview

Mikael Karlsson
Fredrik Antelius
Björn Möller

mikael.karlsson@pitch.se
fredrik.antelius@pitch.se
bjorn.moller@pitch.se

Keywords:
HLA, Time representation

ABSTRACT: *In the DoD M&S Glossary a simulation is defined as a model operating over time so time is a key element in almost all simulations. Simulations can use time for assigning time stamps to individual data elements as well as for specifying the simulated time for the entire simulator/federate, in particular if the simulation is frame-based. The simulation time needs a binary representation, both in each simulator and in a federation where several simulators interoperate.*

This paper is about how the simulation time is represented and how it is interpreted in HLA Federations and related domains. It mainly focuses on logical or scenario time and doesn't go into detail on sources, usage or correctness of time stamps.

It covers the following:

- *General aspects of binary time representations and in particular time classes used in HLA. An overview of commonly used time representations is given, ranging from the older HLA 1.3 RTI time classes to the standardized time types in HLA Evolved. Issues with floating-point time representations and small time steps are covered in detail.*
- *Specialized time representations such as RPR FOM/DIS time stamps and advanced time classes used for perfect event ordering*
- *Relation to time representations from outside of the simulation domain such as UTC time and other time.*

Finally, this paper identifies some of the typical mistakes that are made in relation to simulation time and some important points and advice for both beginners and advanced developers of federations are given.

1. Introduction

This paper takes a closer look at the representation of time in simulation in general and in simulation interoperability in particular. Special attention is given to the time representations in different versions of HLA, be it fully standardized time representations or commonly used representations.

1.1 Challenges of Discreet Simulation

All software-based simulations are discreet in some sense. The state values, for example the speed of an aircraft, that are calculated and stored are represented in the computer using bits and bytes. These can only have a finite set of values, using for example integers or floating-point representation. In many cases these values represent an approximation of the accurate state value.

The DoD M&S Glossary [1] defines a simulation as a model executed over time. The time representation is also discreet and may in some cases also be an approximation of the intended value. This means that a software based simulation can be seen as discreet both in the state values that are calculated as well as the points in time where these are calculated.

Before taking a closer look at time representations it should be noted that simulations don't necessary

need to be discreet. A mechanical/physical simulator, such as an early pilot trainer, provides a truly continuous simulation over time. It should also be noted that not all software models use models that are calculated over time. An example is damage models where temporal data are sometimes calculated as a function of spatial data.

2 Important Concepts

In order to understand time representations there are several important concepts that need to be understood. This section provides a short introduction with some examples. Most of these concepts are discussed in detail in later sections of this paper.

Wall-clock Time. This is the actual time in the real world when a simulations is executed.

Simulation Time or **Logical Time** this is a representation of physical time within a simulation.

Time Step or **Time Increment**. This is the value by which a simulation repeatedly increments the simulation time. Many virtual simulators (like flight simulators) use a constant time step. These are sometimes referred to as "frame based" since the state in each step is visualized in a frame of a visualization system. The time step may also vary. In event driven simulation the time step will typically be the time to the next known event.

Simulation Executive Loop. The main loop of a simulation program that increments the time value is sometimes known as the Simulation Executive Loop.

Time Resolution or **Precision**. This is the smallest possible difference between two different time values. For an integer the resolution is exactly 1. For a floating-point value the Time Resolution will vary. This concept will be discussed in detail later.

Time Representation. This is the binary representation of the time value, for example a 32 bit little-endian integer or a 64 bit big-endian IEEE-754 [2] floating-point.

Time Interpretation. This is the interpretation of the time value used by the domain model. An integer representation with the value of 47 may be interpreted as 47 seconds or 47 days, depending on the interpretation used.

Time Implementation. This is program code, usually a set of object-oriented classes, which can represent and perform calculation on time values, such as time stamps and time increments. The HLA standard enables the developer to provide his own time implementation classes.

Time Management. HLA defines a set of services for managing the time advance in a federation, as well as the exchange of time-stamped information. An RTI is required to implement these services. These services are described in the HLA standard and will not be covered in detail in this paper.

2.1 Internal vs Shared Time Representations

All members of a federation have to agree on a shared binary representation of simulation time. Each model may have different internal time representations, depending on the model requirements, software technologies and languages used, as well as personal preferences. If the shared representation is different from the simulator's internal representation then it has to provide a conversion between the shared and the internal representation. This conversion has to be accurate enough that any loss of information during the conversion doesn't affect the validity of the simulation. It is not uncommon that information is shared at a lower frequency than the internal model.

3. A Closer Look at Binary Representations

The simulation time needs a binary representation, both in each simulator and in a federation where several simulators interoperate.

3.1 Basic Number Types

There are three basic types of numbers in common use in computers: integer, floating point and fixed point.

The *integer type* represents a mathematical integer. The most common representation is a string of bits, for example a 32-bit integer. The integer type may be signed (capable of representing positive and negative integers) or unsigned (capable of representing only non-negative integers).

The *floating-point type* uses a fixed number of significant digits and is scaled using an exponent. The base for the exponent is usually 2, 10, or 16. A floating-point number can be expressed like this:

$$significant\ digits \times base^{exponent}$$

The following figure shows a floating-point representation of the value 12 648.59 with significant digits 1264859, base 10, exponent 5. The upper row shows the significant digits. The lower row shows the position of the radix point.

```
                      12648 59
1234567890123456789012345567890.1234567890
```

The exponent determines where the significant digits are located in relation to the radix point. Increasing the exponent by one moves the significant digits one step to the left, thereby multiplying the value by 10.

The *fixed-point type* has a fixed number of digits after the radix point. It is essentially an integer with a scaling factor. For example, the number 9.87 can be represented as the integer 987 with a scaling factor of 100. Most computer languages have no built-in support for fixed-point types. The scaling

Type	Description
Integer32	32-bit signed integer
Integer64	64-bit signed integer
Float32	32-bit floating point (24 bits precision, 8 bits exponent)
Float64	64-bit floating point (53 bits precision, 11 bits exponent)

Figure 1: Common binary representations

factor has to be handled manually by the developer when presenting values and when making calculations using values with different scaling factors.

Figure 1 shows the binary representation of common types.

3.2 Interchange Formats

The interchange format for integer and fixed-point types are usually defined as bit strings of a certain length using two's complement.

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) defines interchange formats for floating -point numbers. These formats are encodings (bit strings) that may be used to transfer and communicate floating-point data between systems.

3.3 Precision

Most binary representations have a limited precision. An integer representation has a precision of 1. The typical floating-point representations have a limited number of significant digits. Depending on the value of the exponent, the least significant digit may be 1/4 000 000 or 1/2. This value that the least significant digit (lsb) represents if it is 1 is called the *unit of least precision* (ULP).

It can be said that the fixed-storage representations trade precision for range. When the value goes up, the precision goes down. The figure below shows a decimal representation of the value 12 648.59. The digits are 1264859 and the exponent is 5.

```
                          12648 59
12345678901234567890123455678 90.1234567890
```

The ULP is the value of the least significant digit. In the example above, the ULP is 10^{-2} or 0.01.

If we want to represent a number that is a factor of ten larger then we have to sacrifice precision.

```
                          126485 9
12345678901234567890123455678 90.1234567890
```

The value is 126 485.9 comprised of the same digits as before, 1264859, and the exponent 6. The precision has been reduced, and the value of ULP has increased to 10^{-1} or 0.1. The figure below illustrates how the precision changes as a value increases.

Figure 2: Precision in relation to value

The same rules apply to fixed-storage binary floating-point representations such as float64 although the base is 2 instead of 10. Here's an example with 16 significant digits, 1001011000111010 and the exponent 8. The value is 256 + 32 + 8 + 4 + 1/4 + 1/8 + 1/16 + 1/64 = 300.4531. The ULP is 1/128 or 2^{-7}.

```
                          100101100 0111010
12345678901234567890123455678 90.1234567890
```

3.4 Range

Most binary representations have a limited range. The integer64 representation has a range from $-2^{63} = -9 \times 10^{18}$ to $+2^{63} = 9 \times 10^{18}$. The float64 representation has a range from $(2 - 2^{-52}) \times 2^{1023}$ to 2^{-1074}.
To put these ranges in perspective, the number of seconds in a year is approximately $365 \times 24 \times 60 \times 60 = 31\ 536\ 000$. Thus an integer64 representation can represent a year with a precision better than nanoseconds.
The range of float64 with 1 second as 1.0 and a precision of 1 ms is huge. To represent 1 ms, i.e. 1 / 1 000 accurately, we need at least 11 bits since $2^{-11} = 0.00048828125$. That leaves 42 bits for seconds which can represent almost 140 000 years.

3.5 Epsilon

Epsilon is the name of the smallest value that can be represented in a particular binary representation. The following table shows the epsilon for different binary representations.

Representation	Epsilon
integer32	1
integer64	1
float32	2^{-149}
float64	2^{-1074}

Figure 3: Epsilon for common representations

In the integer representations, the epsilon value and ULP remain equal and constant regardless of the value being represented. The floating-point representations on the other hand have a ULP that becomes larger as the value gets larger.

4. Floating-Point Issues

The use of floating-point calculations in computers have a number of non-intuitive behaviors.

4.1 Exact Values

Binary representations are unable to make an exact representation of some values. For example, a decimal representation can only represent the value 1/3 as an approximation. This is quite intuitive to humans as we have the same problem in the common decimal system. The value 1/3 generates an infinite number of decimals, 0.33333333..... Less intuitive is the fact that float64 cannot represent the value 0.1 exactly. In the same way as the value 1/3 generates an infinite decimal representation, the value 0.1 generates an infinite binary representation.

Most developers would hesitate to use the value 1/3 as time step since it cannot be exactly represented in a decimal form. However, they will use the value 1/10 (0.1) as time step with a float64 representation without realizing that this value cannot be exactly represented in a base 2 format.

4.2 Confusing Arithmetic

Adding large and small values using floating-point representations can yield unexpected results since the smaller value can fall below the ULP of the large value and therefore not affect the large value. See illustration below.

```
                          12648 59
+                           0 0001
12345678901234567890123455678 90.1234567890
=                         12648 59
```

A special case of this is that Epsilon may be smaller than the ULP since ULP may change depending on the value. This means that taking a number and adding epsilon to it is not guaranteed to yield a new number.

4.3 Accumulated Errors

What's 0.1 × 10? Is it the same as 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1? You'd think so, but it actually isn't when using floating-point arithmetic. Here's a simple Java program that performs the calculation.

```
public class TestArithmetic {
   public static void main(String[] args)
   {
      double step = 0.1;
      double multiplied = step * 10.0;
      double accumulated = 0;
      for (int i = 0; i < 10; i++) {
         accumulated += step;
      }
      System.out.println(
         "multiplied = " + multiplied);
      System.out.println(
         "accumulated = " + accumulated);
   }
}
```

The output of this little program is:

```
multiplied = 1.0
accumulated = 0.9999999999999999
```

This discrepancy may cause problems, for example if a federate is planning to perform an action at time 1.0 and is checking the current time to see if it is equal to that value. The values will never be equal and the action will never be performed. It is also very likely that the current time is presented as a rounded value which will be 1.0, thereby hiding the problem and making debugging even more complicated.

4.4 Comparing for Equality

Due to the limited precision of floating-point numbers, comparing for equality is unlikely to give the intended result.

```
if (result == expectedValue)
```

A common attempt to solve this problem is to say that the values are considered equal if they are closer than a predefined distance, say 0.00001.

```
if (abs(result - expectedValue) < 0.00001)
```

This predefined distance may work fine during development of a simulation where simulation time never exceeds 1 000, but when the simulation is put into production and it runs for 100 000 steps the predefined distance 0.00001 falls below the precision of the time representation and the comparison fails.

4.5 Changing Behavior

In some situations, the result of floating-point calculations can depend on such unexpected factors as compiler flags. For example, the following C program prints one value when compiled in optimized mode, while it prints another value when compiled in standard mode.

```
double foo(double v) {
   double y = v * v;
   return (y / v);
}

main() { printf("%g\n", foo(1E308));}
```

The reason for this behavior is that the computation v × v is done in extended precision, with a larger exponent range. In optimized mode, the extended precision result of v × v stored in a register is used in the calculation of y / v. In standard mode, the result of v × v is stored in the double precision variable y as positive infinity due to the overflow. The variable y is then loaded and used in the calculation of y / v to yield positive infinity.

HLA Version	Name	Binary Representation	Comment
HLA 1.3	LogicalTimeDouble (“NG”)	Integer64	Integer value to be divided by one million
HLA 1.3	LogicalTimeDouble (“DMSO”)	Float64	Not Epsilon safe
HLA 1516 (2000 API)	LogicalTimeDouble	Float64	Not Epsilon safe
HLA 1516 (2000 API)	LogicalTimeDouble (alternate)	Integer64	Integer value to be divided by one million
HLA 1516 (DLC API)	LogicalTimeImpl	Float64	Not Epsilon safe
HLA 1516 (DLC API)	LogicalTimeImplFloat	Float64	Not Epsilon safe
HLA 1516 (DLC API)	LogicalTimeImplInteger	Integer64	
HLA Evolved	HLAfloat64Time	Float64	Standardized. Epsilon safe
HLA Evolved	HLAinteger64Time	Integer64	Standardized.

Figure 4: Common and standardized time classes in HLA

5. Time Implementation in HLA

Throughout the lifetime of the HLA standard, different time representations have been used. Here's a historic overview, including time representations in related standards. Figure 4 provides a list of time representations from the HLA 1.3 [3] standard through HLA 1516-2000 [4] and HLA Evolved (IEEE 1516-2010) [5].

5.1 Custom Time Classes

Ever since the first version of HLA, custom time classes have been supported. In fact, the standard hasn't provided any standardized time classes until HLA Evolved. The time types LogicalTimeDouble and LogicalTimeFloat in HLA 1.3 and HLA 1516 have simply been example classes provided by RTI developers. These classes have then become de facto standards.

5.2 Common and Standardized Time Classes

Figure 4 shows the common time classes usually found in HLA 1.3 and HLA 1516-2000 RTIs. Note that none of these common time classes that use floating-point representations are epsilon safe.

HLA Evolved (IEEE 1516-2010) finally introduces two standardized time representations; HLAinteger64Time and HLAfloat64Time, where the names, representations, behavior and even the serialized form have been standardized so they are guaranteed to be compatible between RTIs and RTI vendors. HLA Evolved, like the previous HLA versions, also allows the use of custom time classes.

5.3 Compatibility

The API for time classes has changed between different HLA versions, which means that it is not possible to use a time representation from one HLA version with another HLA version.

The implementation of the time representation has to be available to all federates in the federation. So if the federates are implemented in different environments (e.g. operating system, compiler version, programming language) the time implementation has to be available in all the different environments.

5.4 HLA Evolved and the Vanishing Epsilon

The fact that epsilon can fall below ULP disrupt the internal calculations made in an RTI, for example to guarantee that all messages have been delivered before allowing a federate to advance to a certain point in simulation time.

In HLA Evolved, the time implementations are required to have special handling of the vanishing epsilon. The rule, in a simplified form, is that taking a number and adding a non-zero number to it shall always yield a different number. This requirement doesn't completely fix the problem with vanishing epsilon, but it will prevent the federation from grinding to a halt.

6. Some Other Time Representations

This section provides a overview of other, related time representations. It also gives an example of a more advanced time class.

6.1 UTC Time

UTC [6] time or Coordinated Universal Time is used by many computer application that need a universal time. UTC time is not affected by daylight saving time and it is also the “common” time zone that all other time zones relates to. UTC in essence is Greenwich Mean Time (GMT). UTC is

Type	Unit	Required Time Resolution	Resulting Time Range
integer32	second	1 second	2^{31} / 86 400 / 365 = 68.096 years
integer32	millisecond	1 millisecond	2^{31} / 1 000 / 86 400 = 24.855 days
integer32	microsecond	1 microsecond	2^{31} / 1 000 000 / 60 = 35.791 minutes
integer64	microsecond	1 microsecond	2^{63} / 1 000 000 / 86 400 / 365 = 292 471.209 years
integer64	nanosecond	1 nanosecond	2^{63} / 1 000 000 000 / 86 400 / 365 = 292.471 years
float32	second	1 second	2^{24} / 86 400 = 194.181 days
float32	second	1 millisecond (11 bits)	2^{13} / 60 = 136.533 minutes
float32	second	1 microsecond (20 bits)	2^{4} = 16 seconds
float64	second	1 millisecond (11 bits)	2^{42} / 86 400 / 365 = 139 461.14 years
float64	second	1 microsecond (20 bits)	2^{33} / 86 400 / 365 = 272.385 years
float64	second	1 nanosecond (30 bits)	2^{23} / 86 400 = 97.090 days

Figure 5: Precision and range for different (signed) data types

specified using a date and a time value.

6.2 Time in DIS

The DIS [7] PDU header contains a *timestamp* when the data was generated. This is either an *absolute timestamp* using UTC time or a *relative timestamp* relative to an arbitrary starting point. Relative timestamps are used when the clocks of the simulation applications are not synchronized.

The timestamp uses an unsigned 32-bit integer representation. The least significant bit is used to indicate absolute or relative timestamp and the remaining 31 bits is used to represent time passed since the beginning of the current hour. Each time unit represents 3600 s / (2^{31} - 1) = 1.676 µs.

6.3 Time in GRIM-RPR

The GRIM-RPR [8] in HLA uses the DIS timestamp in the user-supplied tag for some of the HLA services. The first 8 bytes of the user-supplied tag contains the 32-bit DIS representation encoded as an ASCII string (without the usual terminating '\0') using hexadecimal ASCII characters (0-9 and A-F), including leading zeros.

6.4 Tie-breaking Time Implementation

An important property of some simulations are repeatability and independence on the number of nodes used during the execution. Repeatability problems can occur when multiple events are scheduled at the same time. [9] To achieve repeatable results, special care must be taken to ensure that the events are processed in the same order every time the simulation is executed. One approach is that the model developer adds or subtracts tiny epsilon increments to event times during event scheduling to manually produce unique ordering. This type of solution usually breaks down quickly as models get more complex. In addition, this practice violates the modeling since events will not occur at their calculated time.

A better way to guarantee repeatability is a time implementation that provides unique timestamps for all events by adding special tie-breaker fields. The WarpIV Kernel [10], a high-performance computing framework, provides a time implementation with these characteristics.

7. The Time Agility Challenge

HLA doesn't have a predefined information exchange model for any particular domain. Instead a Federation Object Model is used. This is an XML document that describes the shared objects and attributes (like aircrafts) and shared interactions (like start/stop commands or munition fire). Starting from HLA 1516-2000 the FOM defines the exact representation of the data, such as integer size, byte ordering, record layout, etc. A federate needs to format (encode) an attribute or parameter value according to the FOM before sending it to other federates. When it receives data from another federate it needs to decode the data.

One advantage of HLA is that the FOM concept makes it possible to implement general tools that support any FOM. The tool reads the FOM and can then use this information to dynamically decode the data that is received, for example for display purposes.

It is a bigger challenge for an HLA tool to be flexible with respect to the time representation. Time values are received as byte arrays. First this value needs to be decoded, according to the time repre-

sentation table in the FOM. Secondly, an HLA tool that needs to support Time Management, needs to match this to the corresponding time implementation in the code.

The challenge here is that there are numerous ways to represent and interpret time. Time implementation classes (code) need to be provided for each of them in order to be able to perform time calculations.

To reduce this problem HLA Evolved introduces two distinct, standardized time representations: HLAinteger64Time and HLAfloat64Time. If simulation developers adopt these representations it will reduce the effort to make simulations interoperable. It will also simplify the development of time agile tools.

8. Advice on the Choice of a Time Representation

The following is a simple list of advice when choosing a time representation:

a) Understand the limitations of the time representation that you intend to use. Integer representation will have their obvious limitations in resolution and highest possible value. For floating-point representations, big values may result in two different problems. If the time implementation doesn't guarantee an increment then simulation time will effectively stop. If an increment is guaranteed, then time steps may be too large, unintentionally "speeding up" the simulation. Each time representation has a "sweet spot" in its effective range.

b) Use time representations that have a big enough value range for your simulation. If your simulation starts at a large time value, consider offsetting this so that is starts at zero. Remember that large values may have larger ULP. Figure 5 shows the range for a number of types given a specific precision.

c) Consider using integer or fixed-point time representations. These are considerably easier to understand and debug. If a floating-point value is required in the calculation this can be achieved in the interpretation of the value. You may for example consider using an integer that is interpreted as microseconds.

d) If you use floating-point time representations, make sure that it provides safe increments of time. You should still try to stay in the range of time values where this feature isn't necessary.

e) Use the standardized time types according to HLA Evolved (IEEE 1516-2010): HLAinteger64Time and HLAfloat64Time, even if you are developing for an older version of the standard. This allows you to minimize risk and take advantage of the time implementations provided by the RTIs. In some cases RTIs also provide compatible implementations for the older APIs.

f) Consider writing code that can detect where the time values leave the "safe range" and warn the user when the time of his scenario goes outside of this range.

9. Conclusions

This paper has provided an introduction to several aspects of time representations. The challenges of the vanishing epsilon has been introduced. Some challenges on creating time agile tools have also been presented. Some advice on how to choose a time representation has been given.

One important long-term solution is to strive towards more standardized time representations, two of which are provided in HLA Evolved (IEEE 1516 -2010).

No matter which time representation that is chosen, it will have limitations and the simulation developer needs to understand the nature of these limitations.

References

[1] "DoD M&S Glossary (5000.59-M)", MSCO, www.msco.mil.

[2] IEEE: "IEEE 754-2008, IEEE Standard for Floating-Point Arithmetic", www.ieee.org.

[3] "High Level Architecture Version 1.3", MSCO, www.msco.mil.

[4] IEEE: "IEEE 1516, High Level Architecture (HLA)", www.ieee.org, March 2001.

[5] IEEE: "IEEE 1516-2010, High Level Architecture (HLA)", www.ieee.org, August 2010, A.k.a. "HLA Evolved".

[6] "Coordinated Universal Time", http://en.wikipedia.org/wiki/Coordinated_Universal_Time.

[7] IEEE: "IEEE 1278, Distributed Interactive Simulation (DIS)", www.ieee.org.

[8] SISO, "Real-time Platform Reference Federation Object Model 2.0 ", SISO-STD-001 SISO, draft 17.

[9] Steinman, J., "Introduction to Parallel and Distributed Force Modeling and Simulation", 09S-SIW-021, www.sisostds.org.

[10] "WarpIV Kernel", http://www.warpiv.com

Author Biographies

MIKAEL KARLSSON is the Chief Architect at Pitch overseeing the world's first certified HLA IEEE 1516 RTI as well as the first certified commercial RTI for HLA 1.3. He has more than ten years of experience of developing simulation infra-

structures based on HLA as well as earlier standards. He also serves on several HLA standards and working groups. He studied Computer Science at Linköping University, Sweden.

FREDRIK ANTELIUS is a Lead Developer at Pitch and is a major contributor to several commercial HLA products. He holds an MSc in computer science and technology from Linköping University, Sweden.

BJÖRN MÖLLER is the vice president and co-founder of Pitch, the leading supplier of tools for HLA 1516 and HLA 1.3. He leads the strategic development of Pitch HLA products. He serves on several HLA standards and working groups and has a wide international contact network in simulation interoperability. He has twenty years of experience in high-tech R&D companies, with an international profile in areas such as modeling and simulation, artificial intelligence and Web-based collaboration. Björn Möller holds an MSc in computer science and technology after studies at Linköping University, Sweden, and Imperial College, London. He is currently serving as the vice chairman of the SISO HLA Evolved Product Support Group.