Building Time Managed Federations with Object Oriented HLA

Fredrik Antelius, Pitch Technologies, Sweden
Martin Johansson, Pitch Technologies, Sweden
Björn Möller, Pitch Technologies, Sweden

fredrik.antelius@pitch.se
martin.johansson@pitch.se
bjorn.moller@pitch.se

Keywords:
HLA, Time Management, OO-HLA

ABSTRACT: *A popular approach to simplify the use of HLA is to use an object oriented HLA middleware with an API that is tailored to a specific FOM and federation agreement. This is informally known as Object Oriented HLA, or OO-HLA.*

Time Management in HLA is a powerful set of services to achieve deterministic distributed simulations. The Time Management services in HLA support different kind of simulations; everything from frame-based simulations like the SISO Smackdown federation to analytical simulations run multiple times using the Monte Carlo method and even optimistic, event-based simulations where data that is sent and used optimistically can later be retracted and any calculations based on the data have to be redone.

This paper contains an introduction to HLA Time Management and Object Oriented HLA. It describes common use case for HLA Time Management and Object Oriented HLA. It shows how the advanced features of Time Management can be simplified and presented in a powerful way in an existing, commercial OO-HLA tool.

1. Introduction

This paper gives an introduction to the Time Management services in HLA and it also gives an introduction to Object Oriented HLA (OO-HLA). This paper describes how support for the time management services can be added to a commercial OO-HLA tool, the best practices used and finally some thoughts and insights gained during the development.

1.1 About simulation and time

The US DoD M&S Dictionary defines a simulation as "a method for implementing a model over time" [1]. When several federates interoperate in a federation, simulation time needs to be managed. This is what the HLA Time Management services do.

There are many reasons that you may want to use the HLA Time Management services, they all have the requirement for determinism and repeatability in common.

The result of a deterministic and repeatable simulation need to be based on the input parameters and the actions of each system, and shall not be affected by external factors, for example changes in latency between systems or processing time of messages. Messages are delivered to the systems in a well-specified order that is guaranteed to be the same in every simulation run.

The simulation run can be repeated and the results after every run will the same.

The determinism and repeatability is relying on the order of events, i.e. when messages are received. The order has to be well defined. There has to be some form of the "happened before" relationship between events [1a]. For example the "fire event" *happened before* the "detonation event".

If we only have two systems, this can be solved by using TCP/IP to send the "fire event" and "detonation event". The TCP/IP stream will reassemble and order the received network packages so that the messages are guaranteed to arrive in the intended order. Similarly, in a Client-Server architecture this is also not a problem since all messages are ordered by the server. When building distributed simulations, we often have more than two systems that need to exchange messages. We will show how Time Management in HLA can be used to solve this problem.

1.2 Drawbacks of not using Time Management

The classical paper "HLA Time Management and DIS" by Richard M. Fujimoto and Richard M. Weatherly contains a simple example [2]. We have three systems, A, B and C. System A send a "fire event", that system B reacts to by sending a "destroyed event". System C is an observer of the 2 events, and they may be received so that the "destroyed event" is seen before the "fire event". That can clearly cause problem in the simulation.

This simple example shows what can happen when we are not able to control the "happened before" relationship between the "fire" and the "destroyed" event. System C sees the "destroy event" before the fire event that actually caused the event. The effect is seen before the cause!

The problem is that we need to order events between A, B and C together, pairwise between A-B, A-C, and B-C is not enough.

For system C we need to order the "destroy event" so that the fire event is processed first. See the dotted line in the picture.

With HLA Time Management we can use the RTI to perform this ordering and make sure that all messages are seen and processed in the correct order and we can have a well specified "happened before" relationship between multiple events in multiple systems.

Instead of relying on the real-time reception of events, we assign logical time to events. The logical time is just a value that the RTI and the simulators can use to compare with each other to determine which "happened before".

Messages can be received in a random order and then be delivered to the simulator in an order that is well specified by the assigned logical times.

1.3 Applications

Frame-based simulations typically simulate a continuous system, for example a platform based training simulator. The data is exchanged at a fixed frame rate, where some systems may use a multiple of the frame rate, e.g. only exchanging data in every 5th frame. The simulation can run in real-time when there are humans or hardware in the loop, but may also run in non-real-time (as fast as possible or time scaled) when possible. Simulators both send and receive time-managed messages. The data in the current frame is used to produce data for the next frame.

An example of a frame-based simulation is the SISO Smackdown event where data is exchanged at 1 Hz. NASA has provided two time managed federates, one that simulates the positions of the

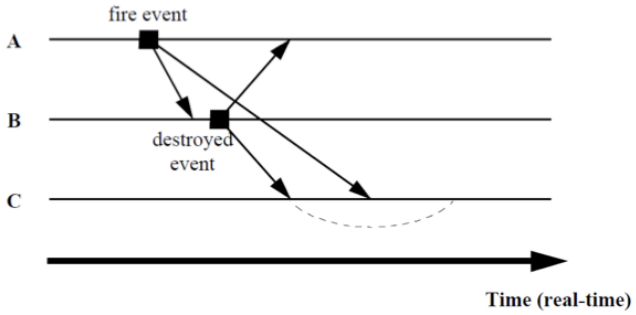

Figure 1: Example from "HLA Time Managment and DIS"

Sun, the Earth and the Moon [3]. This federate is also responsible for pacing the logical time. The other federate is a transfer vehicle that flies from the Earth to the Moon.

Students from all around the world have then added simulators for rovers and astronauts on the Moon, space vehicles, observers, visualizers and communications. This collaboration has required an extensive documentation and specification, for example in the Federation Agreement [4].

The frame-based approach to designing the simulation can be seen in the diagram of the Running state from the Federation Agreement, see figure 2. One iteration in the diagram is one frame.

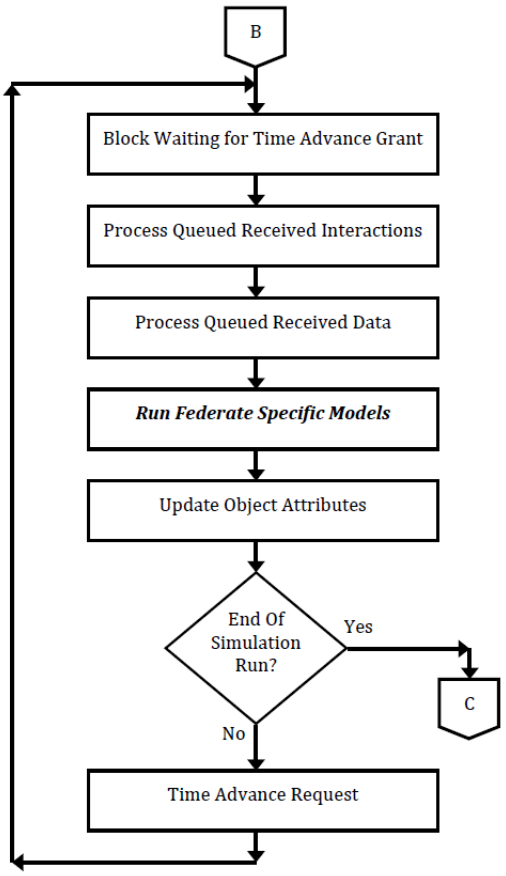

Figure 2: Running State from the SISO Smackdown federation agreement

An *event-based* federate does not perform the same processing in every time step, instead it performs the processing as a result of an internal or external event.

An example is a judge federate that does a damage calculation as a result of a fire event. When a fire event is received, the damage is calculated and update on the targeted instance. Something called zero lookahead can be used so that the fire and the updated damage will be received "simultaneously" by the other federates.

A simulator may be a mix of frame and event-based in different stages of the scenario. It may be frame-based during launch and docking but event based during flight to the moon. The frame length may also be modified to be longer during the flight phase.

Based on the data exchange, it may be preferable to use frame based if you expect to receive data at every frame, but if the data is received sporadically and usually not in every frame, event based may be preferred. The internal model used to implement the simulator should also be considered.

An *analytical simulation* using time management is typically based on the Monte Carlo method [5], in which repeated random sampling is used to compute the result. For the Monte Carlo method to be applicable, the simulation run must be deterministic; changes in the result must be caused by the input, not by random behavior in simulators or data exchange.

This type of simulations uses time management for all data exchange and is usually not paced but running as fast as possible. Save and restore is often used to restart the simulation between the Monte Carlo runs.

Frame overrun detection is also a common use case for time management when building distributed simulations. The data is exchanged with logical time stamps but the messages are delivered in receice order rather than logical time stamp order. The federates still use the HLA Time Management services to control the logical time and when a message is received with an unexpected logical time a frame overrun has been detected. This can result in a warning and/or that the data can be dropped.

1.4 Making it easier to build Time Managed federations

When building more advanced distributed simulations, an increasing number of simulators need to use the HLA Time Management services. The available guidance, documentation and examples of time management usage are very limited so there is a great need to make it easier to build and use time managed federations.

The support for the HLA Time Management services in Pitch Developer Studio and this paper is part of an effort to make it fun and easy to build time managed federations.

2. Time Management in HLA

2.1 The Happened-Before Relationship

For developers of multithreaded applications, the happened-before relationship between events is a constant source of problems:

"What happens if an element is removed from the list while I'm iterating the same list?

- I need to guarantee that I have iterated the complete list before someone else may remove an element"

This is achieved using locks, mutexes, comapare and swap (CAS), lock-free data structures, signals or whatever other tools the programming language provides for concurrent programming. All these are based on the notions of a happened-before relationship between events.

For example the Java Language Specifications defines a Memory Model where the Happens-before order is described in great detail, that operations that may be reorder (for example by the CPU) and when a thread "sees" a write from another thread [6]. C++ did define a Memory Model in C++11, previous versions relied on the processor architecture to define Happens-before (so the same program may behave differently on different processors) [7].

2.2 Happened-Before in Distributed Simulations

For distributed simulations where the Happened-before relationship needs to be guaranteed between different applications or simulators, HLA provides the following support [8, 9, 10, 11, 12]:

- Synchronization points or Save and Restore.
- Request/Response interactions or updates.
- The Time Management services.

If we look at some of the other architectures for building distributed simulations, we note that DIS, TENA and DDS have no native support to guarantee causality [13, 14, 15]. To achieve causality, you are required to use manual Request/Response messages to guarantee any Happened before relationships. The use of unreliable communication in DIS and DDS makes this extra error prone.

2.3 Time Managed operations

In addition to the HLA Time Management services to manage and control the logical time, the following operations are optionally time managed:

- Update instance with `updateAttributeValues` and `reflectAttributeValues`
- Exchange events with `sendInteraction` and `receiveInteraction`
- Remove instances with `removeObjectInstance` and `deleteObjectInstance`

Some additional services are often requested to have time managed support. They can easily be handled in the federation agreement, for example:

- **Ownership transfer**: use a pattern like the Transfer of Modeling Responsibility (TMR) pattern where the administrative exchange uses time managed interactions and then the ownership is exchanged as agreed [15a].
- **Create instance**: treat the first update as the time managed operation when the "instance was created". This may use an empty update of the `HLAprivilegeToDelete` attribute if no other attribute is applicable.

2.4 Other types of Time Handling

When discussing HLA Time Management, and especially when comparing it to other mechanisms to handle time, there are often some confusion. This section describes other types of time handing that is not related to HLA Time Management.

Time stamps

The RPR FOM uses timestamp sent in the User Supplied Tag [16]. The sender assigns a time stamp when the data was valid or generated, instead of when the receiver received or processed the data. This is useful for dead reckoning for example. Note that synchronized clocks are not needed, the RPR relative time stamps do only rely on the time differences between updates and use that for dead reckoning.

Real-time

Timing may also be implicit, all systems know that all other system are connected to the same data bus or wired to the same sync signal. All messages sent or received are implicitly valid for the previous, current or next frame.

2.5 Definitions

There are some terms that we need to define before we continue.

Regulating: a federate that is time regulating has the capability to send time managed messages.

Constrained: a federate that is time constrained has the capability to receive time managed messages.

Federate Time: each federate has its own logical time. The federate time is regulated and constrained by other federates in the federation.

Lookahead: the lookahead defines how soon in the future messages may be sent (only applicable if the federate is time regulating). The smallest logical time of messages to send is the federate's current time + lookahead.

Time Representation: the data type used to represent the logical time.

HLA 1516-2010 Evolved contains two standardized time representations, `HLAinteger64BE` and `HLAfloat64BE`. They use a 64-bit integer or a 64-bit float in your programming language, for example an `int64_t` or `long` and a `double`.

You may choose not to use the standardized time representations. It is possible to represent the logical time as a string and 4 floating-point values. It is not recommended as some tools may only support the standardized time representations [17].

2.6 Advancing the Logical Time

When the federate increases or advances its logical time, it moves between two states: the *Granted* state and the *Advancing* state.

The federate starts in the *Granted* state. In this state, messages for the next logical time may be sent. When all messages for the next logical time

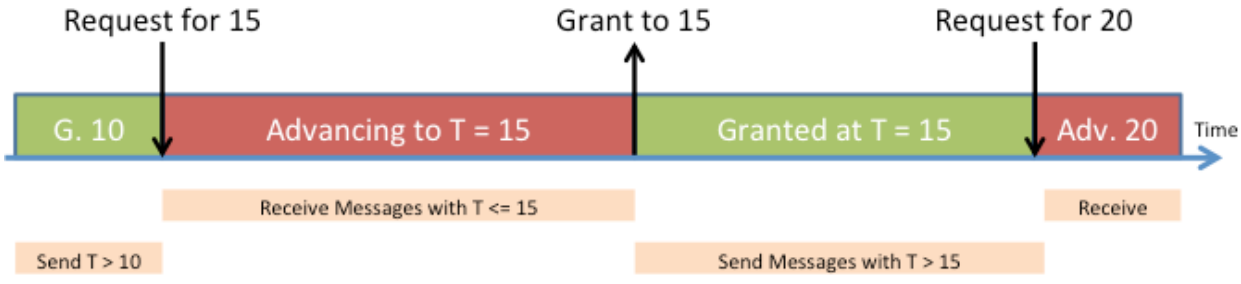

Figure 3: HLA Time Managment states

have been sent, the federate will send a request to the RTI to advance its logical time. With this call the federate enters the *Advancing* state and no more messages for the next logical time may be sent.

When in the *Advancing* state, messages from other federates are received. When the RTI can guarantee that no more messages with the requested time will be received, the advance is granted and the federate is returned to the *Granted* state. The time spent in the *Advancing* state is controlled by the RTI and is based on the behavior of the other federates in the federation.

3. Object Oriented HLA

Object Oriented HLA (OO-HLA) is a way to represent the HLA API using the object oriented programming paradigm, where the instances in the HLA federation are actual objects in the programming language. The programming language object classes, for example `HlaAircraft`, correspond to object classes in the FOM, for example `HLAobjectRoot.Aircraft`. The main advantage is that this makes OO-HLA very intuitive and easy to use, but it assumes a particular FOM.

OO-HLA can be developed in many different ways, so there is no standard that defines how this must be done. OO-HLA is a design pattern for the middleware, commonly referred to as the HLA Module, which adds HLA support to a simulator [18, 19].

The HLA Module is generated, automatically or manually, by selecting the attributes, object classes and the interactions that should be used. Some additional properties may also be added to specify how the attributes and interactions are intended to be used. The generated code has an API that has been tailored to the FOM and the specific requirements of a particular simulator, instead of the very general HLA API.

3.1 Proxy Objects

The HLA Module contains two types of proxy objects.

1. If another federate registers an HLA object instance, then the HLA Module creates a C++ or Java object that corresponds to that instance. The attributes of that object are populated as updates are received. To get these values you use a type-safe "get" operation.
2. Your federate can create your own "local" object instances. The HLA module will then register the object in the federation. The attributes of that object can be updated with type-safe "set value" operations. These attribute values will automatically be sent to other federates.

3.2 Example

An example of using an HLA Module generated with the OO-HLA tool Pitch Developer Studio™.

```
// connect to the RTI
_hlaWorld.connect();

// create an aircraft with callsign AirforceOne
HlaAircraft aircraft = _hlaAircraftManager
    .createLocalHlaAircraft("AirforceOne");

for (int i = 1; i < 20; i++) {
  // create an updater and set the (dummy) position
  HlaAircraftUpdater updater = aircraft
      .getHlaAircraftUpdater();
  updater.setPosition(PositionRec.create(i, i*20, i*300));

  //send the position update
  updater.sendUpdate();

  // sleep for 1 second
  Thread.sleep(1000);
}

// delete the aircraft
_hlaAircraftManager.deleteLocalHlaAircraft(aircraft);

// disconnect from the RTI
_hlaWorld.disconnect();
```

This example will connect to the RTI and create one aircraft. The position of the aircraft will be updated 20 times, once per second, and then delete the aircraft and disconnect from the RTI. In a later sample we will see the modifications needed to use time management.

3.3 Pitch Developer Studio

Using Pitch Developer Studio™ is like "getting an HLA expert in a box". It generates code that gives you best-practice patterns for HLA integration. It supports HLA functionality like create, join, publish, subscribe, register, discover, update, reflect as well as sending and receiving of interactions. It supports functionality like request and provide of attributes for efficient initialization and to support late joining federates. It takes care of encoding and decoding values for simple as well as complex data types. It allows you to work with local and remote objects using set and get on local objects. It allows you to work with HLA interactions. And it handles the handles.

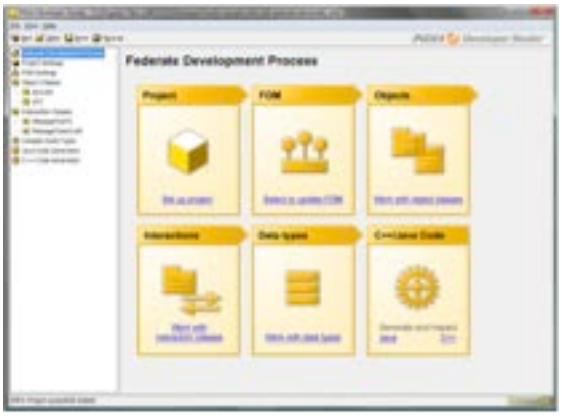

Figure 4: Pitch Developer Studio™

Pitch Developer Studio™ is based on a number of best-practice patterns both for HLA and for programming in general. The following design patterns are often used in the generated code [20]:

- Factory
- Listener (Observer/observable)
- Adapter

The current version of Pitch Developer Studio™ generates both C++ and Java code and supports common Windows, Linux and Mac OS X versions.

4. Adding Time Management to OO-HLA

This section describes how support for the HLA Time Management services was added to the OO-HLA tool Pitch Developer Studio™.

4.1 Principles

There were some principles that guided the design of HLA Time Management support for Pitch Developer Studio™, for example:

1. Easy to switch between using time managing and not using it.
2. Easy to use for beginners, but support more advanced use cases.
3. Support gradually increasing time management usage.
4. Support the most common ways of handling time.
5. Paced by another system or support for internal pacing with custom time sources.
6. Support commonly used time representations.

The HLA Module uses a fixed frame length by default. The lookahead is the same as the frame length. This works well for frame-based simulators. Event-based simulators can set the frame length to ∞, and set the lookahead to 0. This makes the generated HLA Module easy to use for both types of simulators. Both the frame length, i.e. the step size for time advancement, and the lookahead can be modified at runtime if needed.

All "get" methods will return the value for the current frame. The frame when the update was received can also be returned.

All "set" methods and the `sendUpdate()` method produce values for the next frame. An optional parameter can be used to send an update in another (future) frame.

The generated HLA Module keeps track of the default logical time that is used if no other value is assigned when sending an update or interaction. This time is automatically updated by the HLA Module. The HLA Module also keeps track of the currently granted time and the time of the next frame.

4.2 Advancing the Time

The logical time can be advanced using three different methods with slightly different semantics.

- `advanceToNextFrame()`: advances the time to the next frame. This method is used by frame-based simulators.
- `advanceToNextEvent()`: advances to the logical time of the next event. This may be the same as the current time. This method is used by event-based simulators. This is the only method that fully supports zero lookahead.
- `advanceToLogicalTime(LogicalTime)`: advances the time to the requested time.

All methods are blocking, so they will not return until the request has been granted. The federate is in the granted state before the call, and then in the advancing state during the call. When the method returns, the federate is back in the granted state, see figure 5.

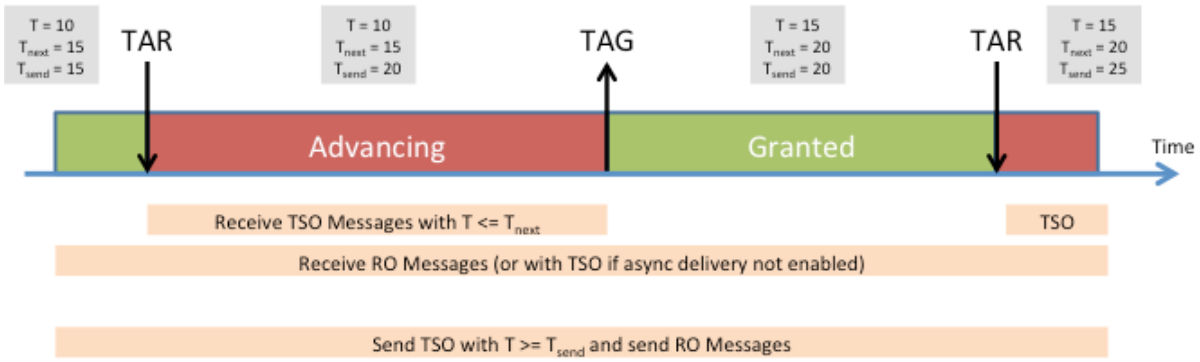

Figure 5: Advancing the logical time

4.3 Simple Example

The simple example that we looked at in section 3.2 can easily be modified to use time management.

```
// connect to the RTI
currentTime = _hlaWorld.connect();

// create an aircraft with callsign AirforceOne
HlaAircraft aircraft = _hlaAircraftManager
    .createLocalHlaAircraft("AirforceOne");

for (int i = 1; i < 20; i++) {
  // create an updater and set the (dummy) position
  HlaAircraftUpdater updater = aircraft
      .getHlaAircraftUpdater();
  updater.setPosition(PositionRec.create(i, i*20, i*300));

  //send the position update for the next frame
  updater.sendUpdate();

  // advance to next frame
  currentTime = _hlaWorld.advanceToNextFrame();
}

// delete the aircraft
_hlaAircraftManager.deleteLocalHlaAircraft(aircraft);

// disconnect from the RTI
_hlaWorld.disconnect();
```

This example will connect to the RTI and create an aircraft. The position will be updated 20 times, once each frame, and then the example will delete the aircraft and disconnect from the RTI. The updated position will be sent using time managed messages.

The only difference between this sample and the previous sample is that we use time management to pace the simulation instead of running in real-time and sleep. Time management was activated when we created the `HlaWorld`. As we have seen, it is very easy to take a federate and modify it to use time management.

The frame based sample can be compared to another sample that uses the event based approach.

```
// define a listener to interactions
class FireListener extends HlaInteractionListener {
  // method to run when a Fire interaction is received
  public void fire(boolean local,
                   HlaFireParameters fireParameters,
                   HlaTimeStamp timeStamp,
                   HlaLogicalTime logicalTime) {
  // check if any tank is hit
  for (HlaTank tank : _hlaTankManager.getHlaTanks()) {
    if (hitDetected(fireParameters, tank)) {
      // increase the damage with 1 if hit
      HlaTankUpdater updater = tank.getHlaTankUpdater();
      int initialDamageIfNotSet = 0;
      int newDamage =
          tank.getDamage(initialDamageIfNotSet) + 1;
      updater.setDamage(damage+1);

      // send the updated damage
      updater.sendUpdate();
    }
  }
}


// add listener
_hlanteractionManager
    .addHlaInteractionListener(new FireListener());

// connect to the RTI
_hlaWorld.connect();

while(_running) {
  // advance the logical time
  _hlaWorld.advanceToNextEvent();
}
```

This example will connect to the RTI and react to "fire" interactions. If any tank is hit by a fire interaction, the federate will update the damage for the tank.

4.4 Choosing Lookahead

The frame size is usually easy to figure out from the federation agreement and the requirements for the simulator. But the value for the lookahead is harder to choose wisely. The lookahead limits how far other federates may advance their logical time. The lookahead should be as large as possible, so that the other systems may advance their time as far as possible, but the federate may not send messages with a logical time that is smaller than the current time + lookahead.

The lookahead should be the same as your frame length, but:

- It may be larger if the simulated system has an inherent reaction time.
- It may be zero, if the simulator needs to react to events in the current frame.

Let's say that we have two systems, F1 and F2:

- F1 has a frame length of 1, the computation time for a frame is 1 and the lookahead is 1.
- F2 has a frame length of 10 and the computation time for a frame is 10.

Should the lookahead for F2 be 1 or 10?

The total time to run the simulation is about twice as long with one lookahead compared to the other!

- F1 and F2 can work in parallel with $F2_{lookahead} = 10$.
- When $F2_{lookahead} = 1$ then F1 has to wait for F2 for 9 time units after completing the first frame, then F2 has to wait for F1.

The lookahead should be as large as possible, but not larger.

4.5 More Advanced Features

The HLA Module will keep track of a lot of information, handle the small details that are tricky to get correct and provide useful convenience functions, for example:

1. Time management is enabled on `connect()` and disabled on `disconnect()`.
2. Messages that are not time managed are, by default, delivered in both the *Granted* and *Advancing* state using asynchronous delivery.
3. `InvokeWhenGranted()` is a convenience function that makes it easy to schedule an operation to run when the federate enters

the *Granted* state. Since messages are received in the *Advancing* state, there is often a needed to perform an operation when *all* messages for the frame have been received, i.e. when returned to the granted state.

4. Federates that are not time regulating that join a running federation will start at time 0. The HLA Module will take an initial time step to start close to the other federates.
5. Convenience methods for pacing and detection of frame overruns.

All time representations that are compatible with `HLAinteger64Time` and `HLAfloat64Time` from the HLA 1516-2010 Evolved standard can be used with Developer Studio. Since Developer Studio works with any HLA version, this includes time representations used in HLA 1.3, HLA 1516-2000 and HLA 1516 DLC. The previous HLA version does not contain any standardized time representations so time representations from common RTIs are used instead.

If `HLAinteger64Time` is selected, the generated HLA Module will use a `long` in Java API and an `int64_t` in C++ API. For `HLAfloat64Time` a `double` will be used in both the Java and C++ APIs.

4.6 Working with Pilot users

We worked with the existing users of Developer Studio when we specified and designed the support for time management. We listened to their requirements, their use cases and how they wanted their simulators to work with the HLA Module.

We also worked in close cooperation with pilot users at TNO in Holland. They were involved in the design of the API and used pre-releases and beta version of Developer Studio in their real project.

5. Discussion

5.1 Challenges

The HLA API for time management is quite complex, the sequence of calls and the valid values for parameters are not obvious. How and when time managed messages may be sent is not obvious.

It was a challenge to simplify this interaction to an API that was easy to understand for a beginner, but also was powerful enough for experienced HLA developer with existing experience with the time management services. Frame-based federates were straightforward to understand and describe but event-based federates, especially with zero lookahead, were tricky to get correct.

It was also a challenge to decide what parts of the HLA time management services to support and which time representations to support.

5.2 Best Practices for an Easy Solution Pattern

Pitch Developer Studio™ provides a large number of best practices have been incorporated in the generated HLA Module. These range from design and architectural practices down to low-level programming patterns. This makes it possible to easily add support for HLA time management to simulator.

6. Conclusions

It is possible to add support for the HLA Time Management services to a commercial OO-HLA tool. It can be done without major changes for users who have not started using time management, or have no need for it.

An HLA Module is much easier to use than using the general HLA API directly. The OO-HLA middleware can handle a lot of the details and the developer can focus the attention and energy on developing a good model.

The HLA Module has been developed and tested with pilot users, for example at TNO. They have provided valuable feedback. In December 2012, the support for time management was released with Pitch Developer Studio™ version 3.0.

We believe that the support for HLA Time Management in a commercial OO-HLA tool will have a positive impact on the adoption of HLA over the coming years, not only in the defense domain, but also in civilian applications.

References

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

[1a] Leslie Lamport, "Time, Clocks and the Ordering of Events in a Distributed System", (1978), Communications of the ACM, 21(7)

[2] Richard M. Fujimoto, Richard M. Weatherly "HLA Time Management and DIS", (1995)

[3] "SISO Smackdown", www.sisosmackdown.com, January 2012

[4] "SISO Smackdown Wiki", www.smackdown.inarisolutions.com, January 2012

[5] “Monte Carlo method”, www.wikipedia.org/wiki/Monte_Carlo_method, January 2012

[6] “The Java Language Specification”, docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4, January 2012

[7] Hans-J. Boehm, Sarita V. Adve: “Foundations of the C++ Concurrency Memory Model”, www.hpl.hp.com/techreports/2008/HPL-2008-56.pdf, January 2012

[8] “High Level Architecture Version 1.3”, DMSO, www.dmso.mil, April 1998

[9] IEEE: “IEEE 1516, High Level Architecture (HLA)”, www.ieee.org, March 2001.

[10] SISO: “Dynamic Link Compatible HLA API Standard for the HLA Interface Specification” (IEEE 1516.1 Version), (SISO-STD-004.1-2004)

[11] IEEE: "IEEE 1516-2010, High Level Architecture (HLA)", www.ieee.org, August 2010.

[12] Frederick Kuhl, Richard Weatherly, Judith Dahmann: “Creating Computer Simulation Systems: an Introduction to the High-Level Architecture”, Prentice Hall PTR (2000), ISBN 0130225118

[13] IEEE: “IEEE 1278.1-2012, Distributed Interactive Simulation (DIS)”, www.ieee.org, 2012.

[14] “Test and Training Enabling Architecture (TENA)”, www.tena-sda.org, January 2012.

[15] “Data-Distribution Service for Real-Time Systems (DDS)”, portals.omg.org/dds, January 2012.

[15a] Björn Möller, Filip Klasson, Björn Löfstrand, Per-Philip Sollin: “Practical Experiences from Four HLA Evolved Federation”, 12S-SIW-057, SISO, March 2012.

[16] SISO: “Real-time Platform Reference Federation Object Model 2.0”, SISO-STD-001 SISO, draft 17, www.sisostds.org

[17] Mikael Karlsson, Fredrik Antelius, Björn Möller: ”Time Representation and Interpretation in Simulation Interoperability – an Overview”, 11S-SIW-049, SISO, April 2011.

[18] “The HLA Tutorial”, www.pitch.se, September 2012

[19] Björn Möller, Fredrik Antelius: “Object-Oriented HLA - Does One Size Fit All?”, 10S-SIW-058, SISO, April 2010.

[20] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley (1994), ISBN 0201633612

Author Biographies

FREDRIK ANTELIUS is a Senior Software Architect at Pitch and is a major contributor to several commercial HLA products, including Pitch Developer Studio, Pitch Recorder, Pitch Commander and Pitch Visual OMT. He holds an M.Sc. in Computer Science and Technology from Linköping University, Sweden.

MARTIN JOHANSSON is Systems Developer at Pitch Technologies and is a major contributor to several commercial HLA products such as Pitch Developer Studio and Pitch Visual OMT 2.0. He studied computer science and technology at Linköping University, Sweden.

BJÖRN MÖLLER is the Vice President and co-founder of Pitch Technologies. 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 M.Sc. 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 and the chairman of the SISO Real-time Platform Reference FOM PDG.