4.0 Review of Continuous and Discrete Event Simulations

One of the key concepts in any simulation is the manner in which time is advanced. There is a basic distinction in this regard between simulations based on continuous models and those based on discrete-event models (Figure 4.1).

Continuous vs. Discrete Simulations

<table>
<thead>
<tr>
<th>Continuous</th>
<th>Discrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Continuously advances time and system state.</td>
<td>• System state changes only when events occur.</td>
</tr>
<tr>
<td>• Time advances in increments small enough to ensure accuracy.</td>
<td>• Time advances from event to event.</td>
</tr>
<tr>
<td>• State variables updated at each time step.</td>
<td>• State variables updated as each event occurs.</td>
</tr>
</tbody>
</table>

Figure 4.1: Continuous vs. Discrete Simulations

Continuous models assume continuously advancing time and a continuously changing system state. Computationally, this is represented by a time variable which advances in increments small enough to accurately simulate the continuous behavior of the model. Updates to the variables representing the state of the system are typically computed for each step in time (Figure 4.2).
Figure 4.2: Simulation Time Steps

Discrete-event simulations are based on models that assume an unchanging system state until an event occurs that produces an instantaneous change in the state of the system. In these simulations time advances from event to event in chronological time order.

4.1 HLA Classes, Objects, and Attributes

In the HLA, the different kinds of physical objects that are to be simulated are represented as classes (e.g., automobiles). Classes may be further divided into subclasses (e.g., 4-door and 2-door). Specific instances of a class are objects of that class (e.g., myCar and yourCar). Data that describes a simulated object are called the attributes of that object. The set of values of all of an object's attributes define the state of that object (Figure 4.3).
Simulation Objects

A program that simulates one or more classes of objects is known as a federate. Various federates (including multiple copies of some federates), running on one or more different machines, may join into a system simulation called a federation (Figure 4.4). The description of a federate's classes, objects, and attributes is given in the Simulation Object Model (SOM) for that federate. If this information is exchanged between federates, the information also appears in the federation's Federation Object Module (FOM) and Federation Execution Data (FED) files.
Figure 4.4: Simulation Structure

In an implementation of a federate using a specific object-oriented language such as C++, HLA classes may be represented as C++ classes, and HLA objects represented as instances of those C++ classes. HLA object attributes may be represented as those instance variables of an object that collectively define its state. Typically, routines for accessing and manipulating the object's attributes are included in the class. This is an obvious structure for implementing an HLA object, but it is not required by the standard. Also, care must be taken, because HLA is a language-independent standard, and the rules of any specific language for the behavior of classes and subclasses may not exactly match that defined for the HLA class structure.

An example that uses this implementation structure may be found in the HelloWorld federate. This federate defines a class `Country`, as the objects that are being simulated are specific countries in an environment consisting of a world of several countries. Numerous member functions are defined for class `Country`. Some are accessor-type functions such as `Country::SetPopulation`, which sets the population attribute for the Country to the specified value. Other functions in class `Country` actually provide simulation functionality, such as `Country::Register`, which registers its country with the federation.
4.1 Updates and Interactions

The state of the system is defined as a set of values, or attributes, for each object being simulated. Changes to the state of an object may be communicated to other federates in the federation by sending an update message to the federation using the RTI service `Update Attribute Values` (Figure 4.5). Other federates will receive the updated attribute values when the RTI calls the federate’s `Reflect Attribute Values` routine. (Note that the dagger indicates that this is a callback routine, actually located in the federate’s code.)

---

**HLA Messages -- Updates**

**Updates**

- Send/Receive new values of attributes at end of each time-step.
  - Send: `Update Attribute Values`
  -Recv: `Reflect Attribute Values`

- Controlling unnecessary message traffic:
  - Update:
    - `Enable/Disable Attribute Relevance Advisory Switch`
    - `Turn Updates On/Off for Object Instance`
  - Reflect:
    - `Enable/Disable Attribute Scope Advisory Switch`
    - `Attribute In/Out of Scope`

---

**Figure 4.5: HLA Messages -- Updates**

There is a control mechanism to avoid sending unnecessary messages. This mechanism is enabled/disabled by the RTI service call: `Enable/Disable Attribute Relevance Advisory Switch`. The federate will be notified by the `Turn Updates On/Off` RTI callback whether or not there are federates able to receive the updates. Sending updates should be suspended upon receiving the `Turn Updates Off` callback from the RTI. Similarly, the federate that is interested in receiving updates will be notified whether or not there is any federate capable of sending the update if the `Enable/Disable Attribute Scope Advisory Switch` RTI service has been called. If no other federate is currently capable of sending
If a simulation generates events, which cause an immediate change of state, it may send a message containing information about the event to other federates in the federation by using the RTI service: *Send Interaction*. Other federates may receive this message when the RTI calls the federate’s *Receive Interaction* callback routine (Figure 4.6).

Interactions (events) have various data associated with them describing the characteristics (parameters) of the event. Events, like simulation objects, are divided into classes. An interaction class is declared which contains parameters quantifying the exact nature of the event. (For example, a *traffic_light_color_change* event might have a parameter that specifies the current color of the traffic light.) A specific event is considered a member of this interaction class, with specific values for its parameters. The mechanism for controlling unnecessary message traffic for interactions is enabled/disabled using the RTI calls: *Enable/Disable Interaction Relevance Advisory Switch*. The RTI signals whether other federates are able to receive this interaction by using the callback routines: *Turn Interactions On/Off*. 
4.2 Publish and Subscribe Mechanism

It is possible that a federation has a large number of federates, simulating a wide variety of different physical entities. Not every federate is interested in receiving all of the messages generated by every other federate. For example, an automobile simulation may not have any need for data about an aircraft if there is no possible interaction between the aircraft and the automobile. The air traffic control simulation, however, may be very interested in the messages from this aircraft simulation. It is costly (in terms of simulation performance) to send unnecessary messages, and some kind of management is needed for the message delivery system of the federation (Figure 4.7).

The *publish and subscribe* mechanism is used by the HLA to manage the distribution of messages between the federates in a federation. (Both interactions and updates are considered messages.) Each federate defines to the federation what data are to be *published* (sent in a message), for each update or event using *Publish Object Class* and *Publish Interaction Class* RTI service calls. Each federate declares to the federation which updates and interactions (events) it is interested in receiving by subscribing to those messages using the *Subscribe Object Class Attribute* and *Subscribe Interaction Class* RTI service calls.

The “publish” calls are used to describe the data to be sent in messages to the RTI by a class of objects. As particular instances of these objects are created by a simulation, they are registered with the system by the RTI service call: *Register Object Instance*. If any federates have subscribed to objects of this class, they will receive a *Discover Object Instance* callback from the RTI that will inform them of the existence of the new object.

Notice that the subscribing federate does not need to know in advance exactly which, or how many, objects of a particular class will be of interest, only in which classes of objects it will be interested. (E.g., an automobile federate may be interested in all other automobiles, but not in aircraft. How many automobiles will be of interest depends on the number of automobiles that are introduced into the simulation.) Similarly, a federate that is publishing data does not need to know in advance anything about the federates that will be interested in subscribing to its data (Figure 4.7).
Object and Interaction Registration

<table>
<thead>
<tr>
<th>Objects</th>
<th>Interactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Publish/Subscribe</td>
<td>• Publish/Subscribe</td>
</tr>
<tr>
<td>– Publish Object Class</td>
<td>– Publish Interaction Class</td>
</tr>
<tr>
<td>– Subscribe Object Class</td>
<td>– Subscribe Interaction Class</td>
</tr>
<tr>
<td>• Object Registration</td>
<td>• Enable Attribute Relevance Advisory Switch</td>
</tr>
<tr>
<td>– Register Object Instance†</td>
<td>• Enable Interaction Relevance Advisory Switch</td>
</tr>
<tr>
<td>– Discover Object Instance‡</td>
<td></td>
</tr>
<tr>
<td>• Controlling Instance Registration:</td>
<td></td>
</tr>
<tr>
<td>– Enable/Disable Class Relevance Advisory Switch</td>
<td></td>
</tr>
<tr>
<td>– Start/Stop Registration for Object Class</td>
<td></td>
</tr>
</tbody>
</table>

![Image](image.png)

**Figure 4.7: HLA Object and Interaction Class Registration**

On the other hand, if no other federate is interested in the data from objects of a particular class, simulated by federate “A”, then federate "A" could save simulation overhead if it stopped sending the data at all! There is a mechanism in the HLA for handling this situation. A federate may express an interest to the RTI in controlling the sending of unnecessary messages by using the RTI service *Enable Attribute Relevance Advisory Switch* for updates, or the RTI service *Enable Interaction Relevance Advisory Switch* for interactions. (*Disable ...* routines are available in both cases.) When any other federate subscribes to the data published by a federate (and who has issued the above enable commands to the RTI), the RTI will call the federate’s *Turn Updates On For Object Instance*† or *Turn Interactions On*† callback routines. (*Turn ... Off* routines are also used in both cases.)

Also, there is no need to register an object of a certain class, if no federate exists that has subscribed to it. The RTI will call the federate routines *Start/Stop Registration for Object Class*‡ to control the object registration calls if the federate has previously called the RTI service *Enable Class Relevance Advisory Switch*. This advisory may be stopped by calling the RTI service: *Disable Class Relevance Advisory Switch*. 
4.3 Time Stamp Ordered and Receive Ordered Updates

Most simulations are written so that messages containing events or updates have an associated "time-stamp" and must be delivered to the receiving federate in the correct order with respect to this time-stamp. Messages that must be delivered in the correct time order are said to be delivered in Time Stamped Order (TSO). A simulation that is publishing time-stamped data (TSO) may publish messages in any time order, and the RTI will guarantee to deliver the messages in correct time order (Figure 4.8).

Other types of messages, such as informational messages, may not have an associated time stamp, and are delivered upon arrival, without regard to the time when the message was sent. Messages delivered in the order in which they are received are said to be delivered in Receive Order (RO). An example might be an event that signals "The federation will shut down at 12:00." This message should be delivered as soon as it is available, asynchronously with respect to the current federate's simulated time.

Message Order

Two types of Message Ordering

- **TSO (Time Stamped Order)**
  - Messages delivered to federate in order of time stamp
  - RTI guarantees that no messages will be received from past

- **RO (Receive Order)**
  - Messages delivered to federate in order received
Time Management is concerned with mechanisms for controlling the advancement of time for each federate in consonance with the federation time axis. This advancement of time for each federate is controlled by the time-management services so that federates will receive messages in the correct time order, and no federate will receive messages from a time in the past (after it has already advanced time past the time stamp of the message).

4.4 Time-Regulating and Time-Constrained Federates

In order to guarantee the correct, time-ordered delivery of messages to a federate, the time for each federate must be coordinated with the time of other federates. Some federates may finish processing all known data before others; however, no federate could be allowed to suddenly advance its time to the next time step, until it has received all data up to that point in time from all other federates. Hence there are limitations on the advancement of time by a federate that receives time stamped data, and the federate is said to be time constrained (Figure 4.9).

Regulating and Constrained Federates (1)

- To receive TSO messages in Time Stamped Order, Federate must declare itself Time Constrained.
- To send TSO messages, Federate must declare itself to be Time Regulating.
- By default, Federates are neither time constrained nor time regulating.
- To become time constrained, use RTI service Enable Time Constrained.
- To become time regulating, use RTI service Enable Time Regulation.

Figure 4.9 Time Regulating and Time Constrained Federates

In order for time management services to properly control the advancement of time to time "T" for time-constrained federates, the RTI must be informed when all publishers
data are finished sending any messages up to time "T." The senders of time-stamped messages therefore regulate the advancement of time as simulation time cannot be advanced to time "T" until all publishers are finished sending data up to that time. Any federate that publishes time-stamped data is known as a time regulating federate.

When a federate first joins the federation, it is considered to be neither time regulating nor time constrained. In this state, the federate will receive time stamped messages in arbitrary order, as RO messages, until it becomes time constrained by calling the RTI service Enable Time Constrained. A federate will not correctly send time stamped messages, as the messages will be sent without a time stamp as RO messages, until it has called the RTI service Enable Time Regulation. The RTI responds to these service requests by calling the federate's Time Constrained Enabled’ routine or its Time Regulation Enabled’ callback routine. Federates may be both time constrained and time regulating, neither time constrained nor time regulating, or any combination of the two.

Time regulation may be disabled by calling the RTI service Disable Time Regulation and time constrained may be disabled by calling the RTI service Disable Time Constrained. In these cases, the RTI does not respond, and all future messages will be sent and received in receive order (RO).

4.5 Lookahead, LBTS, Logical Time, Effective Logical Time

In a federation, each individual federate has its own idea about time, and the local value of time for a federate is called its logical time. The logical time is assumed to be zero when the federate first joins the simulation. If the federate is to receive time-stamped data from other federates, it must first become time constrained. Time is constrained in the sense that the advancement of logical time by a federate is constrained by time management to not advance beyond the least time of a time-stamped message that can possibly be received from any other federate. The maximum time to which a time-constrained federate may advance, and follow the above rule, is known as the Lower Bound Time Stamp (LBTS) (Figure 4.10).
In order for the RTI to be able to compute the LBTS for a federation, it must have a "promise" from each time-regulating federate (one that publishes TSO data) as to the minimum time stamp that will be used for any future message. This is provided by the federate when it enables time regulation through a parameter known as its lookahead. The federate is not allowed to send any message with a time stamp less than the current logical time plus the value of lookahead. (There is a special case where the federate has a zero lookahead. In this case, it must send messages with time stamps greater than its current logical time.) The first time that the federate is allowed to use as a time stamp is its effective time, and is equal to the current logical time plus the lookahead (Figure 4.11).
As soon as a federate becomes time regulating, it is subject to the federation's requirements for time stamps. The federation already promises not to allow any messages to be delivered to any federate before the value of its LBTS. To ensure proper behavior after a new federate becomes time regulating, the RTI service Enable Time Regulation has an implied time advance request. The logical time of the federate making the request is increased from its current value (possibly zero) to a new logical time that would prevent it from being able to use a time stamp for any message that was less than or equal to the minimum LBTS of all previously existing federates. The new logical time will be communicated to the federate when the time advance is granted through the federate's Time Advance Grant' callback routine, and will be equal to the requested time, or the current LBTS minimum of other existing federates, whichever is greater. (Note that the lookahead for the federate is not used in this calculation.)
4.6 Time Advancement and Message Reception

During the course of a simulation, a federate may increase its logical time by calling the RTI service: Request Time Advance. The logical time is actually increased when the RTI calls the federate’s Time Advance Grant† callback function (Figure 4.12).

**Requesting Time Advancement**

- **Continuous Simulations**
  - To request an advance in time, use the RTI service: *Request Time Advance*.
  - RTI will notify when it's ok to advance time by calling: *Time Advance Grant†*.

- **Discrete event simulations**
  - The RTI service: *Next Event Request (t1)*, requests time advancement to time of next event, or to *t1*, whichever occurs first.
  - RTI will notify when to advance time by calling: *Time Advance Grant†*, and will specify the amount of the granted time advance.

A federate implementing a discrete-event simulation does not have a known, or predictable, time step. Instead of making an explicit time advance request, a discrete-event simulation may call the RTI service: *Next Event Request*. This requests that the federate's logical time be increased to the time of the next event, or to the time specified as an argument if no events will be received before this time. Like the *Time Advance Request*, this call guarantees that no future messages will be sent with time stamps before the specified time plus the *lookahead*. 
Federates That Are Both Time Regulating and Time Constrained

If a federate both receives TSO data (in order of the time stamp) and sends TSO data, then that federate must declare itself to be **both time regulating and time constrained** (Figure 4.13). This federate is then both constrained to not advance time beyond the LBTS, and regulating. As a regulating federate, it must not attempt to send data before its lookahead period (Figure 4.14).
Figure 4.14: Time Regulating and Time Constrained

When does a federate receive its messages? Messages are normally only received when the federate is in the time advancing state (Figure 4.15). This is the interval between the time advance request and the subsequent grant. In other words, during this interval, all of the messages are received for the next "time step". The time advance is granted; processing for the time step is completed, and the next time-advance request is made.

RO messages are normally received only during the time advancing state. This could result in quite a delay for the receipt of potentially urgent messages. A federate may make the RTI service call: Enable Asynchronous Delivery to allow RO messages to be delivered any time tick is called, regardless of the temporal state of the federation.
Messages are only received when in a **time-advancing state**.

- A Federate is put into a **time-advancing state** by:
  - *Time Advance Request* OR
  - *Next Event Request*
- To enable receipt of RO messages at other times:
  - *Enable Asynchronous Delivery* (Prevents excessive delay for urgent events)

### 4.7 The HelloWorld Federate

The HelloWorld federate makes the *Enable Time Constrained* and the *Enable Time Regulation* RTI service calls, and as a result is both a time-constrained and a time-regulated federate. Time constraint is necessary for the federate to receive time-stamped messages in order of increasing time, and time regulation is necessary to send time-stamped messages to other federates.

The *Enable Asynchronous Delivery* RTI service is called to allow the delivery of RO events while in the simulation processing part of the program as well as when the federate is in the time-advancing state. These messages will only be received when the federate relinquishes control to the RTI by calling the function *tick*. Note that *tick* is a part of the a specific implementation of the RTI, not a part of the standard.

The lookahead value for this federate is one, although the timestep is ten! This results in an offset of one time unit in the values of time used at each time step between two federates running in the federation.

**Assignment**
Use the HLA Course Lab Notes to help you to create a new, passive federate, the UNFederate, that collects and displays data from any running HelloWorld federates.

Suggested Readings


2. Read Chapters 3, 5, 7, and 8 of the High Level Architecture Run-Time Infrastructure Programmer’s Guide. Also look up the interface specifications for the interfaces used in this lesson in the Appendices to this Guide. Compare the information to that obtained in [1] above. The Guide gives the actual C++ prototypes for the RTI services, for a particular implementation of the RTI. Details of operation are provided here that are not possible to specify in the implementation-independent descriptions of the services provided by the standard.