6.1 Views of Time by Various Types of Simulations

Parts 1 through 5 of the course notes have presented various aspects of the HLA, and the practical exercises have applied some of these concepts to a simple HLA federation, *HelloWorld*. In this section we look again at the way in which time is handled in HLA-based simulations. We will review some of the earlier material on time management, add some new information, and apply it to different simulation scenarios.

6.01 Types of Simulation Models

In this section we will address the question of applying the HLA to different kinds of simulation. In a general sense, simulation may be used for a variety of reasons:

- to assist in the design of a new system
- to better understand the behavior of an existing system
- for training purposes
- as an educational tool, etc.

In addition, the modeling approach can vary greatly from one application to another. For example, the simulation of an aircraft may be used to understand how a new design will function under a variety of conditions, some of them severe, such as the effect of wind shear or of a sudden loss of engine power. In cases like these, it is important to represent the underlying behavior of the aircraft structural components and control surfaces as accurately as possible, based on the laws of physics. A training simulator for the same aircraft, however, may operate largely on the basis of table look-ups in which pre-computed behaviors are stored and replayed. The same aircraft is being simulated, but the models may be very different. The combination of the modeling approach and the purpose of the simulation produces a variety of different simulation scenarios.

We will begin by reviewing, in a general way, a number of different types of simulation. We will then address these types of simulation from the perspective of the HLA. Let us start by reviewing the differences between continuous simulations and discrete simulations. As stated in Part 4:
Continuous models assume continuously advancing time and a continuously changing system state. Computationally, this is represented by a time variable that 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.

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.

6.02 Types of Simulation Implementations

While the observance of a distinction between continuous and discrete simulations has long been a practice in the simulation community at large, HLA literature often uses a different terminology and refers to time-stepped and event-driven simulation. (See Figure 6.1.)

## Types of Simulation Models

- Types of Simulation Models
  - Continuous Models
  - Discrete Models
- Types of Simulation Implementations
  - Event-Driven Implementations
  - Time-Stepped Implementations
  - Real-Time Implementations
  - Scaled Real-Time Implementations
  - Non Real-Time Implementations

Figure 6.1 Types of Simulation Models

Time-stepped means that time advances in equal increments. This would apply to the simulation of an electronic digital circuit, for example, in which the outputs
of its electronic components change only synchronously with the pulses of a system clock. It also applies to many implementations of continuous models in which the step size of the numerical integration algorithm is constant (fixed-step integration) as is the case with HelloWorld. Some integration algorithms use a variable step-size (to control errors) but still communicate with other simulation modules at fixed communication intervals comprising several steps. These would still qualify as time-stepped. If, however, a variable step-size simulation communicates with other modules after each time step, then the end of each step could be regarded as an event, in which case the process would be classified as event-driven.

6.03 HelloWorld as a Time-Stepped Example

The HelloWorld Federate (see Figure 6.2), provides an example of a time-stepped (continuous) model, because the changes in population are described by an ordinary differential equation and this equation is solved using a simple, fixed-step, integration algorithm (Euler). The differential equation states that the rate of increase of population, at any instant in time, is proportional to the size of the population itself. The larger the population, the greater the rate of increase. This leads to an exponentially increasing population. In practice, some limiting process is bound to apply eventually, such as shortage of food, disease caused by overcrowding, etc., but that is not built into the simple HelloWorld model.

Another approximation in the HelloWorld continuous population model is that it does not recognize that population is essentially an integer quantity. Population is represented by a real (floating-point) variable and can have non-integer values even though a fractional person is not possible. As long as the population is large, this does not matter, but for very small populations, it could be significant.

Looking at this in another way, population growth is essentially discrete: it occurs one person at a time! So we have a continuous model of a discrete system. This is common in other areas involving large populations, for example, traffic flow on a freeway. We would normally consider the vehicles in a traffic simulation to be discrete objects and use a discrete model to simulate the system. In some freeway models, however, traffic is modeled like a continuous fluid flow in vehicles/hour. So once again we have a continuous model of a discrete system.

In summary, a model is a mathematical abstraction that captures some aspects of the behavior of a real-system that are of interest. Strictly speaking, we should refer to continuous models and discrete models rather than systems.
6.04 Real-Time vs. Non Real-Time Simulations

A simulation can also be characterized as either a real-time or a non real-time simulation. Many simulations execute as rapidly as possible, as determined by the time taken to complete all the necessary calculations. In some cases, the simulated behavior might be generated in a period of time that is much greater than the duration of the actual behavior. Simulations of elementary particle motions or high-speed electrical phenomena might fall in this category. Other simulations might execute much faster than real time, such as in the simulation of an astronomical system, or in human population studies (such as HelloWorld). Most simulations like these would be referred to as non real-time. There are, however, a number of situations that might be qualified as real-time.

One type of real-time simulation occurs when human players provide inputs to an executing simulation and so influence the subsequent course of the simulation. Such simulations do not necessarily guarantee that the timing of the results of the simulation will accord precisely with the timing of the behavior of the system being simulated. Many simulated training exercises, computer war games, and management games fall into this category.

There is a special situation in which the time scale of the simulation bears a fixed relationship to the time scale of the real system (say 100 times faster or 1000
This can be called scaled real-time, but the most restrictive definition of real-time simulation requires that the outputs of the simulation occur with exactly the same timing as the corresponding outputs of the real system.

Each of these approaches to simulation requires special handling of the advancement of time and in order to be able to provide for them, we need to understand how the time-management features of the HLA would work in each case.

### 6.1 Schemes for Time Management in HLA

Let us first review some of the basic aspects of time management in the HLA.

First of all, it is important to recognize that time management in the HLA is not about constraining the federates in a federation to execute in real (wall-clock) time, although the designers of federates can use the HLA time-management services to help them achieve this, if they wish. HLA does not support the concept of wall-clock time for a federation as a whole. Each federate has a logical time and the purpose of time management is to coordinate the advance of the logical times of all federates.

The HLA allows flexibility in the way a particular federate handles time. It supports a number of time-management schemes, including the following (see Figure 6.3):

1. **No time management**, in which each federate advances time at its own pace

2. **Conservative synchronization**, in which federates advance time only when it can be guaranteed they will receive no past events

3. **Optimistic synchronization**, in which a federate is free to advance its logical time, but has to be prepared to roll-back its logical time if an event is received in its past

4. **Activity scan**, in which federates proceed through periods during which they exchange messages at the same time until they agree to advance their logical times together.
Time-Management Schemes

- No time management
  - Each federate advances time at its own pace
- Conservative synchronization
  - Federates advance time only when guaranteed that no past data will be received
- Optimistic synchronization
  - Free to advance logical time, may have roll-back
- Activity scan
  - Advance time by mutual agreement with other federates

Figure 6.3 Time-Management Schemes

In all cases the logical time belonging to a federate conforms to the following (see Figure 6.4):

1. It has an initial value

2. Its value is not tied to any system of units. Unit logical time could represent a microsecond, a second, an hour, a decade or whatever the federation convention requires

3. It is well-ordered. The RTI can determine which of two times is the greater

4. It is always greater than the initial time
Logical Time Restrictions

- Initial value
- Value not tied to any system of units
- Well-ordered
- Always greater than (or equal to) initial time
- Time is effectively discrete
- Has special value of positive infinity that is greater than any other value

Figure 6.4 Logical Time Restrictions

5. It is effectively discrete. It has an epsilon value representing the smallest possible difference between two logical times

6. It can assume a value of positive infinity, which is greater than any other value.

Each federate determines its own degree of involvement in the time-management process. It can choose not to participate in time management. (It is said to be neither time-regulating nor time-constrained as discussed in Part 4.) This is the default state of a federate when it joins a federation. It can choose to be time-regulating, in which case it is capable of generating time-stamp ordered (TSO) events. It can choose to be time-constrained, in which case it is capable of receiving TSO events. It can also choose to be both time-regulating and time-constrained, in which case it can both generate and receive TSO events. The important point about TSO events is that it is possible to determine the order in which they were sent, and they are delivered to the receiving federates in this order. Receive ordered (RO) events, on the other hand, are delivered in the order in which they are received and, lacking time stamps, it is impossible to determine the order in which they were sent. Since a federate that is not time-constrained cannot receive TSO events, an event that is sent with a time stamp (sent as a TSO event) will be received as a RO event, without the time stamp, if the receiving federate is not time-constrained. RO events bear no relationship to logical time.
To summarize, for a time-constrained federate operating with conservative synchronization:

1. TSO events are delivered to the federate in time-stamped order, irrespective of the order in which the originating events are sent.

2. No event will be delivered to the federate with a time stamp less than the current logical time of the federate.

To guarantee the above conditions without the possibility of a deadlock in which no federate is able to advance its clock, it is necessary for each time-regulating federate to declare a lookahead, which is a time period beyond its current logical time for which it is forbidden to send events. In other words, it is a notice period after which events can be sent and it gives the receiving, time-constrained federates the leeway to advance their own logical time free of concern that in doing so they will be open to the danger of receiving events with a time stamp less than their current logical time, i.e., in their past.

Looking at this from the perspective of a time-constrained federate, capable of receiving events from a number of other federates, there is a time, beyond its current logical time, before which it knows that no federate will send it an event. (See Figure 6.5.) It is able, therefore, to advance its time to this new time without fear of subsequently receiving an event in its past. This time is known as the Greatest Available Logical Time (GALT) of the federate.

Note that a federate is allowed to change its lookahead during the federation execution. It must request such a change and if it is reducing its lookahead by an amount $\delta t$, then it must wait until it has advanced its own logical time by at least $\delta t$ to avoid violating the time-management guarantees.
6.2 Real-Time Execution

Many HLA federations execute at a rate that is approximately equal to real-time. In other words, events appear to occur at a normal rate, but there is no attempt to ensure precise agreement in time with real events. In the HLA literature, this is what "real-time simulation" means. That is, intervals between events are perceived as normal by an observer. Federations do not naturally execute in this way. Unless the real-time property is designed into a federation, it will simply execute as fast as the computers are able. To achieve the appearance of real time, it may be necessary to make a federate go into a waiting state so as to slow the simulation down.

Imagine a federation that models traffic flowing through an intersection. Vehicles, traffic signals, and pedestrians might all be represented by different federates. Vehicles move through the intersection when the light is green and their intended path (straight on, left or right turn) is not blocked by other vehicles or pedestrians. Similarly pedestrians cross the road when the signal says WALK and the path is clear. Traffic signals change state, either according to a strict time schedule, or controlled by the vehicles passing a sensor in a particular location. If all of these operations were programmed without regard to time-management, it is likely that the whole simulation would run very much faster than real time. Constraints must be introduced which make the traffic lights wait
an appropriate time before changing from red to green, or which ensure that vehicles and pedestrians don't appear to move at supersonic speed through the intersection.

Consider a vehicle that is stopped at the traffic signal intending to move straight ahead when the signal changes. Assume there are no obstacles to forward progress when the light changes. The logical time of the vehicle federate, \( T \), coincides with the time at which the light change occurs. In this particular run, this occurs at a wall-clock time of \( T_c \). The vehicle federate does a calculation that determines it will take, say, 5 seconds to clear the intersection, but the calculation is completed in 0.1 second at a wall-clock time of \( T_c+0.1 \) sec. The federate advances its logical time to \( T+5 \) seconds and must wait for another 4.9 seconds for wall-clock time to catch up. After the waiting period is complete, the federate issues its updated position. In other words, the federate waits until wall-clock time catches up before it issues (and receives) updates and then it immediately proceeds to a new set of calculations of the next set of updates.

The appearance of running in real time can be maintained as long as the computers can process the necessary information faster than real time and then wait for real time to catch up.

Time management is effected by the sending and receiving of events (an event is characterized by its ability to have an associated time stamp). A federate sends
an event when it calls one of the following (see Figure 6.7):

1. update attribute values
2. send interaction
3. delete object instance

A federate receives an event when the RTI calls one of the following on it:

1. reflect attribute values†
2. reflect interaction†
3. remove object instance†

Sending and Receiving Events

- Sending events
  - Update attribute values
  - Send interaction
  - Delete object instance
- Receiving events
  - Reflect attribute values†
  - Reflect interaction†
  - Remove object instance†

A federate that is time-regulating should make regular requests to the RTI to allow it to advance its clock because this affects the rest of the federation. It is these requests that co-ordinate the advance of time in the federation, not the sending of time-stamped events. The reception of events is controlled by the advance of time, but events do not advance time.

Conservatively synchronized federates, such as we are discussing here, will call one of two RTI services to advance time, either Time Advance Request or Next Message Request(). The corresponding callback service in either case is Time Advance Grant†. Conservative federates are constrained to not advance their local time until permitted by the Time Advance Grant†. This advance in time is
limited by the GALT. Optimistic federates are not constrained by the GALT and can advance local time into the future, beyond the GALT. These federates require additional time-management services, which are discussed in Module 2 under the topic Advanced Time Management.

6.3 Synchronization of Federate Timing

One of the features that makes the HLA so flexible is that, as mentioned earlier, it does not dictate a timing regime to any federate. Each federate is autonomous with respect to its determination of a timing regime. This has some consequences which, for those who may be familiar with different simulation environments, may be unexpected. As an example, let us examine once again the operation of *HelloWorld* (see Laboratory Notebook, Section 1.3.5).

You will recall that, in accordance with normal HLA procedures, the country federates of *HelloWorld* join the federation one at a time. The first federate to join creates the federation and starts to execute. Federates are made both time-regulating and time-constrained. Each country federate calculates its own population increase during each time step and publishes this information to all other country federates. It also receives the population data from all the other federates.

In the original version of *HelloWorld* we noted that the second federate, C2, to join the federation was, in general, a little out of step with the original federate, C1, so that its time stamps differ by a fixed amount from the time stamps of C1. This problem was fixed in the modified version. Notice also that in Figure 10 on p. 10 of the Laboratory Notebook, which displays output from the modified *HelloWorld*, the populations of C1 and C2 are reported as 194.774 and 167.1 respectively at time 670 and 671, respectively. The model for population growth used in these two countries is identical with identical data, so why the difference? The reason is that C2 has taken one large step from time=0 to time=671 and has calculated the population to be $100 + 67.1 \times 0.01 \times 100 = 167.1$. On the other hand, C1 has been advancing in steps of 10 and increasing its population by 1% in each step, which leads to the higher figure of 194.774.

This example is not presented to show that there is anything wrong with the HLA, but rather to show how important it is to consider carefully the way in which time is handled in each federate and the time-dependent relationships between different federates. Otherwise unexpected results may occur.

This discussion raises the question: “How could we launch a federation in which all the federates start executing and advancing their logical times at the same instant in time?” In other words, we want to establish a world federation,
consisting of a number of countries, which are given some initial populations at a starting time represented by time=0, and which all advance time in a synchronized fashion.

Regulating and Constrained Federates

![Diagram of Federates](image)

**Figure 6.8 Time-Regulating and Time-Constrained Federates**

Clearly, there would have to be a way to prevent the federations that join first, from advancing time, at will, until the last federation joins. The federation, then, needs some way of recognizing that the last federation has joined. Maybe a synchronizing federate would join first and would then count the countries as they joined until some agreed number of countries had joined. It could then generate an interaction that would effectively start all federates advancing their times at once. Note that we are still not synchronizing the federation with wall-clock time, but merely ensuring that the logical times of the federates all advance together, avoiding the long initial step we saw in the earlier HelloWorld examples.

Another way of achieving the same result would be to have a modified federate, to be used specifically as the last federate to join, so that when this modified federate joined the federation, it generates the required interaction to start time advancing.

The above discussion provides an example of "Late Arrival" which is discussed in the *HLA RTI Programmer's Guide*, Chapter 3.4.
The problem with the late-arriving federate is that, if it requires to be both time-regulating and time-constrained, it must assume a time that guarantees that it cannot generate a TSO message earlier than the GALT of the remaining federates. This is illustrated in Figures 6.8 and 6.9, in which Federate 2 is a late-arriving federate and is assigned an initial time of 20.

It is important to recognize that three things must have happened for an event to be delivered TSO (see Figure 6.10):
1. The sender must be time-regulating
2. The receiver must be time-constrained
3. The event itself must be designated TSO.

Requirements for TSO Delivery

- Sender must be time-regulating
- Receiver must be time-constrained
- Message must be designated TSO

6.4 Review of Time Management Services

Events are sent and received by the Local RTI Component (LRC) of the appropriate federates. Each LRC contains two queues, a FIFO receive queue and a time-stamp queue. Events that meet the above TSO criteria are placed in the time-stamp queue, which is ordered based on the time value of its contents. The receive queue contains RO events (events that do not meet all the criteria) entered in the order in which they arrive. Since there are no time constraints on the delivery of RO events, the receive queue is drained any time the federate provides sufficient time to libRTI, using \textit{tick()}, for example. See Figure 6.11.
Receiving Events

Per Federate Queues

- FIFO Receive Queue
- Priority Time-Stamp Queue
- Receive ordered events are queued as they arrive (i.e., on a first-come-first-serve basis). The FIFO queue will be drained if the federate provides sufficient time to libRTI.
- Time-stamp ordered events are queued based on the associated time value. The queue will be drained up to and including messages at the current federate time if the federate provides sufficient time to libRTI.
- Receive ordered events are queued as they arrive (i.e., on a first-come-first-serve basis). The FIFO queue will be drained if the federate provides sufficient time to libRTI.

In the example illustrated by Figure 6.9, Federate #3 generates a TSO event. Federate #6 sees this as a RO event. The event does not arrive as a TSO event because Federate #6 is unconstrained and therefore unable to receive events in time-stamped order. The same event is received by Federate #2, which is time-constrained, as a TSO event.

The RTIambassador and FederateAmbassador functions associated with time regulation and time constraint are (see Figure 6.12):

- `enableTimeRegulation()`
- `timeRegulationEnabled()`
- `disableTimeRegulation()`
- `enableTimeConstrained()`
- `timeConstrainedEnabled()`
- `disableTimeConstrained()`

By default, federates are not time-regulating. A federate uses the member function `enableTimeRegulation()` to request that it be made time-regulating. The LRC uses the callback function `timeRegulationEnabled()` to inform the federate that the request has been granted and informs the federate of its (possibly new) logical time to which it must advance to ensure that the GALT of existing federates will be honored. The time-regulating status can be canceled dynamically by calling `disableTimeRegulation()`, which takes effect immediately.
By default, federates are not constrained. A federate uses the member function `enableTimeConstrained()` to request that it be made time-constrained. The LRC uses the callback function `timeConstrainedEnabled` to inform the federate that the request has been granted. The time-constrained status can be canceled dynamically by calling `disableTimeConstrained()`,

Several additional time-management functions, listed below, are available to allow federates to make inquiries about time status and to modify lookahead (their purpose is self-explanatory):

- `queryLogicalTime()`
- `queryLookahead()`
- `modifyLookahead()`
- `queryGALT()`
- `queryLITS()`

Although the `tick()` method is not part of the HLA specification, it is an important method for RTI 1.3-NG, which is not fully multithreaded and which uses `tick()` to provide an opportunity for the LRC to do its work. Failure to "tick the LRC" can lead to federation-wide problems. A late-arriving federate can be prevented from joining a federation because information that needs to be passed to the LRC of existing federates is blocked by the failure of the federate to tick the LRC. This
could cause the entire federation to stall.

### 6.5 Review of Federation Management Services

Federation management includes such tasks as creating and destroying federations, joining and resigning federates, observing federation-wide synchronization points, and effecting federation-wide saves and restores.

![Federation Management Life Cycle](image)

**Figure 6.13 Federation Management Life Cycle**

These tasks are implemented through `RTIambassador` services and `FederateAmbassador` callback functions. For example, calling the `RTIambassador` method `createFederationExecution()`, causes the LRC to communicate with the RtiExec process, which creates a new FedExec process and associates it with the given federation name (see Figure 6.13). If the specified federation already exists, a `RTI_FederationExecutionAlreadyExists` exception is thrown. For federates that can both create a federation and join an existing federation, such as the HelloWorld federate, the call to `createFederationExecution()` can be made robust by catching and ignoring the exception.

The `RTIambassador` method `joinFederationExecution()` is called to associate a federate with an existing federation execution. It provides the names of the federate and of the federation it is trying to join. It also provides a pointer to an instance of a class implementing the `FederateAmbassador` callback functions.
The request to join may be issued before the federation has actually been created, even though an earlier call has been made to `createFederationExecution()`. In this case, a `RTI_FederationExecutionDoesNotExist` exception is produced. The code should make repeated attempts to join the federation, until the join is successful or until a predetermined number of join attempts is exhausted.

A federate resigns from a federation using the `resignFederationExecution()` method. Federation management has to make arrangements for the objects for which the resigning federate has update responsibility. The method has a single argument of type `RTI_ResignAction`. Sample choices are:

1. `unconditionallyDivestAttributes()`
2. `deleteObjects()`
3. `deleteObjectsThenDivest()`
4. `cancelThenDeleteThenDivest()`
5. `noAction()`.

The `destroyFederationExecution()` method attempts to terminate an executing federation. If successful, the `FedExec` associated with the federation terminates. If the invoking federate is not the last federate participating in the targeted federation, a `RTI_FederatesCurrentlyJoined` exception is thrown.

In the federate `HelloWorld`, where the federation is created and joined, respectively, the `RTI_FederationExecutionAlreadyExists` exception is caught and essentially ignored. Most remaining exceptions are caught, logged, and rethrown.

### 6.5.1 Federate Synchronization

The HLA provides functions for synchronizing activities between federates participating in a federation. The RTI provides mechanisms for exchanging information between federates. It is possible to associate times with exchanged information and thereby coordinate federate activities. The Federation Management Synchronization functions allow federates to communicate explicit synchronization points. Figure 6.14 illustrates the `RTIambassador` service calls extended to a federate and the resulting `FederateAmbassador` callback functions that together support a synchronization capability. The `RTIambassador` method `registerFederationSynchronizationPoint()` accepts a label, a tag, and (optionally) a set of target federates. (By default, all federates are targeted.) The label and tag are communicated to targeted federates. The specific role of the label and tag are outlined in detail in the appendices.
The RTI provides functions for coordinating federation-side saves and restores (Appendices A and C).

Figure 6.14 Federation Management Synchronization

Figure 6.15 Federation Management Save
6.6 The HLA Rules Revisited

To conclude this module, let us review the HLA Rules with which we started in Part 1 (see Figures 6.17 and 6.18).
Federation Rules

1. Federations shall have a FOM, documented in accordance with the OMT.
2. All representation of objects in the FOM shall be in the federates, not in the RTI.
3. During a federation execution, all exchange of FOM data among federates shall occur via the RTI.
4. During a federation execution, federates shall interact with the RTI in accordance with the HLA interface specification.
5. During a federation execution, an attribute of an instance of an object shall be owned by only one federate at any given time.

Figure 6.17 Federation Rules, Reviewed

Rules 1 through 5 deal with federations and 6 through 10 with federates.

Rules 1 and 6 specify that the federation and all the federates must be documented, and how this should be done (FOM and SOM). This topic was addressed in Part 3.

Rule 2 requires that the objects defined in the FOM must be represented in the federates and not in the RTI and Rule 3 specifies that data exchange between federates (i.e. FOM data) occurs via the RTI. Rule 4 introduces the HLA interface specification and its role in defining the interaction between the federates and the RTI. Taken together, Rules 2, 3 and 4 define the specific roles played by the federates and by the RTI in an HLA federation.

Rule 5 establishes the important requirement that only one federate can own an attribute of an object instance at any one time, but it does not rule out the possibility that the ownership of such an attribute might be passed from one federate to another during a federation execution.

Rules 7, 8 and 9 deal with the control and transfer of relevant object attributes. Rule 7 establishes that a federate can update the object attributes it owns and reflect (receive values of) objects in which it is interested, and that are owned by other federates. It also allows federates to send and receive interactions as specified in the SOM. Rule 8 allows federates to exchange ownership of
attributes dynamically (during a federation execution) and Rule 9 gives a federate control of the conditions under which it is required to provide updates of the attributes it owns. Note that we are talking about ownership of attributes, not of the objects themselves. Ownership of the attributes of a given object can be shared between several federates and passed back and forth between them, as long as this is provided for in the SOM.

Federate Rules

6. Federates shall have a SOM, documented in accordance with the OMT.

7. Federates shall be able to update and/or reflect any attributes of objects in their SOM, and send and/or receive SOM interactions externally, as specified in their SOM.

8. Federates shall be able to transfer and/or accept ownership of attributes dynamically during a federation execution, as specified in their SOM.

9. Federates shall be able to vary the conditions under which they provide updates of attributes of objects, as specified in their SOM.

10. Federates shall be able to manage local time in a way which will allow them to coordinate data exchange with other members of a federation.

Figure 6.18 Federate Rules, Reviewed

Rule 10 deals with time management. Note that it requires that federates be free to manage their own local time "in a way which will allow them to coordinate data exchange with other members of a federation". This is rather non-specific but Rule 10 needs to be interpreted in conjunction with Rule 4, which states that "federates shall interact with the RTI in accordance with the HLA interface specification". In other words, most of the restrictions that apply to the management of time are found in the RTI interface specification.

6.7 Concluding Remarks

The flexibility of the HLA provides for a wide variety of needs and scenarios in distributed simulation. With this flexibility comes a responsibility for the designer of HLA federates and federations to take considerable care in choosing how to handle time. Outputs of new simulations need to be carefully scrutinized to ensure that they handle time in the way that was intended.
In this Part we have focused considerable attention on the topic of time management. There are, however, several more advanced aspects of time management that have not been addressed, such as the detailed behavior of federates that use optimistic or activity scan approaches to time management. These topics are dealt with in the section on Advanced Time Management in Module 2.

**Assignment**

Modify the UNFederate Federate in HelloWorld so that it causes the country federates to wait until the last federate has joined the federation. Country federates should not advance beyond the point at which a joining federate can still be assigned a logical time of zero. This can be done by assuring that no country advances to the second time step until all have completed the first. The number of countries in the federation should be provided by user input to the UNFederate Federate.

**Suggested Readings**