

Communications, Data Distribution and Other Goodies in the HLA Performance Model

Sudhir Srinivasan
Mystech Associates, Inc.
5205 Leesburg Pike, Suite 1200
Falls Church, VA 22041
sudhirs@mystech.com

Paul F. Reynolds, Jr.
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
reynolds@Virginia.EDU

KEYWORDS

HLA, performance analysis, simulation modeling, federation analysis tool

ABSTRACT

At the last workshop, we reported on the design of a performance model for the High Level Architecture. The model is designed to provide federation designers a capability to conduct first order performance analyses before constructing the federation. This paper provides an update on the model, describing the design of some new features, including: (1) a general capability to specify different communications structures between federates, (2) modeling of data distribution mechanisms, and (3) modeling of event-driven federates. The powerful communications submodel allows the user to create multiple instances of predefined types of communication “cells” and connect them arbitrarily. Currently, the model establishes connectivity between federates based solely on the publish/subscribe relationships. We describe our design for modeling the Data Distribution Mechanisms in the Interface Specification, which permit more precise modeling of the connectivity among federates based on attributes and attribute values. The modeling of event-driven federates, though similar to the modeling of time-stepped federates, required a modification to the event selection loop in the federate submodel, described here.

1. INTRODUCTION

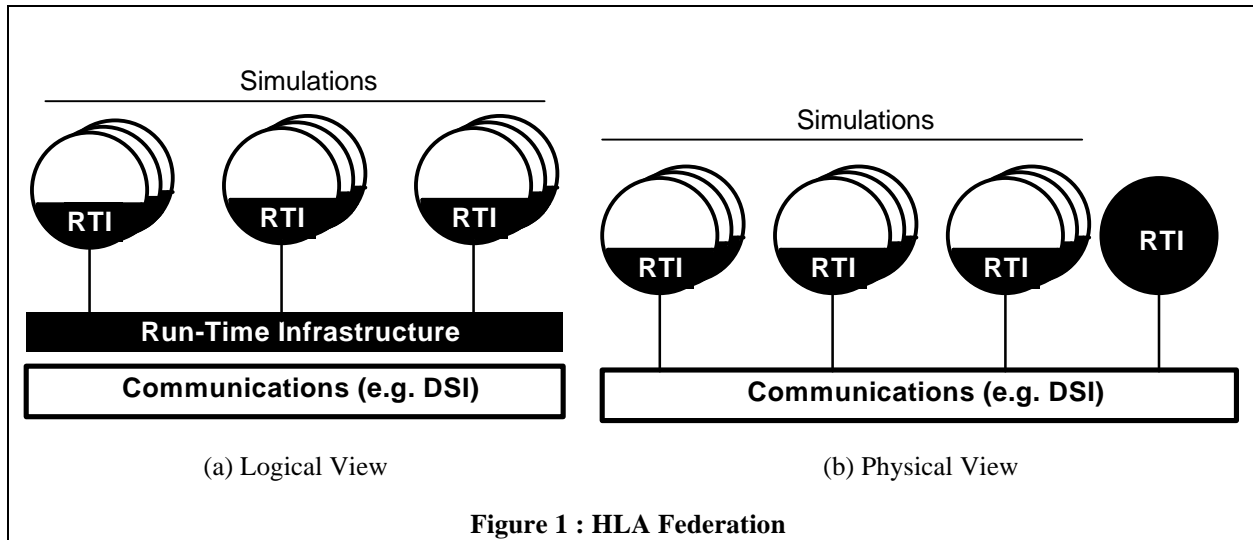
The goal of the performance model described in [SrRe95] is to provide users with a capability to model those aspects of a federation that are relevant to its performance. Since the publication of that paper, we have added several new features to the model, which are described here. These features, including flexible communications modeling, the modeling of data distribution mechanisms and the modeling of event-driven federates, facilitate the modeling of more realistic federations using the performance model. In Section 2, we summarize the basic design of the model briefly for convenience (for details, see [SrRe95]) and then describe the new features of the model in subsequent sections.

2. SIMULATION MODEL DESIGN

Our goal is to construct a simulation model with sufficient flexibility (i.e. it should be adequately parameterized) so as to allow users to model foreseeable HLA-compliant federations with ease. Since this is a performance model rather than a functional model, it is a stochastic abstraction of an HLA federation, focusing on resource usage and contention and ignoring the application-level details

of the simulated federation. Rather than simulating the various application-level details of a federation, we represent them by stochastic processes. However, those aspects of a federation that concern relevant performance metrics (such as resource usage) are modeled in detail. For example, while the generation of a message may be based on a stochastic representation of a federate, the transportation of the message from sender to receiver is modeled explicitly.

Figure 1 shows the logical and physical views of a typical HLA federation (we have omitted the layer commonly called “middleware” since it is typically not relevant to the performance model). While the RTI appears logically as a layer above the underlying communications of the federation, physically, the RTI is implemented as a combination of distributed components located at each federate (some designs can have an additional centralized component). Based on the physical view, our model consists of two submodels: a *federate submodel* (including the local RTI component) and a *communications submodel*. A separate model of the central RTI component is not required since it can be captured using the federate submodel. The communications model encapsulates *all* physical communications in the federation.



2.1 Federate Submodel

We consider a federate as a simulation residing on a single *machine*, with a single interface to the RTI. Since we do not focus on the semantics of federates, we can model federates that span multiple machines easily by considering a federate as a collection of machines (although the issue of a federate with multiple interfaces to the RTI is still open and will be considered as the model evolves). The federate submodel has three components:

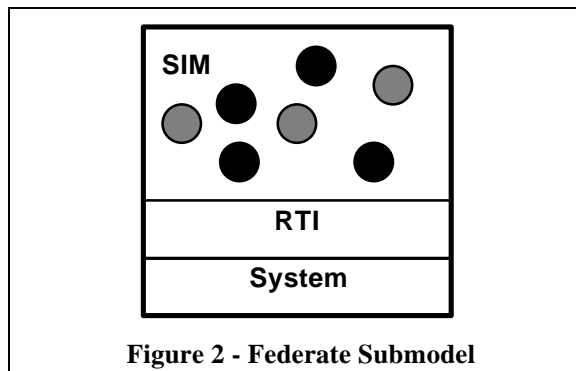
- the workload characterizing the federate simulation, which constitutes the major source of resource usage,
- the local RTI component, which will place additional load on the machine resources and is therefore of interest, and
- a *system* component that models relevant low-level activities in the machine (such as sending and receiving of messages), which can become an important source of resource utilization depending on the implementation of the RTI.

Figure 2 shows the logical structure of a federate submodel. The first component is SIM, which consists of a collection of simulation objects representing the simulation. Objects can be public (solid) or private (shaded) depending on whether they interact with other objects outside the federate. Public objects generate RTI calls that are passed on to the RTI component for processing and routing. Processing may entail book-keeping operations that maintain the routing information as well as generation of system calls which are passed on to the System component. The System component simulates the actions of sending/receiving messages

to/from the communications submodel. We describe each of the three components.

2.1.1 SIM component

This component models the actual execution of events by the federate. We do not model individual events explicitly; rather, the execution of an event is considered the same as a single simulation or execution of the object that is responsible for that event. Thus, the sequence of event executions translates to a sequence of object executions. A fundamental design choice we have made is that the majority of the characterization of the federate is placed in the description of the objects. Thus, most of the stochastics in a federate are stored in the objects and the SIM component provides the dynamics for object execution. A more detailed discussion on objects and the data associated with them is deferred to Section 2.2



The SIM component is a loop, in each iteration of which it executes the following actions:

- Examine each object for execution and execute it if the data stored in the object so dictates. Execution of

the object usually consists of consuming machine resources (such as CPU) and generating RTI calls.

- Advance simulation time of the federate. For time-stepped simulations, this would be performed once in each iteration of the SIM component whereas for discrete-event simulations, it would be performed after each object execution.
- Perform overhead tasks such as computing line-of-sight and dead-reckoning remote entities. Currently, these tasks are performed once per iteration of the SIM loop; in the near future, the model will include the capability to specify these overheads for each object execution.

2.1.2 RTI Component

RTI calls made by executing objects in the SIM component are handled by the RTI component. This component has two main responsibilities:

- maintaining the data structures that determine where updates and interactions should go, and
- generating the messages and the lists of destinations for updates and interactions.

Note, we take the approach that the sender of a message determines the list of destinations. However, this does not imply that the actual implementation of the RTI is also sender-based (in fact, it is very likely it will *not* be sender based). The point is that the functionality is concentrated at the sender, *but not necessarily the costs*. We have separated the functionality from the costs by adopting a separate *cost model* for the RTI (explained in Section 2.3). This provides the flexibility of capturing various RTI implementations simply by adjusting the costs in the RTI cost model appropriately - the functionality is fixed (and located at the sender for convenience).

The RTI component maintains a complex set of data structures to track the publish/subscribe information among federates and uses this information to determine where updates and interactions should go. This determination is done stochastically, based again on parameters stored in objects invoking the RTI calls. At the sender, the RTI generates system calls that are passed to the System component. At the destination, the RTI component receives messages from the System component and forwards them to the SIM component after due processing. As noted in Section 2.3, the costs in the RTI cost model are implemented by consuming machine resources.

2.1.3 System Component

The System component acts primarily as a conduit for messages, incurring appropriate costs as messages are sent and received. This component was included since these low-level costs can dominate performance depending on the implementations of the federates and the RTI.

2.2 Objects

As noted earlier, a federate is essentially determined by the objects it is comprised of. Objects contain no application specific data (such as attributes of a tank), but rather contain stochastic parameters that determine their dynamic behavior. Parameters for an object include the following:

1. The class to which it belongs, which determines the default set of classes to which this object is subscribed[†].
2. Mean time to execute the object once, which determines the CPU time requirement for executions of the object.
3. Current activity state, transition matrix and selection vector. To model the spectrum of objects encountered in typical simulations, we have introduced the concept of the *activity state* of an object. An object

	High	Medium	Quiescent
High	0.8	0.15	0.05
Medium	0.1	0.4	0.5
Quiescent	0.1	0	0.9

Table 1 - State Transition Matrix

can be in one of three states of activity: *high*, *medium* or *quiescent*. The activity state determines primarily, the probability that the object will be executed at any time. Thus, each object has a *selection vector* consisting of three probabilities, one for each activity state. It is expected that $P[\textit{high}]$ will be greater than $P[\textit{medium}]$, which will be greater than $P[\textit{quiescent}]$, which will be nearly zero. Note, these probabilities directly affect resource utilization. The selection vector is used by the SIM component in deciding whether to execute a particular object in any given iteration of its loop. The *state transition matrix* controls the movement of an object from one activity state to another. This is a 3x3 matrix of probabilities as shown in Table 2. The probability in any cell is the probability that the object will move to the column-state given that it is in the row-state. It can be verified that the matrix in Table 2 describes an

[†] The *publish/subscribe* mechanism of the HLA is used to specify the information flow between federates [DMSO96].

object that tends to be highly active but that can become quiescent occasionally. The state transition matrix is used by the SIM component each time it examines an object for possible execution. Irrespective of whether the object is executed in that particular iteration, its state may be changed according to the matrix.

4. The RTI calls generated by an object are determined by a combination of three parameters. First, a pair of Boolean flags determines whether this object is capable of generating messages to the external world or not (i.e. whether it is public). Two flags are required since an object can generate attribute updates and interactions independently of each other. The second parameter is an *RTI call vector* of probabilities for various RTI call types. Each element of this vector corresponds to a particular RTI call type and indicates the probability that a generated RTI call is of that type as well as the number of calls of that type generated. It follows from this definition that the elements of this vector should sum to 1.0. Only non-zero probabilities are maintained, to reduce memory requirements. If an RTI call type is selected based on this probability, the SIM component generates n calls of that type, where n is also specified in the vector. This is done to model “bursty” call generation as would be required when instantiating a set of new objects, for instance. The third parameter is a *burst probability* that is used to control the generation of multiple RTI calls at a time. After generating an RTI call, the SIM component uses the burst probability to determine whether another call should be generated. If so, the *RTI call vector* is renormalized to eliminate calls that have already been generated during this execution of the object. Note, specifying a probability p is equivalent to making $1/(1-p)^2$ calls on average.

In addition to these parameters, the simulation program computes and/or maintains other pieces of information for each object, including:

- an identifier
- pointers to access those federates that should receive updates and/or interactions generated by the object
- time since the last update was generated (for those objects that specify *minimum rate* attributes)
- other book-keeping information

Although class structures capture the hierarchy among object types, they do not capture the logical aggregations in military units (e.g. a brigade). To this end, the model allows the definition of *groups* of objects. Groups are specified at run-time through configuration files. Objects from different classes can be part of the same group. A group specifies all information related to activity state (i.e. initial state, transition matrix and selection vector) and RTI calls (Boolean flags, RTI call vector and burst probability). The main property of a group is that its constituent objects all use the parameters specified for the group and thus exhibit *identical* dynamics with respect to activity and RTI call generation. The rationale is that this capability allows a user to model an aggregate unit such as a platoon of tanks since the constituent tanks will likely act together.

To reduce the amount of information to be specified by the user, object parameters are specified at the class and group levels. Group parameters override those specified for the class. If necessary, future versions of the model will provide the capability to specify parameters for individual objects.

The HLA provides federates with the capability to express interest in only a subset of the attributes of another object, through the *subscribe* mechanism. In conjunction with the *publish* mechanism, the RTI uses this information to route messages appropriately[‡]. Since our design philosophy (no application semantics) precludes the modeling of the attributes of simulation objects, we have chosen to model attribute-based subscription using a probability parameter. Each subscription specifies a probability that may be thought of as a *level of subscription*. The SIM component uses this probability when an update is generated to decide whether a subscribing federate should receive that update or not. This construct may be used to model attribute-level subscription as follows. If federate A is interested in 40% of federate B’s attributes and federate B updates only 60% of its attributes regularly of which 30% overlap with the 40% interest of federate A, then federate A should specify a subscription probability of 0.5 (30/60).

In our model, as in HLA federations, subscriptions are specified at the federate level, rather than at the object level. This has the desirable effect of reducing the amount of configuration information the user has to specify. We expect that each federate will include a *federate controller object* that is responsible for

[‡] Data Distribution Management (DDM) builds upon the basic interest management of the publish/subscribe mechanism - see Section 4.

executing federate-level actions such as publishing/subscribing and instantiating/deleting new objects. The distinguishing feature of the federate controller object is that it will be the only object in the federate with a non-zero probability of generating these federate-level RTI calls.

2.3 RTI Cost Model

One of the goals of the model is to help evaluate alternate RTI implementations and/or design choices. As such, this requires that the model should be easily configurable to model different RTI designs. As noted earlier, we have taken an approach that separates the functional aspects of the RTI from the cost. The rationale behind this approach is that the functionality is going to be essentially the same across all implementations (dictated by the HLA Interface Specification [DMSO96]), and different implementations can be characterized by the costs they impose on the federation.

For the present, the RTI cost model is focused on object and declaration management services. With regard to these services, the RTI essentially acts as an information pipe. When updates and interactions are generated, they must be transported through the RTI, which will incur the following potential costs:

- determination of receiver set - this could be as simple as a table lookup to determine a multicast group address
- generation and reception of physical messages - this will occur in the System component of the federate submodel
- any data distribution management overheads
- processing of received messages by the RTI at the receiver
- generation of corresponding calls to the federate by the RTI at the receiver

While this corresponds to the “push” model for data exchange, similar costs will be incurred for the “pull” model where a particular receiver initiates the data exchange.

Many RTI calls result in changes in the state of the RTI rather than data exchange between federates. This can involve the exchange of messages between the RTI components at various federates. The RTI cost model provides the user with the capability to specify, for each type of RTI call, a sequence of message exchanges. Each step in this sequence could be: a unicast to a specific destination (such as a central RTI component), unicast to the sender (i.e. a reply), unicast to the originator of the sequence,

multicast to a uniformly selected set of destinations or broadcast to all federates. Finally, users can specify the local costs of all RTI call types (data exchange as well as control) using a general method that allows for constant values, distributions and general functions of other variables.

3. COMMUNICATIONS MODELING

The communications submodel encapsulates all communications in the federation. Figure 3 shows the structure of this submodel. Each federate connects to the submodel through a bi-directional *port*. The ports are interconnected by instances of models of communication subsystems as shown. The submodel consists of first-order models of various communication subsystems, which can be instantiated as desired to construct the topology. Communications are modeled at the message level. As messages are generated, they are routed through the topology using specified routes to their destinations. Any multicasting capability in the communication subsystems is modeled explicitly. Currently, the communication subsystem types modeled are ATM, Ethernet and point-to-point links. We believe these will be sufficient to capture typical communication topologies.

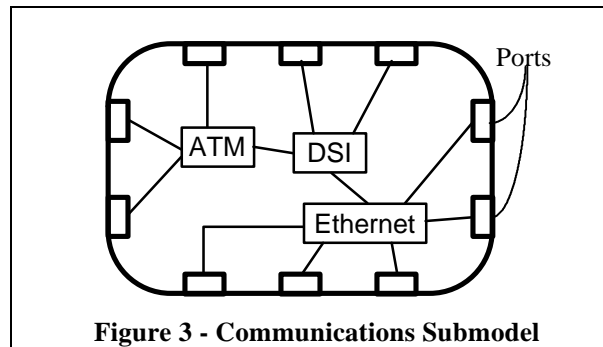
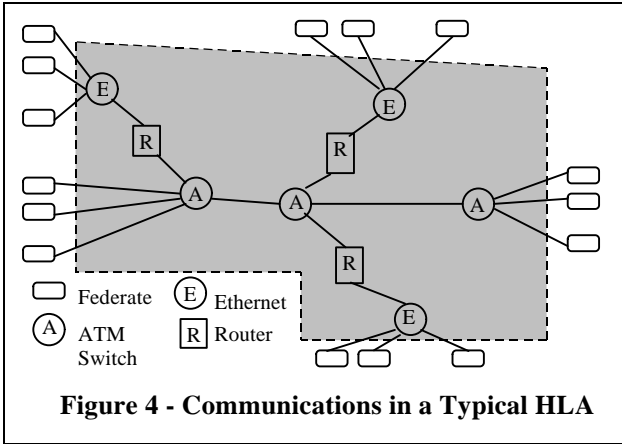


Figure 3 - Communications Submodel

A typical HLA Federation configuration may appear as shown in Figure 4. Our goal was to model everything in the shaded box in such a way that a user can instantiate and connect the elements almost arbitrarily. This was a significant challenge given the static nature of the SES/Workbench simulation tool (especially the language) used to implement the model. We used the following approach:

1. The federates are isolated from the communications submodel using *pools* - each federate has an *in pool* to send messages into the communications system and an *out pool* to receive messages from the communications system. At the sender's in pool, a message contains a list (generated by the federate



submodel) of all the destination federates to which it must be delivered.

- Each of the elements of the communications system (ATM, Ethernet, router, etc.) is called a *cell*. A cell appears as in Figure 5. Each cell has an *InBin* and an *OutBin* which form the *interfaces* to a cell. This standardization of the interface is important because it enables uniform access to different cell instances without knowledge of the names given to particular cells. Specifically, the InBins and OutBins are arranged in an array so that individual bins can be accessed by the Workbench code using array indices computed at run-time. Thus, for example, Ethernet[5] “knows” that ATM[3] is attached to InBin[7], so that message transfer can be made possible simply by determining the index 7 using a table look-up. Each cell has some number of internal *InBufs* and *OutBufs*. This number could be different for different cells and is a user specified parameter.
- A global *FedCell* table is provided by the user describing which cells the federates are connected to immediately. A sample table may appear as below. The row numbers correspond to Federate ID’s. The first column indicates

which cell the particular federate is connected to and the second column indicates which InBuf within that cell receive messages from that federate. Note, the same index is used to determine the OutBuf from which the Federate receives messages from that cell.

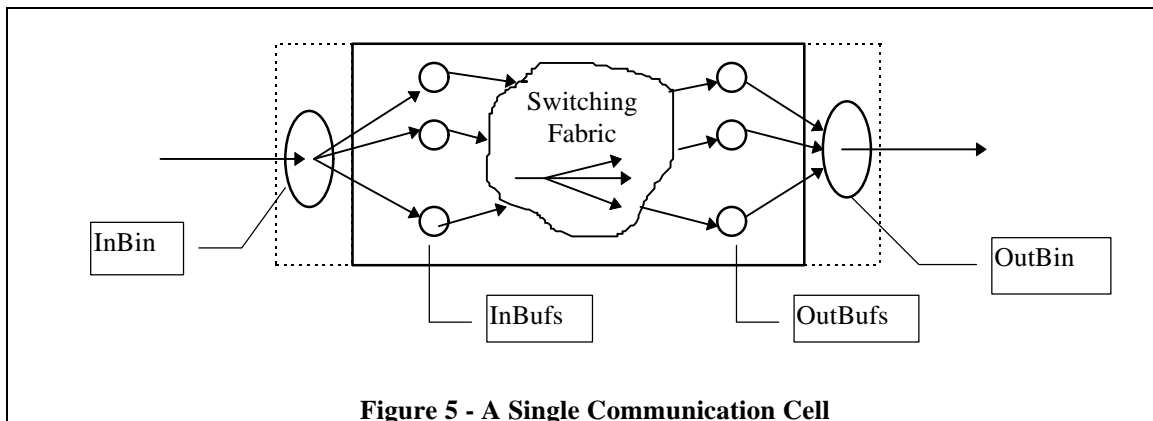
	Cell Index	InBuf# & OutBuf#
0	4	2
1	4	1
2	2	5
...
126	23	3

- For each cell, the user provides a *Routing* table as below. The row numbers indicate the *destination cell* - i.e. the ultimate cell that a message needs to go to so it can be delivered to its destination federate(s). The first column indicates which immediately connected cell the message should go to for a destination cell, the second column indicates which local OutBuf to use and the third column indicates which InBuf to use at the next Cell.

Destination Cell	Next Cell	Local OutBuf#	Remote InBuf#
0	3	2	2
1	-	-	-
2	2	5	5
...
25	3	4	4

These two tables together specify the communication topology of the federation.

- Each cell is associated with a *Walker* process whose sole responsibility is to take messages from the OutBin (whenever there are any) and move them to the appropriate InBins of the next downstream cells or to the *out pools* of the destination federates. The combination of the InBufs, OutBufs and the Walkers provides the desired plug-and-play capability.



Here's an example of how the pieces work. Let's say Federate 10 (F10) has generated a message intended for F2, F3, F6, F12 and F15. These destinations are specified in the message itself.

- a) The SYSTEM transaction of the F10 looks up the FedCell table determines it is connected to Buf3 in Cell5 and accordingly drops the message into InBin5 with an internal variable indicating it should use Buf3.
- b) Let's assume Cell5 is an Ethernet. It picks up the message from InBin5 (it's been waiting on that InBin) and routes it to its internal InBuf3. It then simulates an Ethernet message transmission according to its design. Now the message is ready to be transmitted out of Cell5.
- c) Some code inside Cell5 uses the destination list in the message and the FedCell table to determine that this message needs to go to Cell4 (for F2 and F3), Cell6 (for F12 and F15) and also needs to be delivered to F6, which is directly connected to Cell5 at OutBuf7. This process is called *partitioning*. The partitioning results in three copies of the message, one with F2 and F3 as destinations, another with F12 and F15 as destinations and a third with F6 as destination. The last of these is dropped in OutBuf7. This code then uses the Routing table for Cell5 to determine that: (i) to get to Cell4, it must go to Cell3 through local OutBuf2 and remote InBuf1, and (ii) to get to Cell6, it can go directly to Cell6 through local OutBuf5 and remote InBuf4. So it drops the first partition message into OutBuf2, after setting internal variables to indicate the next cell (4) and InBuf (1), and then drops the second partition message into OutBuf5, after setting internal variables to indicate next cell (6) and InBuf (4). *Note:* This partitioning function must be performed within each cell. However, it can be performed at different points, depending on the type of cell. In the case of Ethernet, the partitioning happens physically after message transmission on the wire and thus, it is described as happening after the transmission has been simulated. In the case of ATM, the partitioning will happen at the input, *before* the message is transmitted (the partitions are created and dropped into the appropriate OutBufs to be picked up with the output port transactions).
- d) Since OutBufs are not really relevant to the Ethernet model, in the previous step, the Ethernet model can drop its messages directly

into the OutBin, after setting the internal routing variables appropriately.

- e) The Walker for Cell5 then picks up these three partition messages and delivers the first to InBin4, the second to InBin6 and the third to OutPool6 (for F6).
- f) Process is repeated at Cell4 and Cell6 and so on, until all destination federates receive a copy of the message.

4. Data Distribution Management Mechanisms

The main challenge in modeling mechanisms for data distribution management (DDM) is to model the effect of DDM on federate connectivity (who can talk to whom), since many costs in distributed simulation (including DDM costs in some implementations) are proportional to the connectivity among federates. For example, dead reckoning costs and the cost of computing the line of sight are proportional to how many remote entities can be sensed by a given entity. Once the connectivity is captured accurately, the simulated message traffic will also be accurate, resulting in more accurate costs.

The challenge in capturing connectivity is that, with DDM techniques, connectivity can depend on several progressively more challenging factors:

- class-based publication/subscription;
- *attribute*-based publication/subscription (e.g. tank class is interested only in position and velocity attributes of other tanks);
- *attribute-value*-based filtering (a tank is interested only in those units that are located within a 10 kilometer radius of its own position).

The latter two pose a significant challenge because our model does not track which attributes are being updated and consequently, it does not track values of attributes either.

In the version of the model described in the previous paper, a single probability is used to attempt to capture the attribute-level connectivity. This is inadequate since it does not capture value-based filtering and does not even capture attribute-based connectivity accurately. What we really require is a set of n probabilities for each subscribed class at the subscriber where n is the number of objects. n has to be the number of objects rather than federates because two different objects at *one* federate could be updating different portions of the attribute space of the same class and thus, the subscriber could require different percentages of the updates of each object. This is not an unrealistic situation: consider the following - all tanks are objects of the same class but some tanks can only generate a subset of the attributes of all tanks. So

let's say tank model A has 3 attributes and tank model B has 5 attributes and subscriber S requires 4 of the 5 attributes. Then the probabilities for the two subscriptions would be 1.0 and 0.8 (even this is an approximation since we really need to take into account the distribution function of which attributes are updated - this model assumes it to be uniform).

Even with n probabilities at each subscription, the scheme is not attractive for several reasons:

- specifying $O(n^2)$ probabilities becomes very tedious for the user even for moderately large n ;
- estimating these probabilities from traces is a challenging task and has a significant potential to invalidate the results obtained from the model (garbage-in-garbage-out);
- the estimation is further complicated if the scheme is used to model value-based filtering because this would require the probabilities to change dynamically, which reduces the statistical validity of the model.

A scheme is needed where these probabilities are achieved (indirectly) through more accurate modeling of the connectivity.

Our basic approach is to use a “shell” implementation of the *routing space* technique to specify and establish connectivity. Routing spaces [VH96] permit data distribution connectivity based on attributes and attribute-values. A routing space consists of d dimensions. An attribute update can be mapped into this d -dimensional space based on its value. In the real RTI, connectivity is established based on overlaps among *update* and *subscription* regions. Given such an overlap, all updates in the update region are sent to all receivers with overlapping subscription regions.

The basic idea behind our modeling approach is that each object will be associated with a set of update regions at any time in the execution. Generated updates will be assumed to fall into one of these regions. Each object will also specify subscription regions. The simulation model will maintain at all times, information regarding the overlaps and thus establish the connectivity.

The following describe the new mechanisms added to the model to incorporate the modeling of DDM (we assume familiarity with DDM concepts):

1. The class-based publication/subscription scheme in place previously was retained. The only

change is that the probability of subscription (i.e. the level of subscription) is no longer relevant. In the new scheme, each sender maintains a list of “listeners” based on class publication/subscription. In a sense, this is the coarsest level of filtering, one that will change infrequently. For these reasons, we retain this structure and mechanism because it reduces the cost of computing connectivity as follows: we will use this class-based list as a base and prune it with the routing space mechanisms to get the actual connectivity. In this way, for each update, we only have to check the members of the list. If we were to not use this mechanism, we would have to check *every* object for a potential receiver, which would be prohibitively expensive.

2. The user associates one or more *routing spaces* with each federation. Each routing space is specified as a number of numeric dimensions. Each dimension is represented internally as a real-number range even though the dimension may actually be integral. The range for each dimension is also specified by the user.
3. Each object is associated with two sets of *regions*:
 - a set of *update* regions, each with a probability of selection
 - a set of *subscription* regions

Each region in the routing space is specified as a set of ranges, one per dimension of the routing space. Two special range specifiers are *all* (“*”) - which means the entire range, and *nothing* (“-”) - which means not interested in that dimension. A region A is said to overlap with a region B if there is overlap on corresponding ranges of A and B in at least one of the dimensions of the routing space.

4. Based on the overlaps, each federate maintains a list of subscription regions that overlap with any of the update regions of its objects (recall, this list is generated relatively inexpensively by pruning the already existing class-based list of subscribers).
5. When an object generates an update, one of the associated update regions is chosen based on the probabilities. Then, each of the entries in the subscriber list for that federate is checked to see if it contains this update region. If so, the federate is added as a destination for the message.

Clearly, since our scheme mimics the functionality of the DDM mechanisms being implemented in the RTI currently, it is able to capture the connectivity at least as accurately as the actual mechanisms. A potential drawback of this approach is that it has the same limitations as the actual routing space mechanism - connectivity is not as precise as it can be. Even marginal

overlaps in regions will result in a connection which could be seldom used. If new DDM mechanisms are devised in the future that can extract more precise connectivity than the current ones, the model may require significant changes to reflect them. Ideally, we would like to capture the perfect connectivity and then artificially inflate it as required for performance studies of less accurate DDM mechanisms. We are currently in the process of enhancing the design of our DDM model to achieve more accurate connectivity.

Additionally, we would like to incorporate a capability to read *scripts* specified by the user, that describe the dynamics of update and subscription regions. For example, a script might specify (for some objects) a list of $\langle \text{region}, \text{time span} \rangle$ pairs, where a pair specifies the time span for which the region is valid. This will allow the user to study performance under dynamic DDM conditions. Naturally, the added power of being able to model dynamic DDM situations comes at the cost of a burden on the user to provide the scripts.

5. Modeling Event Driven Federates

The current scheme for selecting objects in any one event execution iteration at a federate is to step through the entire list of objects at a federate (either in sequence or in a random order), testing each for selection based on its activity level (probability). This is well suited for a *time-stepped* federate where such a mechanism is typical in the actual event processing loop of the federate.

In an *event-driven* federate, events still correlate with objects. The difference is that the entire list of objects be tested in each iteration of the event execution loop. In reality, in an event-driven federate, an object can schedule itself for execution in subsequent iterations. In general, an event-driven federate can be modeled by a selection process where each object is given a probability of selection so that the sum of all probabilities is 1.0. The selection process then rolls a uniform die to determine which object to pick in *any* one iteration of the event execution loop. Note that it is possible under this scheme, for an object to be selected in subsequent iterations of the event loop (the probability of this happening in the current scheme is *very* low and also depends on the selection probabilities of other objects - an unnatural side-effect).

We would like to retain the concepts of selection probabilities and groups as defined in the current scheme as these are very natural concepts for a user of the model and thus facilitate the task of providing accurate input parameters to the model. Unfortunately, with these probabilities, the sum may not be 1.0 (it could very easily exceed 1.0). To accommodate these probabilities, changes are required only in the object selection process. What is required is a *mapping* of the selection probabilities as defined currently to a corresponding set that sums to 1.0. Note, it is possible to put this onus on the user by requiring that event driven federates must be specified so that their object selection probabilities *always* sum to 1.0. This is a significant responsibility especially since objects can asynchronously change their activity states (and hence their selection probabilities). Consequently, we would like to retain the selection probability externally and *internally* perform the mapping described above.

The following scheme has been implemented. This is inside a single federate.

1. Each object has a probability of selection based on its state of activity (as in the current model)
2. The federate maintains a running sum of the selection probabilities. Each time a probability changes, the sum must be reevaluated. A probability can change under the following conditions:
 - activity level changes (the probabilities of the *entire* group must be changed)
 - object is instantiated
 - object is deleted
3. In a single event execution the federate does the following in sequence:
 - draws a uniform random number in $[0,1], p$
 - starts with the first object in the object list
 - computes the *effective selection probability* of that object as $p_i = s_i/s_T$, where s_i is the original selection probability of the object and s_T is the running sum of selection probabilities
 - if $p_i \geq p$, the object is selected and the event execution loop terminates.
 - otherwise, the next object is tried.
4. When an object is selected, it is executed as in the current scheme. Additionally, *each* member of its group is also executed (again, as in the current scheme). In each execution, the activity state may change (if one changes, all change). If so, each change must be accounted for in the running sum of probabilities.

NOTE: the conditions under which the running sum can change can occur only *between* event execution loops (i.e. at the end of an object or group execution). This prevents race conditions due to a changing sum in the midst of an iteration.

This scheme has some very attractive properties:

- It preserves the natural model of selection probability at the user level - the user can simply tie the selection probability to the current activity level of the object, irrespective of the states of other objects (groups always stay together).
- If all objects are highly active (with selection probability = 1.0), we have effectively, a uniform selection process, as expected.
- If a small set of objects have selection probability 1.0, that set will have a higher effective selection probability than others, but among them, each will have the same probability.

Thus, we see that the effective selection probability is tied neatly to the natural activity level of an object, as desired.

The only potential drawback appears to be the need to search through the list of objects to find one, in each iteration (an $O(n)$ search with n objects). One way to alleviate this problem is to periodically sort the list of objects based on the selection probabilities (note, there is no need to use effective selection probabilities since the ranking won't change).

6. SUMMARY

We have described enhancements to the design of a simulation model for first-order performance analyses of HLA-compliant federations. Enhancements include flexible communications features, data distribution management mechanisms and event-driven federates. These new features make the model more realistic and mostly complete. As simulation programs move towards HLA compliance, our federation analysis tool can be used to support the transition process by demonstrating feasibility and alleviating concerns before bending metal, supporting design choices and identifying potential trouble-spots.

ACKNOWLEDGMENTS

The simulation model is being implemented by Greg Borek of Mystech Associates, Inc. using the SES/Workbench simulation tool. This work is

sponsored by DMSO under contract number N61339-96-D-00020023 of the ADST II program.

REFERENCES

- [DMSO96] Defense Modeling and Simulation Office, "HLA Interface Specification", <http://www.dmsomil/hla/>, August 1996.
- [SrRe95] Srinivasan, S. and Reynolds, P.F. Jr., "NPSI Adaptive Synchronization Algorithms for PDES", *Proceedings of the 1995 Winter Simulation Conference*, Dec. 1995, 658-665.
- [VH96] Van Hook, D. J., "Approaches to RTI Implementation of HLA Data Distribution Management Services.", *Proceedings of the 15th DIS Workshop*, Paper #96-15-084, September 1996.

AUTHOR BIOGRAPHIES

SUDHIR SRINIVASAN is a research scientist at Mystech Associates, Inc., conducting research in parallel and distributed modeling and simulation. He received his Ph.D. in Computer Science from the University of Virginia in 1995 and Bachelor of Engineering also in Computer Science from the Bangalore University, India, in 1990. From August 1995 through February 1996, he was also a research associate at the University of Virginia, working on identifying fundamental issues in linking models at different levels of resolution. His main research interest is in parallel and distributed simulation, especially the High Level Architecture. His other interests are in parallel and distributed computing and networking. He has published several papers in conferences and journals and is a member of the ACM and the IEEE Computer Society.

PAUL F. REYNOLDS, JR., Ph.D., University of Texas at Austin, '79, is an Associate Professor of Computer Science at the University of Virginia. He has published widely in the area of parallel computation, specifically in parallel simulation, and parallel language and algorithm design. He has served on a number of national committees and advisory groups as an expert on parallel computation, and more specifically as an expert on parallel and distributed simulation. He has been a consultant to numerous corporations and government agencies in the systems and simulation areas. He is a member of the ACM and the IEEE Computer Society.