Integrating a Distributed Agent-Based Simulation into an HLA Federation

Gary Kratkiewicz
Amelia Fedyk
Daniel Cerys
Distributed Systems & Logistics Department
BBN Technologies
10 Moulton Street
Cambridge, MA 02138
617-873-2476, 617-873-7568, 617-873-4534
kratkiewicz@bbn.com, afedyk@bbn.com, cerys@bbn.com

Keywords:
Agent-Based Simulation, Distributed Systems, Cougaar, HLA, Logistics

ABSTRACT: *Distributed agent-based architectures can be used to build powerful models and simulations. Although such architectures can themselves be used as a means of interoperating with other simulations, it can more useful to integrate them into a standard simulation interoperability architecture such as HLA, leveraging the benefits of each.*

Instances of one such agent-based architecture, Cougaar, have been integrated with other models as federates in HLA federations. Cougaar is a Java-based architecture for the construction of robust large-scale distributed agent-based applications. It has been used to demonstrate the feasibility of using advanced agent-based technology to conduct rapid, large scale, distributed logistics planning, execution, and replanning in extremely chaotic environments in DARPA's Advanced Logistics Program and UltraLog Program, and to demonstrate effective analysis of logistics support for an FCS Unit of Action in the Army's Future Combat Systems program.

This paper describes how a Cougaar-based agent society modeling a small logistics supply network was integrated as a federate in V1.3 and 1516 HLA federations. It describes how to interface the society and the HLA RTI; how to synchronize society time with HLA RTI time; and how to map agents, objects, and actions in the society to HLA objects and interactions. The lessons learned in this demonstration integration effort can be applied to large-scale efforts, such as integrating large Cougaar-based logistics simulations with combat simulations (e.g., OneSAF).

1 Introduction

Distributed agent-based architectures such as Cougaar [1] and others [2] can be used to build powerful models and simulations.

Agents are independent software entities that react to events and initiate actions by themselves. Different agent-based architectures yield agents with varying capability levels. Agents can have or define roles, tasks, beliefs, desires, or intentions. Agents can be static (remain on their original platform) or mobile (can move to another platform). Some agents are considered intelligent and some agent systems are considered intelligent [3].

A distributed system is a collection of separate processes or information systems that can act together as a single system. Distributed agent computing involves agent-based systems that operate in a distributed manner. They are used to implement complex behavior or to model or simulate complex systems.

Although distributed agent-based architectures can themselves be used as a means of interoperating with other simulations, it can be more useful to integrate them into a standard simulation interoperability architecture such as HLA, leveraging the benefits of each and allowing them to interoperate with existing compliant simulations. In this work we demonstrated integration of a distributed agent-based architecture (Cougaar) with a standard simulation interoperability architecture (HLA) by designing and developing a prototype Cougaar-based logistics simulation that was also an HLA federate.

2 Distributed Agent Computing with Cougaar

Cougaar is a distributed multi-agent system infrastructure developed to provide a flexible framework in which to solve complex problems.

2.1 Cougaar Architecture

Cougaar (for Cognitive Agent Architecture) is a Java-based architecture, which is well suited to integration with HLA because its rich design can model diverse objects and interactions between those objects. Problems modeled within the Cougaar architecture are capable of reacting to a dynamically changing environment. As plans change, Cougaar, seen as a workflow engine, adapts. The solution is reworked creating a modified workflow for the plan, composed of viewable, traceable components [1].

The Society is an important concept in the Cougaar architecture. The Cougaar Society is composed of a collection of Agents with various capabilities (plugins) that work together to solve a particular problem (see Figure 1). The agents within the society can be organized into Cougaar Communities. The communities can be multi-tier with logically related agents, which usually tackle a particular sub-task or may share common information that is not made available to the rest of the agents in the society. Membership in Cougaar Communities is not distinct, so an agent may be a member of one, many, or no community.

Organizing the conceptual society onto a physical platform necessitates the definition of the Cougaar Node, defined as a single Java Virtual Machine (JVM) instance. Agents and communities may be grouped onto a node based upon proximity to a data source, such as a database, or to facilitate inter-agent communication, as the JVM is used to shortcut the message transport layer. Agents sharing a single JVM benefit from loop back in-memory transport. It might be cogent to place agents with heavy interactions on the same node as long as CPU and memory constraints are not overburdened.

Agents are Cougaar components with a defined functionality and a local memory store called a Blackboard. The functionality (or behavior) of the agent emerges from the composite of plugins within the agent's makeup. The agents are responsible for scheduling the execute cycles of the plugins and the management of the messaging system that is responsible for inter-agent communications. Removing the details of messaging from the plugins allows the plugins to focus on domain specific functionality instead of infrastructure details. The plugins themselves represent the business logic, which passes information between itself, other plugins within the agent, and other agents, by publishing objects to the local Blackboard.

The Blackboard is the collective memory store of the agent. It implements a publish/subscribe API. Each plugin is able to view objects on the Blackboard by creating subscriptions. The subscriptions show the plugin a view of the Blackboard in which it is interested. Likewise, plugins add or publish objects to the Blackboard. The published objects are available to all plugins within the agent. These Blackboard objects can be used as a means of communication and are persisted. The Cougaar infrastructure facilitates notification of all changes affecting a plugin's subscriptions within a transaction. The transaction represents the adds/removes/changes made to objects described by the subscriptions since the plugin's last execute cycle. This ensures that the plugin is processing a complete and consistent set of interesting Blackboard objects. All changes that a plugin makes to the Blackboard are kept local to that plugin until the plugin has completed its cycle at which time the entire set of changes is advertised to all agent members via updates to their subscriptions.

The Logic Provider (LP) is an agent component, which watches Blackboard activity. They are very lightweight and are responsible for messaging and Blackboard modifications. All adds, rescinds and changes to objects on the Blackboard invoke an LP. The use of a Logic Provider is nearly transparent to the plugin developer. A plugin need only publish an inter-agent object to the Blackboard and the LP will silently handle the message transport.

Two common communication patterns used in the Cougaar Architecture are Plugin-Plugin and Agent-Agent. The plugin-to-plugin communications uses asynchronous messaging within a JVM. This usually takes the form of a query or object published to the Blackboard resulting in one or more responses received back. The agent-to-agent messaging relies upon its Logic Providers to translate the message into "message space" [4].

2.2 Running a Cougaar Society

Cougaar societies can be defined within an XML file and run manually with the scripts provided with the Cougaar infrastructure. The XML files define the agents and their plugin content. Most societies require a boilerplate set of plugins within each agent. Plugins that may be found in most agents are Service plugins that establish a client/server relationship between agents, Yellow Pages plugins which assist the agent in finding services, White Pages plugins which do lookups to help agents find other agents (similar to DNS), PrototypeProviders that

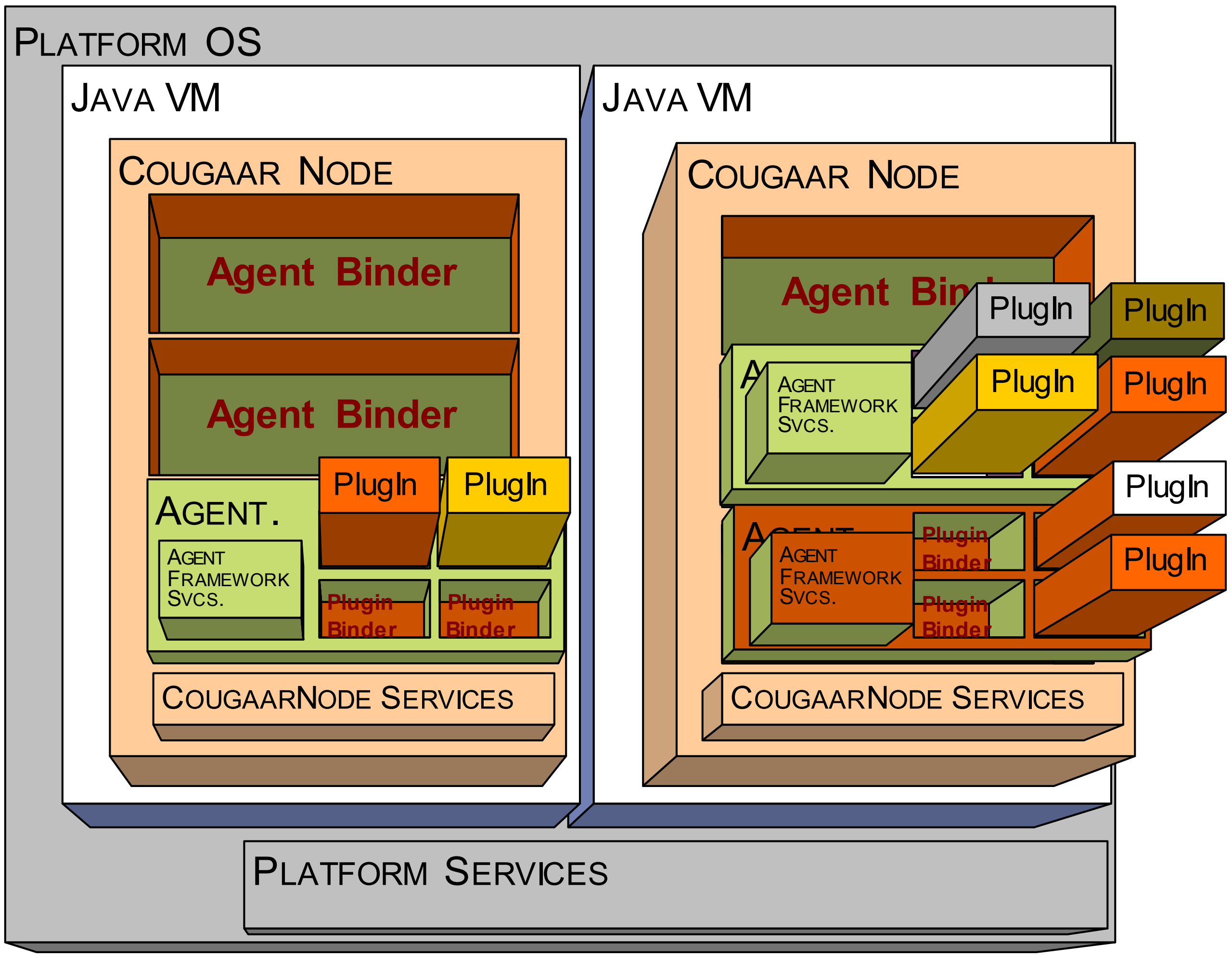

Figure 1. Plugin, Agent, and Node Structure of the Cougaar Architecture [6]

contribute to the creation of Assets, Servlets for data visualization, and domain plugins to handle the job the society is tasked to do.

Many tools have been developed under the UltraLog project (an extension to Cougaar, described below) to aid the user in society run and control. Although bringing up a group of agents on each node can run a distributed society, it becomes difficult when your society grows to more than two or three machines. Acme was developed under the UltraLog project for automated testing and scripted tailoring of a society. Acme uses Ruby scripts and rule files to modify the behavior of the society pre-run by adding/subtracting plugins or plugin parameters. Acme also allows the user to script time advances, perturbations, and data retrieval into a society's run, which fosters growing complexity.

The preferred user interface for the Cougaar society is the Java servlet as it easily lends itself to a distributed architecture. Servlets are server side, browser enabled modules that are very similar to a plugin. Many servlets aid the user in the visualization of interesting Blackboard objects. The UltraLog society uses servlets to monitor society completion and correctness, profile memory usage during runs, view objects on the Blackboard and traverse their lineage, manually advance society time, and much more.

2.3 UltraLog as an Application of Cougaar

UltraLog is a four-year project sponsored by the Defense Advanced Research Projects Agency (DARPA) [5]. It is a layer built on top of the Cougaar architecture with the goal of adding survivability to Cougaar by incorporating robustness, scalability, security and stability into its design. At the core, UltraLog was developed to model military logistics within a distributed multi-agent system. It is composed of plugin components, containing the business logic for modeled agents, domain specific data as well as additions to the Cougaar infrastructure.

The UltraLog test society models the behavior of a large set of military organizations provided by a set of logistics support organizations, which interact with each other as they plan and execute a military operation. Each agent models a single military organization with its physical assets, business rules, and relationships to other organizations. The present full UltraLog society has over 500 agents running on more than twenty nodes. Each agent's behavior is defined by its collection of plugins.

All military organization agents produce demand for materiel based on their physical assets and their planned activities. The agents, also called organizations in this context, equipped with the appropriate demand plugins will produce demand tasks. The assets themselves encapsulate their demand rate information. Special plugins called PrototypeProviders are called during the creation of an asset to populate it with necessary information, usually derived from a database. For example, demand for DF2 (diesel fuel) can be easily traced within the society to the Major End Item (MEI) that produced that demand, such as a tank or a truck asset. Demand tasks are objects published to the Blackboard that communicate need for re-supply from inventory points. Each task has a lineage with a parent. Inventory points within the same organization or a supporting organization will receive the message and respond to the task with its ability to comply. The subtleties in the task structure, including preferences, scoring functions and confidence, allow a rich communication between agents.

Like MEI assets, inventory assets are created with the necessary business logic that represents the practices of that organization. Inventory points are aggregation points. UltraLog's inventory managers aggregate demand over a commodity and item pair. Although each demand task is distinct on the Blackboard with its own lineage, individual demand becomes indistinct within the aggregate bin. The inventory plugins will manage their inventory bins using detailed business logic tailored to the specific agent in which it resides. Re-supply tasks are issued by the inventory plugin to stock enough of a particular item to safely accommodate demand from its customers for a defined period of time and to keep the supply chain from falling behind the demand.

In the UltraLog society, some of the re-supply tasks produced by inventory points are expanded into transportation tasks and sent to transportation agents. The transportation agents model ground, air, and ship transport and make use of Genetic Algorithms (GA) for scheduling. The extent that the transportation plan meets the request is returned in the results attached to the transport task. The Cougaar infrastructure propagates that information up through the expanded supply task so all dependent agents receive notification of results.

The society can run strictly in a planning mode where the solution is predictive or in an execution mode where the solution is generated through simulation or captured from real-world operations. The planning mode is based on static information and the operation takes place entirely in the future. One method of simulating execution of the plan involves jumping Cougaar's society time forward by some expected time increment. As time moves forward, demand generators (plugins built to produce simulated 'actual' demand) are employed. The actual demand is derived from the predicted demand with some deviation generated using a Poisson distribution. Another method could be based on integration with combat simulation models that model individual vehicle activities. The execution demand is received by the inventory points who replace the predicted customer demand with the actual demand, where available, and modify the existing plan to accommodate any significant variance from the prediction.

2.4 FCS Supportability as an Application of UltraLog

The FCS (Future Combat Systems) Supportability prototype extends UltraLog functionality. The FCS Supportability (FCSS) society runs within the full UltraLog society but it centers on the Unit of Action (UA). The UA is modeled after the rapidly deployed and self-sufficient military brigade envisioned for FCS. The duration of its scenario is relatively short encompassing days instead of months. Initially, FCSS is handling a single commodity, fuel. The agents in the UA are designed to reduce the logistics footprint, requiring much less support than a traditional brigade. The UA agents are also highly mobile; they traverse a hostile environment that makes the act of re-supplying more difficult.

In keeping with this design, the functionality of the inventory plugin was extended to bring inventory management down to the MEI level. Each inventory asset, for example, can now refer directly to the vehicle as well as the item it maintains. Agents, in addition to modeling organizations, also model trucks and tanks. Because these mobile agents maneuver in and out of hostile territory, it is necessary to refill at convenient points. The inventory plugin was also extended to add new business rules and algorithms to the refill generation module of the inventory plugin that will find acceptable re-fueling points.

The needs of the UA society also had a dramatic impact on the transportation plugins. The UA society has an extremely flat support structure; all the re-supply requests are now expanded into transport tasks. Instead of refueling an organization, as in the original UltraLog society, the fuel trucks must travel to each individual UA vehicle and meet the vehicle during its requested time. Being late or early may result in the truck arriving during an offensive period, putting the truck at higher risk. New GA scheduling rules were developed to handle this interesting scheduling problem.

Simulation in the UA took on the characteristics of the society itself, in that its time advance is at a greater pace, hourly instead of daily, to better mirror the quickly changing scenario. The demand generators were also enhanced for FCSS to generate hourly actual demand as well as accept the simulation data from an outside source. This opens the way for FCSS to interact with third party simulation tools.

3 The High Level Architecture (HLA)

The High Level Architecture (HLA) is a standard technical architecture for the interoperation of simulations.

3.1 Overview of HLA

HLA is based on the idea that a single simulation cannot meet the needs of all users. Originally developed for the US Department of Defense under the leadership of the Defense Modeling and Simulation Office, it was specified as the standard technical architecture for all DoD simulations in 1996. The latest version of this HLA specification (1.3) was adopted in 1998. This specification formed the basis for the draft IEEE standard for simulation interoperability (IEEE 1516), which was approved as an open standard in 2000 [7, 8].

A group of simulations interoperating via HLA is known as a federation. A federation has three main functional components. The first component is the set of simulations themselves, referred to as federates. Federates can also be interfaces to live players and tools that passively collect simulation data and monitor simulation activities.

The second component is the Runtime Infrastructure (RTI), which is a distributed operating system for the federation. It provides the following services for the federation:

- Federation management
- Declaration management
- Object management
- Ownership management
- Time management
- Data distribution management

A reference RTI was originally available through DMSO. A variety of RTIs are now commercially available in HLA 1.3 and IEEE 1516 versions. Some of these RTIs are fully asynchronous while others are partially asynchronous (requiring periodic calling of a "tick" method to allow the RTI to perform operations).

The third component of the federation is the RTI interface. This allows the federates to interface to the RTI and access its services. The federates interact with each other through these services. The HLA interface specification mandates APIs in various languages including Java and C++.

The HLA Federation Object Model (FOM) describes the set of objects, attributes and interactions shared across a federation. It is specified in a file read by each federate at startup.

Three general time synchronization methods can be used with HLA [9]:

- No synchronization: Each federate advances time at its own pace. This results in federations running with divergent representations of time.
- Conservative synchronization: This avoids the possibility of processing events out of time stamp order. Messages sent under this synchronization are given time stamps. In federates that can receive or send these messages, their time is referred to as logical time. Different federates can have different logical times. The RTI manages messages so they cannot be received by a federate until no earlier messages can be received; it manages time advancement requests with the same restriction.
- Optimistic synchronization: This allows the possibility of processing events out of order, but provides a mechanism for rolling events back when this occurs.

3.2 Purpose and Goals of Integration

Existing military combat simulations lack sophisticated logistics components. As recent reports indicate, logistics support is critical to combat operations [10].

Although Cougaar-based logistics simulations provide capabilities not available in current military simulation systems, a roadblock to their use has been a lack of interoperability with these systems. The primary purpose of our work was to demonstrate the ability of Cougaar societies to interact with other simulations via HLA. Showing that Cougaar-based logistics capabilities are applicable to existing simulation test beds would allow credible proposals of simulation systems based on Cougaar-HLA linkages. Specifically, this would allow the addition of realistic simulation of logistics to combat simulations such as OneSAF.

Our specific technical goals with respect to our demonstration prototype were as follows:

- Demonstrate that a Cougaar-based society of agents can act as an HLA federate, successfully publishing information to the federation and receiving information from the federation.
- Demonstrate that Cougaar time mechanisms can successfully integrate with HLA time synchronization.
- Demonstrate that a Cougaar society can work with a variety of RTIs:
    - o HLA 1.3 and HLA 1516.
    - o Fully asynchronous and partially asynchronous (tick-based)
    - o Java and non-Java based
- Develop techniques for creating or modifying Cougaar societies to be HLA federates.
- Identify further areas of research as well as additional functionality required in a full-scale Cougaar HLA federate.

4 Integration Approach

Our integration approach involved reducing in size an existing Cougaar-based logistics simulation society, splitting the society into two societies that modeled the same organizations but which performed different functions, augmenting the societies to operate as HLA federates, and then interfacing them with various RTIs.

4.1 Federation and Society Design

We started with a small Cougaar-based UltraLog society that simulated the logistics aspects of a military hierarchy and its use of various supply items such as fuel, ammunition, and supply parts. We pared the society down to model only bulk fuel consumption and to contain only two fuel-consuming battalions at the bottom of the hierarchy. These were an artillery battalion (1-35-ARBN) and an infantry battalion (1-6-INFBN). The society contained a total of 19 organizational entities, which included the two battalions, their organizational chain of command (with NCA, National Command Authority, at the top), and their supply hierarchy.

A Cougaar agent modeled each organizational entity. Each of these agents contained a variety of plugins to perform their actions. The fuel-consuming organization agents contained a demand generation plugin to project their bulk fuel requirements and to generate bulk fuel Supply tasks to model these requirements. Other plugins would then obtain these tasks from the blackboard and order fuel.

For the initial proof-of-concept demonstration we created an HLA federation with two federates. The first federate was a general simulation of the above logistics organization. The second federate was a specific fuel demand model. The first federate obtained fuel demand data from the second federate via HLA.

To create the federation we duplicated our demonstration society into two societies:

- A high-fidelity logistics society that performed the general simulation
- A demand generation society to model fuel demand

In the high-fidelity logistics society we removed the demand generation plugin for the fuel consuming organization agents. In the demand generation society we left this plugin, but removed others that dealt with the Supply tasks after they were generated. We also removed all organizations not necessary for the demand generation function. This pruned society contained seven agents.

4.2 Non-Cougaar Federate

To ensure that our prototype experiments were valid with non-Cougaar-based federates, we performed verification experiments where the demand generation society was replaced with a non-Cougaar federate that performed the same function. This was a Java-based, non-agent program that performed the demand modeling with no Cougaar code.

4.3 RTI Selection

We tested our prototypes with three HLA RTIs that were readily available to us and which covered the capabilities we want to test:

- Pitch 1516 LE
- Pitch 1.3 LE
- DMSO NG 1.3

This selection allowed us the feasibility of using Cougaar with both DMSO 1.3 and IEEE 1516 RTIs, with fully asynchronous (Pitch) and partially asynchronous (DMSO) RTIs, and with Java-based (Pitch) and non-Java-based (DMSO) RTIs.

4.4 Simulation Models

There are two types of simulation models that HLA is designed to handle: continuous (time-stepped) and discrete (event-driven). A Cougaar society can map into either of these, depending on how it is set up. Time moves ahead in a Cougaar society in two ways. First, it continuously advances at the same rate as real (wallclock) time. Second, time can be advanced ahead to some future time.

If time is left to continuously advance with real time or is advanced ahead in equal steps, the Cougaar society maps into the time-stepped simulation model. If time is advanced in unequal steps, the Cougaar society maps into the event-driven model.

A simulation can operate in real-time, scaled real-time, or non-real-time. Although Cougaar societies generally are not designed to operate in real-time, there is nothing preventing them from doing so, since Cougaar by default runs in real-time. A properly designed Cougaar society with agents appropriately distributed across adequate machines can operate in real-time in conjunction with other real-time simulations or with humans. The design must ensure that the society operation is not constrained by inadequate computing resources.

However, the great majority of existing Cougaar societies perform simulations lasting on the order of minutes to hours of wallclock time that model operations and events on the scale of days or months. They require time to be advanced ahead to perform useful analyses. In these cases, if time is stepped evenly, the society would operate in scaled real-time. If stepped unevenly, it would operate in non-real-time.

4.5 Time Management

In a Cougaar society, simulation time (also known as society time) is initialized to the start of physical time (the time in the system being modeled by the simulation) at the beginning of a run. Simulation time then advances at the same rate as wallclock time (one second per second).

Simulation time can be stepped to a future point, at which point it resumes advancing at the same rate as wallclock time. For an accurate simulation, time should not be stepped unless the society is in a state of quiescence (reached a steady state after a planning change or previous time step). Although Cougaar does not prevent this by default, a Cougaar society can be set up to wait for quiescence before allowing a time step.

Here is how the general HLA time synchronization methods apply to Cougaar federates:

- No synchronization: This is not useful for a Cougaar society in execution (simulation) mode. However, if a society is only used as a planning model, this would be fine as there would be no point in expending effort to synchronize the society.
- Conservative synchronization: This is the appropriate mechanism for a Cougaar society in execution mode. It fits well with Cougaar's time advancement mechanism.
- Optimistic synchronization: This is not a good match for a Cougaar society because Cougaar currently cannot roll back time. Since federations can contain federates with different synchronization methods, this does not limit the ability of Cougaar societies to be federation members.

4.6 Development Approach

To minimize development time and maximize learning, we used an iterative development approach with three main phases. This approach allowed us to get the prototype working in a simple, albeit non-standard, manner before moving on to the correct but more complex way.

The technical details of these steps as well as the lessons learned are covered in Section 5.

1. Build a federation without HLA time management

This federation included two federates, each mapping to a Cougaar society. It involved building the prototype societies and checking that they ran independently on the same computer with no Cougaar interactions, interfacing the Cougaar society to a Java-based 1516 RTI via a single ambassador class, communicating demand via supply interactions, and coordinating time via time interactions. It also involved updates to some plugins and the time advance servlet. Once this operated successfully, we tested it with a non-Cougaar federate performing demand generation, and then moved it to a Java-based 1.3 RTI and confirmed operation.

2. Build a federation with HLA time management

This federation also included two federates, each mapping to a Cougaar society (Figure 2). It involved upgrading the society-RTI interface from a class to a Cougaar service, moving to a non-Java-based, partially asynchronous 1.3 RTI, and replacing time interactions with a standard RTI time synchronization mechanism. It also involved further updates to some plugins and the time advance servlet.

3. Build a federation with a multi-node Cougaar society:

This federation included three federates, two mapping to each of the two nodes in the high-fidelity logistics society, and the third mapping to the other society (Figure 3). It involved splitting the high-fidelity logistics society into two nodes and retesting the federation.

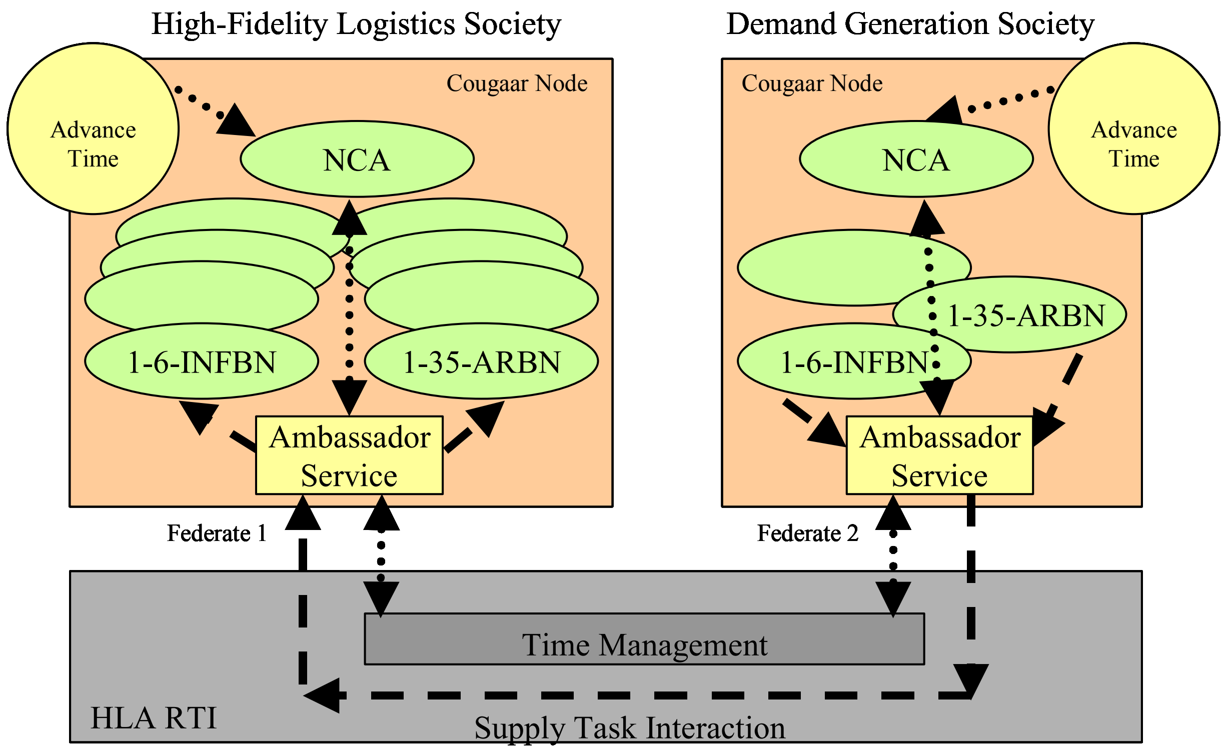

Figure 2. HLA Federation with Cougaar Federates

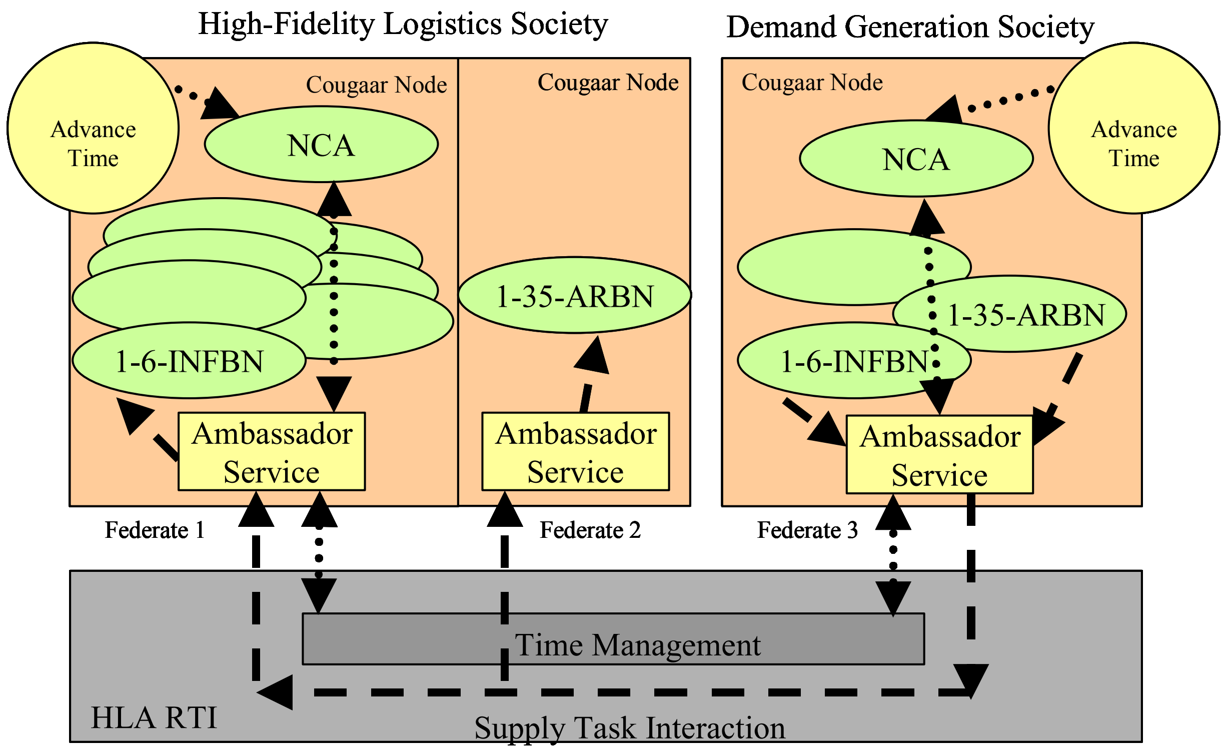

Figure 3. HLA Federation with Cougaar Federates Including a Multi-Node/Multi-Federate Cougaar Society

4.7 Results of Integration Experiments

We met all of our technical goals during development and testing of the prototype system.
We demonstrated interoperability in the following types of federations:

- Two Cougaar societies as federates in an asynchronous and a partially asynchronous HLA 1.3 federation
- Two Cougaar societies as federates in an HLA 1516 federation
- A multi-node Cougaar society as two federates and another society as a third federate in an HLA 1516 federation
- A Cougaar society and a simple non-Cougaar simulation as federates in an HLA 1.3 federation
- A Cougaar society and a simple non-Cougaar simulation as federates in an HLA 1516 federation

We demonstrated the following operations:

- Subscribing a Cougaar federate to specific HLA interactions
- Detecting tasks in one society, transferring the task information via HLA interactions, and reconstituting the tasks in a second society
- Synchronizing society time via HLA interactions
- Interfacing a Cougaar society to the RTI via a single ambassador class
- Subscribing plugins to specific HLA interactions
- Interfacing a Cougaar society to the RTI via a Cougaar service
- Synchronizing society time via the HLA mechanism using conservative synchronization

We learned techniques for creating or modifying Cougaar societies to be HLA federates (described in Section 5).

We identified further areas of research as well as additional functionality required in a full-scale Cougaar HLA federate (described in Section 6).

5 Lessons Learned

We learned or developed a number of techniques for creating or modifying Cougaar societies to be HLA federates.

5.1 Interfacing Between Society and HLA RTI

There exist a number of ways to interface between a Cougaar society and an HLA RTI. In our prototype we first created a singleton ambassador class that handled the communication. We used different ambassador classes for different RTIs instead of modifying the same one. This simplifies moving back and forth when necessary.

The ambassador class performs interactions between the society and federation, including instantiating the RTI's ambassador class, joining the federation execution (creating it first if necessary), connecting to interactions and objects. It also handles the interactions and object attributes from the federation and distributes them to the appropriate agent in the society.

We later changed the interface to a Cougaar service, which essentially provided the same functionality (by wrapping the ambassador class) but in a Cougaar-compliant manner. In a multi-node Cougaar society, the ambassador class is instantiated separately for each node. Each node thus becomes a separate federate. If this is not desired, the ambassador service would need to be modified to create only one federate per society.

5.2 Implementing Time Synchronization

We implemented conservative time synchronization in two different ways. For maximum flexibility, we used a servlet-based, user-directed time advancement mechanism in our demonstration societies. This allowed us to operate with a variety of simulation models.

In phase one of the prototype, we implemented time synchronization without HLA time management. In this implementation, the high-fidelity logistics society advanced its own time directly and advanced time in the demand generation society using HLA interactions. When the user advanced time in the high-fidelity logistics society, it sent a time interaction to the RTI. When the demand generation society received the interaction, it advanced its time to the specified value. This scheme worked because the second society was so small it was already ready to advance time when the other society requested. We implemented it so we could focus on learning about interactions at that point in our development cycle. Although it works in special situations, it is not appropriate for general-purpose use.

In phases two and three of the prototype, we implemented time synchronization with HLA time management. This implementation involved performing conservative time synchronization using the standard HLA mechanism and a modified version of the user-directed Cougaar time advancement mechanism. As described earlier, the existing Cougaar mechanism for advancing time directly advances time when the user requests it. To work with HLA, this mechanism must be expanded to obtain permission from the RTI before advancing time in the society. This augmentation involved implementing the ambassador class, adding plugins to handle time, and modifying the time advance servlet.

5.3 Details of Time Management Integration

Time synchronization with HLA time management requires close coordination between the two mechanisms.

At initialization the ambassador class joins the federation execution, gets handles, and sets up interactions. It then performs the following operations to set up for time synchronization:

- Enable asynchronous delivery of messages
- Enable time-constrained mode (allows the federate to receive time-stamped messages); wait for confirmation
- Enable time-regulating mode (allows the federate to generate time-stamped messages) with default lookahead and default current time; wait for confirmation
- Request a time advance to the current society time

From that point on, the ambassador class handles interactions and time advance requests asynchronously.

The Cougaar society uses a servlet at the top-level agent, which allows the user to advance time. In this prototype, the user needs to manually confirm that the society has reached quiescence before attempting to advance time (in a full system this would be automated). For experimental purposes, the servlet allows the user to manually drive a time-stepped or event-driven style simulation (this also would be automated in a full system).

Normally, this directly changes the society time. To work with the RTI time synchronization method, this needed to be changed by modifying the servlet and adding a plugin to the agent. Instead of directly changing the society time, the servlet now places a time change request object on the Cougaar blackboard for the agent.

The plugin added to the agent subscribes to the time change request object. When it appears on the blackboard, the plugin reads the time request and calls the ambassador service with the time and a pointer to itself. The service makes sure that there isn't a pending request for a time change and sends the time advance request to the RTI.

If there is a pending request, the new request gets dropped in the prototype. This could be handled differently in a full implementation.

If no other messages less than or equal to the new time are forthcoming, the RTI grants the time advance to the society. If the time advance request is granted, the ambassador service calls a callback method on the plugin to actually change the time. This design keeps all society time interaction in the plugin code and all RTI time interaction in the ambassador code. The ambassador class knows about the callback method through a simple Java Interface, so all society information is hidden from it.

Whenever a time advance request is granted, the ambassador class stores the new federation time so that it can check whether the next request is successful. The RTI delivers messages to the society with times less than or equal to the new time, and the society starts handling them. If the time advance is not granted, the ambassador service does nothing. In this prototype it is up to the user to repeat the request.

In this implementation, the user must make time requests in both societies using the Cougaar time advance mechanism. A time advance in a society was not granted until the other society had requested an advance to that time or beyond.

5.4 Mapping Agents, Actions, and Objects

A Cougaar society contains many objects, so mapping these objects to HLA actions and objects must be done with care to ensure acceptable performance. Data can be transferred in HLA via interactions (events) and/or object attributes. How these entities map to Cougaar entities is heavily dependent on the society design.

In general, Cougaar agents or asset objects would map to HLA objects. This mapping should only be set up for those agents and objects where information needs to be transferred. This could be the case for equipment agents (i.e., where an agent is modeling a tank).

Events in Cougaar would map to HLA interactions. Again, the mapping should only be made for those events where information needs to be transferred throughout the federation.

Cougaar objects such as UltraLog tasks could be mapped either way. In our prototype, we only needed to know when the tasks appeared in the demand generation society and what they contained. In this case, it made sense to model them as interactions. The information was transferred throughout the federation and then the interaction disappeared.

In other cases, tasks could be modeled as objects. Tasks can exist for an extended period of time and their contents can change. If this is important to capture in other federates, it would be appropriate to model the tasks as objects.

6 Design Considerations for Full-Size Societies

This section discusses some considerations for designing a full-size Cougaar society for integration with an HLA federation.

6.1 Society and Federation Organization

Society and federation organization need to be carefully considered when designing a full size Cougaar society that is HLA compliant.

One factor to consider is the federate granularity: the level of mapping of federates to portions of a Cougaar society. The federate granularity should be based on what makes sense for the size and organization of the society, as well as what makes sense for interoperability with the non-Cougaar federates. In our demonstration prototypes, each Cougaar node mapped to a federate. In a large complex society, however, a logical group of agents (covering multiple nodes) might logically constitute a single federate. For example, this might be a group of agents defining the transportation organization or modeling a specific combat organization. Single agents could even map to individual federates. However, if this design causes many of the normal Cougaar mechanisms to be bypassed, it might make sense to implement the agents differently.

If a federation contains multiple Cougaar societies, a factor to consider is society design. Specifically, which agents are present in each society? There are two ways to organize the Cougaar societies. In our prototype we used a functional split where the same (or subset of the) agents had different functions. This allows the entire simulation organization to be modeled in a single society, with various organizational functions handled within the society or by other simulations in the federation.

Another way to design the societies is with an organizational split. Each society would contain different agents in logical groupings. For example, one society could contain the combat organization agents while a second society could contain the transportation organization agents. In this case, a society could be replaced by another federate if it better modeled the organization.

6.2 Time Synchronization

The time advance mechanism in our demonstration prototype requires manual operation for experimental purposes. In a full operational society, this mechanism would need to be automated. The time advance requests would be triggered automatically, either according to a regular time schedule or driven by an appropriate event or events. The mechanism would also need a check to ensure that the society had reached quiescence before requesting a time advance.

An important point to keep in mind is that Cougaar society time is always advancing at wallclock time, even though its HLA logical time is still that of its last time advancement grant. The time step in these federations needs to be large enough so that the society time advances an insignificant amount between these time steps.

6.3 Adaptive Society Design

There are enhancements that can be made to a society to allow it to operate more flexibly in the presence of a federation.

The prototype HLA ambassador class hardcodes the HLA interactions and objects to which it subscribes, along with their attributes. The problem with this is that when we want to use a new interaction or when an interaction definition in the FOM file changes, we need to modify this class.

A better way to do this would be to hardcode nothing in the ambassador class. Each plugin would register with the ambassador class and tell it which interactions and objects it was interested in. The ambassador class would then dynamically subscribe to the requested interactions, read the FOM file to get the attributes for each interaction, and then dynamically get the attribute handles.

In addition, instead of hardcoding knowledge of objects throughout the federation, it could use object discovery to match up objects in federates with those in the society. Cougaar incorporates a sophisticated service discovery mechanism; object discovery could leverage it.

6.4 Operation Independent of Federation

For more flexibility and survivability, a society can be designed to operate whether or a not a federation is present, and whether or not the appropriate federates are present. For example, the society could be designed to contain various default plugins that perform default simulation operations. However, if a federation with more advanced simulation elements was present, these would be preferable to use. The society could check for the presence of federates that perform these functions; if not found, it would use the default plugins.

Another approach would be to use the Cougaar Message Transport Service (MTS) and create an HLA-specific link protocol. This approach would be appropriate in an environment where a society may or may not be a member of a federation and where it has a number of other communication channels at its disposal. MTS would then select the appropriate communication method based on availability and prioritization of channels. This could also allow interoperation with simulations that exist outside of both the Cougaar society and the HLA federation.

7 Conclusion

We have described how a Cougaar-based agent society modeling a small logistics supply network was integrated as a federate in V1.3 and 1516 HLA federations. Although distributed agent-based architectures can themselves be used as a means of interoperating with other simulations, it can be more useful to integrate them into a standard simulation interoperability architecture such as HLA, leveraging the benefits of both architectures. Our work established how to interface the society and the HLA RTI; how to synchronize society time with HLA RTI time; and how to map agents, objects, and actions in the society to HLA objects and interactions. We also examined considerations for integrating a full-size Cougaar society with an HLA federation. The lessons learned in this demonstration integration effort can be applied to large-scale efforts, such as integrating large Cougaar-based logistics simulations with combat simulations such as OneSAF. Such integrated simulations would provide greater effectiveness than combat-only simulations.

8 References

[1] "Cougaar Open Source Web Site," http://www.cougaar.org/.

[2] J. Stavash, B. Chadha, J. Wedgwood, J. Welsh, M. Parker, D. Teitelbaum, "Agent Based Models for Logistics in Wargaming," Proceedings of the Fall 2003 SISO Simulation Interoperability Workshop.

[3] "About Distributed Agents," IEEE Distributed Agents Online, http://dsonline.computer.org/agents/about.htm.

[4] M. Thome, "Multi-Tier Communication Abstractions for Distributed Multi-Agent Systems," 2003 IEEE KIMAS Conference, Boston, MA, 2003.

[5] "UltraLog Web Site," DARPA, http://www.ultralog.net/.

[6] K. Kleinmann, R. Lazarus, R. Tomlinson, "An Infrastructure for Adaptive Control of Multi-Agent Systems," Presentation Slides, 2003 IEEE KIMAS Conference, Boston, MA, 2003.

[7] "High Level Architecture," Defense Modeling and Simulation Office, US Department of Defense, https://www.dmso.mil/public/transition/hla/.

[8] J. Dahmann, R. M. Fujimoto, R. M. Weatherly, "The DoD High Level Architecture: An Update," Proceedings of the 30th Winter Simulation Conference, Washington, D.C., 797 – 804, 1998.

[9] R.M. Fujimoto, "Time Management in the High Level Architecture", Simulation, 71:6, 388-400, 1998.

[10] E. Schmitt, "Army Study of Iraq War Details a 'Morass' of Supply Shortages," New York Times, Feb. 3, 2004.

Author Biographies

GARY KRATKIEWICZ is a Scientist at BBN Technologies in Cambridge, Massachusetts. He has developed software for a variety of advanced systems, including a data-intensive Web-based logistics modeling system for the U.S. Defense Logistics Agency and a large-scale distributed agent-based logistics system for DARPA. Gary has spoken at a number of technical conferences, including JavaOne, the Lightweight Languages Workshop, and the SISO Simulation Interoperability Workshop. He holds an S.B. from MIT and an M.S. from Stanford University.

AMELIA FEDYK is a Senior Software Engineer at BBN Technologies in Cambridge, Massachusetts. She has developed plugin and infrastructure software for Cougaar, an open source agent architecture for large-scale, distributed multi-agent systems. Prior to working with BBN, Ms. Fedyk worked primarily in time-critical interfaces including air traffic control systems and distributed battle simulators. She holds B.A. and B.S. degrees from St. John Fisher College.

DANIEL CERYS is a Senior Scientist at BBN Technologies in Cambridge, Massachusetts, where he has applied advanced software capabilities in the logistics domain for over a decade. He has been applying distributed agent-based technologies to military logistics programs for DARPA, DLA, and the Army. He holds B.S. and M.S. degrees from Stanford University.