Proceedings of the 2002 Winter Simulation Conference
E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, eds.

THE UMBRA SIMULATION FRAMEWORK AS APPLIED TO BUILDING HLA FEDERATES

Eric J. Gottlieb

Orion International Technologies, Inc.
2201 Buena Vista Drive, SE, Suite 211
Albuquerque, NM 87106, U.S.A.

Michael J. McDonald
Fred J. Oppel
J. Brian Rigdon
Patrick G. Xavier

Sandia National Laboratories
P.O. Box 5800, M/S 1004
Albuquerque, NM 87185-1004, U.S.A.

ABSTRACT

Sandia's Umbra modular simulation framework was designed to enable the modeling of robots for manufacturing, military, and security system concept evaluation. Umbra generalizes data-flow-based simulation to enable modeling of heterogeneous interaction phenomena via a *multiple worlds* abstraction. This and other features make Umbra particularly suitable for developing simulation federates. Umbra's HLA interface library utilizes DMSO's HLA Run Time Infrastructure 1.3-Next Generation (RTI 1.3-NG) software library to federate Umbra-based models into HLA environments. Examples draw on a first application that provides component technologies for the US Army JPSD's Joint Virtual Battlespace (JVB) simulation environment for Objective Force concept analysis.

1 INTRODUCTION

The Umbra simulation framework was designed to enable the modeling of robots for manufacturing, military, and security system concept evaluation. Many robots in these applications are embodied agents, which are entities having behavior, control, geometry, sensing, physics, communications, etc., and that are affected by and interact with their environment and its inhabitant entities.

Umbra generalizes certain aspects of modular data-flow-based simulation in a way that also enables linking together heterogeneous modeling tools. Users can quickly build models and 3D interactive simulations for system development, analysis, experimentation, and control. Model components can be built with varying levels of fidelity and readily switched; models built for conceptual analysis can be gradually converted to high fidelity models for detailed analysis. Umbra has been used in modeling robots ranging from manipulators to swarms of autonomous mobile robots with sensors and radio communications.

Several of the features and software abstractions—particularly, multiple worlds for heterogeneous interactions—that enable the Umbra framework to flexibly support embodied agent simulation also make it ideal for developing simulation federates for High-Level Architecture (HLA) federation. The Defense Modeling and Simulation Office (DMSO) has developed and supports a software library called the Run Time Infrastructure, or RTI, which implements the HLA interface specification and facilitates building HLA-compliant codes. The RTI manages all inter-process communications, such as data exchanges between federates, in an HLA federation. We have developed Umbra's HLA interface library to interact with DMSO's RTI 1.3-NG library (DMSO 2001) to enable Umbra models to be federated into HLA environments.

The US Army JPSD tasked Sandia with developing robot models that operate within its Joint Virtual Battlespace (JVB) simulation framework for Objective Force concept analysis. Unmanned Air Vehicle (UAV) and Unmanned Ground Vehicle (UGV) models and others we developed in the first year of effort provide examples of Umbra simulation and our HLA interface implementation.

2 UMBRA

2.1 Background

Umbra's development was driven by a need to analyze a wide range of robot systems in stages of development from conceptual design to hardware-in-the-loop experimentation. There are extensive modeling and simulation technologies for many domains. For example, LabView (National Instruments 2002), Simulink (Mathworks 2002), SMART (Anderson 1997), and Chimera (Stewart et al. 1997) enable modular control system simulation and/or development via various port-based composition abstractions. Commercial software such as ADAMS (Mechanical

Dynamics 2002) can be applied to certain mechatronic and physical systems, while modular, composable architectures such as that of Diaz-Calderon et al. (1998) and modeling languages such as ModelicaTM (Otter et al. 1999) attempt to cover them more generally. AVS (AVS 2002) offers a modular framework for combining simulations, and many other specialized simulators exist. None of these technologies met our needs, although some of them proved the utility of certain abstractions and features. Furthermore, specific simulation needs for future robot system exploration and development are unpredictable.

2.2 Basic Umbra Overview

Instead of attempting to be an all-encompassing architecture for simulation, Umbra is a simple modular, composable simulation framework that is conducive to both writing simulation components from scratch and to leveraging existing external simulation capabilities.

Umbra builds on continuous-time (timestepped) data-flow-based simulation. The basic component unit of Umbra simulation is a *module*, which encapsulates a state and a computational model. A module also may have *input ports* from which it can read data and *output ports* at which it can present data. An output port can be connected to input ports, making the output port data readable there. A *dynamic module* has a virtual *update* (work) function that is automatically called once each time through the simulation loop, which is known as the *Umbra update* cycle. Ports and connections are typed and can present interfaces. Specific model classes are derived from the base module classes. Non-native model libraries and legacy codes are integrated by writing encapsulating module libraries.

The Umbra framework is implemented in C++, and Umbra module libraries are usually written in C++. Umbra integrates Tcl for an interactive shell and scripting language. Modules are typically named upon creation and are addressable via these names in Tcl. Furthermore, data ports on named modules are automatically accessible in Tcl via their labels, as are designated member functions. Tk is optionally used for constructing GUIs. Umbra also includes mechanisms that support event-driven simulation, such as C++ and Tcl callback objects. A scene graph and an interactive, OpenGL-based viewer are integrated optionally into Umbra and are used by configuring kinematics and geometry modules appropriately.

A start-up Tcl script is typically used to first load the Umbra libraries containing the desired module classes and then to call C++ code to create the desired modules and connect their ports as required. Umbra enables extensive computational steering, allowing users, for example, to add obstacles to terrain models to examine dynamic control response, to add/delete modeled entities, and to interrupt simulations and swap in different component models. A significant departure from the usual data-flow paradigm is discussed in Section 2.4.

2.3 General Robot Modeling Issues

The Umbra framework naturally supports modular system-level models that mimic system structures. Figure 1 shows (simplified) the typical modular organization of a model of a robot with a classical control system. The individual modules are connected into a *meta-module* – a module network connected in the same way as the real robot components. The model includes a collection of sensor modules (shown as a shadowed box), a behavior/controls algorithm module, and a physical plant or physics module. The update function of a continuous-process module computes its transfer function. For example, an aircraft physics module might take, as input, a set of control surface angles and engine thrust and output position (and orientation). The sensor module might use the aircraft's state values to compute a set of sensed values, which the control module would use to compute the new flight control values. Command input, not shown, is typically introduced to the behavior and control modules as an asynchronous event.

Non-linear components of robots are readily modeled in Umbra. For example, vehicle physics models can be made responsive to the geometry of the environment. Sophisticated behavior, planning, etc., components can be modeled by Umbra modules as long as there are incremental models of their computations and adequate resources. A simulator clock module (not shown) provides the current time and step size to other modules via its output ports.

To prevent causality violations due to fortuitous update order, the update functions of modules must be computed in a constrained order. In the simple application of the Umbra framework, each component model is a dynamic module. Their update order is computed from the directed graph whose nodes are modules and whose edges are connections. This graph is made acyclic (enforced as connectors are added) by specifying *feedback* connections and therefore determines a partial order. Other constraint mechanisms, including dummy connectors, exist.

2.4 A Worlds Abstraction for Simulating Interactions

It is necessary to model robots whose state evolutions are dependent on their environments and the other entities in them. For example, concurrent incoming RF communications signals may interfere with each other, resulting in message loss or corruption. A meta-module as shown in Figure 1 might be used to model a single robot in a static environment; however, without a mechanism for sharing

data across meta-module boundaries, a set of such meta-modules would be limited to modeling multiple robots that never interact and that cannot sense each other.

While enabling the modeling of interactions, Umbra's *Worlds* abstraction preserves the ability of meta-modules to follow the system structure of the entities they model. We model each interaction phenomenon with a specific *world module* that computes the coupled part of the update of the modules participating in the interaction. The world module governs the world of interactions that are its domain. We have found it convenient for the world module for a phenomenon to also serve as the factory (Gamma 1994) for the participant modules and sometimes to manage their updating.

Figure 2 illustrates the relationship between meta-modules and worlds. The world module that governs the Communications World is responsible for computing the "signal" at each of the Communications In modules due to the transmissions of the Communications Out modules. Since this computation is done by that world module's update function, it is atomic with respect to other modules' updates and prevents violations of causality. Except for connections to the simulation clock (not shown), the Communications World module and each of the vehicle meta-modules are separate components of the data-flow graph. The modules outside the Communications World do not even know it exists.

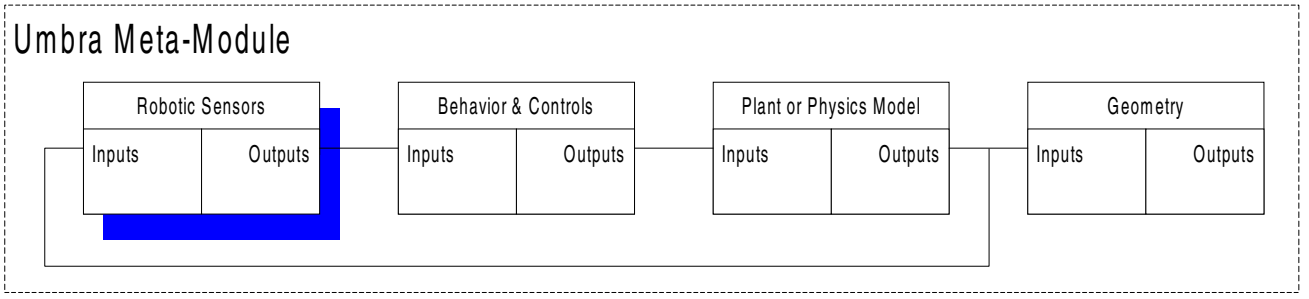

Figure 1: Conceptual Umbra Robotic Meta-Module. Geometry model module is used in visualization.

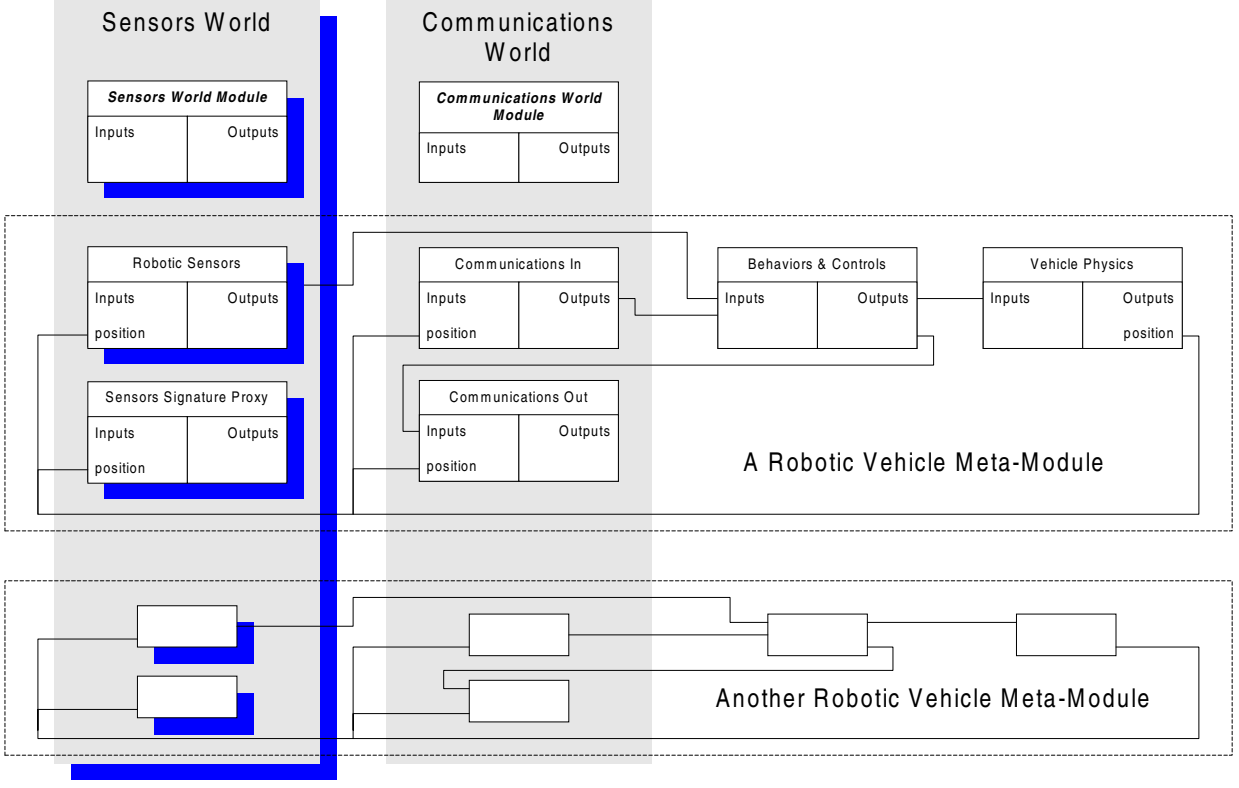

Figure 2: Umbra Robotic Vehicle Example Showing Relationship between Meta-Modules and Multiple Worlds. Simulation clock module and related ports and connectors are omitted.

Because signal reception is dependent on the positions of the senders and receivers within the environment, the position of each Communications Out and Communications In module is determined from the position output of the VehiclePhysics module of the meta-module to which it belongs. It is the world module's responsibility to model the effects of position and environment, which it may do using its own geometric model or using a shared one, for example, provided by a Geometry World (not shown).

While robots often carry only one communications device, they typically use several sensors (for example, Figure 3). The shadowed Sensors World box and shadowed boxes for associated modules represent this plurality. In practice, there are usually multiple Sensor Worlds. For each sensor type, the sensor world needs adequate information to simulate each sensor's response to its environment, including dynamic inhabitant entities such as robots. In each sensor world, for every dynamic entity that can be sensed, this data is provided by a sensor signature proxy module that is a part of the meta-module for that entity.

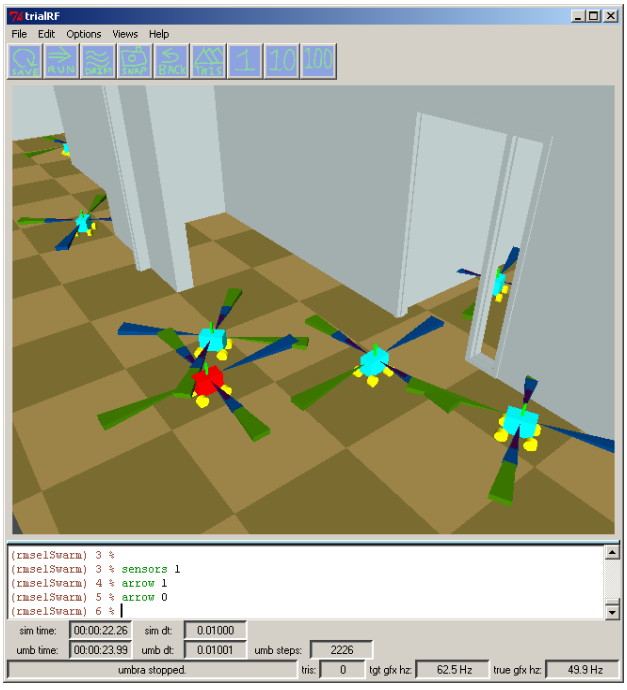

Figure 3: Umbra Simulation of Robots Collaboratively Exploring, with Sensor Response Regions Shown

3 UMBRA AND HLA

3.1 Overview

The natural level of modularity in Umbra models makes it well suited for both the entity-level and component-level composable integration found in federated simulations. The HLA paradigm allows various aspects of the models to be distributed among separate federates at a component level. For example, Behavior Services might generate commands, status and high-level reports. Other elements of the HLA simulation environment might transfer task and report data to and from the Behavior Services while propagating this C4ISR data through the environment. Platforms, sensors, and other objects might be attached, either temporarily or permanently, to one another to represent systems with combined functionality.

The Umbra HLA Interface Library (Gottlieb et al. 2002a) enables models to be composed conforming to an HLA federation's interface as defined through its Federation Object Model (FOM). For example, robotic system models for the JVB are implemented in accordance with the JVB-FOM with separate Behavior and Platform Services. Where important (e.g., for efficiency), behavior may be tightly coupled to sensor input or platform motions within Umbra. At the same time, HLA Objects separately representing Platform and Behavior Services are presented to the HLA as separate services. This internal coupling is transparent to the FOM to allow maximum flexibility.

Figure 4 shows a conceptual diagram of how Umbra robot vehicle meta-modules may be integrated via HLA for federation. Here, Umbra publishes behavior and platform data through separate HLA objects, while Umbra sensors and physics models subscribe to HLA sensor, environment, and Mobility Services. Published services thus may depend both on internal (Umbra) models and, for loosely coupled systems, on external ones through subscribed services. For example, robotic tanks might model mobility and battle damage within Umbra or externally by having Umbra subscribe to services.

Through the HLA interface, other systems instantiate, command, and monitor individual as well as integrated collections of platforms. In addition, Umbra can monitor HLA objects that it does not control. This monitoring is important for allowing Umbra to interact with the entire HLA simulation environment.

3.2 Umbra HLA Ambassador and Proxy Modules

The Umbra HLA library is built on DMSO's HLA Run Time Infrastructure 1.3-Next Generation (RTI 1.3-NG) library. DMSO's RTI provides an abstract class, called FederateAmbassador, that identifies the callback functions that each federate is obliged to provide. The Umbra HLA library provides an Ambassador class that implements these functions and that is also a subclass of UmbDynamic (dynamic module). For example, Ambassador implements create/destroy, join/leave, and object and interaction subscription and publication.

The Umbra HLA implementation builds on the Worlds abstraction to achieve a close matching of Umbra modules

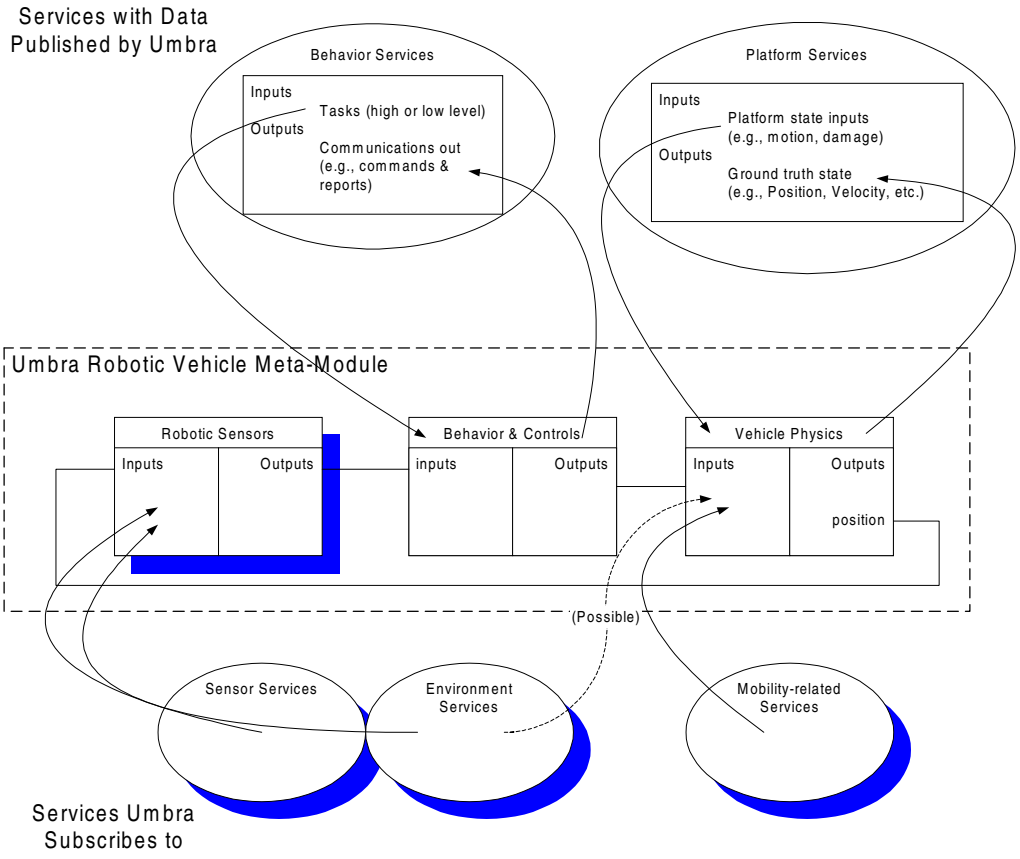

Figure 4: Conceptual Diagram Showing Umbra-to-HLA Integration for a Robot Vehicle

to HLA components. An Ambassador module establishes conceptually and governs an HLA World. Umbra simulations publish and subscribe to RTI interaction parameter and object attribute data, and subsequent updating and reflecting is done through proxy modules. For each federation an Umbra simulation joins, a separate Ambassador module manages communications between the RTI and the associated proxy modules. Each HLA Interaction Class or Object Class that an Umbra simulation publishes or subscribes to is represented within Umbra with an application-specific interface proxy module class. These modules proxy the HLA by maintaining local state data concerning the attributes and parameters and by providing mechanisms for moving HLA state and interaction data between other Umbra modules and the HLA environment. Interface proxy modules do not typically provide modeling services computationally, except for dead-reckoning. Rather, they communicate with other modules within the Umbra environment that in turn provide modeling services.

For example, recall that a robot is typically modeled within Umbra by various connected modules. Elements of this model may have counterpart HLA objects. Umbra HLA proxy modules, such as the HlaRobotModel module described later, are connected to these elements to transfer state (attribute) data between the corresponding Umbra modules and the HLA world.

The HlaModule class, derived from UmbDynamic, is the base or virtual proxy class for handling interactions. HlaClassModule, derived from HlaModule, adds the ability to proxy instances of HLA Object Classes. Application-specific classes are derived from these module classes; in particular, instances of HlaClassModule subclasses are incorporated into meta-modules to implement reflection and updating of HLA object attributes. Figure 5 shows the class hierarchy of these modules in relation to module classes developed for particular applications. HlaModule modules can be programmed to send or receive any interaction allowed by federation management. More than one module can be programmed to receive the same interaction. Here, the Ambassador module provides a copy of the interaction data to each module so that it can perform its separate processing function.

In addition to its other roles Ambassador also provides HLA module factory and scheduling services, as well as time-management services. The Ambassador module is a factory for all HLA object and interaction proxy modules, while a specific interaction proxy module serves as the factory that constructs meta-modules as needed to support re-

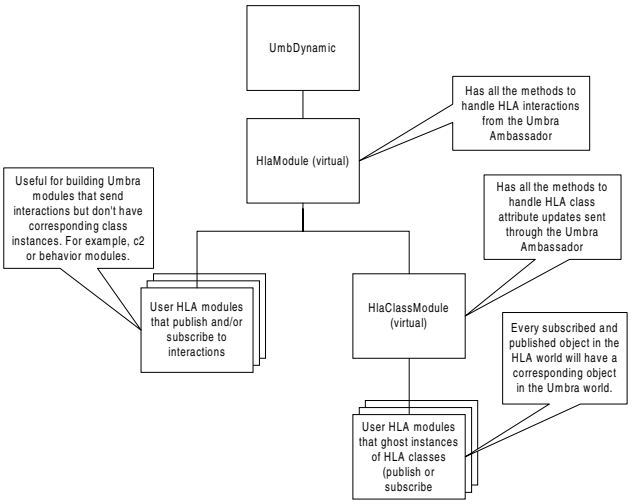

Figure 5: Typical Class Hierarchy for Application-Specific HLA Objects

mote creation of HLA object instances. Use of callbacks to Tcl-procedures that construct these modules and meta-modules enables these factories to be application independent. The Worlds abstraction is used to provide an added level of control between the Ambassador and the interaction and object proxy modules.

3.3 Data and Interaction Exchanges and Time Management

We now describe the Umbra HLA library's mechanisms for data exchange, interaction processing, and time management, which exploit the Umbra framework. See Gottlieb et al. (2002a) for descriptions of mechanisms for object class and interaction class publication and subscription – i.e., "signing up" to produce or receive object attribute updates or interactions.

The Ambassador coordinates all data exchanges and interaction exchanges between an Umbra simulation and its associated HLA federation. At a user-adjustable frequency no greater than Umbra's main update frequency, the Ambassador walks its HLA proxy module list and sends each an *Ambassador Update* message. Within the call, each of these modules can make calls to the RTI, e.g., to update HLA object attributes or request updated HLA object attributes.

After the last proxy module has been updated, the Ambassador does an RTI tick. The tick allows the RTI to make its callbacks to the Ambassador. The most typical is a call to process an interaction or reflect attributes. Within each RTI-invoked callback, the Ambassador maps the object handle provided by the RTI to the appropriate proxy module, calls its standard callback method and passes appropriate data. Some modules cache the exchange request and finish processing it later. For example, attribute data to be reflected is simply processed, for presentation to other modules, during that proxy module's subsequent Umbra update. Other requests might result in a (C++) callback object being stored for invocation during the next Ambassador update cycle. In other cases, the proxy module forms a Tcl callback and posts it, possibly with some delay, in the Tcl event loop. Later, the Tcl event loop processes the function (outside the RTI callback).

HLA time-management services must be supported to control time disparity among federates. The Umbra HLA Ambassador implements a scheme that allows the HLA federates to take large time steps (e.g., greater than 1 second) while the Umbra environment uses small internal time steps (e.g., less than 100 milliseconds). In the limiting case, the design allows Umbra to operate with arbitrarily large federation time steps. Our solution implements a time base that interpolates federation time to derive a local (interpolated) time that can be used by timestep size sensitive modules such as physics. Supported are: non-regulating, unconstrained time management; regulating and constrained time management; and non-regulating, constrained time management. The Ambassador serves as the Umbra simulator clock module for the Umbra models in its federation, providing a connector-based interface to its local interpolated time and timestep size *dt* that simulation modules can use for computation.

4 EXAMPLES IN JVB AND JVB DEVELOPMENT

4.1 Umbra Robotic Vehicle Simulation for JVB

Sandia's involvement in JVB provides examples of how the above can be used in HLA federated simulation and to support federated simulation development. Within an Umbra federate, tasks and sensor-readings interactions are treated as communications and data received through Umbra communications and sensor modules. Robotic ground truth state information (vehicle positions, velocity, etc.) is computed by Umbra Vehicle Physics modules and reported back to the HLA layer as Platform Services. Communications, including status reports, are generated by Umbra and reported through the HLA layer to Behavior and Communications Services. Robot models might also command and exchange data with other automated devices, such as sensors or other robots, during their normal operation. These interactions are coordinated as Behavior Services that are invoked by the control, command, and communications (C3) grid. Other functions and state changes, including damage states, are propagated to the sensor and physics modules as appropriate.

Robotic platforms types include UGVs, UAVs, and exotics. Platforms transmit state data and changes according to

the JVB FOM. Position, speed, acceleration (both Cartesian and angular) are reported in JVB required formats at required frequencies. Umbra Platform Services support instantiation and destruction operations. Instantiation calls include data that describe specific platform features.

A number of specific Umbra models and libraries have been developed for the JVB project (Gottlieb et al. 2002b). A central task for Sandia has been notional UGV and UAVs models and their HLA interfaces. In order for vehicles published by other federates to be tracked and rendered in Umbra, "stealth" models were developed. An HLA proxy module class represents platform state of both UAVs and UGVs. HLA interaction interface modules receive remote_create commands from the HLA control layer and various movement commands from the C2 Grid.

Umbra models in JVB use Compact Terrain Database (CTDB) data via the Umbra CTDB Terrain (UCT) library interface to DMSO's LIBCTDB library from MODSAF and ONESAF Testbed. Umbra interface modules that utilize CTDB terrains form a key part of the UCT library. These modules include terrain lookup modules as well as physics, sensor and communications models. The CTDB interface modules exploit Umbra's Worlds technology. A Terrain world module is a factory module that also supports a variety of CTDB initialization and lookup functions. The Terrain world module is also the factory for physics modules, including the Constrainer and RoadSensor module mentioned below.

4.2 Notional Robotic Vehicle Models

The UGV meta-module in Figure 6 shows the module configuration for the notional UGV developed for JVB's November 2001 experiments. Each robot model includes WayQueue, CarWayPoint, SCMachine, CarPhysics, Constrainer, Model(-in-Scene), RoadSensor, and HlaRobotModel modules. The Ambassador is shared.

The WayQueue module generically feeds a list of positions and times to the WayPoint module. WayQueue takes its input from a Tcl command with a list of positions and outputs them in an ordered way on its wayPoint connector. Typically, input comes from HlaOrder modules described later, and an arrived connector is used to pulse the wayPoints generation. WayQueue modules generate waypoints for one-way, bi-directional, and cyclic paths. The CarWayPoint module outputs a vehicle steering and speed command that attempts to drive the vehicle toward the goal and arrive at the specified time. It updates this command with respect to the time output by the Ambassador module and the position reported by the Constrainer module. The RoadSensor module contains a list of configurable sensors that monitor the state of pre-determined ground conditions. For example, sensors can be programmed to sense whether the terrain at a given point is part of a road.

The SCMachine module takes an input control vector (steering and speed) and overrides or offsets it on the basis of sensor input. The module is programmed via Tcl with a list of offset operators that correspond to each state generated by a sensor module. In the notional model, the SCMachine uses inputs from RoadSensor in overriding the input driving commands, up to a point, to try to keep the vehicle on the road; it drives more slowly off-road. The CarPhysics and Constrainer modules together compute the motion of the vehicle. The CarPhysics module computes the vehicle's instantaneous velocity (including rotational) based on either an Ackerman steering or a skid-steering model. The Constrainer module corrects the velocity based on the terrain geometry and mobility values and in-

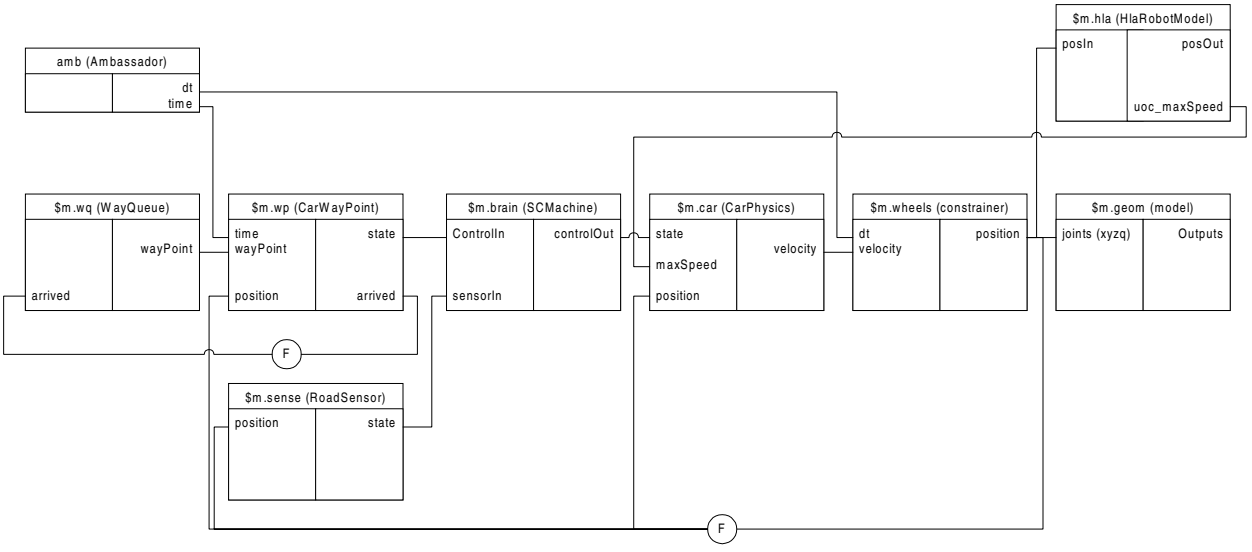

Figure 6: Notional UGV Model. Circled 'F' explicitly denotes feedback connector.

tegrates the adjusted velocity to compute the vehicle's new position on the terrain. The optional Model-in-Scene module (Model) is a generic module provided by the Umbra core library. It is created by the 3D visualization module (Scene, not shown), which renders the vehicle's geometry in the 3D viewing window.

The HlaRobotModel module is a generic HLA proxy module that represents the platform state to the JVB federation. It is created by the Ambassador module as part of an HLA world. In a JVB simulation, Umbra generates individual proxy modules for each Platform Object Class instance in the simulation. When Umbra creates a platform, it creates a proxy *publisher* module that publishes the Object Class instance and updates its object attribute data. When Umbra discovers a platform (i.e., a platform published by another federate), it creates a proxy *subscriber* module that reflects the externally updated attribute data. A single module can even play both roles, dynamically relinquish ownership of some individual object attributes, and request and gain ownership of others.

For convenience, HlaRobotModel modules are used to proxy all Platform Object Class Instances. HlaRobotModel performs dead reckoning using algorithms similar to those in Singal & Zyda (1990), both as subscriber and publisher (to estimate error). Since module updates in Umbra federates are computed 10 to 1000 times more frequently than time requests are made, Umbra can accurately integrate behaviors over the relatively large time steps used in the JVB federation. A high-frequency filter is used to remove noise due to terrain-induced accelerations.

The notional UAV model developed for JVB's November 2001 experiments leverages the WayQueue, HlaRobotModel, and Model-in-Scene (configured appropriately) modules from the UGV model, and interacts with HLA via the same Ambassador. A UavController module takes wayPoint output directly from the WayQueue module and produces control outputs to fly to the current waypoint; if the WayQueue runs out of waypoints, UavControl flies a figure-eight about the last one. Control outputs are fed directly to UavPhysics, a flight model.

4.3 Stealth Models

In a variety of situations it is important to track and render vehicles entirely published by other federates. *Stealth* vehicle meta-modules, typically consisting of an HlaRobotModel module connected to a Model-in-Scene module, were developed to serve this purpose. Stealth models are subscriber models. Unlike the UAVs and UGVs described earlier, Stealth models are created upon object discovery. The Tcl command used to request that Umbra's Ambassador module subscribe to an object contains the script name for creating the stealth vehicle in its arguments. When the Ambassador discovers (via an RTI callback) that an object of the given class has been created, it calls this function with the name of the new object in its parameters. The callback script then calls the necessary functions for creating the stealth vehicle. Because object attribute data associated with platform instances is provided in separate callbacks from the discovery callback, the stealth model creation script is broken into two Tcl procedures. The first instantiates the HlaRobotModel and scene model modules and schedules a call to the second procedure with the Tcl after function. Instantiating the HlaRobotModel module provides a mechanism to parse and store attribute data received by the Ambassador. The second procedure (incrementally) fleshes out the HlaRobotModel module if data is available; if it is not, the procedure schedules itself to be called again later or requests new attribute data if there has been too much delay.

4.4 HLA Interaction Interface Modules

Two modules were developed to handle JVB-related interactions that Umbra responds to. The first, the HlaOrder module, handles command and control-related interactions. These interactions typically originate in C3 federates and contain commands for robots. The second, the HlaRemoteCreate module, supports JVB's remote_create simulation control function. These interactions typically originate in federation control tools (like HLA Control) and command simulation federates to dynamically create robot instances. While designed to receive interactions, these modules can also be used to send interactions, for example, to support Umbra and small federation testing.

Only one HlaOrder and one HlaRemoteCreate object are created in a typical Umbra application. During initialization, the Ambassador subscribes to the interactions of interest through the RTI. This ensures that the RTI listens for and passes on interactions to the Ambassador. (At this time, the Ambassador may also be commanded to request permission to publish the interactions.) Next, the modules are commanded to subscribe to the particular interactions through the Ambassador. This ensures that the Ambassador passes interactions to the individual modules.

Callback script names are provided to the interaction modules in their interaction subscription. Later, when an appropriate interaction is received, the modules call the named Tcl functions while also providing interaction data in a parsed form to the script. These scripts then typically execute the requisite function. The callback script provided to the HlaRemoteCreate module uses the data from the remote_create interaction to construct an entire robot meta-module. This includes creating, naming and initializing each module and connecting the modules into a network. Callback scripts provided to the HlaOrder module

typically use the data from the interaction to program the control modules of the indicated robot to execute particular commands. For example, Move orders include the name of the entity being commanded and route information. The callback script for Move orders uses the Tcl interface on the WayQueue module to load and execute the desired path. In addition, the script performs various error checking, error recovery, reporting, and bookkeeping functions.

5 CONCLUSION

Umbra is a highly modular framework for interactive simulation. It enables execution in C++ and scripting, simulation configuration, and interaction in Tcl/Tk with OpenGL-based visualization. Umbra naturally supports models that follow system structure through their organization into module networks known as meta-modules. Umbra's Multiple Worlds abstraction conceptually cuts across meta-module boundaries to enable modeling multiple interaction phenomena.

The Umbra framework and associated technologies are convenient for building HLA support. We developed the Umbra HLA interface library to integrate HLA federation capability into Umbra simulations supporting the US Army JPSD's JVB simulation framework for Objective Force concept analysis.

Future work should increase scale and broaden domains. Specifically, we would like to explore multiprocessing, cluster-based Umbra simulations, and apply Umbra to more complex systems, such as ones including embodied agents having human cognitive and perceptual models.

ACKNOWLEDGMENTS

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL8500. Portions of this work were sponsored by the U.S. Army Joint Precision Strike Demonstration (JPSD) Project Office.

The authors thank Maj. Rick Schwarz of US Army JPSD, Russell Richardson of SAIC, Rich Briggs of Virtual Technology Corporation, Ray Harrigan, and Terri Calton for their programmatic support and technical advice. Thanks also to David Hensinger and David Schoenwald for their comments.

REFERENCES

Anderson, R. A. 1997. Building a modular robot control system using passivity and scattering theory. *Proc. 1997 IEEE Int'l Conf. On Robotics and Automation*, Minneapolis, MN.

AVS Advanced Visual Systems. 2002. AVS.

Defense Modeling and Simulation Office (DMSO). 2001. High Level Architecture Run Time Infrastructure RTI 1.3-Next Generation Programmers Guide, Version 4. U. S. Department of Defense DMSO.

Diaz-Calderon, A., C. J. J. Paredis, and P. K. Khosla. 1998. A modular composable software architecture for the simulation of mechatronic systems. *Proceedings of DETC'98, ASME 18th Computers in Engineering Conference*, Atlanta, GA.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley.

Gottlieb, E. J., M. J. McDonald, and F. J. Oppel. 2002a. *The Umbra High Level Architecture Interface*, Sandia Technical Report SAND2002-0675.

Gottlieb, E. J., M. J. McDonald, F. J. Oppel, J. B. Rigdon, and P. G. Xavier. 2002b. *Umbra's Joint Virtual Battlespace (JVB) Models*, Sandia Technical Report SAND2002-xxxx. [Publication Pending].

Mathworks, The. 2002. Simulink.

Mechanical Dynamics. 2002. ADAMS.

National Instruments. 2002. LabVIEW.

Otter, M., H. Elmqvist, and S. E. Mattsson. 1999. Hybrid modeling in Modelica based on the synchronous data flow principle. *Proceedings CACSD*'99, Hawaii.

Singhal, S. and M. Zyda. 1999. *Networked Virtual Environments: Design and Implementation*, Reading, MA: Addison-Wesley.

Stewart, D. B., R. A. Volpe, and P. K. Khosla. 1997. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Transactions on Software Engineering*, 23(12):759-776.

AUTHOR BIOGRAPHIES

ERIC J. GOTTLIEB is a consultant for the Intelligent Systems and Robotics Center at Sandia National Laboratories. He recently joined Orion International Technologies.

MICHAEL J. McDONALD, FRED J. OPPEL, J. BRIAN RIGDON, and **PATRICK G. XAVIER** are respectively Principal, Principal, Senior, and Principal Members of the Technical Staff in the Intelligent Systems and Robotics Center at Sandia National Laboratories. They collectively have over five decades of experience in robotics and simulation.