Rapid synthesis of high-level architecture-based heterogeneous simulation: a model-based integration approach

Graham Hemingway, Himanshu Neema, Harmon Nine, Janos Sztipanovits and Gabor Karsai

SIMULATION published online 17 March 2011
DOI: 10.1177/0037549711401950

Simulation

Simulation: Transactions of the Society for Modeling and Simulation International
0(00) 1–16

DOI: 10.1177/0037549711401950
sim.sagepub.com

Rapid synthesis of high-level architecture-based heterogeneous simulation: a model-based integration approach

Graham Hemingway, Himanshu Neema, Harmon Nine, Janos Sztipanovits and Gabor Karsai

Abstract

Virtual evaluation of complex command and control concepts demands the use of heterogeneous simulation environments. Development challenges include how to integrate multiple simulation engines with varying semantics and how to integrate simulation models and manage the complex interactions between them. While existing simulation frameworks may provide many of the required run-time services needed to coordinate among multiple simulation engines, they lack an overarching integration approach that connects and relates the interoperability of heterogeneous domain models and their interactions. This paper outlines some of the challenges encountered in developing a command and control simulation environment and discusses our use of the Generic Modeling Environment tool suite to create a model-based integration approach that allows for rapid synthesis of complex high-level architecture-based simulation environments.

Keywords

domain-specific modeling language, distributed simulation, heterogeneous simulation, high-level architecture, model-based integration, multi-paradigm modeling

1. Introduction

The evaluation of emerging command and control (C2)[1] concepts necessitates a sophisticated modeling and simulation infrastructure that allows for the concurrent modeling, simulation and evaluation of (1) the C2 system architecture (advanced system-of-systems modeling), (2) the battle space environment (scenario modeling and generation), and the (3) human organizations and (group and individual) decision-making processes (human performance and man–machine interaction modeling). Using simulated C2 environments to evaluate design concepts, validate new systems and components, and explore hazardous, as well as ambiguous, scenarios is easily justified from both a cost and a practicality perspective. However, complex C2 environments have many disparate facets that need to be orchestrated, all of which cannot be handled by a single simulation engine. As a result, a heterogeneous collection of integrated simulations, all acting in a tightly coordinated environment, must be employed. This collection is referred to as a simulation federation.

In a federation, individual simulations are composed of two parts: an underlying simulation engine, such as Simulink, and a domain-specific model, such as an automatic flight control system, designed to run on the engine. In addition, each simulation engine may have its own unique execution semantics that need to be accounted for. All of the engines and models in a federation must be coordinated in a meaningful way in order for the larger C2 simulation environment to

Institute for Software Integrated Systems, Vanderbilt University, TN, USA.

Corresponding author:
Janos Sztipanovits, Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, USA
Email: sztipaj@isis.vanderbilt.edu

be useful. Issues encountered in developing these environments relate to integrating simulations at both the engine and model levels.

Existing frameworks, such as the high-level architecture (HLA),[2,3] provide application programming interfaces (APIs) that have helped to reduce the complexity of integrating multiple different simulation engines, but many challenges remain in such environments. HLA, for example, does not specify any tools to design or deploy a federation. It primarily standardizes run-time support for various tasks, such as coordinated time evolution, message passing, and shared object management. As a result, these frameworks require a significant amount of tedious and error-prone hand-developed integration code.

Additional complications stem from the fact that existing frameworks typically do not have centralized repositories for the configuration of the entire environment. Ideally, this repository would include configuration data for each of the individual simulation engines and models involved, as well as for the deployment and execution of the overall simulation. Lacking such a repository leads to the possibility of conflicting or inconsistent configurations. These types of problems can be difficult to diagnose and can greatly increase the burden of manually coordinating the design, deployment, and execution of the simulation environment.

In this paper, we present the C2 wind tunnel (C2WT), a graphical environment for designing and deploying heterogeneous C2 simulation federations. Its primary contribution is to facilitate the rapid development of 'integration models', and to utilize these models throughout the lifecycle of the simulated environment. An integration model defines all the interactions between federated models and captures other design intent, such as simulation engine-specific parameters and deployment information. This information can be leveraged to streamline and automate significant portions of the simulation lifecycle.

The C2WT uses a custom developed domain-specific modeling language (DSML) for the definition of integration models. This language facilitates the easy capture of all of the design details for the simulation environment. C2WT integration models follow the conceptual architecture depicted in Figure 1. A simulation environment is composed of multiple 'federates', each of which includes a simulation model, the engine upon which it executes, and some amount of specialized glue code to integrate the engine with the simulation bus. Both the engine configuration and the integration (or 'glue') code needed for each federate is highly dependent upon the role the federate plays in the environment, as well as the type of simulation engine being utilized.

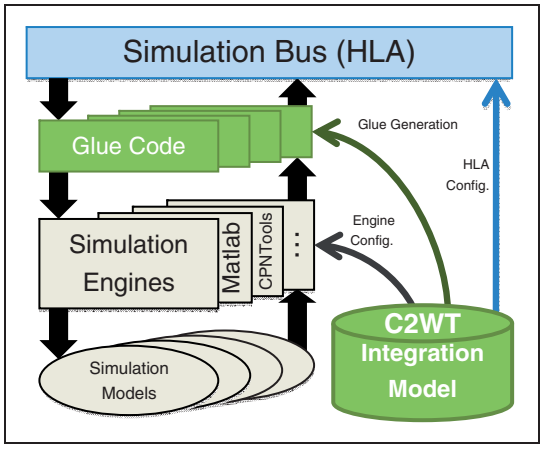

Figure 1. Conceptual command and control wind tunnel architecture. HLA: high-level architecture.

While manually developing the glue code is possible, by leveraging the integration model, the C2WT is able to synthesize all of the code, greatly reducing errors and effort. We developed a suite of tools, called model interpreters, integrated directly with the DSML, that automatically generate engine configurations and glue code, as well as scripts to automate simulation execution and data collection. The integration model DSML combined with our generation tools provides a robust environment for users to rapidly define complex, heterogeneous C2 simulations.

The remainder of this paper is organized as follows. The next section discusses background material. Section 3 details the C2WT DSML and our approach for model integration using the C2WT environment. Section 4 reviews the details of the integration of several simulation engines. Section 5 discusses our integrated approach to simulation deployment and execution. Section 6 covers results from using the framework in a real-world scenario. Section 7 discusses some of the decisions and factors that drove the tool suite's development. Section 8 reviews work related to our research. Section 9 concludes the paper and outlines planned future work.

2. Background information

One of the primary contributions of our effort is our focus on developing a completely model-based integration approach. Our efforts leverage the Generic Modeling Environment (GME)[4] tool suite for designing the integration model DSML and a customizable HLA integration and configuration framework to provide run-time support as the 'simulation bus'.

The HLA is a standardized architecture for distributed computer simulation systems. Communications between different federates is managed via the run-time

infrastructure (RTI) layer – an implementation of the HLA standard. The RTI provides a set of services, such as time management, data distribution, message passing, and ownership management. Other components of the HLA standard are the object model template (OMT) and the federate interface specification (FIS).

The HLA standard focuses on three primary areas. First is time coordination throughout the federation. The evolution of time is a key thread through each of the integrated simulators. Each simulation engine must slave its progression of time to that of the overall HLA clock. The HLA standard provides several methods to accomplish this. Second is coordination of inter-federate messages and shared data objects. The HLA standard provides a publish-and-subscribe mechanism for passing messages and object updates throughout the federation. Thirdly, the HLA standard provides for basic simulation execution control. A limited ability to start, pause, and stop the execution of a simulation is built directly into the HLA standard. The C2WT relies upon the services provided by the RTI during run time.

As HLA is an accepted standard, a number of commercial, academic, and alternate RTI implementations are available. Currently, we use the Portico RTI,[5] which provides support for both C++ and Java clients and is compliant with version 1.3 of the HLA standard.

The GME is a meta-programmable model-integrated computing (MIC)[4] toolkit that supports the creation of rich domain-specific modeling and program synthesis environments. Configuration is accomplished through meta models, expressed as unified modeling language (UML)[6] class diagrams, specifying the modeling paradigm of the application domain. Meta models characterize the abstract syntax of the DSML, defining which objects (i.e. boxes, connections, and attributes) are permissible in the language and how they compose. Another way to view this is that the meta model is a schema or data model for all of the possible models that can be expressed by a language. Using finite state machines as an example, the DSML would support states and transitions. From these elements any state machine can be realized. The inherent flexibility and extensibility of the GME via meta models make it an ideal foundation for the C2WT environment. Alternate meta modeling frameworks, such as AToM^3,[7] MetaCase,[8] and Microsoft DSL,[9] exist but extensive prior experience with the GME led to its selection for the C2WT.

3. Model-based integration environment

Complex C2 simulations require coordination between multiple heterogeneous simulation engines. The HLA provides a standard for the RTI that supports the coordinated execution of distributed simulations. However, designing the model integration, coding the engine-to-RTI glue code, and testing and deploying all of the various run-time components across multiple engine-specific simulation tools remains a challenging problem. Our project introduces a new approach for the simulation integration problem. The primary contributions of this effort are the development of a comprehensive modeling and management environment built around a custom DSML, implemented in the GME, and a related suite of model interpreters to coordinate between the integration model and the engine-specific simulation tools involved in the overall environment.

3.1. Integration overview

A common problem with developing large-scale heterogeneous simulations is the complexity and effort required to integrate distinct simulation tools into the larger environment. In the case of a HLA-based environment, not only does the RTI require a common federation definition, but each involved simulation tool must also be integrated (via simulation engine-to-RTI glue code) and configured (in an engine-specific way) according to its role in the environment. Existing approaches treat the definition and creation of these artifacts as separate, but not necessarily related, steps. Our custom DSML is able to capture all of this integration information and provide a single view of the entire simulation environment. In this section, we will discuss the design of the DSML, our approach for creating integration models, and the execution semantics of these models.

Throughout this paper we will refer to an example scenario we have modeled using the C2WT, illustrated in Figure 2. It involves the deployment of one or more unmanned aerial vehicles (UAVs) (simulated using Simulink[10] models) into a combat zone. The deployment zone and the physical positioning of the ground and aerial vehicles are modeled using a custom Java federate and visualized using Google Earth.[11] The UAVs may have objectives including: data collection, target acquisition and engagement, or battle damage assessment. Video sensors (implemented and simulated using custom-written Java federates) mounted on the UAVs must collect information and relay it via a communications network (implemented in OMNeT++[12]) back to a centralized decision-making organization (implemented as a colored Petri-net (CPN) model in CPN Tools[13]). The organization must react appropriately to the information and provide guidance to the vehicles. In addition, the UAVs are themselves highly autonomous and must utilize collected sensor data to pursue their given objectives.

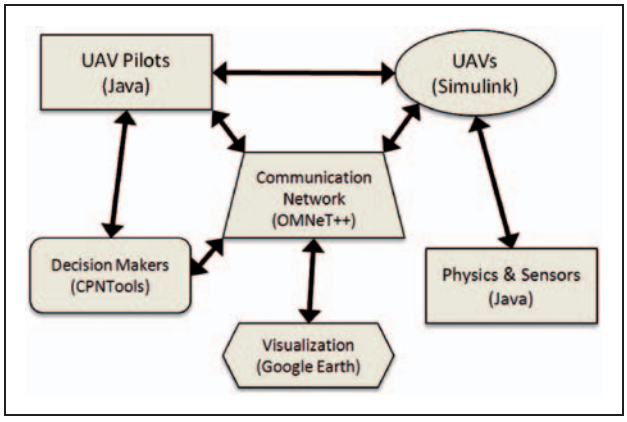

Figure 2. Example simulation. UAV: unmanned aerial vehicle.

3.2. Integration model DSML

Fundamental to the C2WT environment is the overarching model integration DSML. This language is considered overarching in the sense that it provides all of the modeling primitives required to specify the integration, deployment, and execution of the federated simulation. Once the integration model has been defined for a given environment, a set of reusable model interpreters are executed to automatically generate engine-specific glue code and all deployment and execution artifacts. All generation and deployment steps directly rely upon the initial integration model.

Figure 3 shows the primary portion of the DSML that defines the universe of composition elements. The three primary elements in a federation (defined by the model *FOMSheet*) are *Interaction* (on right-hand side of Figure 3), *Object* (on the left-hand side), and *Federate* (in the center), representing a HLA-interaction, HLA-object, and a HLA-federate respectively. Note that proxy elements are simply references to their respective target model elements and can be used in place of their targets to help structure or organize a model. As defined by the HLA standard, federates in a federation communicate among each other using HLA-interactions and HLA-objects – both of which are managed by the RTI.

Interactions and objects, in an analogy with interprocess communications in operating systems, correspond to message passing and shared memory respectively. As seen in Figure 3, the meta model fully supports the key HLA-defined attributes of these communication elements, such as delivery method, message order (timestamp or receive order), and parameters. Also notable is that via the *InteractionInheritance* and *ObjectInheritance* connection elements, interactions and objects can form inheritance trees where derived types inherit the parameters or attributes respectively of base types. The *ParameterType* attribute on the *Parameter* and *Attribute* elements defines the data type of that element (i.e. float, int, string). The *Interaction* and *Attribute* elements also support the HLA-defined attributes of *Delivery* and *Order*. The *Delivery* attribute specifies the desired method of delivering the interactions (and attribute updates), such as *reliable* or *best-effort*. The *Order* attribute specifies the order in which the interactions (and attribute updates) must be delivered, such as *time-stamped*, or *receive*. When *time-stamped* order is used, the RTI maintains a time-stamped queue of interactions (and attribute updates) and delivers them in that order. When the *receive* order is used, the RTI forwards the interactions (and attribute updates) as soon as it receives them without guaranteeing the order in which they will be finally received by the recipient federates.

The *Federate* element directly corresponds to any single instance of a simulation tool involved in the federation. The primary attribute of a federate, as far as HLA-based synchronization is concerned, is its *Lookahead* – the interval into the future during which that federate guarantees that it will not send an interaction or update an object.

We have sub-classed the *Federate* element with *CPNFederate* and *OmnetFederate* elements. These elements are used to represent two supported simulation engines that benefit from having more detailed federate models. *Places* and *EndPoints* represent contained elements within larger CPN Tool and OMNeT++ models, respectively. These 'children' elements appear as ports on their parent container, see Figure 4, and allow the federate to relate interactions or objects directly to the child elements. For many federate types, such as Matlab/Simulink or Java/C++, children elements do not have a semantic equivalent, and as such do not need specific support in the meta model.

Lastly, the attributes of the *FOMSheet* element capture the names and locations for configuration code that enables the integration of supported simulation engines. We will describe this capability in detail in Section 4.

Collectively, these language elements are required to define the relationships between all federate types. Developing an actual integration model using these special simulation elements is discussed in the subsequent section.

Figure 5 shows additional portions of the C2WT DSML. The top portion of the figure shows the language elements necessary to model federates publishing and subscribing interactions, objects, and attributes. A *Federate* (bottom center of the top portion) can be related to an interaction (inherited via the *InteractionBase* element – bottom right).

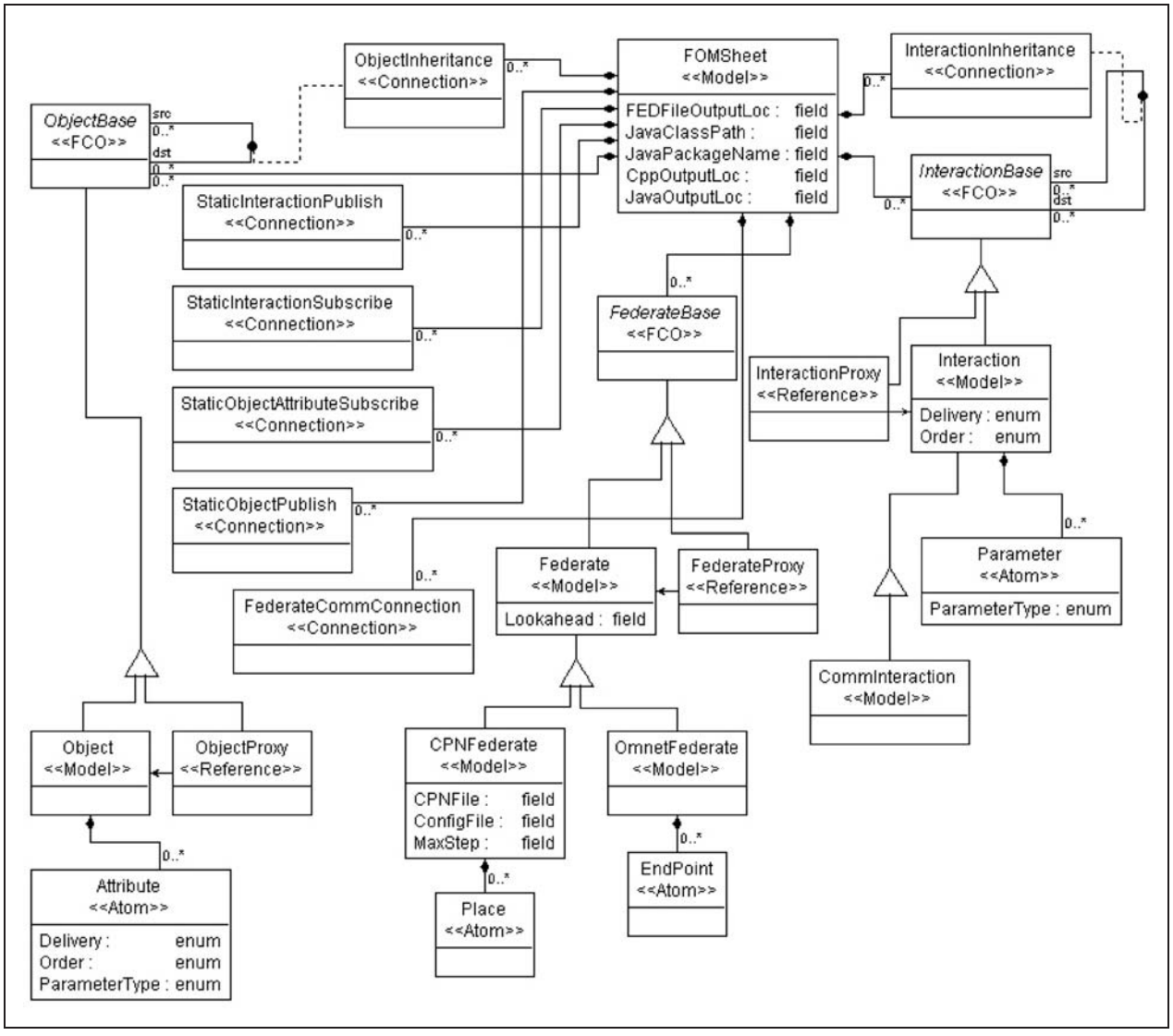

Figure 3. Integration model meta model.

Two relationships are possible, publish (via a *StaticInteractionPublish* connection – middle right) or subscribe (via a *StaticInteractionSubscribe* connection – top right). Similar relationships exist between objects (via the *ObjectBase* element) and federates. Since the HLA standard allows federates to subscribe to individual object attributes, the *Attribute* element supports the subscribe connection to federates. The *PubSubFilter* element provides a simple means to organize publish-and-subscribe relationships.

The lower two portions of Figure 5 are extensions specific to the CPN Tools and OMNeT++ engines integrated into the C2WT. The middle portion of the model captures publish-and-subscribe links with *Place* elements. Similarly, the model at the bottom of the figure captures the connection with special-purpose *EndPoint* elements for the integration of OMNeT++ federates.

This language, and its set of modeling elements, is very closely related to the HLA standard for future extensions to other RTIs in addition to the one currently supported (viz. Portico RTI[5]). With these elements a designer is able to completely specify the integration model of the entire federation and its constituent simulation engines. Federates define the details of the engine-specific models that are involved, and their relationships are captured via publishing and subscribing to various interactions and objects. Further extensions to the language have been implemented that include primitives for specifying

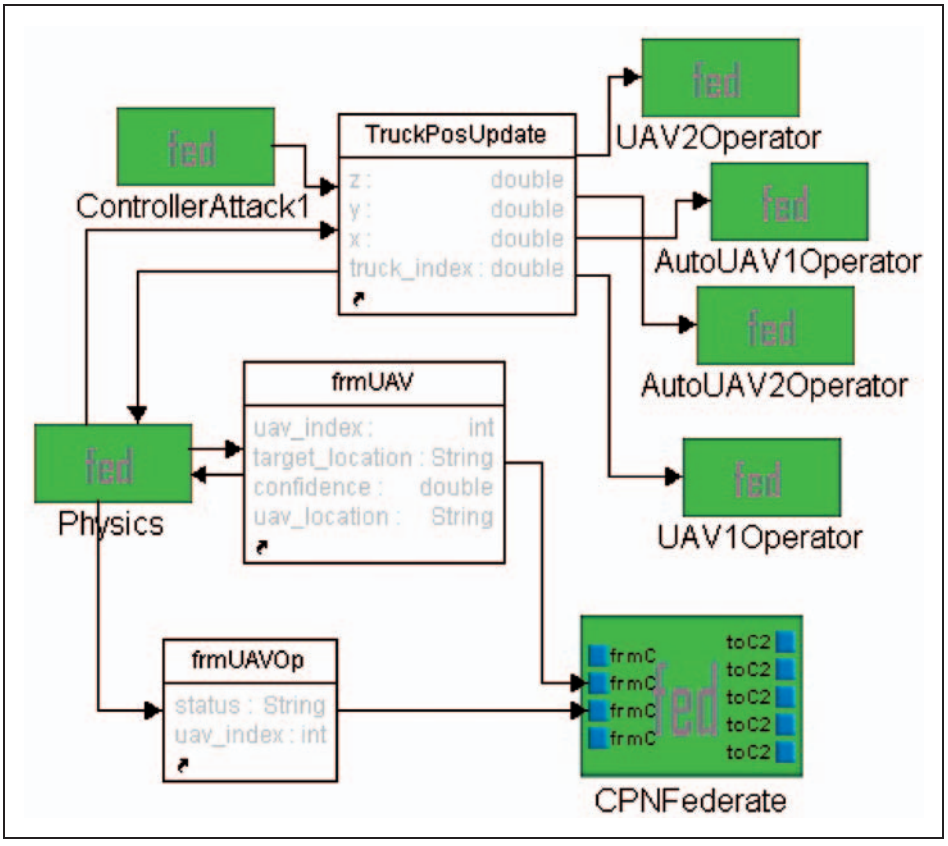

Figure 4. Example publish/subscribe relationships.

deployment and execution. These extensions are discussed in Section 5.

3.3. Creating integration models

The semantic relationship between federates can be defined primarily using two main aspects: the data representation and the data flow. These are common elements of most simulation modeling paradigms, and we have adopted these as the key concepts of our integration models. An integration model describes both data representation and data flow elements, and, in some cases, includes special elements as the placeholders for concepts specific to particular simulation engines.

Data representation models consist of interaction and object models. Interactions are stateless, and can have parameters, while objects have states, which are represented as a set of attributes. Both interactions and objects are permitted to have inheritance hierarchies. These data representation models directly map to the HLA federation object model (FOM).

Figure 6 shows two data representation models from our example: an interaction class-inheritance tree on the top (elements not shaded), and an object class-inheritance tree on the bottom (elements shaded). All interactions must initially inherit from the *InteractionRoot* element; likewise, all objects must inherit from the *ObjectRoot* element. While root inheritance is mandated in the HLA standard, we have directly incorporated this concept into the DSML to both clarify the visual representation and to simplify the interpretation of the model tree. Deriving elements via inheritance is an intuitive approach readily understood by modelers.

Once the data representation models are created, the modeler must define publish–subscribe data flow relations with federates. This is accomplished by connecting federates to interactions or object attributes with directional links. Federates publish and subscribe to any set of interactions or objects, dictated solely by the desired operational semantics. Federates can also publish or subscribe to entire data elements or to a subset of their attributes. Figure 4 shows a simple data flow from our example specifying the publish-and-subscribe relationships between federates (elements shaded) and interactions (elements not shaded).

Integrating engine-specific models together in the central modeling environment is simply a matter of connecting federates to those interactions and objects with which they have a publish or subscribe relationship. This greatly simplifies the designer's job, since they no longer need to directly incorporate

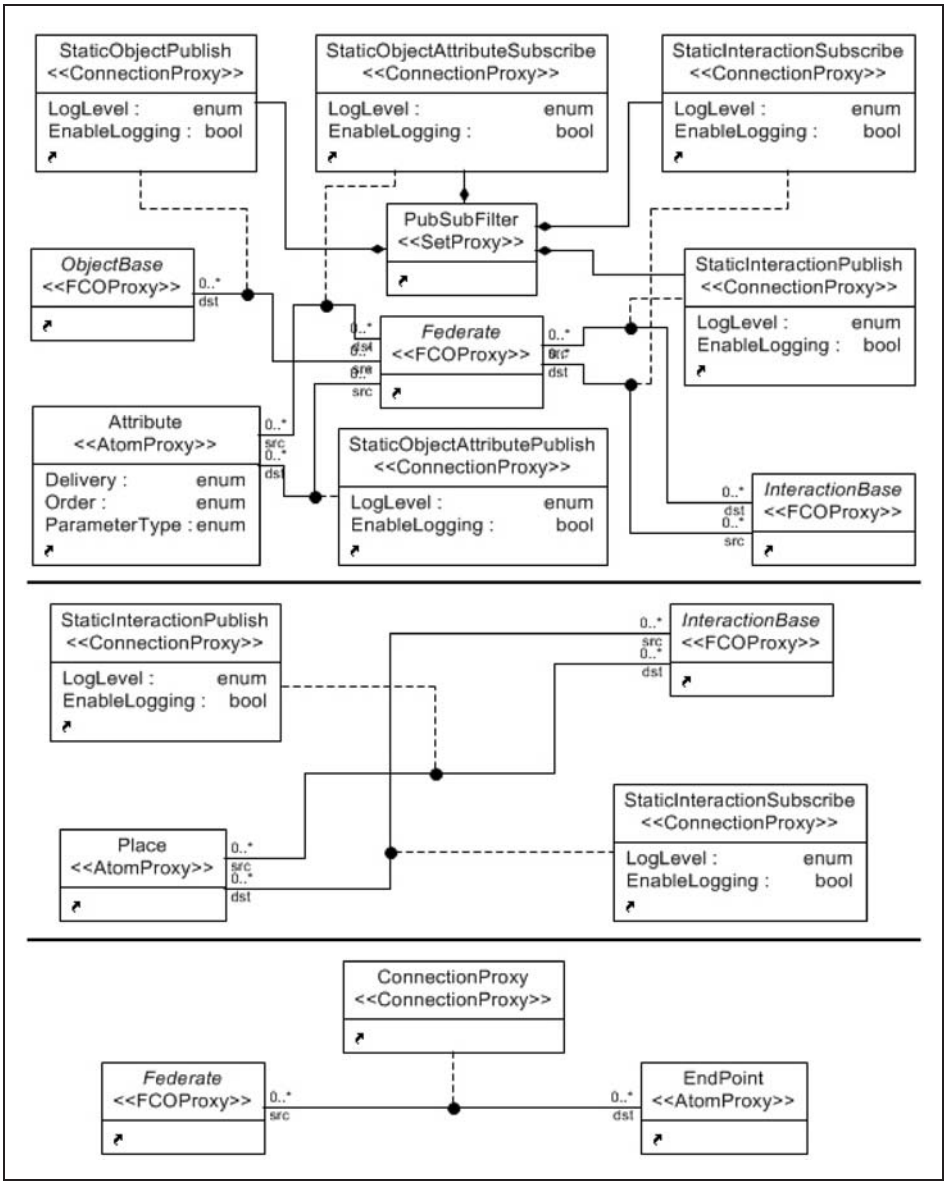

Figure 5. Command and control wind tunnel meta model continued.

engine-specific considerations and can focus solely on the high-level interactions of the model. The lower-level integration details, such as clock management and message passing, are addressed once and for all when the simulation engine is integrated into the general C2WT environment.

3.4. Federation execution semantics

The integration model defines all of the relationships between federates via publish-and-subscribe mechanisms on interactions and objects. We have purposefully not included any timing-related information into the integration model beyond the *Lookahead* parameter within each *Federate* object.

The HLA standard provides numerous schemes for coordinating time among federates. These can range from completely lacking time synchronization, where one federate can execute arbitrarily far into the future, to completely synchronized, where all federates evolve time within a tightly bound window. Our integration models always assume that time is strictly controlled. Using HLA terminology, all federates must be both 'time regulating' and 'time constrained'. A time-regulating federate's progression constrains all time-constrained federates. Likewise, a time-constrained federate's advance is controlled by all time-regulating federates. Because within the C2WT framework all federates are both time regulating and time constrained, time progresses only when all federates are ready to

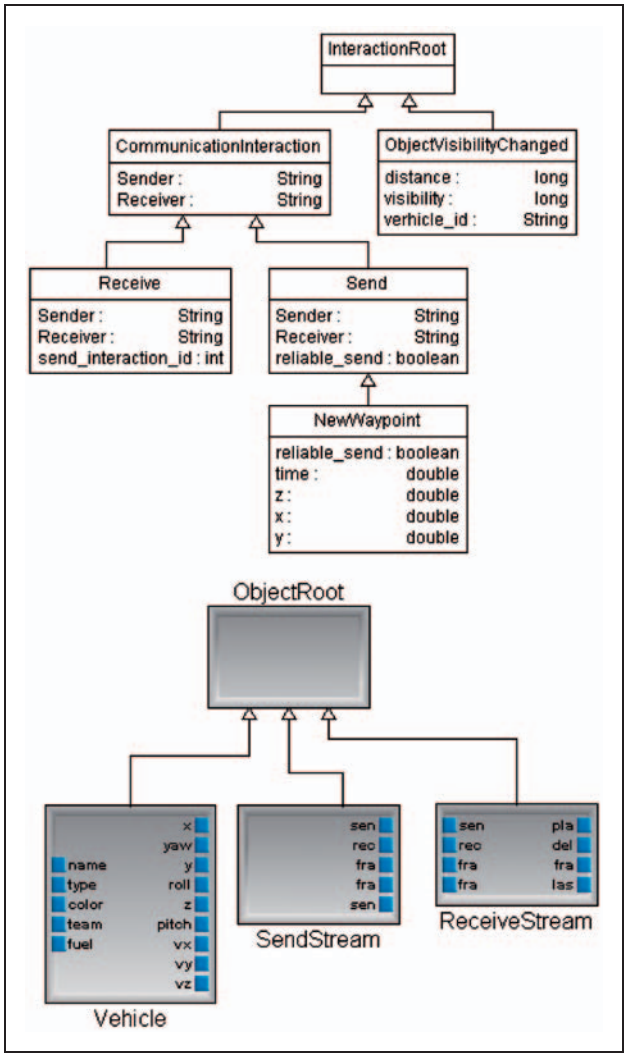

Figure 6. Interaction and object hierarchies.

proceed and then only as far as the smallest permissible time step across all of the federates. Without both characteristics for all federates, the overall behavior of the simulation can become non-deterministic due to reordering of events in time. Determinism is necessary if scenarios must be executed multiple times without variance in the events or outcomes, such as for scenario analysis. While our envisioned scenarios rely upon determinism, other uses, such as for training purposes, may not, and therefore the requirement for federates to be both time constrained and time regulating could be relaxed.

The C2WT environment also assumes that all interactions and object updates are strictly time ordered and must have timestamps. The HLA standard specifies that messages can be sent at any time but may only be received while the federate is waiting for a time advance request to be granted. This ensures that all incoming messages will have a timestamp greater than or equal to a federate's current time, that is, no timestamps are allowed on a message that make a message appear that it was received in the past. Once a time advance request is granted a federate can simulate forward in time and processes incoming messages according to their timestamp order.

Given these assumptions, the operational semantics of a federation become straightforward. Each federate operates in a loop consisting of two steps: request a time advance from the RTI and wait, receive a time advance grant from the RTI, and simulate up to that time. The glue code generated from the C2WT integration model must be able to control the simulation engine execution to abide by this scheme. In the next section, we will provide several examples of how various simulation engines are controlled.

A complete C2WT integration model does not contain information about the detailed execution of each federate. Nor does it replace in any way the internal operational semantics of any simulation engine. Every federate references an engine-specific model (e.g. our Simulink-based UAV model) and it is within this model that the details of the internal semantics of the federate are contained. The C2WT only builds upon the standard time management and message passing mechanisms provided by the underlying HLA infrastructure.

4. Integrating simulation engines

This section describes the process of integrating several example domain-specific simulation engines into the overall C2WT environment. For each engine, we outline how the engine aligns with the overall framework and the primary considerations involved in integration. The three example engines are OMNeT++, Matlab/Simulink, and CPN Tools. This set was selected as representative samples of the stereotypical types of engines that are incorporated into C2 simulations.

Each integrated simulation engine has its own unique underlying execution semantics, CPNs for CPN Tools, synchronous data flow for Simulink, discrete event systems for OMNeT++, etc. These execution semantics directly impact the approach of how an engine is integrated into the C2WT environment. Integration approach details, such as how an engine coordinates clock management or how it passes inter-simulation interaction events to the HLA, must be solved for each engine individually.

In addition to those simulation engines covered below, the C2WT environment currently supports DEVSJava,[14] Delta3D,[15] Google Earth, and C/C++ and Java-based custom federates. This set of supported engines allows for the simulation of quite complex and diverse scenarios.

4.1. OMNeT++ – communications federate

In a C2 simulation environment it is essential to model and simulate communication networks in order to study mission critical situations, such as network failures or attacks. After evaluating multiple public domain network simulators, OMNeT++ was selected as our network simulation engine. A primary advantage of OMNeT++ is its modular architecture, which allows for the event scheduler to be easily replaced – a requirement for HLA integration.

We developed a tool called *NetworkSim*, which is a HLA-compliant reusable communication network simulator based on OMNeT++. NetworkSim provides a set of high-level communication protocols (e.g. reliable send, streaming) while internally maintaining a faithful simulation of the full network stack. A key advantage of NetworkSim is its ability to utilize communications network models built using the standard OMNeT++ modeling tools. It simply handles the translation of messages from the RTI into appropriate network actions, and vice versa, and injects these messages onto the correct simulated network node. In addition to maintaining the underlying semantics of OMNeT++, this mechanism also serves to isolate general RTI traffic from traffic on the simulated network.

Each OMNeT++ model deployed onto NetworkSim must have some code synthesized for integration with the RTI. All OMNeT++ models are composed of connected nodes that form a communications network. When simulated via NetworkSim, some of these nodes are end points, responsible for passing messages between the RTI and the OMNeT++ engine. The code that implements these nodes must be generated.

A GME-based model interpreter traverses the C2WT integration model and generates the C++ code needed for end-point nodes within an OMNeT++ model. The integration model provides all of the information to understand, for a given OMNeT++ federate, which interactions may be sent or received and which objects attributes may be published or updated. As seen in the bottom center of Figure 3, an OMNeT++ federate contains a set of end-point atoms. In addition, as demonstrated in Figure 4, any federate may be related to a set of interactions and objects. The interpreter understands these relationships and synthesizes code for each end point in an OMNeT++ federate. The generated code builds upon the OMNeT++ API and is compiled directly into the model-independent NetworkSim tool.

In addition to inter-federate communication, evolution of the OMNeT++ internal simulation clock must also be synchronized with the RTI. NetworkSim includes a reusable class that extends the basic OMNeT++ scheduler. Figure 7 shows the key

```
cMessage *HLAScheduler::getNextEvent()
{
    cMessage *msg = sim->msgQueue.peekFirst();
    if (!msg)
        throw new cTerminationException(eENDEDOK);

    while( msg->arrivalTime() > rti->getTime() )
    {
        rti->advanceTime();
        msg = sim->msgQueue.peekFirst();
    }

    return msg;
}
```

Figure 7. OMNeT++ scheduler function.

scheduler function that implements RTI time synchronization. The function is called by OMNeT++ to determine the next event, originated either internally or externally. If the timestamp on the next message places it outside of the window of time granted by the RTI, then a time advance is requested using *rti->advanceTime( )*.

An internal dispatch mechanism routes all RTI interactions to the appropriate OMNeT++ protocol module, which interprets them and can schedule new internal OMNeT++ messages. A similar mechanism interprets and routes OMNeT++ messages bound for external dispatch into the RTI. Using these mechanisms, both the evolution of time and message passing within an OMNeT++ federate is tightly coordinated via the RTI with the federation.

4.2. Matlab/Simulink – plant and controller federate

Matlab/Simulink[10] is a widely used simulation environment for dynamic and embedded systems, such as communications, controls, and signal processing. It is a visual language that uses a set of pre-built block libraries for designing and controlling the simulation.

Integration of the Simulink simulation engine is similar to that of the OMNeT++ engine in that all of the engine-specific glue code is generated based on the overarching integration model. The GME-based model interpreter generates code that, in conjunction with several generic classes, is used to directly integrate any Simulink model with a C2WT federation. The generic classes are completely reusable Java code and provide all of the fundamental RTI integration requirements: providing interfaces for converting between Simulink types and RTI types, encapsulating interfacing with the RTI for initializing the federate, synchronizing the Simulink engine's simulation clock, and managing any publish-and-subscribe relationships with other federates. The automatically generated Java and Simulink files for the engine integration depend on this reusable code.

Within any given Simulink model the user must insert an S-function block (a visual block that calls some textual code, specified in a '.m' file, for execution) for each interaction or object to which the model either publishes or subscribes. It is via these blocks that the Simulink engine interacts with the remainder of the federation. The modeler must specify whether the block either publishes or subscribes an interaction or object. This is done by instantiating the corresponding *sender* or *receiver* S-function from those that were generated from the integration model. The modeler must also tell the S-function block which interaction or object it should call. This is done by passing the name of the interaction or object via a string parameter to the block. The naming convention of the. m files and of the parameters is standardized and easily derived from the primary C2WT model. Once the S-function blocks have been incorporated and their values set, no further manual steps are typically necessary to prepare the model to be integrated. Some effort may have to be spent to properly order the signals entering and exiting the S-function blocks so that they correspond to the attribute ordering of the corresponding RTI interaction.

The key mechanism for synchronizing the clock progression of the Simulink model with that of the RTI is the basic time-progression model for S-function blocks. During its execution, the Simulink engine consults each block in a model about when it can generate an output. With all S-function blocks, code must be supplied, via an implementation of the *mdlGetTimeOfNextVarHit( )* method, to respond to this request from the engine. For our integration, the synthesized integration code in an S-function block uses this method to synchronize the model with the RTI and allow simulation time within Simulink to progress only when the RTI allows it to proceed. Until the RTI allows federation time to progress, we do not return from the method call within the S-function block, thus not allowing the Simulink engine to progress. We keep the Simulink engine step size low (typically $\sim$0.1 seconds) to minimize any event timing errors due to the passing of input and output events between the Simulink model and the HLA. For incoming events, the glue code uses a polling scheme at every time step to check if the federate has received an input from the remainder of the federation.

General experience shows that very small step sizes in any Simulink model can lead to a significant slowdown in simulation speed. In the context of the C2WT, possible performance penalties due to having small step sizes must be weighed against minimizing timing errors due to overly large time steps. After thorough evaluation, we found that the performance penalty is negligible in comparison to the basic lock-stepped simulation we use for synchronizing federates.

4.3. CPN – organizational decision-making federate

A very important goal in C2 simulations is to evaluate the response of decision makers to the evolving situation. The C2WT environment integrates CPNs to model and simulate human decision-making organizations.

We use CPN Tools augmented with the BRITNeY[16] extension. This extension provides a low-level bridge between the native CPN Tools API and Java, which simplified integration with our Java-based RTI.

The primary challenge involved in integrating the CPN Tools engine into the C2 environment was correct time synchronization. In order to ensure that the CPN model execution stops at desired times one extra place and a transition, which is set to fire with a predefined frequency of 1 kHz, are added into a CPN model. The CPN Tools engine optimistically progresses ahead of the HLA clock, but when needed, it can be rolled back to a desired time. This save and restore functionality might be useful for increasing performance using an optimistically large step size and *lookahead*. However, with our experiments, we found that the performance penalty incurred by using the small step size and *lookahead* was negligible. Thus, we currently use a step size of $\sim$1 second and *lookahead* of $\sim$0.1 seconds for the CPN federate. Internally, while executing the CPN model via the BRITNeY Java library, the CPN clock moves forward 1 millisecond at a time. While time progresses internal to the CPN simulation, we compare its current time with the time granted by the RTI to the CPN federate. If the CPN cannot proceed in time, it requests the RTI to advance time and waits until it receives a time advance grant. While the small internal time step of CPN federates has the potential to generate significant HLA communications (if there was an interaction every time step for example), our experience has shown that typical CPN models do not need to interact with the HLA every time step and thus do not incur a significant overhead.

As illustrated in Figure 8, CPN models are imported into a C2WT integration model via an automated interpreter. Upon importing a CPN model, a *CPNFederate* element is created within the integration model and the CPN places become corresponding ports on this federate. The ports on this element can then be connected to either interactions or objects to specify inputs and outputs for the CPN model, as discussed in Section 3.2. This graphical step is the only integration effort necessary for CPN models. All of the engine-specific code to communicate via the RTI and to synchronize the CPN federate with the rest of the federation is automatically generated from the GME integration model.

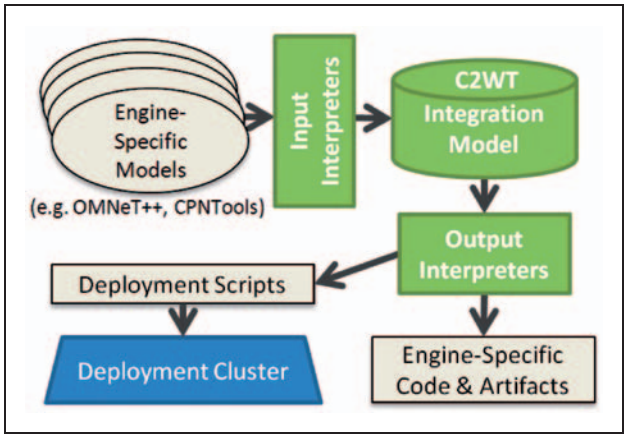

Figure 8. Integration model workflow.

A custom GME output interpreter generates an extensible markup language (XML) file that describes all of the input–output bindings. The run-time CPN execution engine reads this file and simulates the CPN according to its specification. The set of places to monitor during execution can also be specified. Tokens on these monitored places are shown during run time in a simple Java graphical user interface (GUI) provided by the C2WT framework.

5. Simulation deployment and execution

The deployment and execution of large-scale heterogeneous simulations can be quite complex. Typically, deployments span multiple computers and execution requires the coordination of many independent processes.

The HLA specification does not provide any mandate for how simulations are to be deployed or controlled, and available RTIs also do not provide such facilities. As C2 scenarios grow larger they must span multiple computers and the deployment of multi-domain simulations can impose a significant administrative burden. Manual approaches, such as hand-crafted batch files, tend to not scale well and are typically not well suited to highly dynamic environments where deployment parameters change frequently. Our team has made additional contributions in this area in order to ease the administrative burden imposed by these complex environments.

5.1. Deployment modeling

Our team encountered numerous deployment-related hurdles as we tried to execute scenarios built upon our environment. As the complexity of our scenarios grew, manual deployment processes quickly began to consume more time than the actual execution of scenarios. Our solution to this problem was to incorporate a model for deployment and execution directly into our central modeling environment. Figure 9 shows several additional elements that augment the C2WT meta model: *Experiment*, *Host*, *Computer*, *Network*, and *Deployment*. Now a single model incorporates both the federation integration design and the deployment information. With this extension, a model interpreter automatically generates all of the necessary scripts and files, copies the files to the appropriate computers, and prepares the environment for execution.

As discussed in previous sections, the overall simulation model is a composition of federates and their relationships via interactions and objects. For any given experiment, a simulation scenario may only utilize a subset of the federates defined in the model. Similarly, each engine-specific model involved may be parameterized to allow for run-time flexibility. Example parameters are the duration of a network attack (for the network simulator engine) or the weight a given command decision may be given (for the CPN Tools engine). An experiment is the set of federates included in a specific deployment and their run-time parameterization.

Frequently, an experiment is run on more than one hardware setup. A designer may run the entire simulation on one machine during development, while deploying the simulation onto a cluster for full-scale demonstrations. The network element of the language extension is the physical set of computers involved in a specific deployment.

The deployment element is where an experiment configuration is mapped to a network configuration. Specific federates are assigned to hosts in the network, thus allowing complete flexibility in defining which simulation tools execute on which hardware.

A model interpreter reads the deployment configuration from the model and generates all of the script files necessary to support the deployment. In cases where modeling deployments may only be partially specified, for example in very large-scale or rapidly changing environments, the interpreter generates the deployment for whatever portion is defined. Once generated, the environment is fully prepared for experiment execution.

Finally, the generated scripts manage the actual movement of files and code to the various hosts being used. Upon invocation, the scripts remotely connect to each machine and create local copies of all necessary files before the simulation begins execution. The scripts then coordinate the execution of all federates. After the experiment is concluded, the scripts remotely stop all processes, collect output files, and clean all local copies to restore the hardware to its original state.

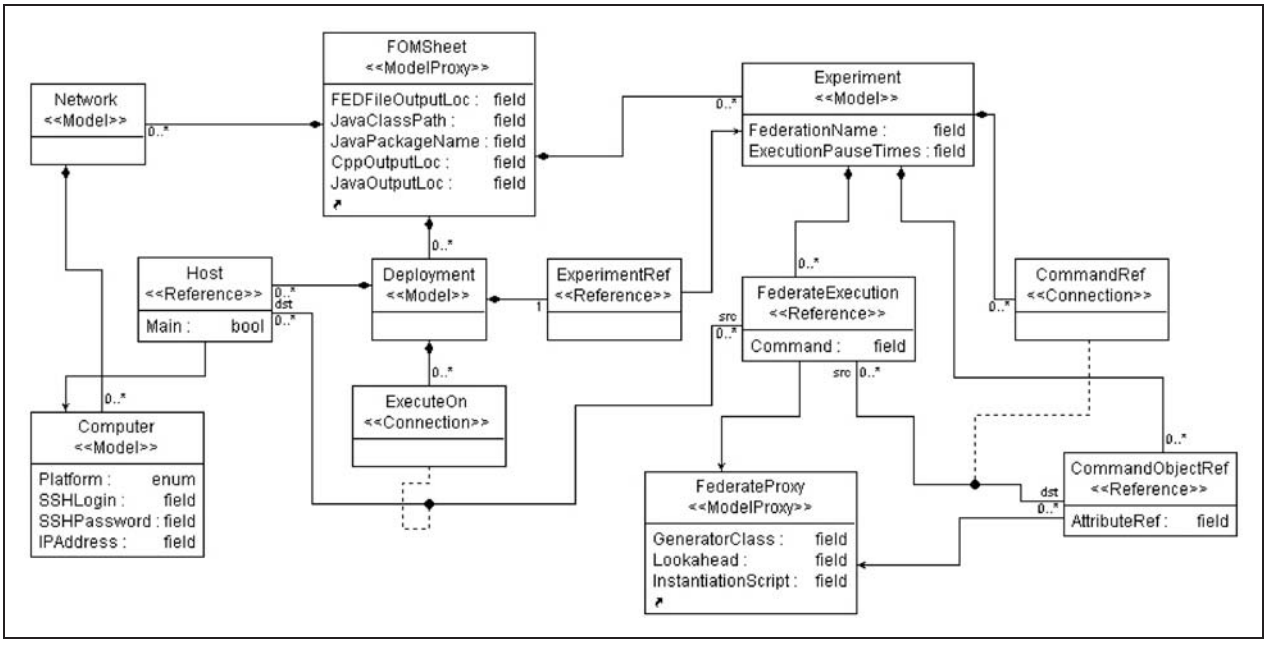

Figure 9. Command and control wind tunnel deployment and execution meta model.

5.2. Federation manager

The HLA standard prescribes basic methods for controlling the execution of a simulation: start, stop, and pause. However, the C2WT environment extends much greater control and coordination of federate execution throughout a simulation. This is achieved by using a special federate called the federation manager (FM).

The FM is a generic federate, and so can be used as a part of any federation. It coordinates a simulation by: (1) waiting for all federates in an experiment to join the federation before allowing it to begin simulation; and (2) making sure all federates are initialized and ready to begin the simulation before allowing it to proceed.

The first item above is achieved by listing which federates are part of the simulation in a configuration file that is read by the FM upon its initial execution. Using this information, along with the HLA's built-in 'FederateObject' class, the FM can detect when each federate joins the federation, and allow the simulation to proceed only when all federates have joined.

Making sure all federates are fully initialized is necessary to avoid one or more federates proceeding with simulation execution before others are ready. Such behavior can corrupt an execution and therefore invalidate the simulation. The FM uses 'synchronization points', as specified by the HLA standard, to guarantee that all federates are ready to proceed with the simulation before any of them begin execution. In particular, it registers a synchronization point to allow federates to report when they are initialized and ready to proceed with the simulation. Once all federates have reported that they have reached the synchronization point, the FM allows the simulation to proceed.

The coordination the FM exerts over a simulation is extremely important in that it allows simulations to be precisely repeated. Without the FM, for any sufficiently complex simulation, it would be nearly impossible to guarantee that all federates are always initialized and begin simulation simultaneously. The FM also allows the user to exercise greater control over the execution of the simulation. This is realized via several different mechanisms.

Firstly, the FM is capable of pacing the simulation in synchronization with the wall clock, or allowing the simulation to run as fast as possible. This is accomplished by coding the FM to monitor the wall clock and to use RTI calls to keep the wall clock and the simulation clock synchronized. The FM is time regulating and time constrained, similar to all other federates, and can therefore restrict or allow federation-wide time evolution through control of its own virtual clock. This behavior can be turned on and off using the FM's GUI. Turned off, the FM allows the simulation to proceed as fast as hardware and network speeds allow.

Secondly, the FM provides fine-grained controls to allow for the simulation to be paused for examination (and subsequently resumed), or terminated, at any point in the simulation. This works by placing 'Pause', 'Resume', and 'End' interactions directly into the integration model. The FM sends out 'Pause', 'Resume', and 'End' interactions either on demand

(via GUI buttons) or at times pre-specified in its configuration file. Each federate is prepared to respond to these interactions automatically by way of the automatically generated glue code.

Thirdly, the FM also allows federation-specific interactions to be injected into the simulation at pre-specified times. This is very useful for both debugging and quick 'what if' considerations. The FM is automatically configured to publish every interaction in the integration model. Interaction injections are controlled by specifying in the FM configuration file which interactions are to be injected with what parameter values and at what times. When the appointed time arrives the FM publishes the interaction to the rest of the federation.

Finally, the FM allows interactions to be monitored and logged as they are sent by federates during a simulation. This is also specified in the FM configuration file. Monitored interactions, as they occur, are displayed in a text box in the FM's GUI.

6. Experimentation with the framework

The C2WT project's goal is to create a heterogeneous simulation environment tailored towards scenarios involving, for instance, the interaction of multiple UAVs, their operating conditions, and the associated C2 organization and infrastructure. Rapid evolution of scenario details and easy evaluation of C2 effectiveness are key motivators.

Our first goal was to demonstrate that we could rapidly synthesize such simulations using our model-based integration environment involving all the domain-specific simulation engines described above. For demonstration purposes, we used an urban scenario with two 'good-guy' UAVs, and two 'bad-guy' ground vehicles that must be tracked in order to prevent an attack. Each UAV has video sensors and continuously transmits video data to the control station. The control station remotely controls the UAVs in a formation flight and assesses the targets one by one based on the initial position estimates of the targets.

All network communications are simulated using an OMNeT++ federate. We tested how a denial-of-service network attack affects mission performance. The abstraction of low-level physical layer communication in OMNeT++ makes it straightforward to implement various types of network attacks.

Using a collection of 'zombie' nodes in dedicated sub-networks, either parameterized or controlled by a master node, any type of 'dumb' or non-adaptive attacks can be simulated. For our purposes, a distributed denial-of-service (DDoS) attack was sufficiently disruptive. Each zombie machine sends, at specified intervals, service requests to every discovered valid node. As communications degrade, the effect on UAV formation flight is pronounced. The formation flight does not loosen: rather it collapses altogether. The UAVs continue in their individual directions, entirely uncoordinated. Sensor information is lost, so the operator becomes unable to repair the flight formation manually.

Table 1. Model information for experimental scenario

Federate	Engine	Update rate
UAV dynamics	Simulink	100 Hz
UAV operators	Simulink	100 Hz
Controller attack	Java	10 Hz
Physics	Java	10 Hz
Visualization	G. Earth	10 Hz
Comm. Network	OMNeT++	20 Hz
Decision making	CPN Tools	1 Hz

UAV: unmanned aerial vehicle.

The impact of these attacks on the C2 of the UAVs closely mirrors predicted theoretical consequences. This gives the experimenters confidence that the results of simulation can be directly applied to the modeled scenarios.

Table 1 captures the engine type and update rate for each of the federates involved in our simulation. Our hardware configuration used during demonstrations consists of six dual-core 3.0 GHz-class machines networked via a dedicated 100 Mb/s switch all with nVidia GTX 280 graphics cards. In a typical deployment, each machine had between one and three federates running on it. Despite highly complex engine-specific models, we have not experienced any significant performance bottlenecks during simulations lasting an average of 30 minutes. If our scenarios are deployed entirely onto one of our typical development machines (a clone of those in our demonstration cluster) performance is acceptable but noticeably slower. This is especially true if multiple visualization federates are present.

Using our framework, new scenarios can be created relatively quickly. It took our team less than three weeks to create a new scenario as compared to anecdotal evidence from our sponsors that, prior to our framework, individual scenarios could take between one and two years to develop. It is important to note that our scenario did not necessitate the integration of any new simulation engines and was able to leverage portions of existing custom Java federates.

7. Discussion

The design of the C2WT has evolved considerably over the three years it has been under development.

A primary driver of its evolution has been to see how a model-based approach can reduce the effort required to develop distributed simulations. Generally, with each iteration of the tools, previous functionality was refined and new automation was added. Our approach of refine and extend has allowed us to incrementally build a tool suite that would likely have been difficult to arrive at in a first attempt.

Our initial focus for the tool suite was on creating scenario models that captured the HLA configuration, namely the interaction and object hierarchies. Manually creating the federation configuration file (the. fed file) was tedious and error prone even for simple scenarios. With the core meta model in place, a very basic model interpreter was created to synthesize the. fed file. These tools made configuring the HLA very simple. At this point the largest burden in creating a scenario was integrating each simulation engine-specific model that would partake in the scenario.

The next iteration of the tool suite focused on integrating various simulation engines into the modeling environment. Integrating engine-specific models directly is not very reusable, so we instead focused on integrating as deeply as possible with the simulation engines themselves. Our first attempt at engine-specific integration led us to make numerous extensions to the meta model, namely sub-classing of the *Federate* element, as discussed in Section 3.1. The integration model interpreter was extended for each engine so that engine-specific glue code could be generated, as discussed in Section 4. Now the HLA configuration and all of the engine-specific code was being synthesized automatically from the integration model.

At this point, creating a new scenario was simpler but the repeatability of scenario execution was proving problematic. Our RTI did not offer tools to coordinate starting or stopping numerous federates. Depending on the order in which, and even the relative time at which, federates were started the scenario outcome would vary. We needed a means to allow arbitrary initialization of federates and then a coordinated means to start progression of the simulation. At the same time we needed a means of injecting interactions into a simulation at specific times. This functionality was needed to support debugging of individual engine-specific models and to take the place of models that were not yet ready to be incorporated into the federation. The FM was created to solve both of these problems. The integration model interpreter was again extended to synthesize a simple XML file that was used to configure the FM. With the FM in place, scenarios could be executed repeatedly with perfect consistency and we gained very flexible control over interaction injection.

The most recent iteration of the C2WT tool suite sought to alleviate a problem we had been experiencing since the project began. Individuals developing engine-specific models tended to run the entire federation on their local machine, while production runs of the federation were performed on a dedicated cluster of machines. Developers sacrificed absolute performance for the convenience of having the federation running locally. This approach worked well but developers were spending significant amounts of time manually creating scripts to run their local instance of the federation. Each developer's local setup was different, but they all relied upon a common scenario definition in a repository. A change in even a small scenario parameter often forced every developer to have to manually rework their scripts. It occurred to the team that if deployment configurations were directly incorporated into the scenario model then each developer could maintain a separate configuration but a common interpreter could generate all of the necessary deployment scripts. The core meta model was extended, as discussed in Section 5.1, and a new deployment-specific interpreter was developed. With these changes in place, the current C2WT automates many formerly manual processes and allows developers to focus on creating engine-specific models and the overall integration model.

8. Related work

There is prior research on the integration of multiple diverse simulation packages, both incorporating and not incorporating HLA. Multi-paradigm modeling (MPM)[17,18] addresses the methods, tools, and applications related to engineering problems involving the modeling of multiple different domains. Similar to MPM, the most challenging problem for distributed heterogeneous simulation is composing domain-specific models.

One of the previous efforts that relates to heterogeneous simulation, at our institute, is the MILAN framework.[19] MILAN provides multi-domain simulation of system-on-chip (SoC) designs and integrates several simulation tools in order to optimize performance versus power consumption. This approach is very SoC specific and is not a general engine for heterogeneous simulation. Co-simulation[20–22] also involves many similar aspects of heterogeneous simulation, although all specifically focused on hardware/software systems.

Other efforts exist that relate to integrating simulation packages via HLA, including OPNET,[23] MATLAB-HLA,[24] SLX,[25] JavaGPSS,[26] DEVSJAVA,[27,28] and PIOVRA.[29] As mentioned earlier, the HLA APIs provide run-time support but the problem of model integration is not addressed in these efforts.

Relevant commercial integration software does exist, such as the HLA Toolbox[30] for MATLAB federates by ForwardSim Inc.[31] and MATLAB and Simulink interfaces[32] to HLA and driver information system (DIS)-based distributed simulation environments by MÄK Technologies.[33] These products focus on integration of models running on the same simulation engine and do not provide support for heterogeneous simulations.

In addition, there have also been some efforts on enhancing HLA support by complementary simulation methodologies, such as in Sarjoughian and Zeigler[34] and Lu et al.[35] However, these efforts, similar to those above, pursue HLA integration of isolated simulation tools. Moreover, these efforts, except MILAN, do not have any support for model-based rapid integration of simulation tools, and limited, or no, support for automated deployment and execution of the resulting simulation environment. All of these are facilities the C2WT environment natively provides.

9. Conclusion and future work

Integration of complex C2 simulations composed of numerous heterogeneous engines is a challenging problem. Each simulation engine may have its own operational semantics and requires integration at not only the engine level but also at the engine-specific model level.

Pervasive use of models throughout simulation engines opens the door to the use of model-integrated methodologies for defining integration among these tools. In this paper, we discussed the use of a custom DSML-driven modeling environment for heterogeneous simulation. In this environment, it is possible to rapidly integrate domain-specific models from diverse simulation engines and to dynamically generate all of the needed configuration and integration code. The environment also provides automated facilities to manage the deployment and execution of the simulation itself. Together these tools greatly reduce the time required to design, modify, and test C2 scenarios.

In the future, we expect to integrate additional simulation engines to expand the range of possible scenarios. Consequently, new simulation engines will require expanding the existing integration DSML and infrastructure. These enhancements should allow for greater scenario flexibility and reduced development, configuration, and operational costs. In addition, we are exploring alternatives to include capabilities in the C2WT framework to alter and configure the entire simulation during run time, and to capture and analyze additional performance statistics during execution. The public distribution of the C2WT code is available at http://wiki.isis.vanderbilt.edu/OpenC2WT.

Acknowledgements

The authors acknowledge the invaluable contributions to this project from our collaborators: Alexander Levis and Lee Wagenhals' group at George Mason University, Claire Tomlin's and Shankar Sastry's groups at the University of California at Berkeley and Jonathan Sprinkle's group at the University of Arizona.

Funding

This work was supported by the US Air Force Office of Scientific Research under the project 'Human Centric Design Environments for Command and Control Systems: The C2 Wind Tunnel' (contract number FA9550-06-1-0267).

Conflict of interest statement

None declared.

References

1. Alberts DS and Hayes RE. *Understanding command and control*. CCRP Publication Series, 2006.
2. HLA standard. 'IEEE standard for modeling and simulation (M&S) high-level architecture (HLA) – framework and rules', http://ieeexplore.ieee.org/servlet/opac?punumber=7179.
3. Dahmann JS, Fujimoto RM and Weatherly RM. The Department of Defense high level architecture. In: *Proceedings of the 1997 Winter Simulation Conference*, 1997, pp.142–149.
4. Sztipanovits J and Karsai G. Model-integrated computing. *IEEE Comput* 1997; 30: 110–112.
5. Portico RTI. http://www.porticoproject.org.
6. UML – Unified Modeling Language. 'The Object Management Group', http://www.uml.org.
7. de Laura J and Vangheluwe H. AToM^3: a tool for multi-formalism and meta-modeling. *Lect Notes Comput Sci* 2002; 2306: 174–188.
8. Tolvanen JP and Lyytinen K. Flexible method adaptation in CASE. The metamodeling approach. *Scand J Inf Sci* 1993; 5: 71–77.
9. Cook S, Jones G, Kent S and Wills A. *Domain-specific development with visual studio DSL tools*. Addison-Wesley Professional, 2007.
10. Matlab/Simulink. 'Get Started with MATLAB', http://www.mathworks.com.
11. Google Earth. http://earth.google.com.
12. OMNeT++. http://www.omnetpp.org.
13. CPN Tools. http://cpntools.org.
14. DEVSJAVA. http://devs-suitesim.sf.net.
15. Delta3D. http://www.delta3d.org.
16. Westergaard M. The BRITNeY Suite: A Platform for Experiments. In: *Proceedings of 7th Workshop on Practical Use of Coloured Petri Nets and the CPN Tools*, 2006.
17. Vangheluwe H, de Lara J and Mosterman P. An introduction to multi-paradigm modeling and simulation. In: *Proceedings of AIS2002*, 2002, pp.9–20.

18. Mosterman J and Vangheluwe H. Computer automated multi-paradigm modeling: an introduction. *Simulation* 2004; 80: 433.
19. Agrawal A, Bakshi A, Davis J, Eames B, Ledeczi A, Mohanty S, et al. MILAN: a model based integrated simulation. In: *Proceedings of the 2001 ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems*, 2001, pp.82–93.
20. Hoffman A, Kogel T and Meyr H. A Framework for fast hardware-software co-simulation. In: *Proceedings of the Conference on Design, Automation and Test in Europe*, 2001, pp.760–765.
21. Rowson J. Hardware/software co-simulation. In: *Proceedings of the Design Automation Conference*, 1994, pp.439–440.
22. Becker D, Singh RK and Tell SG. An engineering environment for hardware/software co-simulation. In: *Proceedings of the 29th ACM/IEEE Conference on Design Automation*, 1992, pp.129–134.
23. Zhang F and Huang B. HLA-based network simulation for interactive communication system. In: *Proceedings of the First Asia International Conference on Modelling & Simulation*, 2007, pp.177–180.
24. Pawletta S, Drewelow W and Pawletta T. HLA-based simulation within an interactive engineering environment. In: *Proceedings of the Fourth IEEE International Workshop on Distributed Simulation and Real-Time Applications (DS-RT'00)*, 2000, p.97.
25. Henriksen JO. SLX: The X is for extensibility. In: *Proceedings of the 32nd Conference on Winter Simulation (WSC '00)*, 2000, pp.183–190.
26. Klein U, Straburger S and Beikrich J. Distributed Simulation with JavaGPSS based on the high level architecture. In: *Proceedings of the 1998 International Conference on Web-based Modeling and Simulation*, 1998, pp.85–90.
27. Zeigler BP and Lee JS. Theory of quantized systems: formal basis for DEVS/HLA distributed simulation environment. In: *Enabling Technology for Simulation Science (II), SPIE AeoroSense*. Vol. 3369: 1998, pp.49–58.
28. Zeigler BP, Ball G, Cho HJ, Lee JS and Sar-joughian H. Implementation of the DEVS formalism over the HLA/RTI: problems and solutions. In: *Spring Simulation Interoperability Workshop (SIW)*, 1999, SISO.
29. Zacharewicz G, Frydman C and Giambiasi N. Mapping PIOVRA in GDEVS/HLA environment. In: *Proceedings of Summer Simulation Multiconference (SummerSim'07), The Society for Modeling and Simulation International*, 2007, pp.1086–1093.
30. HLA Toolbox for developing and executing HLA federates from MATLAB. http://www.mathworks.com/products/connections/product_main.html?prod_id=696&prod_name=HLA%20Toolbox.
31. ForwardSim Inc. http://www.forwardsim.com/.
32. MATLAB and Simulink interfaces to HLA and DIS based distributed simulation environments. http://www.mathworks.com/products/connections/product_main.html?prod_id=696&prod_name=HLA%20Toolbox.
33. MÄK Technologies. http://www.mak.com/.
34. Sarjoughian HS and Zeigler BP. DEVS and HLA: complementary paradigms for modeling and simulation. *Trans Soc Comput Simulat Int* 2000; 17: 187–197.
35. Lu T, Lee C, Hsia W and Lin M. Supporting large-scale distributed simulation using HLA. *ACM Trans Model Comput Simulat* 2000; 10: 268–294.

Graham Hemingway is a graduate student at the Institute of Software Integrated Systems, Vanderbilt University, USA.

Himanshu Neema is a staff engineer at the Institute of Software Integrated Systems, Vanderbilt University, USA.

Harmon Nine is a senior staff engineer at the Institute of Software Integrated Systems, Vanderbilt University, USA.

Janos Sztipanovits is the E. Bronson Ingram Distinguished Professor of Engineering and director of the Institute of Software Integrated Systems, Vanderbilt University, USA.

Gabor Karsai is Professor of Electrical Engineering and Computer Science and associate director of the Institute of Software Integrated Systems, Vanderbilt University, USA.