A Gateway to Easily Integrate Simulation Platforms for Co-Simulation of Cyber-Physical Systems

Thomas Roth and Martin Burns
Smart Grid and Cyber-Physical Systems Program Office
National Institute of Standards and Technology
Gaithersburg, Maryland

Abstract—Cyber-physical systems (CPS) research leverages the expertise of researchers from multiple domains to engineer complex systems of interacting physical and computational components. An approach called co-simulation is often used in CPS conceptual design to integrate the specialized tools and simulators from each of these domains into a joint simulation for the evaluation of design decisions. Many co-simulation platforms are being developed to expedite CPS conceptualization and realization, but most use intrusive modeling and communication libraries that require researchers to either abandon their existing models or spend considerable effort to integrate them into the platform. A significant number of these co-simulation platforms use the High Level Architecture (HLA) standard that provides a rich set of services to facilitate distributed simulation. This paper introduces a simple gateway that can be readily implemented without co-simulation expertise to adapt existing models and research infrastructure for use in HLA. An open-source implementation of the gateway has been developed for the National Institute of Standards and Technology (NIST) co-simulation platform called the Universal CPS Environment for Federation (UCEF).

Keywords—co-simulation; cyber-physical systems; functional mock-up interface; high level architecture

I. INTRODUCTION

Cyber-physical systems (CPS) consist of interconnected devices that use logical computation to actuate changes on the physical world based on measurements of their environment. These systems lead to improved quality of life and technological advances in critical infrastructures such as personalized health care, smart manufacturing, autonomous vehicles, and the smart grid [1]. A strong assurance case must be made to validate each CPS prior to its deployment because failure has the potential for severe physical consequences that will impact humans and society. However, due to the scale and cost of these systems, it is impractical to prototype each design decision and validation relies heavily on computer simulations. An accurate simulation of a CPS requires the integration of simulators from multiple domains. For example, the common approach to evaluate new control algorithms for the smart grid is to integrate a network simulator with a power system simulator to enable researchers to measure the impact of control algorithms on the physical grid under realistic network conditions [2]. This integration of multiple simulators into a joint simulation is a technique called co-simulation.

IEEE 1516-2010 High Level Architecture (HLA) is one standard for the co-simulation of distributed processes [3]. HLA defines the heterogenous, domain-specific simulators as federates, and the collection of federates that comprise the joint simulation as a federation. The federates communicate and coordinate using software called the runtime infrastructure (RTI) that implements the common set of services described in the HLA federate interface specification. The most useful service that HLA provides is an abstraction of logical time that allows for time synchronization between simulators with different time semantics.

HLA is a comprehensive standard for distributed simulation that defines a rich application programming interface (API) containing over 168 services [4]. However, HLA is difficult to both learn and use because a conscious decision must be made on which HLA services to implement for each federate. This usability problem is compounded in CPS where often decades of research have gone into models and simulations of various sectors such as smart grid in well-established domain-specific testbeds. Almost all this research infrastructure was built without the vision of a co-simulation standard such as HLA, and it is difficult to imagine these testbeds are readily compatible with the HLA federate interface specification.

This paper introduces a gateway federate that defines a reduced API to perform the minimum services required to function as a HLA federate. It retains the time synchronization and data exchange services from HLA while reducing the API from 168 services to under a dozen methods. An open-source implementation of the gateway was developed in Java using the Portico1 RTI, but the architecture described in this paper is applicable to other languages and for other RTI. Using this gateway federate, a researcher can readily integrate their existing research infrastructure into HLA for co-simulation of complex CPS with little to no knowledge of the standard.

Section II provides an overview of other co-simulation methodologies and the related work in this area. Section III discusses the gateway federate architecture, and Section IV summarizes which HLA services the gateway federate implements. The gateway life cycle is presented in Section V, and the results of the work are concluded in Section VI.

1 Portico available at https://github.com/openlvc/portico
II. RELATED WORK

There are two popular co-simulation standards: HLA and Functional Mock-up Interface (FMI) [5]. Unlike HLA where federates are standalone processes that are peers in distributed system, FMI uses a master-slave architecture where the master algorithm imports each simulator as a shared library and makes direct function calls into the slave code. A simulation library in FMI is called a Functional Mock-up Unit (FMU). Each FMU is stored in a ZIP archive that contains either source code or a compiled shared library, an XML file that describes the model inputs and outputs, and other documentation. The FMI standard for co-simulation prescribes the function definitions that each slave must implement to be interoperable with the master. The advantages of FMI over HLA are that it has an intuitive API and FMU are not dependent on a specific RTI implementation. However, FMI is less comprehensive than HLA and is more difficult to use in distributed simulations that require multiple networked computers.

One challenge with using FMI for CPS co-simulation is that it does not provide a re-usable implementation of the master algorithm. A small body of research has investigated how to bridge the two standards by using HLA as the FMI master algorithm, which is equivalent to a federation of FMU federates [6] [7] [8]. This integration makes FMI more viable for CPS co-simulation since the master algorithm would not change between different sectors such as transportation and smart grid. However, FMI still uses shared libraries for the slaves in its architecture and it remains a challenge to use FMI for experiments that involve processes distributed across multiple computers. The distributed process approach of HLA is easier to use for the co-simulation of CPS which can require multiple networked computers for the cases of hardware-in-the-loop simulations and federation between remote testbeds.

There are several ongoing development efforts working on environments that use a combination of graphical domain-specific languages and code generation to ease the production of HLA federates. Vanderbilt University has evolved the Command and Control Wind Tunnel (C2WT) over several major releases to aid in federate development for Java, C++, and multiple simulation engines using a graphical language designed in their Generic Modeling Environment (GME) [9]. The National Institute of Standards and Technology (NIST) released an image of a virtual machine for a variant of C2WT called the Universal CPS Environment for Federation (UCEF) [10]. A parallel effort at the University of Rome Tor Vergata uses an HLA profile for the Systems Modeling Language (SysML) to go from requirements analysis to co-simulation in a single modeling environment [11]. A more recent effort uses an Eclipse plug-in to design HLA federates in the Parallel Object-Oriented Specification Language (POO SL) [12]. It is worth noting that each of these approaches uses a different RTI implementation and federates produced from one environment will be incompatible with the others. All these platforms focus on the creation of new federates for their supported simulation engines, and none of them provide guidance on how to integrate new simulation engines or research infrastructure into the workflow.

The HLA Development Kit takes a different approach and provides a Java library with a simplified API rather than using a modeling language to ease the development of federates [13]. It uses Java annotations in classes that resemble JavaBeans to define the data structures used for data exchange, and contains libraries and documentation useful for the development of Java federates. However, the requirement to implement classes for each message type is too restrictive for a gateway federate that might not be aware of the federation data model until runtime. This paper adopts a similar approach to the HLA Development Kit and defines the interface and life cycle for a gateway federate made available as a Java library to ease the integration of existing infrastructure into HLA. The gateway described in this paper exposes a small API that resembles a FMU from the FMI standard for co-simulation, but retains the richness and flexible architecture of HLA.

III. GATEWAY ARCHITECTURE

This section describes the architecture of a HLA gateway that exposes an existing model as a federate with an open-source implementation for UCEF. The UCEF Gateway is written in Java for Portico and as such is not available in other languages or for other RTI. However, this paper mentions few implementation-specific details and the presented architecture could be used for the design for another gateway in a different language for a different RTI. The gateway federate was designed to satisfy the following set of requirements:

- **Usable without HLA expertise**: a significant barrier to the use of HLA for co-simulation is that the standard is complex with limited documentation. A gateway federate must not presume knowledge of the HLA standard to be useful for researchers.

- **Easy to integrate new things**: a significant gap in the current approaches towards accessible co-simulation is the assumption that all federates will be created within a dedicated platform. This assumption is not true for CPS where significant testbeds already exist in multiple domains such as smart grid that were developed without using co-simulation platforms. A gateway must be able to integrate existing research infrastructure into HLA with minimal effort.

- **Agnostic to the federation data model**: several gateway applications such as protocol conversions are independent of the data exchanged at runtime. A gateway must not need to be recompiled each time there is a change in the data model that doesn’t affect the logic of its execution.

An instance of the gateway federate is a single-threaded Java application that consists of two parts: the gateway library that defines the program control flow and interacts with the RTI, and a user application that implements the set of callback methods defined in the gateway library. Each user application must determine how best to implement these callbacks to interact with the simulator, model, or research infrastructure.

---

2 UCEF Gateway available at https://github.com/usnistgov/ucef-gateway
that should be integrated as a HLA federate. The gateway library yields control to the user application during specific points in the federate life cycle by invoking the callbacks, during which the user application uses the gateway library to perform basic HLA services. Fig. 1 shows the structure of the gateway federate where the shaded areas represent the user application.

The lower dotted line in Fig. 1 represents the service set defined in the HLA standard, while the upper dotted line represents the reduced number of services that are exposed to the user application. Section IV discusses the services exposed by the gateway library, almost all of which relate to the data exchange within a federation. However, although the gateway library eliminates the need to consider most HLA services, the gateway federate is still HLA compliant and can participate in co-simulations with federates produced from the platforms described in Section II. Also, while a developer is shielded from the complexity of the HLA standard when using the gateway library, the user application could still choose to implement the other HLA services if the additional complexity is desired. This architecture satisfies the first requirement that the gateway be usable without extensive knowledge of the HLA standard because the user application only needs to touch the well-documented gateway library.

The minimum requirement for a user application is the implementation of the callback methods—6 in total—defined in the gateway library. These callback implementations can be as complex or simple as required to connect the new hardware or simulation with the gateway library to turn it into a HLA federate. The list of callback methods, as well as when each will be invoked during the gateway life cycle, are discussed in Section V. This addresses the second requirement on ease of integration because there are only 6 integration points where the hardware or simulation interacts with the HLA federation.

To satisfy the third requirement on independence from the data exchanged at runtime, the message structures used in the gateway library are represented entirely by string name-value pairs. A HLA message in the gateway library is structured as a string identifier and a map that points the property names to their byte array values. Data conversion, when necessary, must be performed in the user application through queries to the gateway library about the current object model. Several gateway applications are simple protocol conversions from the HLA message structure to another format such as a JSON string used in a representational state transfer (REST) API. This design decision allows the gateway federate to be data driven through configuration files as opposed to instance-specific compiled code.

This architecture is supported by two configuration files. A JSON configuration file is used to set the basic configuration options related to how to join the HLA federation and how to progress logical time. An extensible markup language (XML) configuration file specifies the structure for the messages that the gateway plans to send and receive during runtime. This XML file is compliant with a format prescribed by the HLA standard called the Federate Object Model (FOM), and is loaded into the gateway federate memory at runtime. Once a gateway federate has been developed through implementation of a specific user application, the same executable can be reused without code modification across multiple federations solely through modifications of these two text files.

IV. GATEWAY SERVICES

This section is a primer on HLA that describes the services relevant to the gateway federate. The HLA federate interface specification defines a set of services that are categorized into service categories. The Federation Management services relate to the creation, destruction, and synchronization of federation executions. The Declaration Management services declare publications and subscriptions to different HLA message types, while the Object Management services send and receive specific message instances. The Ownership Management services transfer the right to update data from one federate to another, and the Time Management services relate to the progression of logical time at individual federates. The Data Distribution Management services apply filters to restrict who receives specific messages to reduce the amount of network traffic in the federation. The remainder of this section will describe the services implemented by the gateway federate.

A. Federation Management

A federation is a named group of federates that can be joined using a specific network address. Federates can join and leave an existing federation at any time during its execution, and all joined federates have access to the same set of HLA services. However, one federate must choose to create the federation, and only this federate starts with the permission to delete the federation. The group of federates in the federation execution can be synchronized at named points along the logical time axis called synchronization points. Whenever a federate tries to synchronize on a registered synchronization point, its execution is blocked until all federates in the federation reach the same synchronization state.
The gateway federate has a limited role in the federation management services. It assumes the existence of a single federate designated as the federation manager whose responsibilities include creation of the federation, registration of the critical synchronization points, and determining when an experiment starts by being the last federate to synchronize. As such, the gateway federate cannot create federations and has a largely reactive role in federation management. It assumes the existence of three synchronization points, and will not run unless these synchronization points are registered by the federation manager. The services for joining, leaving, and synchronization are automated by the gateway library, and the user application can choose to ignore the federation management services.

B. Declaration Management

A federation uses the publish-subscribe message pattern for data exchange between federates. Each federate must declare its interests in terms of publications and subscriptions before it can send or receive data. These publications and subscriptions are restricted to the message classes in the current object model of the federation. The gateway federate requires an XML configuration file that describes its publication and subscription interests in the format defined by the HLA Object Model Template (OMT). At initialization, the gateway library will parse this configuration file to automate the declaration management service calls based on which message classes are annotated with publish and subscribe. The gateway configuration will not be passed to the federation for dynamic configuration of the object model, and the gateway federate assumes that the federation was created to include the message classes defined in its configuration file. The user application has no role in declaration management other than loading the gateway library with a configuration file that has the desired publication and subscription interests.

C. Object Management

HLA has two primitive message types: object classes, and interaction classes. An object is a persistent entity that contains one or more variables that change with time. For instance, a transportation simulation might define multiple vehicle objects which have attributes for current speed and position. An interaction is a one-time event with no continuity between two instances of the same interaction class. For instance, a transportation simulation that contains sensors that trigger an alert when a vehicle exceeds a certain speed limit might define an alert interaction. Objects have a life cycle where individual, named instances must be created and maintained over the course of a federation execution, while interactions are ad hoc and can be sent without preparation. Both object updates and interactions can be sent with an optional logical time stamp to ensure the causal ordering of messages given certain time management constraints.

The gateway federate has full support for the sending and receiving of both objects and interactions, either with or without logical time stamps. The gateway library exposes all the services related to object life cycle in its API, and expects the user application to manage its own object instances.

D. Time Management

The time management services control how an individual federate uses logical time within the federation. A federate can set two different flags to control its logical time progression: time constrained, and time regulating. A time constrained federate cannot advance its logical time beyond the current federation logical time. A time regulating federate dictates the pace of logical time progression, and the federation logical time will advance at the rate of the slowest regulating federate. A federate can use all four permutations of time constrained and time regulating, with a federate that is both constrained and regulating operating in lockstep with respect to federation logical time.

HLA defines two services for federates to advance logical time: time advance request, and next event request. A federate cannot advance its local logical time until it receives a time grant from the federation. When a federate receives the time grant depends on whether it is time constrained, and the current state of the set of time regulating federates. A time advance request will step the federate logical time by a fixed value, while a next event request will act as an interrupt signal and update the logical time by the smallest increment necessary to receive the next event in the federation.

The gateway federate is both time constrained and time regulating to operate in lockstep with the federation logical time, and this behavior cannot be reconfigured. It uses time advance requests with a step size that is static for the duration of the federation execution, and does not support the next event request service. All time management services are automated by the gateway library and hidden from the user application, but the user application configures the logical time step size.

V. GATEWAY LIFE CYCLE

The gateway library executes a well-defined life cycle with callbacks to the user application that can be implemented to perform custom functions. During these callbacks, the library exposes a small number of methods that can be invoked by the user application for sending data and querying the federation object model. As described in Section IV, the services relevant to the user application are primarily concerned with object management. Fig. 2 shows the complete gateway life cycle, and Table 1 describes the gateway callback methods.

The life cycle is bounded by three synchronization points named readyToPopulate, readyToRun, and readyToResign. At each of these synchronization points, the gateway federate will block its execution until the federation synchronizes on the listed name. These synchronization points divide the gateway life cycle into three separate stages of initialization, logical time progression, and termination. This section describes the behavior of the gateway federate during each stage.

A. Initialization

Each federation has a set of expected federates, without which the purpose of the federation—often the execution of some experiment—cannot be realized. It is assumed that the federate designated as the federation manager keeps track of the names and current states of all expected federates. When
the federation manager detects that all expected federates have joined the federation, it will achieve the synchronization point readyToPopulate. The federates will then initialize themselves against their peers in an initialization stage that proceeds until all federates have reached readyToRun. The initialization stage of the life cycle covers the period from when the gateway federate is instantiated until the federation execution achieves the synchronization point for readyToRun.

The gateway library does most of its automated functions when it joins the federation. It parses the FOM configuration file to build its internal object model, calls all the declaration management services to create the required publications and subscriptions, and invokes the time management services to set itself as time constrained and time regulating. It then yields control to the user application for two different phases of initialization. The first phase, self initialization, allows the user application to initialize or connect to the research infrastructure being integrated into HLA, and to register any object instances that are needed to expose the infrastructure’s internal state to the federation. The second phase, peer initialization, lets the user application exchange initial values with its peers in the federation before the first logical time step.

B. Time Progression

After the readyToRun synchronization point, the gateway federate enters the time progression stage of its life cycle. This stage will continue until either the federation manager decides to destroy the federation execution, or the user application requests to leave the federation. The only automated function of the gateway library during this stage is synchronization of the doTimeStep callback to the logical time progression of the HLA federate. However, the user application must implement almost all the object management services during this stage.

The interactions and object updates received during a time step will be sent to the user application one at a time using the gateway callback methods. While Fig. 2 represents receiving data as a single state, different callbacks are used for receiving interactions and for receiving objects as listed in Table 1. All received interactions and object updates will be handled at the start of the logical time step. Then the doTimeStep callback will be invoked with the current logical time to allow the user application to implement how a time step affects the research infrastructure being integrated using the gateway federate. If the user application expects to receive a message during the middle of its time step—which is possible if the message was sent without a time stamp—the gateway library exposes a tick method which can be used to poll for new messages.

C. Termination

The termination stage of the life cycle starts when either the user application requests to exit, or the federation manager decides to destroy the federation. If the federation manager causes the termination, then the gateway federate will wait for the readyToResign synchronization point which is achieved when all federates have transitioned into a terminating state. Otherwise, the readyToResign synchronization point will be skipped under the assumption that the federation will continue to execute after a termination request initiated by the user application. Both cases will invoke one final callback method for the user application before the Java application exits.

The gateway library automates the process of cleaning up all registered object instances and leaving the federation. When the terminate callback is invoked, the gateway federate will no longer be a member of a HLA federation, and none of the methods defined in the gateway library will be usable. The purpose of the terminate callback is for the user application to perform any necessary cleanup functions, such as disconnect or shutdown procedures required by the connected research infrastructure, before the Java application exits.

VI. CONCLUSION

CPS research and development activities use co-simulation as one technique to validate increasingly complex systems that are integrating more and more domains. HLA is one co-simulation standard that considers federates as distributed, peer processes rather than shared libraries. However, HLA is a comprehensive standard that can be difficult to integrate with existing research infrastructure such as testbeds due to its rich service set. This paper proposes the architecture for a gateway federate with a simple interface that resembles FMI to address this integration challenge. It identifies the minimum set of

<table>
<thead>
<tr>
<th>callback</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initializeSelf</td>
<td>Perform any initialization that depends on the existence of a joined federation, such as registering new object instances.</td>
</tr>
<tr>
<td>initializeWithPeers</td>
<td>Perform any initialization that depends on communication with other federates, such as the exchange of initial values.</td>
</tr>
<tr>
<td>receiveInteraction</td>
<td>Handle a single received interaction.</td>
</tr>
<tr>
<td>receiveObject</td>
<td>Handle a single received object reflection.</td>
</tr>
<tr>
<td>doTimeStep</td>
<td>Perform the logic that must be executed each logical time step, such as updating the states of registered objects.</td>
</tr>
<tr>
<td>terminate</td>
<td>Perform any cleanup, such as closing output files and stopping simulations.</td>
</tr>
</tbody>
</table>
services required to participate in a HLA co-simulation, and proposes a federate life cycle based on the assumption of a designated federation manager. An instance of the gateway federate for UCEF was discussed which uses a Java library with the minimum set of methods required to participate and exchange data in a federation. Through implementation of the callback functions defined in the gateway library, new simulators and new physical resources such as domain-specific testbeds can be integrated into HLA federations.

An implementation of the gateway federate for a specific user application requires a smaller number of integration activities—and significantly less co-simulation expertise—than a full implementation of the HLA standard. Several common activities for the integration of a new simulator are:

- Define how to represent the state of the simulation model as a set of HLA object and interaction classes;
- Determine how to progress simulation time during the fixed logical time steps of the gateway federate;
- Launch or connect to the simulation model during the initialization stage of the gateway federate life cycle;
- Implement the gateway federate callback methods to exchange inputs and outputs with the model;
- Terminate or disconnect from the simulation model during the termination stage of the life cycle.

The open-source implementation of the UCEF Gateway federate is being continuously refined to support the integration of new applications. Future development efforts for the UCEF Gateway will focus on how to best integrate both real-time hardware-in-the-loop and entire research testbeds to make it easier to run federations between geographically remote facilities. The UCEF Gateway will also be updated to support both the 2010 Evolved version of the HLA standard, and the upcoming release of Portico v2.2.0 to take advantage of several new quality of life features.

NIST also has multiple ongoing research activities that are using the UCEF Gateway to integrate different simulation engines into the Portico RTI. These simulation engines include the EnergyPlus™ building energy simulator, the GridLAB-D™ power distribution system simulator, and the LabVIEW design and development environment. NIST has additional activities to incorporate both the Cucumber™ testing environment and an experiment scripting language called courses of actions into HLA as implementations of the gateway federate. All these activities are based on the architecture described in this paper, with most choosing to implement the user application as a TCP/IP server using a custom socket protocol.

ACKNOWLEDGMENT

Official contribution of the National Institute of Standards and Technology; not subject to copyright in the United States. Certain commercial products are identified in order to adequately specify the procedure; this does not imply endorsement or recommendation by NIST, nor does it imply that such products are necessarily the best available for the purpose.

REFERENCES


