The UCEF Approach to Tool Integration for HLA Co-Simulations

Thomas Roth, Christopher Lemieux and Martin Burns
National Institute of Standards and Technology
Gaithersburg, Maryland, U.S.
thomas.roth@nist.gov, christopher.lemieux@nist.gov, martin.burns@nist.gov

***Abstract*—Cyber-Physical Systems (CPS) are complex systems that require expertise from multiple domains in their design, implementation, and validation. One cost-effective technique for validation of CPS is the integration of two or more domain-specific simulators into a joint simulation called a co-simulation. Standards such as the High Level Architecture (HLA) have been developed in part to simplify the co-simulation development process. However, CPS co-simulation still requires significant expertise, especially when the goal is the integration of a new domain-specific tool or simulator. The U.S. National Institute of Standards and Technology (NIST) has released a software platform called the Universal CPS Environment for Federation (UCEF) to simplify the development of CPS co-simulations. UCEF provides two approaches to integrate tools and simulators. The first approach is a Java library called the UCEF Gateway that limits the development effort to a list of callback functions in a well-defined simulation life cycle. The second approach is a Representational State Transfer (REST) server developed using the gateway for applications that can implement a Transmission Control Protocol (TCP)/Internet Protocol (IP) client. This paper describes how both approaches are implemented to expedite the integration of new domain-specific tools and simulators.**

***Keywords*-application programming interface, co-simulation, cyber-physical systems, high level architecture, tool integration**

I. Introduction

The design and implementation of Cyber-Physical Systems (CPS) requires significant expertise from multiple domains to ensure smooth operation. For a more formal definition, CPS consist of devices that use logical computation informed by measurements of the environment to actuate physical changes. CPS are common in critical infrastructure such as smart manufacturing, autonomous vehicles, and smart grid [1]. Failure of these systems has great economic and social costs, and validation is required to minimize the risk of failure prior to deployment. However, deployed CPS can be larger than city-scale and it is impractical to prototype all design decisions due to the immense cost associated with deployment. One cost-effective validation technique to overcome this challenge is co-simulation. Co-simulation is the integration of multiple domain-specific simulators into a common execution environment to produce results that more closely resemble the deployed system. The integration of existing simulators is a more scalable solution than the development of new, more complex simulators that can model all CPS dynamics in a single environment. It is quite common in smart grid research, for instance, to perform co-simulation that integrates a network simulator with a power system simulator [2].

The IEEE 1516-2010 High Level Architecture (HLA) is one standard for the co-simulation of distributed processes [3]. A single simulator or process is defined as a federate, and the collection of interacting federates is defined as a federation. The federation communicates and coordinates over middleware called a Runtime Infrastructure (RTI) which can be thought of as a shared message bus. The RTI provides a set of standardized services to the participating federates to facilitate the co-simulation. HLA was designed to be comprehensive and defines all the services that could be useful in distributed simulation whether or not those services are frequently used.

While HLA provides a rich and complete service set [4], the standard is complex and has features that may see minimal use in practical applications. Of the more frequently used HLA services, several can be implemented the same across all federates regardless of the domain or objective of any given experiment. But there is little publicly available information on which services are frequently used, and little guidance on how the services could be implemented to be reusable in a wide range of use cases. None of the services defined in the standard are labeled as optional, and it is not clear which parts of the standard must be implemented and which parts can be safely ignored. From the authors' experiences, learning HLA is a significant many-month process that does not greatly simplify the challenges of co-simulation.

The U.S. National Institute of Standards and Technology (NIST) is one of many groups that are developing software tools to reduce the burden of co-simulation development. The NIST tool, the Universal CPS Environment for Federation (UCEF), was released as a virtual machine that provides code generation of the HLA services for different simulators based on simple user-designed models [5]. One goal of UCEF is to provide a portable development environment where users can develop co-simulations without a background in distributed computing and the HLA standard. However, UCEF is only as powerful as the number of its supported simulators and the ease at which new simulators can be integrated.

This paper presents the approach to tool integration in UCEF. This approach makes assumptions on the HLA service set — in particular, it assumes that most of the services are not used — and uses those assumptions to produce a checklist of functions that must be implemented to integrate new simu-

lators. Two methods for tool integration are presented: a Java library that can be extended to implement a new federate type, and a Representational State Transfer (REST) Application Programming Interface (API). These methods have been used to integrate several smart grid simulators into UCEF, and were developed out of a need to support software developers with no prior co-simulation experience.

The remainder of the paper is organized as follows. Section II provides an overview of related research into simplification of the co-simulation development process. Section III presents a brief overview of the UCEF software platform. The first approach to tool integration using a Java library is presented in Section IV, and the second approach using a REST API is presented in Section V. The paper is then concluded with Section VI.

II. Related Work

Several software platforms have been created to accelerate the HLA development process. These platforms let users model a CPS using a Domain Specific Modeling Language (DSML) and leverage code generation to transform user models into code that executes a subset of the HLA services. For instance, a user might define the input and output requirements of a federate in a table, and the software platform could then transform that table into skeletal code that requires minimal implementation from the user. At the forefront of these software platforms are the commercial design tools released by different HLA RTI vendors to simplify the use of their products [6][7]. While these tools are compatible with other RTI implementations, the HLA research community has attempted to develop open-source alternatives that perform similar functions due to concerns over cost.

One of the earlier open-source software platforms for HLA was the Command and Control Wind Tunnel (C2WT) produced from the Institute for Software Integrated Systems at Vanderbilt University. C2WT uses extensions to a graphical modeling environment called the Generic Modeling Environment (GME) to support the modeling and code generation of HLA federations [8]. Because GME was a desktop application that could only be accessed by one user, it was difficult to use in organizations that required model sharing between collaborators, and therefore a web-based variant was developed called the Web-based Generic Modeling Environment (WebGME) [9]. Vanderbilt University updated C2WT to use WebGME in a new software platform called the Cyber-Physical Systems Wind Tunnel (CPSWT). Public instances of CPSWT are hosted in the cloud at Vanderbilt University and the source code is available online through their GitHub repository [10]. NIST collaborated with Vanderbilt University to produce an offline version of CPSWT with additional support for several smart grid simulators. NIST released this software platform as an Ubuntu virtual machine called UCEF [5]. Other similar approaches have used Systems Modeling Language (SysML) diagrams to generate C++ executables compatible with Simulink models [11], and extensions to Eclipse that incorporate a DSML for HLA that can generate code for C++ federates [12]. All these software platforms attempt to minimize the implementation burden on the user by making assumptions on the default implementations of certain HLA services and using code generation.

Another significant contribution to the HLA open-source community is the HLA Development Kit Framework (DKF) [13]. The DKF is not a software platform, but an open-source Java library based on Java annotations. It provides a basic class structure that can be extended through inheritance to implement Java federates, and provides default implementations of most HLA services in parent and helper classes. In addition to the Java source code, the DKF is packaged with examples, tutorials, and documentation for the creation of federates using its simplified federate life cycle.

Another example that defines a federate life cycle with default implementations of the HLA can be found in [14]. This is not a software platform, nor a reusable library, but an example implementation of one federation using a well-defined life cycle. It defines a modular Federation Object Model (FOM) for data exchange between independent systems and prescribes a specific life cycle for federates in the form of state machines.

The Functional Mockup Interface (FMI) is a more recent co-simulation standard run as a Modelica Association Project [15]. While HLA attempts to define the complete set of services that could be useful in a distributed simulation, including services not commonly used in practice, FMI takes the opposite approach of trying to define the minimal set of functions required for co-simulation. FMI research efforts face similar challenges in trying to make the standard more accessible to users without deep knowledge of co-simulation. There are tool chains that use SysML models and code generation to automate portions of the federation development process [16], and there is a C++ library that can be leveraged to provide default implementations for most of the FMI functions [17].

In all of these cases, the goal has been to abstract the full range of HLA services and FMI functions that are visible to the end user, and provide default implementations for the set of services and functions that are federate independent. The reduced service set can then be considered as an API which minimizes user interaction with the standards documents. As a consequence, all the implementations are incompatible as they redefine in different ways the standardized services and function sets to improve user accessibility. The remainder of this document describes how the HLA services were redefined for the UCEF software platform.

III. Universal CPS Environment for Federation

The software efforts to simplify federation development share several common elements: they are often open-source projects that use some graphical language that leverages code generation to transform user models into federate code. However, there is often an assumption that it's easy - or even possible - for a user to install and configure the software environment. Information technology (IT) policies such as firewall rules can prevent activities such as the installation of

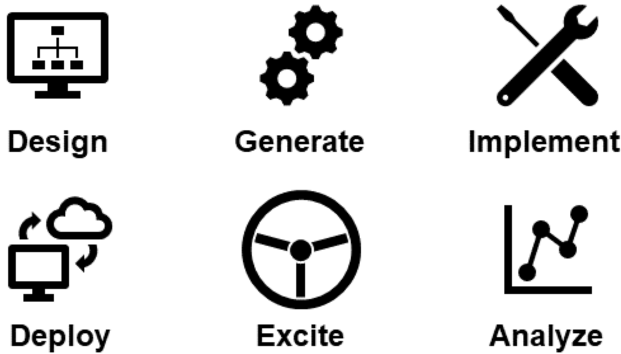

Fig. 1. The stages of Federate Development (top row) and Federation Deployment (bottom row) in the UCEF workflow

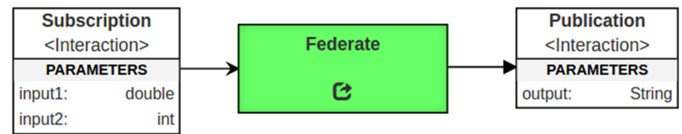

Fig. 2. An example WebGME Federate Model

new software or access to a user account with administrative rights. These policies can make it very challenging to use some of the solutions mentioned in the related work.

This section provides an overview of the NIST software platform UCEF [5]. UCEF is a portable development environment created to expedite the development of HLA federates and federations. The main feature that distinguishes UCEF from similar approaches is its distribution as a self-contained virtual machine. This makes UCEF non-intrusive, easy to redistribute, and easy to install. It is distributed as an Ubuntu 16.04 virtual machine that runs a local WebGME server. The WebGME front end provides a graphical web environment where users can model federations using simple building blocks, and the back end uses JavaScript plugins that transform these models into stub code for different simulators. The current version of UCEF supports several simulators in the smart grid domain that include GridLAB-D, TRNSYS, and LabVIEW with additional support for native Java and C++ applications. The ease of development in UCEF rises from the separation of a federate implementation into two layers: a user layer which implements the federate behavior, and an infrastructure layer generated from WebGME that provides the default implementations for most HLA services.

Figure 1 shows the stages from federate development to federation deployment in the UCEF workflow. The current version of UCEF implements the top row related to federate development which consists of the stages Design, Generate, and Implement. These three stages produce an executable piece of software that can be run on any compute environment ranging from the UCEF virtual machine, to a desktop computer, to a node in the cloud. While UCEF also generates some simple bash scripts for deployment, the bottom row on federation deployment is still under development and the three stages of Deploy, Excite, and Analyze are notional.

The *Design* stage uses WebGME with the HLA meta-language produced at Vanderbilt University for their platform CPSWT. A user produces a graphical model of a federate in a web browser which includes the specification of its simulator type (such as LabVIEW or Java program) and its various inputs and outputs. Figure 2 shows an example of a simple federate designed in this environment that both subscribes to and publishes one HLA message. Because WebGME is a web-application running on the local virtual machine, this modeling phase does not require an Internet connection despite the use of an web browser.

The *Generate* stage is initiated when the user clicks a run button in WebGME to execute its code generation plugins on the federate model. WebGME plugins are written in JavaScript and use Embedded JavaScript Templates (EJS) to define the artifacts that should be generated for each of the supported simulator types. All artifacts produced from WebGME in UCEF are output as Apache Maven projects, regardless of whether they contain Java code. These artifacts have dependencies on the open-source Java RTI Portico and will not work with other RTI implementations [18].

The *Implement* stage varies dependent on the type of federate that was designed and generated, and may occur outside of the UCEF virtual machine. Appropriate domain-specific tools are used to implement each federate type, so Java files are implemented in Eclipse and LabVIEW projects are implemented in LabVIEW. In the cases that require an active license, such as LabVIEW, the generated files will have to be moved to a licensed machine to complete the federate development process.

Figure 3 shows how UCEF implements these three stages of the federate development process. The UCEF virtual machine contains a local WebGME server that is preconfigured with support for various types of simulators. When a user finishes the design and generate stage, stub code for the modeled federates is available outside of the UCEF virtual machine that can connect to an RTI and participate in a federation execution without any additional user implementation. However, the stub federate code contains no behavior and must be implemented by the user to fulfill its design goal. All the simulators shown in Figure 3, in addition to native Java and C++ applications, have been integrated into UCEF using the two approaches described in this paper.

The bottom row of Figure 1 on federation deployment is a notional representation of how deployment could work in UCEF. A federate designed in UCEF can be removed from the virtual machine and deployed in any environment, from a laptop to the cloud. Therefore, the mechanical process of deployment depends on an infrastructure that was provisioned and configured independent of the UCEF virtual machine, and this process may have significant differences from one work environment to another. There are, however, general deployment activities where UCEF could provide useful tools to expedite the deployment process.

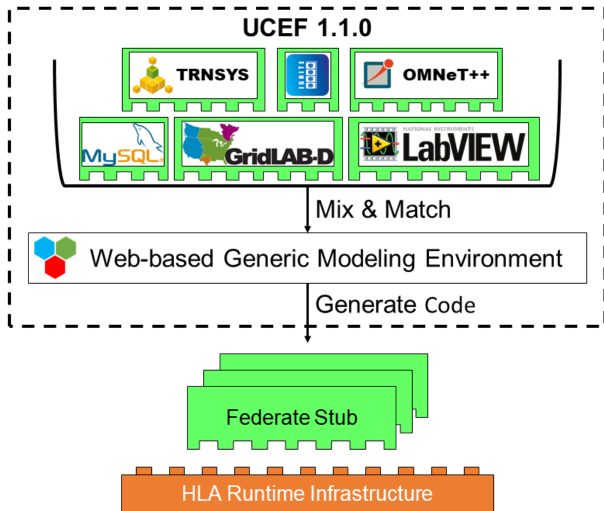

Fig. 3. Federate Development in UCEF

The federation *Deploy* stage needs to package the federates and their dependencies for deployment. First, the user needs to select the federates that should be deployed. It's possible that multiple instances of the same source code could be deployed as different federates in the same federation, and the selection process should handle this case. Then, the artifacts that contain the federates and their dependencies need to be collected from some database and packaged for deployment.

Once deployed, federates will be either configured or driven to execute a desirable scenario. The *Excite* stage perturbs the federation execution using a combination of static configuration files and dynamic runtime messages. For this stage, a simple scripting language could be incorporated into WebGME to generate configuration files that script the runtime behavior of federates. In addition, a suite of federates that enable user interaction – using either a graphical interface or a web server – could be developed and packaged with UCEF.

The *Analyze* phase involves both runtime monitoring of the federation and the use of database storage for offline analysis. For both cases, analytic federates are required to either allow user interaction at runtime or interface with different database systems for data logging.

The current version of UCEF supports the generation of bash scripts to run the federates in the virtual machine, and includes a database federate to store the results of the federation execution. The remainder of this paper focuses on the first row related to federate development and discusses how code generation, and default HLA service implementations, have been used to simplify the federate development process in UCEF.

IV. UCEF Gateway

The first approach to ease federate development is an open-source Java library called the UCEF Gateway that implements a simplified federate life cycle [19]. The gateway implements a main loop that yields control to user-implemented callback functions at specific points in this life cycle. The gateway was developed based on the following three requirements:

1) usable without HLA expertise
2) easy to integrate new things
3) agnostic to the federation data model

To satisfy the first two requirements, the gateway does not support the following HLA services: federation save, federation restore, ownership management services, and data distribution management services. A UCEF federation executes one experiment from start to finish and then terminates, without federates joining or leaving during the federation execution. Therefore, there is no need to load a prior state or handle the distribution of object instances due to sudden changes in the federation membership. The data distribution management services are useful for improved scalability, but the HLA implementation of regions as an unsigned integer is unwieldy. Each federate implementation must have the same region encoder and decoder functions to have consistent interpretations of the integer value, and this creates a new scalability problem due to the difficulty of configuration management. It's better to address scalability using traditional networking approaches rather than the use of HLA regions [20].

The third requirement distinguishes the UCEF Gateway from the HLA DKF which requires explicit Java annotations for declared variables that represent federation data. This requirement was derived from the need for a mechanism to ease the integration of entire simulators, not individual simulations. A simulator requires one reusable federate implementation that can support any simulation with an arbitrary data model. An approach that requires specification of a data model will need additional user implementation whenever a federate is integrated into a new domain or scenario, which defeats the purpose of a gateway library.

Since its release, the UCEF Gateway has been used to integrate several simulators: GridLAB-D, TRNSYS, and LabVIEW. Based on the lessons learned from these applications, a revised version with a modified federate life cycle has been released. This section summarizes the UCEF Gateway and highlights the modifications since its original publication.

A. Time Management Strategy

The gateway executes a well-defined life cycle with callbacks to the user application that can be used to define federate behavior. During the callbacks, the user code can use the public methods of the gateway library to perform functions such as sending data to the federation and querying the FOM. The gateway defines the time management strategy on behalf of the user application, and this strategy cannot be modified. All gateway implementations are both time constrained and time regulating to operate in lockstep with federation logical time. Logical time progression uses the HLA time advance request service with a fixed step size for the duration of the federation execution. The logical step size can be configured by the user in the gateway configuration files. At this time, the next event request service is not supported.

During its life cycle, the gateway assumes the federation has three synchronization points: ready to populate, ready to run, and ready to resign. The gateway assumes that another federate registers these synchronization points, and that federate also determines when the federation synchronizes on each point. In UCEF this federate is called the federation manager, and it gates progression through the federate life cycle by delaying its synchronization until it determines the federation is ready to progress. These synchronization points divide the federate life cycle into three distinct stages: initialization, logical time progression, and termination.

B. Federate Life Cycle

Figure 4 shows the UCEF Gateway federate life cycle. The rectangles are the user-implemented callback functions. Several transitions between callbacks depend on federation synchronization events, which are indicated with a labeled dotted line below the transition. A gateway implementation can block on the first transition, labeled *JoinFederation*, when it tries to join a federation that has not yet been created. The other synchronization events correspond to the three synchronization points discussed in the previous subsection on time management. In addition, the transition labeled *time advance grant* blocks until the federation as a whole advances its logical time to the next logical time step.

Table I describes each callback function in the life cycle. Four of the callback functions that begin with the word *receive* are consolidated in Figure 4 as the single state *receive data*. The order of these callbacks is arbitrary and interleaved, as it depends on the RTI implementation. It is possible that some of the *receive* callbacks do not occur for a given logical time step, and that the callbacks occur in different orders between logical time steps. However, all the *receive* callbacks will be handled prior to the gateway invoking the *step* callback.

The life cycle and callback functions do not show the public methods available in the gateway library for interaction with the federation. Some of these methods are intuitive, such as sending data to the federation and querying for the data type of received data. These methods are summarized in the original gateway publication, and unchanged in the revised version. Of note is that the gateway library provides a polling mechanism to receive new data that can be executed anywhere in the life cycle except *before join federation* and *before exit*. Although Figure 4 seems to indicate that data arrives in one bulk read operation each logical time step, the user application could choose to poll for data where it is needed. The two noted exceptions are merely because the gateway does not exist as a federate in a federation for those two stages of the life cycle, so the poll data operation is undefined.

A few of the callback functions are new since the original release of the UCEF Gateway. The callback *before join federation* was added to give a user a proper initialization method in the life cycle rather than relying on the Java constructor to serve this role. The *before first step* and *before ready to resign* callbacks were added to give unique meaning to the first and last logical time step of a simulation. Before these callbacks were introduced, conditional logic had to be inserted into the *step* callback to determine whether a time step was an intermediate or edge step. This complicated the user code more than the addition of two optional callbacks that could be ignored when the first and last steps are not distinguished. Likewise, the *receive object registration* and *receive object deleted* callback were added to prevent the use of conditional logic inside the *receive attribute reflection* callback when processing an object instance for the first time. While this has led to the introduction of five additional callback functions, the default behavior for each callback is a no-operation.

V. REST API

The UCEF Gateway was intended to simplify federate development by providing reasonable default implementations for the HLA services that did not change between federate implementations. However, in practice, most gateway applications that integrated simulators into UCEF used a simple client-server architecture with TCP/IP sockets. It is much easier to embed a socket in a simulator to pump its data to a server than extend the simulator source code to implement the gateway callback functions. This common use of the gateway led to redundant socket code between gateway implementations — a problem the gateway was designed to alleviate — and the creation of unique communication protocols for each simulator. This section describes the first attempt at a REST API built using the UCEF Gateway to provide a common server implementation for these TCP/IP socket applications. This REST API is available as a standalone federate distributed with UCEF that can be incorporated into any federation.

The reusable TCP/IP server was implemented as a REST API rather than a custom socket protocol for three reasons. First, the client code would be shielded from the potentially long synchronization delays caused by HLA logical time progression. A sensor or small Internet of Things (IoT) device might want to produce a stream of data at a constant frequency and avoid blocking calls. The fast response of a REST implementation will support these devices without the need for a multi-threaded implementation. Second, the REST implementation eliminates the need for constant heartbeat messages between the client and server to ensure a persistent socket connection. This reduces the overhead for the client implementation. Third, the REST API introduces another layer of abstraction that may make it easier for users to develop new federates without knowledge of the HLA standard.

The HLA standard already defines a REST API for interacting with the RTI [21][22]. It is important to note that the standard API exposes the complete set of RTI services, which includes the services that are rarely utilized and the services where default implementations are sufficient for most use cases. Use of the standard API represents a significant implementation burden on the user, and loses the benefits of a simplified approach like the UCEF Gateway. For this reason, a new REST API implemented on top of the UCEF Gateway was developed and is presented in this section.

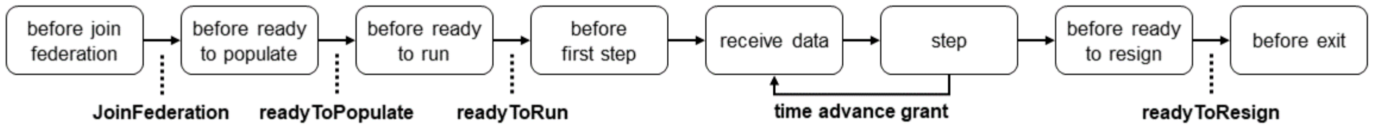

Fig. 4. UCEF Gateway Life Cycle

TABLE I
GATEWAY CALLBACK FUNCTIONS

Callback	Description
before join federation	Perform basic initialization that is HLA independent, such as initialization of data structures
before ready to populate	Perform initialization that requires a joined federation, such as registration of object instances
before ready to run	Perform initialization that requires other federates, such as the exchange of initial values
before first step	Perform one-time actions for the first logical time step, such as starting a simulation
receive object registration	Handle a discovered object instance
receive attribute reflection	Handle an attribute reflection for one discovered object instance
receive object deleted	Handle a removed object instance
receive interaction	Handle one received interaction
step	Perform the logic executed each logical time step, such as updating object attributes
before ready to resign	Perform one-time actions for the last logical time step, such as the exchange of final values
before exit	Perform any cleanup, such as stopping the simulation and closing output files

TABLE II
LIST OF ENDPOINTS

Endpoint	Method	Request Format	Response Format
/status	GET	(none)	FederateStatus
/join	POST	ClientPost	FederateStatus
/dostep	POST	ClientPost	FederateStatus
/ping	POST	(none)	200 OK

A. Endpoints

Table II lists the endpoints defined for the REST API. The request and response formats indicated in this table are defined in the following subsection on payloads. The *ping* endpoint is intended to be a light-weight heartbeat message to check the status of the server. When the federation is starting, or when the client loses connection with the server, the *ping* message can be used to periodically check if the server is online. The *status* endpoint returns the current state of the server. Because the server implements the UCEF Gateway, it represents a federate in some federation, and its status contains information such as the current logical time and the most recent values for data exchanged in the federation. The *join* endpoint is used to tell the server that the client wants to join the federation, and contains some details about the client's identity and data model. The *dostep* endpoint is used to tell the server that the client is ready to advance to the next logical time step, and contains the set of client data that should be broadcast to the federation.

B. Payloads

JavaScript Object Notation (JSON) is used to define all the payload formats. The endpoints define two payloads called *ClientPost* and *FederateStatus*. However, these payloads contain HLA interactions and object instances. This subsection will first define the JSON format for these primitive HLA data structures, and then the formats used in Table II.

Listing 1 defines the JSON for object instances. An object in HLA can be thought of like a variable in a programming language. The user defines a data type (the object class) and a unique variable name (the instance name). The object class determines the structure of the data associated with the object instance in much the same way the data type determines the structure of a variable. The *classPath* field is the fully qualified path for the object class, and the *instanceName* field is the unique identifier of a particular object instance. The *attributes* array is a list of name-value pairs for the data associated with the specified object instance. All of the attribute values are encoded as strings, although the attribute could be any of the data types defined in the FOM. In the current implementation, the client and server are both pre-configured with the FOM so both sides can convert the string value into the correct data type. However, future work will have the client send the FOM to the server by embedding it into the payload of the *join* endpoint.

Listing 1
OBJECTINSTANCE JSON FORMAT

```
{
  "classPath": "ObjectRoot.ClassName",
  "instanceName": "ObjectInstanceName",
  "attributes": [
    {
      "name": "attrName",
      "value": "asString"
    }
  ]
}
```

The interpretation of Listing 1 depends on the direction of data flow. When the client sends an object instance to the server, it is a request to publish updated values for an object instance registered by the client. When the server sends an

object instance to the client, it is a notification that the values for that object instance have changed in the federation.

Listing 2 shows the JSON format for interactions which is almost identical to the format for object instances. Because interaction instances are not assigned unique identifiers, as they have no persistent state that changes over logical time, the *instanceName* field has been dropped. In addition, to conform with the HLA naming conventions, the *attributes* field has been renamed to *parameters*. Otherwise, the format and use of this JSON payload is identical to object instances.

Listing 2
INTERACTION JSON FORMAT

```
{
  "classPath": "InteractionRoot.Class",
  "parameters": [
    {
      "name": "paramName",
      "value": "asString"
    }
  ]
}
```

Listing 3 shows the *ClientPost* JSON format that is sent from the client to the server. The client provides its internal state, represented with three boolean variables, as well as the list of interactions and object instances that should be published to the federation. *isJoining* is a boolean flag that indicates the client is ready to start the simulation, and *isLeaving* is a boolean flag that indicates the client is ready to stop the simulation. *isAdvancing* is a boolean flag that indicates the client has finished its current logical time step and is ready to advance logical time to the next iteration of execution. Not all permutations of values for these flags are valid, as a client cannot simultaneously join and leave. The valid permutations will be elaborated on in the following subsection on the state machine. The *interactions* and *objects* arrays contain updated values for all the interactions and object instances for the current logical time step. If an object instance has not changed from the previous time step, it can be omitted entirely from this array.

Listing 3
CLIENTPOST JSON FORMAT

```
{
  "isJoining": true/false,
  "isAdvancing": true/false,
  "isLeaving": true/false,
  "interactions": [ Interaction ],
  "objects": [ ObjectInstance ]
}
```

Of note is that the *ClientPost* payload provides no mechanism for the client to inform the server that it wants to register a new object instance. The server maintains a list of names for the client's registered object instances. When the client sends an object instance with an unknown *instanceName*, the server will automate the object registration process and update its internal list.

Listing 4 shows the *FederateStatus* JSON format that is sent from the server to the client. The server also provides its internal state using three boolean variables that will be further explained in the subsection on the state machine. *isSimulationActive* indicates the federation exists, and is either preparing to begin or has already begun logical time progression. *isDoStep* indicates the server is idle waiting on input from the client before it proceeds to the next logical time step. *isTerminating* indicates the simulation is over, and the client needs to prepare to exit. An additional *timeStep* value is provided to notify the client of the current logical time in the HLA federation. The *interactions* and *updatedObjects* arrays contain updated values received from the other federates. If an object instance has not been updated since the last status report, it is omitted entirely from this array. The *FederateStatus* JSON also contains a *newObjects* array that lists all the object instances that were registered by other federates since the last status update. If an object instance appears in the *newObjects* array for a given status report, it will not appear in the *updatedObjects* array.

Listing 4
FEDERATESTATUS JSON FORMAT

```
{
  "isSimulationActive": true/false,
  "isDoStep": true/false,
  "isTerminating": true/false,
  "timeStep": 0.0,
  "interactions": [ Interaction ],
  "newObjects": [ ObjectInstance ],
  "updatedObjects": [ ObjectInstance ]
}
```

C. State Machine

Figure 5 shows the state machine implemented by the REST server. This state machine does not show the exceptions that might cause the server to respond to the client with an error code. An exception would occur whenever the server receives a *join* or *dostep* request from the client in a state where there is no explicit transition labeled with that endpoint. The state transitions mirror the synchronization points identified in the gateway life cycle from Figure 4. The same labels are used for *JoinFederation*, *readyToPopulate*, *readyToRun*, *time advance grant*, and *readyToResign*. Three additional labels appear in the state machine transitions: two of the REST endpoints for *join* and *dostep*, and one transition labeled *exit condition*. Because the *ping* and *status* endpoints are valid in all states, they are not shown in the state machine.

The server has an internal state represented by the three boolean flags *isSimulationActive*, *isDoStep*, and *isTerminating*. These booleans have well-defined values for each state in the state machine as indicated in Figure 5. When the server produces a *FederateStatus* payload in response to a client request, the values of these booleans associated with the current

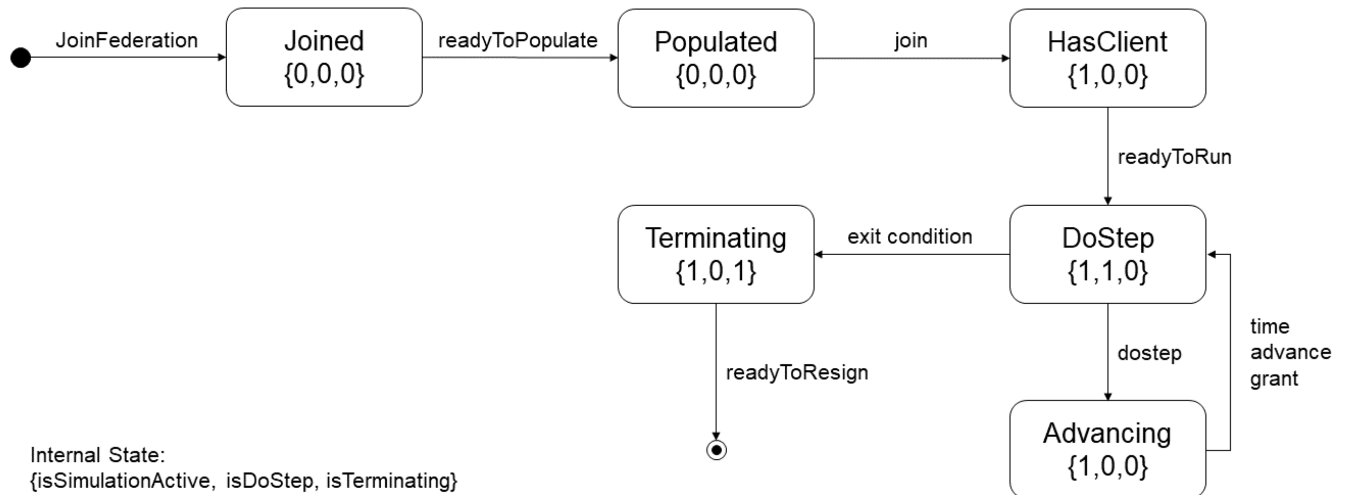

Fig. 5. State Machine of the REST Server

state are used to populate the contents of that payload. The remaining information is retrieved from the HLA federation.

VI. Conclusion

This paper provided a brief overview of UCEF and described two different approaches to tool integration for the platform. The first approach, an open-source Java library called the UCEF Gateway, is a mature implementation that has been used to integrate several new simulators into UCEF by different developers. The second approach, a REST server built using the gateway library, is a more recent development undergoing continuous refinement. Both approaches seek to simplify the tool integration challenge by providing default implementations for HLA services and exposing a reduced API that is more accessible for developers without co-simulation experience.

There are future plans to expand the REST server to support multiple simultaneous client sessions with the goal of creating something akin to an IoT gateway that integrates a large number of devices into a federation as a single federate. The server state machine will have to be updated to support multiple clients, and a session identifier will have to be inserted into the payloads so clients can be identified between calls to the different endpoints. During this process, it is likely the list of endpoints and the payload structure will undergo continuous refinement as the software matures.

One interesting research direction for this work is the performance characterization of different types of user applications using the two approaches. The REST server relies on socket communication rather than direct function calls and should be slower and more prone to bottlenecks than native UCEF Gateway implementations. However, the communication pattern of the user application — such as the frequency of communication and the size of the payloads — could result in vastly different performance profiles. It is likely that some types of user applications are ill-suited to using the REST server, while other types notice little to no performance degradation over a native gateway implementation. An investigation of these different performance characteristics would add another dimension beyond ease-of-use that must be considered when choosing to integrate a tool using one of the approaches.

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

[1] E. R. Griffor, C. Greer, D. A. Wollman, and M. J. Burns, "Framework for cyber-physical systems: Volume 1, overview," Tech. Rep., 2017, doi: 10.6028/NIST.SP.1500-201.

[2] S. C. Müller, H. Georg, J. J. Nutaro, E. Widl, Y. Deng, P. Palensky, M. U. Awais, M. Chenine, M. Küch, M. Stifter *et al.*, "Interfacing power system and ICT simulators: Challenges, state-of-the-art, and case studies," *IEEE Transactions on Smart Grid*, vol. 9, no. 1, pp. 14–24, 2016, doi: 10.1109/TSG.2016.2542824.

[3] "IEEE standard for modeling and simulation (M&S) high level architecture (HLA)– framework and rules," *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)*, pp. 1–38, Aug 2010, doi: 10.1109/IEEESTD.2010.5553440.

[4] "IEEE standard for modeling and simulation (M&S) high level architecture (HLA)– federate interface specification," *IEEE Std 1516.1-2010 (Revision of IEEE Std 1516.1-2000)*, pp. 1–378, Aug 2010, doi: 10.1109/IEEESTD.2010.5557728.

[5] M. Burns, T. Roth, E. Griffor, P. Boynton, J. Sztipanovits, and H. Neema, "Universal CPS environment for federation (UCEF)," in *2018 Winter Simulation Innovation Workshop*, 2018.

[6] (2019) Mak RTI. [Online]. Available: https://www.mak.com/

[7] (2018) Pitch pRTI. [Online]. Available: http://pitchtechnologies.com/

[8] G. Hemingway, H. Neema, H. Nine, J. Sztipanovits, and G. Karsai, "Rapid synthesis of high-level architecture-based heterogeneous simulation: a model-based integration approach," *Simulation*, vol. 88, no. 2, pp. 217–232, 2012, doi: 10.1177/0037549711401950.

[9] M. Maróti, T. Kecskés, R. Kereskényi, B. Broll, P. Völgyesi, L. Jurácz, T. Levendovszky, and Á. Lédeczi, "Next generation (meta) modeling: web-and cloud-based collaborative tool infrastructure," *MPM@ MoDELS*, vol. 1237, pp. 41–60, 2014.
[10] H. Neema, J. Sztipanovits, C. Steinbrink, T. Raub, B. Cornelsen, and S. Lehnhoff, "Simulation integration platforms for cyber-physical systems," in *Proceedings of the Workshop on Design Automation for CPS and IoT*. ACM, 2019, pp. 10–19, doi: 10.1145/3313151.3313169.
[11] M. Bombino and P. Scandurra, "A model-driven co-simulation environment for heterogeneous systems," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 4, pp. 363–374, 2013, doi: 10.1007/s10009-012-0230-5.
[12] T. Nägele and J. Hooman, "Rapid construction of co-simulations of cyber-physical systems in HLA using a DSL," in *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2017, pp. 247–251, doi: 10.1109/SEAA.2017.29.
[13] A. Falcone, A. Garro, S. J. Taylor, A. Anagnostou, N. R. Chaudhry, and O. Salah, "Experiences in simplifying distributed simulation: The HLA development kit framework," *Journal of Simulation*, vol. 11, no. 3, pp. 208–227, 2017, doi: 10.1057/s41273-016-0039-4.
[14] P. T. Grogan and O. L. De Weck, "Infrastructure system simulation interoperability using the high-level architecture," *IEEE Systems Journal*, vol. 12, no. 1, pp. 103–114, 2015, doi: 10.1109/JSYST.2015.2457433.
[15] (2014) Functional mock-up interface for model exchange and co-simulation 2.0. [Online]. Available: http://fmi-standard.org
[16] P. G. Larsen, J. Fitzgerald, J. Woodcock, P. Fritzson, J. Brauer, C. Kleijn, T. Lecomte, M. Pfeil, O. Green, S. Basagiannis *et al.*, "Integrated tool chain for model-based design of cyber-physical systems: The INTO-CPS project," in *2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*. IEEE, 2016, pp. 1–6, doi: 10.1109/CPSData.2016.7496424.
[17] E. Widl and W. Müller, "Generic FMI-compliant simulation tool coupling," in *Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017*, no. 132. Linköping University Electronic Press, 2017, pp. 321–327.
[18] (2019) Portico RTI. [Online]. Available: http://www.porticoproject.org/
[19] T. Roth and M. Burns, "A gateway to easily integrate simulation platforms for co-simulation of cyber-physical systems," in *2018 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES)*. IEEE, 2018, pp. 1–6, doi: 10.1109/MSCPES.2018.8405394.
[20] T. Roth, M. Burns, and T. Pokorny, "Extending portico HLA to federations of federations with transport layer security," *2018 Fall Simulation Innovation Workshop (SIW)*, no. 18F-SIW-038, 2018.
[21] B. Möller, S. Löf *et al.*, "Mixing service oriented and high level architectures in support of the GIG," in *Proceedings of the 2005 Spring Simulation Interoperability Workshop*, no. 05S-SIW, 2005, p. 64.
[22] B. Möller and S. Löf, "A management overview of the HLA evolved web service API," in *Proceedings of 2006 Fall Simulation Interoperability Workshop, 06F-SIW-024, Simulation Interoperability Standards Organization*, 2006.