Reusable Network Simulation for CPS Co-Simulations

Himanshu Neema
Harmon Nine
himanshu.neema@vanderbilt.edu
harmon.s.nine@vanderbilt.edu
Vanderbilt University
Nashville, TN, USA

Thomas Roth
thomas.roth@nist.gov
National Institute of Standards and Technology
Gaithersburg, MD, USA

ABSTRACT

Cyber-Physical Systems (CPS) require seamless integration of physical and computation components through communication networks. They often contain sub-systems from different physical domains (e.g., electrical, mechanical, and electronic) that must coordinate over a communications network to achieve a joint function. Thus, network characteristics significantly affect CPS performance and must be accurately modeled and simulated. However, simulation efforts are often siloed into specific domains (e.g., manufacturing, transportation, and electric grid) due to the level of complexity required to simulate even one domain. Co-simulation standards such as the IEEE High Level Architecture (HLA) attempt to facilitate model sharing between domains through definition of common services such as time management on a shared message bus. However, a well-developed, integrated, and configurable network simulation component that can be readily deployed in CPS co-simulations is lacking due to both the network simulation complexity and customization needed for specific domains. This paper presents a novel approach to create a highly reusable and configurable network simulation for HLA co-simulations which includes a cyber-attack library for analyzing behavior of CPS under attack. This work could provide effective means to quickly develop cyber scenarios for analyzing CPS through networked co-simulations.

CCS CONCEPTS

• **Computing methodologies → Modeling and simulation**; • **Computer systems organization → Embedded and cyber-physical systems**.

KEYWORDS

networked co-simulation, cyber-physical systems, system-of-systems,

1 INTRODUCTION

Cyber-Physical Systems (CPS) [7] [6] achieve their function through the networked coordination of cyber and physical components that sense and control a physical environment. They derive their capabilities from many physical domains, such as electrical, mechanical, and electronic, that interact through network communication. For example, the temperature sensor in a modern thermostat sends its measured ambient temperature to the controller as a network message which the controller uses to react when the temperature crosses a set point. Similarly, when the controller triggers actuation of the heating and cooling unit, that information is also sent over a communication network. Most CPS are large system of systems (SoS) that contain many independent sub-systems that must interoperate to fulfill the objectives of the system as a whole. For example, a typical automotive vehicle contains multiple sub-systems (e.g., the engine, transmission, fuel system, steering, braking, etc.) that are networked for communication and control. Thus, communication networks play a critical and central role in the correct working of CPS, and the accurate modeling and simulation of these systems must consider the potential impact of the network dynamics.

One approach to handle network dynamics is a modeling and simulation technique called co-simulation, where multiple simulators exchange information at run time to execute a joint simulation. A co-simulation might combine a sophisticated vehicle dynamics model with a network simulator, with the information exchange between vehicle sub-systems first routed through the network simulation to induce realistic communication delays. However, real-world networked co-simulations of CPS are highly challenging. This is not only because network simulation is itself complex (e.g., simulation of all networking and application layers, devices, routing, protocols, etc.), but also because these systems continuously evolve and must be evaluated in a variety of application contexts – necessitating a general-purpose, reusable approach to integrating the network simulation. However, a well-developed, integrated, and configurable network simulation component that can be readily deployed in CPS co-simulations is lacking. Therefore, this paper describes a novel approach to create a highly reusable and configurable network simulation for use in networked co-simulations of CPS.

Various co-simulation techniques exist based on standards such as Functional Mock-up Interface (FMI) [2], Distributed Interactive Simulation (DIS) [4], and Data Distribution Service (DDS) [17]. However, these approaches have issues with real-world requirements such as time synchronization, variable time resolution and time scales, flexible information flows with direct, broadcasted, and filtered information, and distributed object management [12]. In addition, co-simulations must support simulators that have different data models, are implemented in different programming languages, and utilize different models of computation (e.g., continuous time, discrete event, discrete time, etc.). The IEEE High-Level Architecture (HLA) standard [1] was designed to address these issues with distributed simulation. In HLA, a co-simulation is called a federation and the different systems that participate in the co-simulation are called federates. However, it can be difficult to make simulators conform to the HLA standard due to their size and complexity.

The authors have previously developed a model-based framework, called the Cyber-Physical Systems Wind Tunnel (CPSWT) [15] [16], for rapidly synthesizing large-scale integrated simulations in HLA. CPSWT uses the Portico Run-Time Infrastructure (RTI) [3] for its implementation of the services defined in the HLA standard. In CPSWT, the different systems that participate in the co-simulation are abstractly modeled with their various information exchanges. CPSWT supports extensive configuration of these systems and its software tools will transform the abstract models into HLA-compliant code for many widely used simulators. The HLA-based reusable network component is included in CPSWT and can be configured for use in a variety of different application domains and scenarios.

The reusable network simulation was developed using OMNeT++ [19] and the INET Framework [11], which are open source and widely used for network simulation. Several network modules were extended to make them configurable to domain-specific use cases in CPS and new modules were created for time synchronization and data exchange with the HLA. A reusable cyber-attack library from prior work was integrated with these modules to allow testing of CPS under various attack scenarios [9]. Further, a novel approach to dynamic and embedded messaging was developed to support processing of HLA message types through the simulated network.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 identifies the key requirements for creating reusable network simulation for CPS co-simulations and Section 4 presents the proposed solution including the detailed HLA integration architecture including the techniques of dynamic and embedded messaging and the generic cyber-attack library. Section 5 demonstrates the presented approach using a brief case study. Section 6 concludes and provides directions for further research.

2 RELATED WORK

Networked co-simulations are critical for evaluation of CPS due to their huge reliance on reliable network communication. Several approaches to co-simulation of CPS are described in [15] and [16]. An evaluation of CPS security and resilience using co-simulation was presented in [9]. This paper focuses specifically on creating a reusable network simulation in the context of CPS co-simulations.

An earlier attempt at creating reusable network simulation based on simplified assumptions can be found in [5], but it requires changing the integration scheme based on changes to network topology and does not handle HLA object updates. An approach using a file to map parameters of the network simulation messages into HLA interaction class parameters appeared in [20]. However, it also does not handle object updates. Moreover, both of these approaches do not address the model integration problem. In the CPSWT framework, model-based integration is at the core of integrating heterogeneous simulations (including network simulation).

This work builds upon previous attempts at solving the network simulation integration in a generic manner as described in [18], which, however, required recompilation of network simulation for different HLA federations, did not handle object updates, and suffered from logical time delays through mapping messages. In contrast, the proposed solution does not require recompilation, handles object updates, and does not introduce any logical time delays with only a little computational overhead (see Section 4).

3 NETWORK SIMULATION REQUIREMENTS

The reusable network simulation for CPS co-simulation was developed based on the following requirements:

- ***Reusability*** It should be possible to reuse the same network model in multiple co-simulations, even those in different application domains, without the need to modify or recompile the model. This network model should be configurable to support variations in network parameters. It should be extensible for enhancing its functionality and for overriding its implemented behavior.
- ***Well-Defined Interfaces:*** The network simulation implementation should modularly allow swapping implemented components for different requirements (e.g., Wired vs Wireless connections). It should have a well-defined external interface for ease of integration into different co-simulations. This interface should be based on a single HLA Federation Object Model (FOM) – or data model – that is reusable in any co-simulation that uses network simulation.
- ***Cyber Attack Modeling:*** The messages propagated through the simulated network should be subject to potential modification due to fault or intentional cyber attack. It should be possible to configure which messages are modified, and to what extent, using a simple scripting or modeling language.
- ***Support for HLA Message Types:*** The network simulation implementation should support the standard HLA message types of interactions and object updates. For object updates, it should be possible to route only a subset of the object attributes through network simulation, while the remainder are delivered using the default HLA delivery services.
- ***Support for Multiple Network Simulations:*** The implementation should support co-simulations that contain more than one network simulator for simulation of different networks, or different parts of a large network. When simulating different parts of the same network, it should be possible to seamlessly pass a message from one network simulator to another when it crosses simulation boundaries.
- ***Tracing and Troubleshooting:*** The modules of the network simulation developed or modified to support co-simulation should be configurable to log (as and if needed) their internal behavior with detailed traces to support troubleshooting.
- ***Model Validation through Testing:*** The implementation should be well tested using testing frameworks such as JUnit and CPPUnit. To support efficient testing and validation, it should be possible to quickly launch a test co-simulation that stresses the implementation's basic features.

4 INTEGRATION ARCHITECTURE

There are two delivery mechanisms for HLA messages. A Receive Order message will be delivered as soon as possible, while a Time Stamp Order message will be delivered at a specified logical time (provided that both sender and receiver are configured to use logical time). To simulate network delays, the logical time axis of the co-simulation must be mapped to a representation of physical time (for example, 1 logical time may be defined to be 5 seconds) and all messages must be sent using Time Stamp Order. Then the HLA messages (interactions and object updates) are routed through a network simulator which determines the exact logical time each message should arrive based on the network conditions. Dependent on the HLA implementation and number of unique message types, it can be difficult to configure the network simulator to support all the messages required for a particular federation. This is because, typically, the implementation of each HLA message type (interaction or object class, heretofore called an "HLA class") consists of a corresponding "language class", i.e., a class written in a programming language such as Java or C++.

This results in two undesirable consequences. First, the network simulator generally requires compilation of federation-specific code to render an executable, which includes the language classes that implement the HLA classes. This means that the executable must be recompiled for each federation due to its unique HLA message types, and each time the data model for an existing federation is modified. This makes it difficult to reuse network models in co-simulations, as it may not be possible to use the network simulator without a manual and time-consuming compilation process.

Second, even after a different set of HLA classes is compiled into the executable, the network simulation still needs to be modified to reference the new data model. So, in addition to the time taken to recompile the executable, the network simulation must be modified to incorporate the changes to messaging.

These problems are a result of binding the implementation of an HLA class to a specific language class, rather than having one dynamic message class that could encapsulate all of the HLA classes.

4.1 Dynamic Messaging

The solution to this problem is to allow one language class to implement any HLA interaction class and any HLA object class by using generic representations of HLA message properties (parameters for interactions, attributes for objects). Rather than storing the HLA class properties as specific language class variables, a map is defined whose keys are the property names[1], and whose values are the corresponding property values[2]. The language class used to implement any HLA interaction class is InteractionRoot and any object class is ObjectRoot. Both InteractionRoot and ObjectRoot are at the top of the HLA interaction and object inheritance hierarchies respectively, and so are called the base messaging classes.

[1]In reality, the keys are a 2-tuple of (full-class-name, property-name), where the full-class-name is the full name of the HLA class in which the property (with name property-name) is directly defined. This disambiguates two properties that have the same name but are defined in classes that are at different levels of the class inheritance hierarchy. The full-class-name of an HLA class is a dotted sequence of the names of the class's full ancestry in its inheritance hierarchy, starting with its most distant ancestor.

[2]The value type of the map must be one that can represent any type of value (e.g., *Object* class for Java.)

```
"interactions": {
    "InteractionRoot": {},
    "InteractionRoot.C2WInteractionRoot": {
        "actualLogicalGenerationTime": {
            "Hidden": false,
            "ParameterType": "double"
        },
        "federateFilter": {
            "Hidden": false,
            "ParameterType": "String"
        },
        "federateSequence": {
            "Hidden": false,
            "ParameterType": "String"
        }
    },
    …,
```

Figure 1: Extended FED with Type Information

Table 1 shows basic operations[3] for the InteractionRoot class in Java. This class can represent any arbitrary HLA interaction class using these basic operations. There is also a method for iterating through the parameters of the HLA class that the InteractionRoot instance is implementing and processing them in a loop. This technique of implementing any HLA class with a single language class is called *dynamic messaging*.

Table 1: Basic Operations of Dynamic Messaging

Operation	Code Example
Instance Creation	InteractionRoot messageInstance = new InteractionRoot("full-class-name")
Set parameter value	messageInstance.setParameter("parameter-name", value)
	messageInstance.setParameter("full-class-name", "parameter-name", value)
Get parameter value	messageInstance.getParameter("parameter-name")
	messageInstance.getParameter("full-class-name", "parameter-name")

With dynamic messaging, the InteractionRoot and ObjectRoot language classes can implement any HLA class and only need to be compiled into the network simulation executable once. In addition, the network simulation can operate on HLA messages using the generic interface defined in Table 1 regardless of the actual federation data model. This allows implementation of features in network modules (e.g., a network attack that scrambles the property values based on their type) that are reusable across all federations.

Fundamental to dynamic messaging is an enhanced federation execution definition (FED) file that specifies type information for the properties of all the HLA classes. The FED file is an alternative specification of the Federation Object Model used in some HLA implementations that defines the names and properties of all the HLA

[3]The getter/setter without HLA 'full-class-name' will search for the parameter named 'parameter-name' starting from the implemented HLA class and working its way up the inheritance hierarchy. The getter/setter with HLA 'full-class-name' searches for the parameter named 'parameter-name' starting from this HLA class. Note that this HLA class must be in the inheritance hierarchy of the implemented HLA class. This allows access to a parameter that is defined higher in the inheritance hierarchy but is shadowed by parameter of the same name defined lower in the inheritance hierarchy.

classes required to run a federation. This file, formatted in JSON, was extended to include type information as shown in Figure 1. Federates (e.g., the network simulation federate) load the FED file before execution and use it to create instances of domain-specific messaging classes through the base messaging classes. Thus, a base messaging class can dynamically define this HLA class within itself, recording all the properties needed for the class and querying the RTI for the handles that pertain to the class and its properties.

4.2 Embedded Messaging

Though dynamic messaging allows a network simulator to work with any HLA class without being recompiled, it remains to describe how the simulator receives and sends interactions and object updates. The simulator must know which HLA classes it needs to handle (subscribe to and publish) in its current federation.

One way to configure the network simulator with the classes it must handle is to use a configuration file that contains the full names of these classes. The simulator can read this file at startup, and register the classes specified in the file. However, it is cumbersome to update this file if the Federation Object Model (FOM) [1] changes over the development of the federation. In addition, propagating object updates through the simulated network is not possible using this method. An HLA object instance can be thought of as a form of shared memory available to the federation. All federates have read access to this memory location, but only one federate designated as the owner of the object instance can write to and update the memory location. The following sequence of steps needed to propagate object updates via the simulated network illustrates the problem:

(1) A federate sends an update of an HLA object instance it owns to the federation.
(2) The network simulator, having used a configuration file to dynamically subscribe to the class of this HLA object instance, intercepts the update and propagates it through the simulated network.
(3) Once the update arrives at its destination in the simulated network, the network simulator tries to relay the update to the rest of federation on behalf of the original sender.
(4) The federation prevents the network simulator from sending the object update, because it does not own the object to which the update pertains.

To address this problem, a novel technique called embedded messaging was developed which utilizes a special interaction, called EmbeddedMessaging, that acts as a level of indirection when sending or receiving HLA interactions and object updates. The EmbeddedMessaging interaction embeds another HLA message within itself and contains two parameters whose values are pertinent to transmitting and processing its embedded message:

- ***command:*** Indicates the type of message being carried, as well as how it should be processed.
- ***messagingJson:*** Embeds the JSON-encoded HLA message.

The parameter “command” generally indicates whether the embedded message is an interaction or object update but could also be used to indicate additional information for more specific processing. Interactions and object updates are easily encoded into the “messagingJson” parameter due to the fact that, as described earlier, the property values are stored in a map – this map simply needs to be translated into JSON. The full class name of the embedded message is stored together with its properties inside messagingJson. Note that additional parameters in the EmbeddedMessaging interaction can allow it to be filtered by federates so that it is received only by those for which it is intended.

Importantly, the EmbeddedMessaging interaction should not be used directly by a federate. This means that a programmer writing the federate code should not try to send or receive the EmbeddedMessaging interaction. Rather, when an HLA message configured for network simulation is sent to another federate, the federate code automatically wraps the message in an EmbeddedMessaging interaction and sends this modified message to the federation. Similarly, when an EmbeddedMessaging interaction is received by a federate from the federation, the message it contains is automatically unwrapped and presented to the federate as though it were received directly from the RTI.

For interactions, the advantage of embedded messaging is that the network simulator needs to register (publish/subscribe) only the EmbeddedMessaging interaction with the RTI because all messages sent to the network simulation will be embedded in this interaction class. Consequently, the aforementioned configuration file is no longer needed. This approach also solves the problem with trying to propagate object updates via a simulated network:

(1) A federate sends an update of an HLA object instance it owns embedded in an EmbeddedMessaging interaction to the federation.
(2) The network simulator receives the EmbeddedMessaging interaction, unwraps the object update and sends it through its simulated network.
(3) Once the update arrives at its destination in the simulated network, the network simulator wraps the object update into an EmbeddedMessaging interaction and sends this interaction to one or more downstream federates.
(4) Downstream federates that receive the EmbeddedMessaging interaction automatically unpack the embedded object update and process it as a normal object update.

Through use of EmbeddedMessaging, the network simulator only needs to send interactions to relay information to the federation. This solves the object ownership issue from the prior case, because the network simulator never tries to send a direct object update.

Note that there is very little computational overhead incurred in using dynamic and embedded messaging. For dynamic messaging, the overhead involved is the table lookup for parameter and attribute values, and for embedded messaging, the overhead is the JSON encoding and decoding of a message into and from an embedded messaging interaction. In both cases, the overhead is small compared to network latencies in sending the messaging across the HLA. Importantly, it does not lead to logical time delays.

4.3 Network Simulation Federate

The network simulation federate was implemented using the INET Framework [11] which is a network modeling framework for the OMNeT++ network simulator [19]. INET consists of several C++ classes that are compiled into modules, where each module represents a TCP/IP component such as network hardware or protocol layers. To simulate a network, these modules are brought together

```
network OmnetFederateTestNetwork {
 parameters:
  @display("bgb=398,294");
  host*.numApps = 1;
  host*.app[0].typename="BasicUdpAppWrapper";
  host*.ipv4.ip.typename="CPSWTIpv4";
  router*.ipv4.ip.typename="CPSWTIpv4";
 submodules:
  configurator: Ipv4NetworkConfigurator {@display("p=174,49");}
  hlaInterface: HLAInterface {@display("p=49,49;i=block/dispatch");}
  host1: StandardHost {@display("p=48,165;i=,yellow");}
  host2: StandardHost {@display("p=348,165;i=,yellow");}
  router1: Router {@display("p=198,165;i=old/router2,lavender");}
 connections: ...
}
```

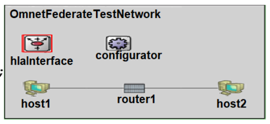

Figure 2: Network Modeling in OMNeT++

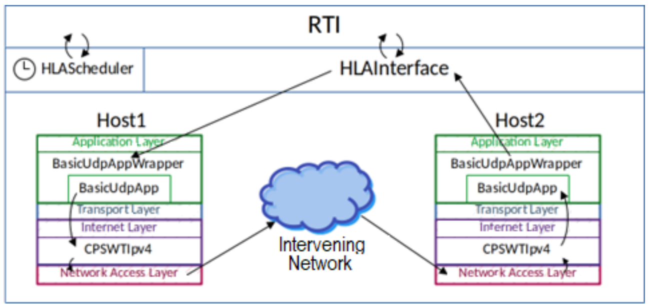

Figure 3: Reusable OMNeT++ Network Federate

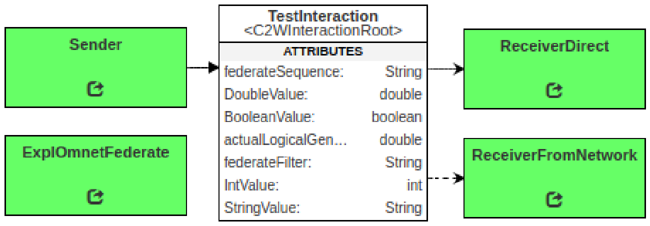

Figure 4: Federation Modeling in CPSWT

into an executable, and the network architecture is specified using a NED (NEtwork Description) file. An example of a NED file is shown in Figure 2, along with a graphical depiction of the network it describes.

To include a simulated INET/OMNeT++ network in a federation, two new modules were developed in C++ and compiled into the executable. The first is the HLAScheduler module, which synchronizes the OMNeT++ simulation time with logical time in the federation. The second is the HLAInterface module, which sends and receives HLA messaging to and from the RTI. The resulting executable that contains these two modules is called the *OmnetFederate*.

A few additional modules are required to allow a message received by the *HLAInterface* to be sent through the simulated network, and be routed back to the HLAInterface once the message has transited the network. One of these is the *BasicUdpApp* module, which acts as an application layer program that sends and receives network packets from the simulated network. Another module, *BasicUdpAppWrapper*, is derived from the BasicUdpApp and extends its functionality to accept messages from and send messages to the HLAInterface (see Figure 3).

4.4 Reusable Cyber-Attack Library

The network simulator in a federation is used to model the flow of messages (information) through a network in which federates correspond to hosts. Given this, it is important to model disruptions in this network, and in particular disruptions that are caused by network attacks.

Modeling of network attacks utilizes the modules HLAInterface, BasicUdpAppWrapper, and BasicUdpApp. In addition, a module called CPSWTIPv4 (see Figure 3) is an IP-layer module to model network attacks at the IP layer (e.g., a denial of service (DOS) attack). These attacks are triggered by sending the OmnetFederate a specific interaction that contains the parameters needed to start, stop, and control specific aspects of the attack, including which hosts participate in or are affected by the attack. The attacks are maintained by a separate class called the *AttackCoordinator*, which stores the attack parameters and can be queried by the OmnetFederate modules implementing the attack. Appendix A provides a list of cyber-attacks currently modeled in CPSWT.

4.5 Deploying OmnetFederate in CPSWT

As previously described, CPSWT [15] [16] enables modeling and synthesizing HLA federations for CPS co-simulations. It is implemented in WebGME [8] - a generic web-based graphical metamodeling environment for creating rich, domain-specific modeling languages (DSMLs). CPSWT provides such a DSML for HLA federations. Figure 4 shows a Federation Object Model (FOM) in CPSWT. Here, the federates are in green, and the interaction that will be sent between them (i.e., "TestInteraction") is shown in a white box. The interaction shows all of the types of parameters it can have. The solid-arrowed-line from the *Sender* federate to *TestInteraction* indicates that the *Sender* publishes *TestInteraction* via the Portico RTI. The solid-arrowed-line from *TestInteraction* to the *ReceiverDirect* federate indicates that *ReceiverDirect* subscribes to *TestInteraction* via the RTI. The dashed-arrowed-line from *TestInteraction* to the *ReceiverFromNetwork* federate indicates that this federate will also subscribe to *TestInteraction*, but will receive it indirectly via a simulated network, in this case the network in the OmnetFederate, named *ExplOmnetFederate*.

CPSWT also provides several model interpreters (using JavaScript and Python) that can interpret and validate the federation models and generate artifacts (e.g., Java/C++ code, scripts, configuration files). These artifacts can be modified by developers to implement the desired simulation behavior. For simulators, CPSWT generates wrapper code that makes them compliant with the HLA standard and allows them to be executed in an integrated manner. In this example, the code generated for *ReceivedFromNetwork* federate will not subscribe it to *TestInteraction* directly from the RTI, but via embedded messaging. This is done automatically without involving the modeler or federate code developer.

CPSWT also provides a courses of action (COA) evaluation modeling language for scenario-based execution [14] [13]. The COAs can utilize attacks from the cyber-attack library and inject them into the running simulation depending on the cyber scenario that is being evaluated. The reusable network simulation component supports simulating these cyber-attacks by design.

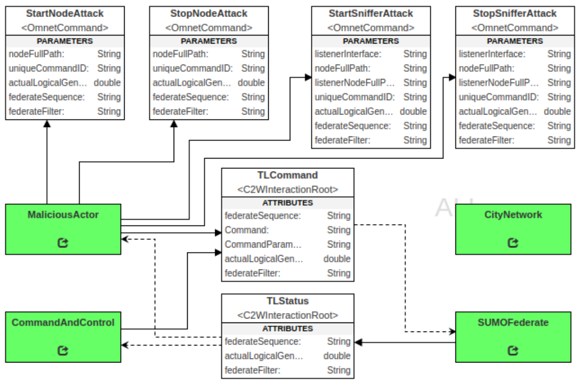

Figure 5: FOM Model for Traffic Scenario

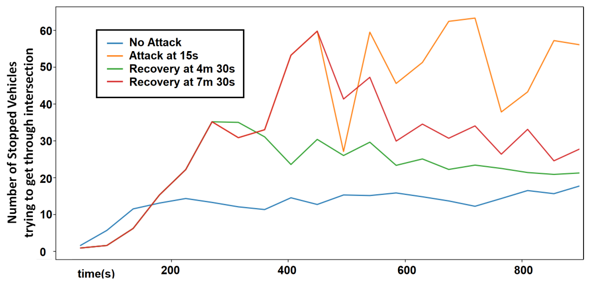

Figure 6: Experiment Results with/without Cyber-Attack

5 CASE STUDY

A concrete example of using the OmnetFederate is for simulating the effects of attacks on the networking infrastructure of smart cities. Such a simulation would use the OmnetFederate to model the city's network in conjunction with a federated city simulator, and federates to present command and control and bad actors trying to perpetrate attacks on the city network. This case study uses the city simulator called SUMO (Simulation for Urban Mobility) [10]. A simple SUMO model of a section of a city involving four traffic light intersections was used.

Figure 5 shows the CPSWT model for this case study. Federates represent the smart city's network (*CityNetwork*), the city itself (*SUMOFederate*), command and control (*CommandAndControl*) and a malicious actor (*MaliciousActor*). Both *CommandAndControl* and *MaliciousActor* publish an interaction that controls traffic lights (*TLControl*), and subscribe, through the network, to an interaction that relays the status of traffic lights. The *SUMOFederate*, which models the city, subscribes to *TLCommand* through the network, and publishes *TLStatus*. In addition, the *MaliciousActor* publishes the interactions that enable it to launch denial of service attacks (called *StartNodeAttack* in the model) and sniffer attacks on the simulated network in the *CityNetwork* (i.e., OmnetFederate). Importantly, the OmnetFederate implicitly subscribes to all of the interactions that start/stop/control network attacks.

The scenario for an attack on the city's traffic lights is:

- *MaliciousActor* launches a sniffer attack on the city network to receive *TLStatus* information.
- *MaliciousActor* then launches a denial of service attack on a network node to prevent *CommandAndControl* from receiving *TLStatus* and from sending *TLControl* commands.
- *MaliciousActor* sends a "spoofed" *TLControl* command to change one or more traffic lights to blinking yellow for one direction and blinking red for the other.
- *MaliciousActor* receives confirmation of attack success via *TLStatus* information.
- *CommandAndControl* is unaware of the change to the traffic lights operation because it is no longer receiving *TLStatus* information due to the denial of service attack.
- *CommandAndControl* receives verbal notification of the attack, but is unable to send *TLCommand* to correct the problem due to the denial of service attack.

The scenario continues with *CommandAndControl* executing countermeasures to the attack and sending *TLControl* commands to try to clear the resulting traffic congestion more quickly. Figure 6 shows the experiment results. Here, the bottom blue line represents normal operation and shows the number of cars waiting to pass intersection increases as simulation starts, but then stays around 15-20. The orange line shows when the system does not recover from attack so the number of cars waiting does not decrease. The other two cases show that when the system does recover from attack, the number of cars waiting gradually normalizes after recovery.

6 CONCLUSION & FUTURE WORK

CPS co-simulations in the real-world are continuously evolving with wide-ranging application contexts, which require a reusable network simulation that can be quickly customized to the varying needs. This is lacking in existing approaches due to the complexity of network simulation and evolving nature of CPS applications and their co-simulations. This paper presented a novel approach to create such a reusable OMNeT++ network simulation based on techniques of embedded messaging and conformance with the IEEE HLA standard. It includes a reusable cyber-attack library, and a courses of action tool that deploys these attacks with required configurations to create various cyber scenarios to explore system security and resilience.

Future work focuses on added support for multiple network simulations to enable simulation of multiple connected networks needed either by design (e.g., intranet with internet, or classified with open) or for scalability of simulation. Further, the cyber-attack library will be extended through creation of a cyber-defense library that will further enhance the capability of CPSWT for experimentation using various cyber-gaming scenarios. Other extensions include network simulation with variable resolution that dynamically varies its fidelity level during co-simulation execution.

ACKNOWLEDGMENTS

This work at Vanderbilt University is supported in part by awards #70NANB21H164 & #70NANB22H216 from the US National Institute of Standards and Technology (NIST). Official contribution of the NIST is not subject to copyright in the US. 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] 2010. IEEE Std 1516-2010, IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)- Framework and Rules. (2010). https://doi.org/10.1109/IEEESTD.2010.5553440

[2] 2019. FMI-standard. https://fmi-standard.org/

[3] 2023. Portico Run-Time Infrastructure. https://github.com/openlvc/portico

[4] Deborah A Fullford. 1996. Distributed interactive simulation: its past, present, and future. In *Proceedings of the 28th conference on Winter simulation*. 179–185.

[5] Emanuele Galli, Gaetano Cavarretta, and Salvatore Tucci. 2008. HLA-OMNET++: An HLA compliant network simulator. In *2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*. IEEE, 319–321.

[6] Edward R Griffor, Christopher Greer, David A Wollman, and Martin J Burns. 2017. Framework for cyber-physical systems: Volume 1, overview. (2017). https://doi.org/10.6028/NIST.SP.1500-201

[7] Janos Sztipanovits. 2007. Composition of Cyber-Physical Systems. *in Proc. of the 14th Annual IEEE Int. Conf. & Workshops on the Engineering of Computer-Based Systems (ECBS'2007)* (2007), 3–6.

[8] Tamás Kecskés, Qishen Zhang, and Janos Sztipanovits. 2017. Bridging Engineering and Formal Modeling: WebGME and Formula Integration.. In *MODELS (Satellite Events)*. 280–285.

[9] Xenofon Koutsoukos, Gabor Karsai, Aron Laszka, Himanshu Neema, Bradley Potteiger, Peter Volgyesi, Yevgeniy Vorobeychik, and Janos Sztipanovits. 2017. SURE: A modeling and simulation integration platform for evaluation of secure and resilient cyber–physical systems. *Proc. IEEE* 106, 1 (2017), 93–112.

[10] Daniel Krajzewicz, Jakob Erdmann, Michael Behrisch, and Laura Bieker. 2012. Recent development and applications of SUMO-Simulation of Urban MObility. *International journal on advances in systems and measurements* 5, 3&4 (2012).

[11] Levente Mészáros, Andras Varga, and Michael Kirsche. 2019. Inet framework. *Recent Advances in Network Simulation: The OMNeT++ Environment and its Ecosystem* (2019), 55–106.

[12] Himanshu Neema. 2018. *Large-Scale Integration of Heterogeneous Simulations*. Ph. D. Dissertation.

[13] H. Neema, G. Karsai, and A. H. Levis. 2015. *Next-Generation Command and Control Wind Tunnel for Courses of Action Simulation.* Technical Report no. ISIS-15-119. Institute for Software-Integrated Systems, Vanderbilt University.

[14] Himanshu Neema, Bradley Potteiger, Xenofon Koutsoukos, Gabor Karsai, Peter Volgyesi, and Janos Sztipanovits. 2018. Integrated simulation testbed for security and resilience of cps. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 368–374.

[15] Himanshu Neema, Thomas Roth, Chenli Wang, Wenqi Wendy Guo, and Anirban Bhattacharjee. 2022. Integrating Multiple HLA Federations for Effective Simulation-Based Evaluations of CPS. In *2022 IEEE Workshop on Design Automation for CPS and IoT (DESTION)*. IEEE, 19–26.

[16] Himanshu Neema, Janos Sztipanovits, Cornelius Steinbrink, Thomas Raub, Bastian Cornelsen, and Sebastian Lehnhoff. 2019. Simulation integration platforms for cyber-physical systems. In *Proceedings of the Workshop on Design Automation for CPS and IoT*. 10–19. https://doi.org/10.1145/3313151.3313169

[17] Gerardo Pardo-Castellote. 2003. OMG data-distribution service: Architectural overview. In *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings*. IEEE, 200–206.

[18] Thomas Roth, Cuong Nguyen, Martin Burns, and Himanshu Neema. 2020. Integrating a Network Simulator with the High Level Architecture for the Co-Simulation of Cyber- Physical Systems. 2020 Simulation Innovation Workshop, Orlando, FL. https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=929390

[19] A Varga. 2001. The OMNeT++ Discrete Event Simulation System. *In: Proceedings of the European Simulation Multiconference (ESM'2001)* (2001).

[20] Fan Zhang and Benxiong Huang. 2007. HLA-based network simulation for interactive communication system. In *First Asia International Conference on Modelling & Simulation (AMS'07)*. IEEE, 177–180.

A REUSABLE CYBER-ATTACK LIBRARY

The table below lists the various cyber-attacks that are modeled in CPSWT framework's reusable cyber-attack library.

Table 2: Cyber-Attack Library in CPSWT

Attack Name	**Attack Description**
Denial of Service	Targeted host is unable to receive or send HLA messages.
Out of Order Packets	On the compromised host, for HLA messages that sent and received by a specified host pair, messages received from the sending host are intentionally sent out-of-order to the receiving host.
Replay	The compromised host records a sequence of HLA messages that are being sent and received by a specified host pair. During the attack, the compromised host repeatedly sends this sequence of messages to the receiving host while dropping packets from the sending host.
Integrity	The compromised host corrupts the property (parameter or attribute) values of every HLA message it encounters according to a specified formula.
Modify to HLA Packets	The compromised host modifies HLA messages it is going to send to the HLAInterface (so the host is at the end of the transit of the message through the simulated network) by setting the properties of the message to their default values.
Modify from HLA Packets	The compromised host modifies HLA messages it is has received from the HLAInterface (so the host is at the start of the transit of the message through the simulated network) by setting the properties of the message to their default values.
Delay	The compromised host delays HLA messages passing through it by a specified amount of time.
Sniffer	The compromised host sends copies of HLA messages being sent and received by a specified host pair and sends the copy to a specified host.