HELICSAuto: Automating the Development of Cyber-Physical Co-Simulation Framework for Smart Grids

Sayeb Mohammad Tadvin
University of Rhode Island
Kingston, Rhode Island, USA
tadvin@uri.edu

Dong Jin
University of Arkansas
Fayetteville, AR, USA
dongjin@uark.edu

Hui Lin
University of Rhode Island
Kingston, Rhode Island, USA
huilin@uri.edu

ABSTRACT

Co-simulation is a powerful technique integrating various simulation tools to create a unified simulation environment. It provides an in-depth understanding of the interplay between cyber and physical infrastructures in industrial control systems like smart grids. HELICS is a framework that facilitates co-simulation development by providing common interfaces to enhance simulators, synchronize their executions, and exchange information. In this paper, we propose HELICSAuto, a code instrumentation procedure that automates the integration of domain-specific simulators with HELICS APIs. HELICSAuto requires developers to label their source codes using a pre-defined syntax, after which an interpreter automatically instruments the code with minimal manual involvement. We demonstrate the effectiveness of HELICSAuto by successfully applying it to simulators based on PandaPower, PowerWorld, OPAL-RT, and PyDNP3 to create a transmission-distribution-communication co-simulation environment for complex smart grids.

CCS CONCEPTS

• **Computer systems organization → Embedded and cyber-physical systems**.

KEYWORDS

Smart Grids, Co-Simulation, Cyber-Physical Systems

ACM Reference Format:
Sayeb Mohammad Tadvin, Dong Jin, and Hui Lin. 2023. HELICSAuto: Automating the Development of Cyber-Physical Co-Simulation Framework for Smart Grids. In *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '23), June 21–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3573900.3591118

1 INTRODUCTION

Evolving from traditional power grids, modern smart grids are becoming increasingly complex. Multiple agents in distribution and transmission systems are replacing the traditional bulky grid. Embedded computing devices, IP-based communications networks, and physical processes continuously interact to ensure efficient and reliable services, constructing today's cyber-physical infrastructure. However, profound differences between the characteristics of cyber and physical components present enormous challenges in analyzing grids' runtime states and planning for long-term stability.

Research communities and industrial sectors heavily rely on co-simulations to obtain an in-depth understanding of the coupling between cyber and physical infrastructures in smart grids. Co-simulation integrates multiple models, which are developed independently for domain-specific tasks, to create a new shared and holistic simulation environment. According to some co-simulation standards and their reference models, e.g., Functional Mock-up Interface (FMI), High-Level Architecture (HLA), and AIOMAS library [5, 11, 20], co-simulation usually includes three logical components abstracted out from detailed implementations, i.e., Time Regulation, Synchronization, and Data Exchange. These components serve as the foundation to allow each involved domain-specific simulator to iterate according to its execution steps but converge at synchronization points, where simulation states are exchanged between them.

After identifying shortcomings in the existing co-simulation environments, Pacific Northwest National Laboratory (PNNL) developed the Hierarchical Engine for Large-scale Infrastructure Co-Simulation (HELICS). Designed to integrate various power grid environments, HELICS enables high-performance cyber-physical co-simulation to reveal complex dependencies between transmission, distribution, and communication components [15]. A recent study shows that HELICS was more scalable than other co-simulation frameworks, e.g., up to 100x more scalable than Mosaik by supporting one million simulation instances [3]. To create "federates" that can be orchestrated under the HELICS co-simulation federation, HELICS provides development APIs to instrument domain-specific simulators. However, this procedure can be manual, repetitive, and error-prone, especially when creating large-scale co-simulations.

In this paper, we introduce HELICSAuto, a code instrumentation procedure that automates co-simulation development based on HELICS. HELICSAuto leverages the fact that HELICS conforms to the general co-simulation standard, which enables us to map specific APIs with standard co-simulation components. With HELICSAuto, simulator developers can mark the location of the Time Regulation, Synchronization, and Data Exchange in the source code following a simple and predefined syntax. HELICSAuto then automates the instrumentation of specific HELICS APIs for corresponding functionalities, creating federates that HELICS can directly orchestrate. Each federate can iterate based on its time step while communicating with the HELICS orchestrate (a HELICS broker) to ensure that all federates converge (e.g., data exchange) at synchronization points before continuing their simulations. HELICSAuto significantly reduces the manual efforts in creating co-simulation federates and significantly reduces the learning curve of HELICS APIs. Furthermore, if HELICS changes its API signatures (e.g., API

SIGSIM-PADS '23, June 21–23, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0030-9/23/06…$15.00
https://doi.org/10.1145/3573900.3591118

name or the parameter types), HELICSAuto can automatically update the source code with the new APIs.

In our evaluations, HELICSAuto has demonstrated the successful implementation on the simulator written in Python or can be controlled through a Python API, e.g., PandaPower [16], PowerWorld (enhanced with SimAuto Python API) [17], OPAL-RT [14], and PyDNP3 library [18]. With minimal manual involvement, we successfully use HELICSAuto to create a complicated transmission-distribution-communication co-simulation for today's smart grid environment.

2 BACKGROUND

As shown in Figure 1, HELICS is a co-simulation framework for power grids' cyber-physical infrastructure. It is capable of running up to 1 million simulation instances [3], making it feasible for a real-world power grid case [4]. HELICS instruments domain-specific simulators in their source codes to achieve functionalities such as Time Regulation, Synchronization, and Data Exchange. Each simulator participating in the HELICS co-simulation is referred to as a federate, which iterates based on its time steps. When encountering a *synchronization point* set through the HELICS API (also referred to as *convergence points*), the federate blocks its local simulation and inquires the HELICS broker to move on to the next iteration. Only when the broker receives requests from all participated federates does it send the responses to the federates, unblocking their local simulation. Federates can maintain specific data exchange orders by performing read or write data before or after the synchronization points through a publish-subscribe mechanism. Following this procedure, HELICS iterates federates' simulations at the discrete synchronization points.

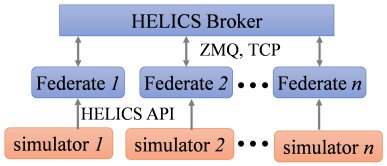

Figure 1: High-level logic in HELICS.

Comparison with Distributed Simulation. Similar to a co-simulation framework, distributed simulation also leverages multiple agents to achieve simulation tasks. However, different from co-simulations that focus on how to orchestrate involved simulators in terms of data exchange, distributed simulation requires algorithms to ensure fine-grained ordering among data consumed and produced by each agent. There are two types of parallel distributed simulation algorithms [1]. The *optimistic* algorithm allows each agent to continue the simulation without blocking until a conflict emerges and requires some agents to roll back to an error-free state. On the other hand, the *conservative* algorithm blocks agents at a certain time stamp and only allows them to continue simulations if doing so introduces no conflicts [13].

HELICS' synchronization logic shares some similarities with conservative distributed simulation algorithms. *We emphasize two major differences.* First, coordinated by HELICS broker, federates are synchronized at specific locations in simulators' source code; it does not have a global virtual time scheduling all federates execution. Second, HELICS does not provide algorithms to define the correct data dependency among different federates within the co-simulation; it leaves developers to determine the appropriate locations to insert its APIs to consume or generate data values.

3 RELATED WORK

Some existing studies develop various designs and implementations related to modeling, analyzing, and optimizing cyber-physical infrastructures of power grids. For example, studies in [6, 21] present comprehensive surveys. The existing co-simulation of the cyber-physical smart grids mainly consists of three layers. The *physical layer* models the physical interconnection of electronic devices. The *network layer* interconnects sensors and actuators. The *application layer* enables human operations.

In addition, various research work has attempted to enhance the fidelity, functionality, and performance of each layer of the smart grid co-simulation testbed. For example, Lin et al. have explored various implementations, such as network emulation, cloud environments, and hardware switches, to achieve realistic communications in a laboratory setting [12]. Some testbeds incorporate software-defined networking (SDN) techniques to enable network programmability [7, 8]. Commercial tools present interfaces that incorporate real electronic devices into simulations in the physical layer. The Real-Time Digital Simulator (RTDS) [19] and OPAL-RT [14], for instance, allow for real-time event injection without the need to reboot simulations, providing more dynamic simulation capabilities. In the application layers, industrial sectors and research communities have different priorities. The former focuses on developing a friendly graphical user interface, while the latter prefers a scripting interface to automate research tasks.

Apart from enhancing individual layers, the primary objective of HELICS is to increase the efficiency of coupling different simulation platforms, rather than providing specific simulation techniques. This work aims to provide additional support to HELICS by automating the process of instrumenting simulation source codes with the appropriate APIs, reducing error-prone manual efforts.

4 HELICSAUTO DESIGN

We present HELICSAuto, a tool that automates federate creation by instrumenting domain-specific simulators with HELICS APIs. Figure 2 illustrates the process. Developers are required to label their simulators with common co-simulation functionalities such as Time Regulation, Synchronization, and Data Exchange. These labels are added as comments for two reasons. First, developers can comfortably add the labels without affecting the execution of the original simulator. Second, HELICSAuto can define a consistent syntax of the labels, irrespective of the implementation language used by the domain-specific simulators. At present, HELICSAuto can instrument simulators written in Python, which is commonly used for physical process simulators, particularly power grid simulators. It is worth noting that because HELICS provides C++ and Python APIs, HELICSAuto can be extended to support simulators built on other programming languages supported by HELICS.

Table 1 outlines the syntax defined in HELICSAuto to label simulators based on common co-simulation functionalities defined in [20]. The labels *#HA:Register* and *#HA:Destroy* correspond to HELICS APIs, such as *helicsCreateValueFederateFromConfig* and *helicsFederateDisconnect*, that establish and terminate connections

Table 1: HELICSAuto labels used to mark simulators.

Functionality	Syntax	Explanation
Time Regulation	#HA:Register / #HA: Destroy	Make / tear down a network connection to the HELICS broker
	#HA: Execute	Start a while loop to execute a synchronized simulation
Synchronization	#HA:Sync	A synchronization point observed by all participated federates (where the federate attempts to obtain the synchronization lock)
Data Exchange	#HA:Pub, VAR_NAME, VAR_TYPE, SHARE_VAR	Publish VAR_NAME in the source code to the shared variable SHARE_VAR, which has the type VAR_TYPE
	#HA:Sub, VAR_NAME, VAR_TYPE, SHARE_VAR	Subscribe VAR_NAME in the source code to the shared variable SHARE_VAR, which has the type VAR_TYPE

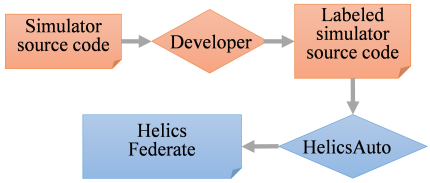

Figure 2: The design overview of HELICSAuto.

with the HELICS broker. The location marked with *#HA:Execute* is where we instrument codes that regulate simulation times, including the duration of each synchronization step and the entire synchronization process. A loop is wrapped around the original simulation codes, starting with *#HA:Execute* and ending with *#HA:Destroy*. Also, *#HA:Sync* labels the synchronization point, which is implemented by HELICS API *helicsFederateRequestTime*. The HELICS broker coordinates these synchronization points with each involved simulator blocking execution until all simulators reach the synchronization point. To exchange data, we use *#HA:Pub* and *#HA:Sub* labels along with associated parameters, enabling HELICS APIs to publish and subscribe VAR_NAME from the simulation source codes to the "SHARE_VAR" recognized by the HELICS broker to exchange data. Since APIs differ depending on the shared variable type, we explicitly include the type (i.e., VAR_TYPE) in the label to enable HELICSAuto to instrument the appropriate APIs to the source code.

HELICSAuto scans the simulator's source code and instruments the appropriate API based on the labels presented in Table 1. One challenge of instrumenting Python codes is to track the indentation in the source codes, which is used to group the same live code range. To overcome this challenge, we use a variable as a stack to track the indentation while scanning the code line-by-line. This implementation is suitable for the most common simulation procedure in power grids. To handle complex situations in more complex simulators (e.g., duplicated variables introduced by instrumentation), we will integrate HELICSAuto with a compiler-assisted tool such as Yacc, which we leave for future work.

To elucidate the procedure, we use a motivating example depicted in Figure 3, which includes two federates representing two simulation processes (named as *Simulator1* and *Simulator2*). We assume that the variable *consumption* in Simulator2 needs the value of the variable *production* updated by Simulator1. To simplify the discussion, we only show critical code snippets while using ellipses to represent other codes. Although both federates run simulations

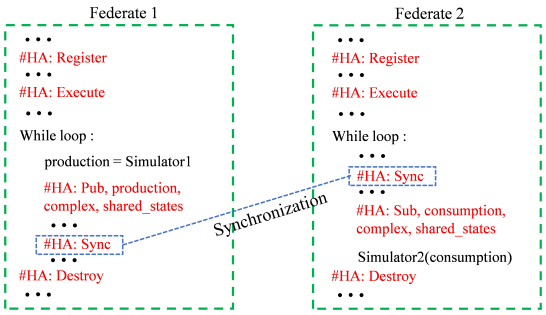

Figure 3: An example of HELICSAuto instrumenting simulator source codes.

independently, their executions are synchronized at *#HA:Sync* within each iteration of the "while" loop. After the instrumentation, we ensure that the variable *production* is updated before the synchronization point, while the variable *consumption* is used after that point. Consequently, federate 2 always uses the updated values from federate 1.

Discussion. HELICSAuto automates the process of federate creation by instrumenting simulation source codes with HELICS APIs. Developers are responsible for labeling their simulators with co-simulation functionalities and determining the appropriate locations to put the labels based on the dependencies of data exchanged in the co-simulation. Because the labels follow a simple syntax, HELICSAuto can be enhanced with a model-checking functionality to verify co-simulation logic, similar to the language proposed in [10]. This enhancement will be a future direction for our work.

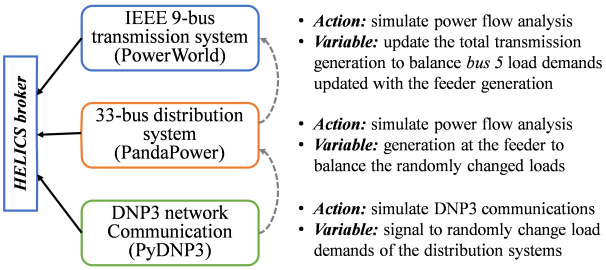

Figure 4: The setup of co-simulation in the case study.

5 CASE STUDY

In this section, we present a case study to demonstrate HELICSAuto's functionality. Figure 4 shows the co-simulation setup

Algorithm 1 Steps to Perform Distribution-Transmission-Communication Co-simulation

1: **Step 1:** The PyDNP3 federate performs a round of network communication
2: **Step 2:** The PandaPower federate changes load demands randomly and performs a steady state power flow analysis, obtaining a feeder generation
3: **Step 3:** Check for the convergence of the synchronization points, i.e., Step 1 and Step 2 are finished
4: **Step 4:** Update the load demands of Bus 5 with the feeder generation and perform a power flow analysis of the IEEE 9-bus transmission system
5: **Step 5:** if the simulation is not finished (by comparing to a pre-set counter), go to Step 1

of three federates running on three computers connected via a routable network, creating a distribution-transmission-communication co-simulation. The details of HELICSAuto's instrumentation are presented in Listing 1 and Listing 2 in the Appendix. The first federate exploits the open-source PyDNP3 library to simulate communications between a master and an outstation based on the DNP3 protocol widely used in today's power grids [18]. The second federate uses the PandaPower simulator to simulate a 33-bus distribution system from [2]. The last federate leverages the PowerWorld simulator to simulate an IEEE 9-bus transmission system. To enhance the PowerWorld with HELICS APIs, we leverage its Python SimAuto interface to configure and execute the simulation. The co-simulation follows the procedure presented in Algorithm 1. We assume the 33-bus distribution system is attached to bus 5 of the IEEE 9-bus system. Consequently, the active and reactive power provided by the feeder bus (equivalent to the slack bus) to the 33-bus distribution system is transferred as newly added load demands for bus 5. In this co-simulation, we perform a control operation in the DNP3 communication to indicate that a change happens in the distribution system, which updates the state of the transmission system.

In our experiments, we run the co-simulation for 7,200 time steps with 432,000 time units defined in HELICS (the actual latency of each time unit is specified by developers). During the simulation, we measure the actual execution time of each simulation performed in Step 1 (i.e., DNP3 network communication), Step 2 (the power flow analysis in PandaPower), and Step 4 (the power flow analysis in PowerWorld). As shown in Figure 5, because the network simulation is performed through the loopback network interface in the same machine, the round-trip time is less than 0.1 millisecond, two magnitudes smaller than the execution time of power flow analyses performed in PandaPower and PowerWorld.

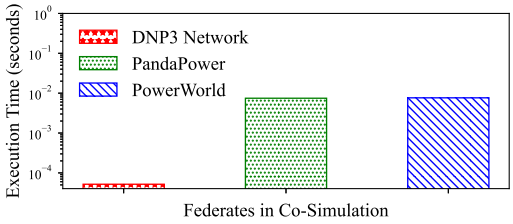

Figure 5: Simulation execution time in each federate.

Based on the measurements in Figure 5, we intentionally add a latency of 0.1 seconds in the federate for the DNP3 network simulation. Even though we add latency in one federate, the synchronization enforced by HELICS makes the simulation in all federates move forward with the same latency. Figure 6 shows the overall runtime communication patterns from the three federates, quantified by the number of packets per second during the co-simulation. By zooming in on the first two seconds of the simulation shown in Figure 7, we observe that the simulation within each federate, even though their starting times are different, moves forward with the same time unit of 0.1 second.

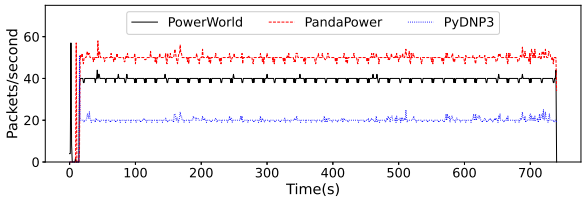

Figure 6: Network communication during co-simulation.

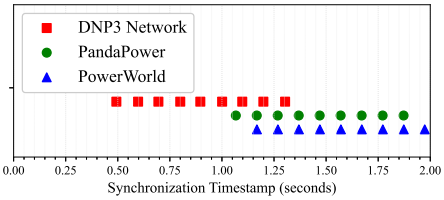

Figure 7: Synchronization timestamps in the first two seconds of three federates.

6 CONCLUSION AND FUTURE WORK

In this paper, we present HELICSAuto, a code instrumentation procedure automating the co-simulation development based on HELICS. HELICSAuto provides labels for simulator developers to mark synchronization logic in their domain-specific tools, and a line-by-line interpreter automatically instruments the source codes with HELICS APIs based on the synchronization logic. Our evaluation has demonstrated the successful use of HELICSAuto to transform simulators written in PandaPower, PowerWorld, OPAL-RT, and a PyDNP3 into federate entities executable in HELICS. In future work, we will improve HELICSAuto by (i) integrating the instrumentation functionality in the Yacc compiler framework to perform code instrumentation in more complicated simulators, and (ii) building a bridge to the model-checking module to verify co-simulation logic.

ACKNOWLEDGMENTS

This material is based upon work partially supported by the National Science Foundation under Award No. CNS-2144513, CNS-2247722, and CNS-2247721. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] J.M. Alonso, A.A. Frutos, and R.B. Palacio. 1994. Conservative and optimistic distributed simulation in massively parallel computers: a comparative study. In *Proceedings of the First International Conference on Massively Parallel Computing Systems (MPCS) The Challenges of General-Purpose and Special-Purpose Computing*. IEEE, Ischia, Italy, 528–532. https://doi.org/10.1109/MPCS.1994.367036

[2] M.E. Baran and F.F. Wu. 1989. Network reconfiguration in distribution systems for loss reduction and load balancing. *IEEE Transactions on Power Delivery* 4, 2 (1989), 1401–1407. https://doi.org/10.1109/61.25627

[3] Luca Barbierato, Pietro Rando Mazzarino, Marco Montarolo, Alberto Macii, Edoardo Patti, and Lorenzo Bottaccioli. 2022. A comparison study of co-simulation frameworks for multi-energy systems: the scalability problem. *Energy Informatics* 5, 4 (2022), 1–26.

[4] Alok Kumar Bharati and Venkataramana Ajjarapu. 2022. SMTD Co-Simulation Framework With HELICS for Future-Grid Analysis and Synthetic Measurement-Data Generation. *IEEE Transactions on Industry Applications* 58, 1 (2022), 131–141. https://doi.org/10.1109/TIA.2021.3123925

[5] T. Blockwitz, Martin Otter, J. Akesson, Martin Arnold, C. Clauß, Hilding Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, and A. Viel. 2012. Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models. In *The Proceedings of 9th International Modelica Conference*. Institute of System Dynamics and Control – Vehicle System Dynamics, Munchen, Germany. https://elib.dlr.de/78486/

[6] Mehmet Hazar Cintuglu, Osama A. Mohammed, Kemal Akkaya, and A. Selcuk Uluagac. 2017. A Survey on Smart Grid Cyber-Physical System Testbeds. *IEEE Communications Surveys & Tutorials* 19, 1 (2017), 446–464. https://doi.org/10.1109/COMST.2016.2627399

[7] Xinshu Dong, Hui Lin, Rui Tan, Ravishankar K. Iyer, and Zbigniew Kalbarczyk. 2015. Software-Defined Networking for Smart Grid Resilience: Opportunities and Challenges. In *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security* (New York, NY, USA) *(CPSS '15)*. ACM, 61–68. https://doi.org/10.1145/2732198.2732203

[8] Christopher Hannon, Jiaqi Yan, and Dong Jin. 2016. DSSnet: A Smart Grid Modeling Platform Combining Electrical Power Distribution System Simulation and Software Defined Networking Emulation. In *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '16)*. 131–142.

[9] helicsauto [n. d.]. HELICSAuto - Instrumentation for HELICS. [Online] Available at: https://github.com/hugolin615/helicsauto.

[10] Yerang Hur and Insup Lee. 2002. Distributed simulation of multi-agent hybrid systems. In *Proceedings Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISIRC 2002*. 356–364. https://doi.org/10.1109/ISORC.2002.1003781

[11] IEEE. 2010. 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)* (2010), 1–38. https://doi.org/10.1109/IEEESTD.2010.5553440

[12] Hui Lin, Bibek Shrestha, and Yih-Chun Hu. 2020. Cyber-Physical Testbed: Case Study to Evaluate Anti-Reconnaissance Approaches on Power Grids' Cyber-Physical Infrastructures. In *Proceedings of the 2020 Learning from Authoritative Security Experiment Results (LASER '20)* (San Diego, CA, USA).

[13] David M. Nicol. 1996. Principles of Conservative Parallel Simulation. In *Proceedings of the 28th Conference on Winter Simulation* (Coronado, California, USA) *(WSC '96)*. IEEE Computer Society, USA, 128–135. https://doi.org/10.1145/256562.256591

[14] opalrthil [n. d.]. Power Hardware in the Loop - OPAL-RT Technologies. [Online] Available at: https://www.opal-rt.com/power-hardware-in-the-loop/.

[15] Bryan Palmintier, Dheepak Krishnamurthy, Philip Top, Steve Smith, Jeff Daily, and Jason Fuller. 2017. Design of the HELICS high-performance transmission-distribution-communication-market co-simulation framework. In *2017 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES)*. 1–6. https://doi.org/10.1109/MSCPES.2017.8064542

[16] pandpower [n. d.]. pandapower pandapower. [Online] Available at: http://www.pandapower.org/.

[17] PowerWorld [n. d.]. PowerWorld The visual approach to electric power systems. [Online] Available at: https://www.powerworld.com/.

[18] pydnp3 [n. d.]. pydnp3 - Python bindings for opendnp3 libary. [Online] Available at: https://github.com/ChargePoint/pydnp3.

[19] rtdshil [n. d.]. Power Hardware in the Loop - RTDS Technologies. [Online] Available at: https://legacy.rtds.com/applications/phil/.

[20] Stefan Scherfke. 2014. Welcome to aiomas' documentation! - aiomas 2.0.1 documentation. [Online] Available at: https://aiomas.readthedocs.io/en/latest/.

[21] Abdallah A. Smadi, Babatunde Tobi Ajao, Brian K. Johnson, Hangtian Lei, Yacine Chakhchoukh, and Qasem Abu Al-Haija. 2021. A comprehensive survey on Cyber-Physical Smart Grid Testbed Architectures: Requirements and challenges. *Electronics* 10, 9 (2021), 1043. https://doi.org/10.3390/electronics10091043

A EXAMPLE OF APPLYING HELICSAUTO ON PANDAPOWER SIMULATION

Listing 1: Example PandaPower simulator enhanced with HELICSAuto labels.

```
import pandapower as pp
import pandapower.networks as pn
import numpy as np

net = pn.case33bw()
ori_load = net.load.loc[:, 'p_mw'].to_numpy()
n_load = ori_load.size
np.random.seed(1)
#HA:Register

#HA:Execute
load_var = np.random.randint(low = 90, high = 110, size =
      n_load) / 100
new_load = np.multiply(ori_load, load_var)
net.load.loc[:, 'p_mw'] = new_load
pp.runpp(net)
feeder_p_mw=net.res_ext_grid.loc[0, 'p_mw']
feeder_q_mvar=net.res_ext_grid.loc[0, 'q_mvar']
feeder_s = complex(feeder_p_mw, feeder_q_mvar)
#HA:Pub, feeder_s, complex, Feeder_S

#HA:Sync

#HA:Destroy
```

Listing 2: HELICS federate of the example PandaPower simulator created by HELICSAuto.

```
import pandapower as pp
import pandapower.networks as pn
import numpy as np
net = pn.case33bw()
ori_load = net.load.loc[:, 'p_mw'].to_numpy()
n_load = ori_load.size
np.random.seed(1)
#HA:Register
import helics as h
fed=h.helicsCreateValueFederateFromConfig('config.json')
#HA:Execute
h.helicsFederateEnterExecutingMode(fed)
hours = 1
total_interval = int(60 * 60 * hours)
update_interval = int(h.helicsFederateGetTimeProperty(fed
     , h.HELICS_PROPERTY_TIME_PERIOD))
grantedtime = 0
while grantedtime < total_interval:
    load_var = np.random.randint(low = 90, high = 110,
         size = n_load) / 100
    new_load = np.multiply(ori_load, load_var)
    net.load.loc[:, 'p_mw'] = new_load
    pp.runpp(net)
    feeder_p_mw = net.res_ext_grid.loc[0, 'p_mw']
    feeder_q_mvar = net.res_ext_grid.loc[0, 'q_mvar']
    feeder_s = complex(feeder_p_mw, feeder_q_mvar)
    #HA:Pub, feeder_s, complex, Feeder_S
    pubid = h.helicsFederateGetPublication(fed,
              'Feeder_S')
    status = h.helicsPublicationPublishComplex(pubid,
         feeder_s.real, feeder_s.imag)
    #HA:Sync
    requested_time = grantedtime + update_interval
    grantedtime = h.helicsFederateRequestTime(fed,
         requested_time)
#HA:Destroy
grantedtime = h.helicsFederateRequestTime(fed, h.
     HELICS_TIME_MAXTIME)
status = h.helicsFederateDisconnect(fed)
h.helicsFederateFree(fed)
h.helicsCloseLibrary()
```

In Listing 1, we show the snippets of the PandaPower simulator used in the case study. In this example, we randomly change all load demands of the 33-bus distribution systems by ±10% and then run a static power flow analysis to obtain the new generation required by the feeder to satisfy the updated load demands. Because this generation determines the load demand of bus 5 in the transmission system, we label the relevant portions of the source code to enable necessary data exchange.

In Listing 2, we present the executable HELICS federates that are created by HELICSAuto based on the source codes in Listing 1. In the federate, we execute the simulation, i.e., power flow analysis on randomly changed load demands, in a "while" loop until the total simulation time expires. After each execution of the simulation, this federate publishes the result, i.e., feeder generation, so that the other participated federate (in this case, the PowerWorld simulator) can use it. In the current implementation, we can also use HELICSAuto to label simulators implemented in PowerWorld and OPAL-RT, similar to the labeled source code shown in Listing 1.

Other federates for the PyDNP3 and PowerWorld simulators created by HELICSAuto are presented at [9]. A comparison of all these federates reveals that the instrumented HELICS APIs are common across different simulators transformed into co-simulation federates, highlighting the versatility of HELICSAuto.