Development Pitfalls: A Case Study in Developing a Smart Grid Co-simulation Platform Based on HELICS

Jeremy Frandon[1], Jun Yan[1(✉)], and Emmanuel Thepie-Fapi[2]

[1] Concordia Institute for Information Systems Engineering, Concordia University, Montréal, Canada
{jeremy.frandon,jun.yan}@concordia.ca

[2] Global AI Accelerator - AI-Hub Canada, Ericsson, Montréal, Canada
emmanuel.thepie.fapi@ericsson.com

Abstract. The recent transformation of the smart grid has led to a complex and heterogeneous cyber-physical system (CPS). The modernizing electric power infrastructure is equipped with multiple systems for sensing, communicating, controlling, and processing an extensive volume of data, for which a platform or a testbed mapping various components of a smart grid is extremely useful for R&D purposes. To this end, a testbed featuring one simulator will not fully characterize the functionalities of such a complex infrastructure. A Co-simulation testbed that federates loosely coupled standalone sub-simulators is appropriate and will accurately represent a Smart Grid. This paper will investigate the process and pitfalls observed in the re-development of ASGARDS-H at Concordia University, Montréal, a co-simulator based on the Hierarchical Engine for Large-scale Infrastructure Co-Simulation (HELICS) for 5G-based smart grid security. The paper will reveal the designs and modifications of the testbed under a microservice architecture via containerization. New tools and code refactorings are proposed and discussed to decouple the simulation logic from the time and value synchronization of the co-simulation. Beyond the final platform that offers improved simulation capacities and functionalities from ASGARDS-H capabilities, the paper also aims to share lessons learned and notable pitfalls that can help smart grid security researchers more effectively and efficiently develop flexible, reliable, and scalable co-simulation testbeds.

Keywords: co-simulation · testbed · smart grid · software engineering

1 Introduction

The traditional electricity grid was designed for a one-way flow of electricity from centralized power plants to customers. However, with the rise of renewable energy sources, distributed energy resources, and electric vehicles, the grid must adapt to become more flexible, reliable, and efficient. This is where the Smart Grid comes into play.

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2025
Published by Springer Nature Switzerland AG 2025. All Rights Reserved
X. Li et al. (Eds.): SmartGift 2024, LNICST 600, pp. 182–195, 2025.
https://doi.org/10.1007/978-3-031-78806-2_11

The Smart Grid refers to an advanced electricity network that integrates digital technology, communication, and sensors to improve the management, monitoring, and control of the electricity system [1]. It enables two-way communication between the grid and its users, allowing for more efficient and effective use of energy resources.

The smart grid presents many benefits, including automated fault recovery, renewable energy integration, and demand-side management. An Automated fault recovery enables the grid to quickly detect and isolate problems, minimizing downtime and improving overall system reliability. By integrating renewable energy sources like solar and wind power, the smart grid promotes sustainability and reduces greenhouse gas emissions, helping combat climate change. Additionally, demand-side management allows for more efficient electricity consumption by providing consumers with real-time data and incentives to adjust their usage during peak and off-peak hours, optimizing energy distribution and reducing costs. These advantages make the smart grid a pivotal technology in creating a more resilient, environmentally friendly, and cost-effective energy infrastructure.

However, as the smart grid becomes more complex and interconnected, it becomes increasingly difficult to model and evaluate the performance and security of the entire system accurately. Co-simulation has been proposed as a solution to this modeling problem.

Co-simulation is a method of simulation that pools domain-specific simulators to interact together and offer a coordinated and connected simulation requirement [2]. In the context of Smart Grid systems, co-simulation enables the testing and evaluating of interactions between different system domains: power flow, network communication, operational logic, and business logic.

A well-implemented co-simulation testbed can provide a more accurate representation of the Smart Grid system [3]. By incorporating real-time data and feedback from the physical system, co-simulation can simulate the system's behavior under different conditions and scenarios. This enables researchers and engineers to identify potential problems and improve the system design before implementation. Advanced co-simulation also allows for evaluating the system's performance in real time. By simulating the system's behavior under different conditions, e.g., load changes or renewable integration, co-simulation can help identify potential bottlenecks, weaknesses, or areas for improvement in the system.

This workshop paper adopts a distinctive approach by focusing on providing valuable insights into the development process of co-simulation testbeds, and it shall serve as a supplement to the existing literature discussing the existing and developed approaches. Through the analysis of a case study, this paper aims to shed light on the intricate process of creating co-simulation testbeds, delving into the underlying methodologies, challenges faced, and best practices employed during their development. By emphasizing the development process, this paper seeks to complement the understanding and knowledge base of researchers, practitioners, and developers, ultimately facilitating the creation of more robust and efficient co-simulation testbeds.

1.1 Related Works

A literature survey [3] outlined the current research trends in the smart grid co-simulation litterature and found that most published research uses purpose-built co-simulators to answer their research questions. The authors compiled a short list of simulators, co-simulation platforms, and simulation architectures. However, there appears to be no consensus about how to select the proper tools and architectures for each co-simulation project. This lack of consensus indicates the existence of major trade-offs made by different research groups during the development of any co-simulation testbed.

For example, co-simulation testbeds like [4], evaluating electric vehicle charging and smart grid security, need to implement the entire ecosystem they want to simulate. The researchers emulated the cars' mobile applications, the smart charging cloud management system, the charging stations' firmware, the human-machine interface, and the car demand for power over time while choosing not to emulate the network communication infrastructure in their design. To the contrary, testbeds like [5] put the focus on accurately simulating data transmission to evaluate the resiliency of the smart grid to cyberattacks but often fail to simulate practical applications that could be targeted by cyber-physical attacks.

On the contrary, projects using the HELICS [6] co-simulation platform seek to capture a holistic and fine-grained view of the smart grid systems they evaluate. For example, [7] extends the Transactive Energy Simulation Platform [8] to evaluate packetized energy management to coordinate energy consumption and supply balance; [9] federates an ns-3 network model with a MATLAB-based power system model to evaluate the impact of non-ideal communication network performance on a proposed Generalized Power System Stabilizer architecture; [10] synchronizes GridLAB-D and Python-based controllers to model an inverter-based microgrid, allowing them to verify a Leader-Follower Consensus (LFC) Architecture for Grid-Forming and Grid-Following Inverter Coordination.

Industry case studies in other fields of CPS discuss the implications of the co-simulation architecture on the development and deployment processes. For example, [11] finds that implementing a microservice-based architecture for DevOps enables continuous deployment, monitoring, and validation of the CPSs they studied (elevator dispatching algorithms). This microservice-based architecture is similar to the one we introduce in this paper. Their findings, which focus on DevOps and continuous deployment, provide yet another insight into CPS testbed developments.

This paper presents ASGARDS-H [12], the co-simulation platform we are building upon. We discuss its design limitations and how they impacted the development of new features. We then present how we overcome those limitations by redesigning and refactoring specific components. We conclude by generalizing our findings based on known software engineering principles and recommending adopting emerging tools for smart-grid co-simulation.

2 Case Study: ASGARDS-H Testbed

ASGARDS-H [12] is a co-simulation platform developed to enable advanced smart grid cyber-physical attacks, risk, and data studies. It aims to provide a modular, complete, and scalable solution for generating standardized datasets for research and development in smart distribution grid security. The platform utilizes the HELICS [6] co-simulation framework and offers capabilities for generating realistic scenarios, including instabilities, faults, and cyber-physical attacks. The generated datasets can be used to develop data-driven approaches and advanced machine learning techniques for enhancing smart grid security. ASGARDS-H is designed to be user-friendly and allows for the customization and extension of its capabilities. It is being developed by a research team at the Concordia Institute for Information Systems Engineering as part of the Ericsson GAIA program and holds potential for future studies on cyber-physical attacks in smart grids.

We chose ASGARDS-H because, compared to other platforms, it is a recently developed platform, its source code was available to us, and [12] commented heavily on its modularity and extensibility. Our team decided to do some development work on top of ASGARDS-H to increase its capabilities and remedy some of its initial design limitations. In this section, we present our understanding of ASGARDS-H design considerations and limitations. In Sect. 3 we present the software engineering challenges encountered when working on ASGARDS-H and how they relate to documented problems in the Evidence-Based Software Engineering literature. In Sect. 4, we document our project-specific solution. This paper aims to provide insight into the development process of a co-simulation testbed by outlining pitfalls to avoid, and possible solutions.

2.1 Design Considerations

The main design consideration was to create a usable testbed capable of generating truthful data about how a smart grid would react under different cyber-attacks. Much thought was put into the usability of the testbed configuration system, with a well-integrated project-generation graphical user interface and both a visual and machine-readable output interface.

ASGARDS-H used the capabilities of its federated simulators, GridLAB–D [13] and OMNeT++ [14] to simulate scenarios that include power grid faults, manual cyberattacks, manual physical attacks, configurable cyber-physical attacks, weather events, and network events. The capacities were built by leveraging the integration between HELICS and GridLAB-D and implementing OpenADR [15] and phase measurement unit communication over an LTE network in OMNeT++. The supporting interfaces and API layers for attack generation and testing were written in Python.

2.2 Design Limitations

In its first development cycle, ASGARDS-H accumulated two elements of technical debt that would hinder its development progress. The first one was that

the project depended on specific software libraries, which are only available for some Linux distributions. The development team shall work on and deploy ASGARDS-H as a virtual machine. Increasing the development overhead.

The other element of technical debt was the highly coupled code structure. Where the project generator would generate OMNeT++ .ned files and GridLAB–D .glm files with hard-coded values that would be depended on by HELICS and the other Python modules, this breach of the separation of concerns principle makes it difficult for the development team to modify one part of the code without having to attend to all other parts.

3 Software Engineering Challenges

As part of its development lifecycle, our efforts with ASGARDS-H faced multiple challenges that are well-documented in the software engineering literature. The first one was a change in requirements to support new features. This kind of requirement change (adding or changing the scope of the work) is the most common type of requirement change [16].

The second development challenge was turnover-induced knowledge loss. As the main developer in our team left the project in 2021, the new developer onboarded to continue the project had to recover that knowledge using various techniques. The new developer relied on the available documentation, interviewed colleagues who had contributed to the project, and attempted to recreate the knowledge from the codebase. Those techniques are consistent with how most teams deal with turnover-induced knowledge loss [17].

3.1 Change in Requirements

The change in requirement was extending the scope of capabilities of the co-simulation platform to include the ability to simulate a new smart grid capability, such as feeder automation or a 5G cellular network. This new scope requires additional capabilities that should be supported by the power and network simulators: feeder switches and relays, fault-current detection, GOOSE messaging, and 5G user-plane communication.

However, some of those capabilities were not supported by the previous deployment. As the most up-to-date 5G simulator, Simu5G [18] was only compatible with a newer version of OMNeT++. The team had difficulties getting millisecond-level transient values from GridLAB-D for the fault-current detection. The change in requirement, therefore, necessitated a major update to multiple components of ASGARDS-H.

Moreover, the tight coupling of the different pieces of code made it difficult to upgrade ASGARDS-H incrementally. Extensive03260997f refactoring and integration efforts were required to accommodate the new capabilities and ensure seamless communication between the power and network simulators. The team had to carefully design and implement the necessary changes to prevent any adverse effects on the overall performance and stability of the co-simulation platform.

3.2 Turnover-Induced Knowledge Loss

Within the development of ASGARDS-H, a significant hurdle was encountered as the main developer departed, accompanied by changes in project requirements, which created difficulties in adapting and delivering the new features. For a large-scale multi-year co-simulator development project, similar key personnel changes are often inevitable. They can cause substantial consequences, as they disrupt the continuity of knowledge keeping/transfer and hampers the understanding of the intricacies involved in implementing complex functionalities [19]. In the case of ASGARDS-H, replacing the main developer resulted in a knowledge gap that posed challenges to the team's ability to comprehend and modify the existing codebase to meet the evolving requirements. The subsequent change in project requirements further compounded the predicament, as the team faced the daunting task of reconciling the new specifications with the existing codebase, leading to delays and potential design compromises. The departure of the main developer and the subsequent change in requirements highlight the criticality of maintaining a robust development team and implementing efficient knowledge transfer processes, ensuring the continuity of project goals and minimizing disruptions during the development lifecycle.

With the arrival of a new main developer, the development team had to rebuild the knowledge loss and generate an upgrade plan for ASGARDS-H. After understanding the technical debt, as defined by [20], of the project and the major components change needed, the team decided to use this opportunity to update the architecture of ASGARDS-H to address the modularity and dependencies issues. This update first took the form of a refactor. The implementation would change, but the user interface should remain the same. Then, the team would upgrade the different modules incrementally.

4 Project-Specific Solutions

The tightly coupled structure of ASGARDS-H was a main obstacle to updating each module incrementally, therefore the primary focus of the refactor in ASGARDS-H was to reduce coupling between the co-simulation federates by making each federate run in a separate container and within its codebase using modular programming principles.

Once each federate was properly decoupled, a concerted effort was made to refactor the existing code using modular programming techniques. The codebase became more cohesive, maintainable, and extensible by breaking down monolithic components into smaller, self-contained modules. This refactoring approach reduced the interdependencies between different parts of the code, allowing for more granular updates and modifications.

Finally, we updated the co-simulator toolchain, in our case a co-simulation scenario generator, to work with this more decoupled design, and to increase the number of scenarios that can be generated. The combined efforts of containerization and refactoring using modular programming principles were instrumental in achieving a less coupled architecture both between the co-simulation federates

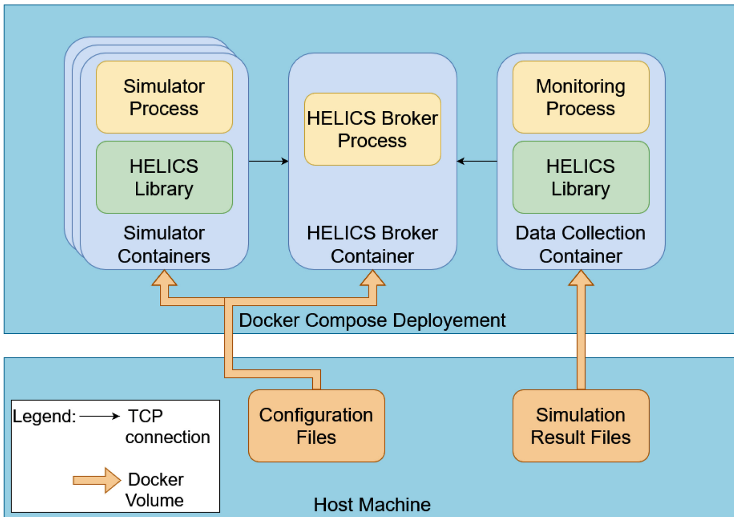

Fig. 1. Containerization architecture.

and within their own codebase, enhancing the overall flexibility, maintainability, and extensibility of the ASGARDS-H co-simulation platform.

4.1 Decoupling the Federates

Containerization is the process of packaging software code with only the operating system libraries and dependencies required to run the code. This creates a single lightweight executable called a container. Containerization was employed to encapsulate each co-simulation federate within a separate container, enabling them to operate independently while communicating through well-defined interfaces. This approach not only enhanced the scalability and portability of the platform but also reduced the dependencies and coupling between the federates, promoting flexibility and ease of integration.

The first step in the refactor was to decouple the various simulators from each other by isolating them in different containers. The separation went as such: the HELICS broker, the user interface, and each of the GridLAB-D, OMNeT++, controller, and attacker simulators were each encapsulated in a Docker [21] container, which allowed a strict separation of the project files for each simulator, as well as enforcing the HELICS broker to be the sole communication interface between the different federates.

Figure 1 shows the containerization architecture. In this architecture, the different federates can then be configured separately. As long as they use the HELICS library to declare the values they need to interface with, the federates can be started together using the Docker management engine and will behave correctly. To create the Docker containers, Dockerfiles are created for both the Broker and the federates. Docker-compose files are created to declare each federate as a service.

For example the HELICS broker Dockerfile reads as such:

```
FROM ubuntu:22.04
LABEL Name=helics:main Version=0.0.1

SHELL [ "/bin/bash", "--login", "-c" ]
ENV DEBIAN_FRONTEND=noninteractive DEBCONF_NONINTERACTIVE_SEEN=true
ENV TZ="US/NY/New York"


RUN apt update && apt install -y #[Dependecies Truncated]

# Clones Helics
WORKDIR /root/develop
RUN git clone \url{https://github.com/GMLC-TDC/HELICS.git} helics
WORKDIR /root/develop/helics/build

#Compile Helics
RUN cmake -DCMAKE_INSTALL_PREFIX=/helics ..
RUN make -j8 && make install

#Export Shared libaries
ENV LD_LIBRARY_PATH="/helics/lib:${LD_LIBRARY_PATH}"
ENV PATH="/helics/bin:${PATH}"
ENV CPATH="/helics/include:${CPATH}"
RUN cp -r /helics /usr/local/
RUN cp /helics/lib/libhelics.so /usr/lib/

ENV PYTHONPATH /usr/local/python
# Python must be installed after the PYTHONPATH is set above for it to
# recognize and import libhelics.so.
RUN apt install -y --no-install-recommends python3-dev \
  && rm -rf /var/lib/apt/lists/*

#Install python requirements
RUN apt-get update
RUN apt install -y openmpi-bin libopenmpi-dev libpcap-dev python3-pip
RUN pip install  numpy scipy pandas matplotlib posix_ipc virtualenv helics
RUN pip install git+https://github.com/GMLC-TDC/helics-cli.git@main
```

The HELICS Dockerfile can be taken as an example for the other federates as it shows the containerization procedure for both C/C++ federates with a compilation step, and python federates with the pip install step.

To see how HELICS is integrated as a service, see the following excerpt from the docker-compose file:

```
broker:
  build:
    context: ./generic
    dockerfile: ./Dockerfile_helics
  expose:
    - "23404"
  volumes:
    - "./generic/broker.sh:/root/broker.sh"
    - "./generic/helics_simplified:/usr/local/python/helics_simplified"
  command: ./broker.sh 2
  networks:
    - cosim
```

The build section points to the Dockerfile we just described; the expose section indicates that the other containers are expected to reach this service on port 23404. The volumes section mounts some scripts and libraries the service will use, in our case `broker.sh` is the startus script for the HELICS Broker, and `helics_simplified` is the federate-level middleware library that we present in the next section. The command section indicates that the broker expects two connections. The network provides separation between co-simulation services and any additional services one may use in conjunction with ASGARDS–H.

The other federates follow a similar structure, mounting libraries or configuration files as volumes.

4.2 Refactoring the Federates

The next step in the refactor efforts was to decouple the simulation logic from synchronizing time and values during co-simulation. The different simulators integrated into ASGARDS-H use various programming languages. For example, OMNeT++ uses C++, and the attacker and controller simulators use Python. To be integrated into the co-simulation, the simulators that do not originally support HELICS need to use the C-style HELICS library. Using the library as-is in the simulator logic code led to a mixed level of abstraction and lack of separation of concern anti-patterns, as defined by [22]. Moreover, the programming idioms of the C-style HELICS library often conflicted with the native programming idioms of the simulators, such as dynamic typing and iterators for the Python-based simulators and the message passing semantic of OMNeT++.

To address this issue during the refactor, Python and OMNeT++ middleware modules called HELICS-simplified were created. These modules introduced an abstraction layer through iterator synchronization and Class-based publication and subscription interfaces, offering object-oriented capabilities to Python. Similarly, the OMNeT++ module provided an abstraction via an OMNeT++ simple module. These modules allowed each simulator to query the simulation time and values from other simulators using idiomatic code without concern for

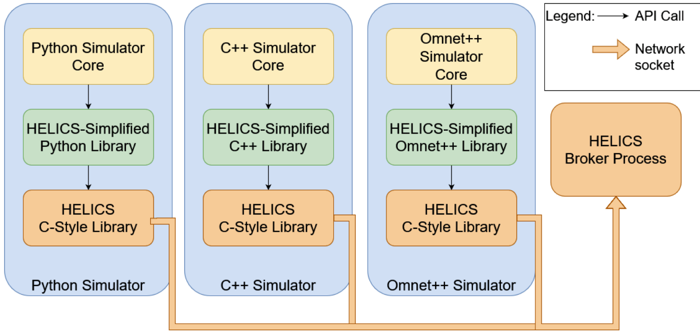

Fig. 2. HELICS-simplified middleware architecture.

the HELICS library semantics. Those modules then use the C-style HELICS library to synchronize time and value among the simulators. The decoupled and middleware-based architecture is illustrated in Fig. 2.

4.3 Updating the Co-simulation Tooling

The original design of ASGARDS-H included a project generator that took a GridLAB-D file as input and produced the necessary configurations for the OMNeT++ simulation, Controller simulator, and HELICS co-simulation framework. This aspect of the project posed challenges from both a compiler theory and toolchain design perspective. The goal of the project generator was to transform simulator specifications (initially obtained from a Graphical User Interface) into a set of configuration files that ensured a coherent co-simulation. However, this tool introduced rigidity in the types of scenarios that could be simulated. After the refactor, we designed a simpler project-generation tool that took as input the scenario files of each simulator (.ini file for OMNeT++ and .glm file for GridLAB-D), as well as the values each simulator expects to exchange through HELICS. The new project-generation tool checks the consistency of the scenario and configures the inter-simulator interfaces before starting a co-simulation. Figure 3 illustrates the overall interaction between the project generator and the co-simulation system.

Table 1. Progress brought by the refactor

ASGARDS-H Property	Before	After
Portability	○	●
Modularity	◐	●
Maintainability	○	●
Versatility	◐	●

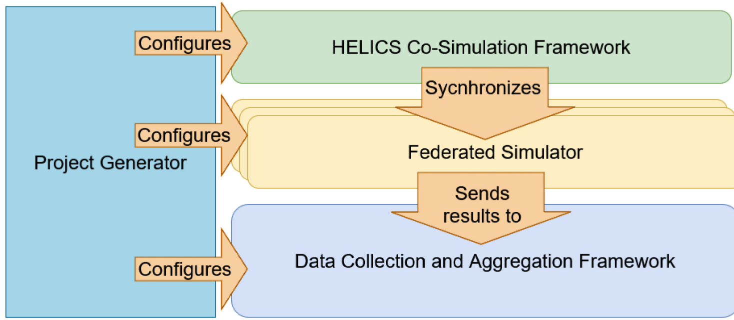

Fig. 3. Co-simulation tooling architecture

By systematically addressing each aspect of the ASGARD-H co-simulation project, the refactoring effort achieved improved modularity and abstraction. Table 1 summarizes the impact of the changes described to the portability, modularity, maintainability, and versatility of ASGARDS-H. The use of docker containers, the introduction of HELICS-simplified modules, the application of the inner dispatch pattern, and the enhancement of the project generator significantly contributed to the overall effectiveness and coherence of the platform.

5 Discussions

As software projects, co-simulation testbeds encounter software development and engineering challenges similar to the ones faced in the industry. As many challenges have been thoroughly documented in the software engineering literature, heeding some of their advice would prevent some of the most common pitfalls for smart grid security researchers.

5.1 Software Engineering Lessons Learned

Adopting evidence-based software engineering practices could have potentially mitigated the shortcomings encountered in the ASGARDS-H co-simulation platform development. By utilizing empirical evidence and best practices to inform decision-making and guide development, several key issues could have been addressed proactively.

Firstly, a systematic approach to software development, including standardized development environments and tools, could have been employed from the outset. This would have ensured greater portability and reduced the ad-hoc nature of the Linux environment, making the platform more accessible and adaptable to different operating systems. Additionally, utilizing established version control systems and documenting the development process would have facilitated knowledge transfer and minimized disruptions caused by the departure of the main developer.

Furthermore, evidence-based practices encourage modular design and code decoupling. By prioritizing modularity and encapsulation during the initial development phase, the platform's maintainability and extensibility could have been improved. Implementing well-defined interfaces between components would have facilitated the integration of control systems and the customization of communication protocols, enabling smoother adaptation to changing requirements.

Moreover, conducting thorough requirements analysis and continuously engaging stakeholders would have reduced the likelihood of significant changes in project specifications. Regular communication with end-users and incorporating feedback into the development process would have ensured that the implemented features aligned with their evolving needs, reducing the need for extensive codebase refactoring.

Finally, adopting rigorous software testing practices, including unit testing, integration testing, and regression testing, would have helped identify and address issues early in the development lifecycle. This would have minimized the impact of bugs and enabled faster iterations, ensuring a more stable and reliable platform.

In conclusion, leveraging evidence-based software engineering practices such as standardized development environments, modular design, thorough requirement analysis, continuous stakeholder engagement, and rigorous testing could have mitigated the shortcomings encountered in the ASGARDS-H co-simulation platform. Embracing these practices fosters a more systematic, adaptable, and reliable development approach, reducing the likelihood of maintainability challenges and facilitating the delivery of robust and feature-rich solutions.

5.2 Advancements in Co-simulation with Functional Mock-Up Interface

The Functional Mock-up Interface [23] (FMI) is a standard for exchanging simulation models between different simulation tools. The last update to the FMI standard, 3.0, released in 2022, includes new features that make it well-suited for creating co-simulation testbeds and digital twins for smart grids.

The Functional Mock-up Interface uses a common interface definition to simulate parts of complex systems encapsulated in Functional Mock-up Units (FMUs). The FMI Application Programming Interface (API) is defined in Common Concepts and allows computations triggered by standardized C functions from the importer into the FMU. Using C as the programming language enables portability and compatibility with all embedded control systems.

Additionally, the FMI Description Schema, defined in XML format, specifies the structure and content of a model description file (modelDescription.xml), containing the definition of all exposed variables, their interdependencies, and capability flags of the FMU. This XML-based approach allows importers to access and store variable definitions with their own representation without the overhead of standardized access functions. The FMU Distribution is done as a single ZIP file, including the modelDescription.xml, the binaries and libraries required for executing FMI functions (.dll or .so files), and the sources of the

FMI functions, with documentation and other data used by the FMU, such as tables or maps.

In the new update to the standard, released after our refactoring efforts, FMI introduces FMI for Co-Simulation, providing a standardized interface for executing simulation models in a co-simulation environment. Unlike FMI for Model Exchange, Co-Simulation FMUs include both the model algorithm and the required solution method. Communication between FMUs is limited to discrete communication points, and the subsystem inside an FMU is solved independently. A co-simulation framework, such as HELICS, should be selected by the researchers to control data exchange and synchronization between FMUs.

As a new standard, FMI for Co-Simulation still needs further investigation. However, it addresses most of the modularity and reusability concerns raised in this paper. A move towards standards like it would foster more flexible and adaptable co-simulation frameworks for the smart grid and CPS research community.

6 Conclusion

In this paper, we discussed the development process and challenges faced in creating a co-simulation testbed for smart grid security. Through the case study of ASGARDS-H, based on the HELICS platform, we highlighted the importance of addressing technical debt, turnover-induced knowledge loss, and other software engineering challenges. We emphasized the need for modularity, maintainability, and versatility in co-simulation platforms to ensure effective and efficient development. By sharing lessons learned and best practices, this paper aims to facilitate the creation of more robust and efficient co-simulation testbeds for smart grid security research.

Acknowledgement. This work is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), RGPIN-2018-06724.

References

1. Bayindir, R., Colak, I., Demirtas, G.K.: Smart grid technologies and applications. Renew. Sustain. Energy Rev. **66**, 499–516 (2016)
2. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Cosimulation: a survey. ACM Comput. Surv. **51**(3), 49:1–49:33 (2018)
3. Mihal, P., Schvarcbacher, M., Rossi, B., Pitner, T.: Smart grids cosimulations: Survey & research directions. Sustain. Comput. Inf. Syst. **35**, 100726 (2022)
4. Sarieddine, K., Sayed, M.A., Jafarigiv, D., Atallah, R., Debbabi, M., Assi, C.: A real-time cosimulation testbed for electric vehicle charging and smart grid security. In: IEEE Security & Privacy, pp. 2–11 (2023)
5. Hammad, E., Ezeme, M., Farraj, A.: Implementation and development of an offline co-simulation testbed for studies of power systems cyber security and control verification. Int. J. Electric. Power Energy Syst. **104**, 817–826 (2019)

6. Palmintier, B., Krishnamurthy, D., Top, P., Smith, S., Daily, J., Fuller, J.: 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), pp. 1–6. (2017)
7. Li, Y., Hou, L., Du, H., et al.: PEMT-CoSim: a co-simulation platform for packetized energy management and trading in distributed energy systems. In: 2022 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm), pp. 96–102 (2022)
8. McDermott, T., Pelton, M., Hardy, T., et al.: Transactive energy simulation platform. (2017). https://doi.org/10.11578/dc.20171025.1920. https://www.osti.gov/biblio/1898731
9. Elliott, R.T., Arabshahi, P., Kirschen, D.S.: A generalized PSS architecture for balancing transient and small-signal response. IEEE Trans. Power Syst. **35**(2), 1446–1456 (2020)
10. Singhal, A., Vu, T.L., Du, W.: Consensus control for coordinating gridforming and grid-following inverters in microgrids. IEEE Trans. Smart Grid **13**(5), 4123–4133 (2022)
11. Aldalur, I., Arrieta, A., Agirre, A., Sagardui, G., Arratibel, M.: A microservice-based framework for multi-level testing of cyber-physical systems. Softw. Quality J. (2023)
12. Lardier, W.: ASGARDS-h: Enabling advanced smart grid cyber-physical attacks, risk and data studies with HELICS, masters, 181p. Concordia University (2020)
13. Battelle Memorial Institute. GridLAB-d simulation software. (2023). [Online]. https://www.gridlabd.org/. Accessed 17 Aug 2023
14. OpenSim Ltd. OMNeT++ discrete event simulator (2019). [Online]. https://omnetpp.org/. Accessed 17 Aug 2023
15. OpenADR Alliance. OpenADR home. (2012).[Online]. https://www.openadr.org/. Accessed 17 Aug 2023
16. Janes, A., Remencius, T., Sillitti, A., Succi, G.: Managing changes in requirements: aempirical investigation. J. Softw. Evol. Process **25**(12), 1273–1283 (2013)
17. Robillard, M.P.: Turnover-induced knowledge loss in practice. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, New York, pp. 1292–1302 (2021)
18. Nardini, G., Stea, G., Virdis, A., Sabella, D.: Simu5g: a system-level simulator for 5g networks. Presented at the 10th International
19. Schneider, K.: Fundamental concepts of knowledge management. In: Schneider, K. (ed.) Experience and Knowledge Management in Software Engineering, pp. 29–66. Springer, Heidelberg (2009)
20. Kruchten, P., Nord, R.L., Ozkaya, I.: Technical debt: from metaphor to theory and practice. IEEE Softw. **29**(6), 18–21 (2012)
21. Docker Inc. Docker: accelerated container application development (May 10, 2022). [Online]. https://www.docker.com/. Accessed 17 Aug 2023
22. Brown, W.H., Malveau, R.C., McCormick, H.W., Mowbray, T.J.: AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, 1st edn. Wiley, New York (1998)
23. Modelica Association. Functional mock-up interface specification. (May 2022). [Online]. https://fmi-standard.org/docs/3.0/. Accessed 17 Aug 2023